Thinking in C++ 2nd edition VERSION TICA18

Revision history:

TICA18: July 29, 1999. Rewrote chapter 8 and added exercises. Replaced all later instances of the "enum hack" with **static const** (which breaks VC++ a lot, since they still haven't implemented this relatively ancient and simple feature). Added compiler support for Visual C++ 6.0 (+Service Pack 3) (although this hasn't been tested with Microsoft's **nmake**; you may need to edit the makefiles, copy **nmake.exe** to **make.exe**, or locate a gnu make in order to get it to work). You can see the results in **CompileDB.txt** in Appendix D. Retested all code with Borland C++ Builder 4 plus the downloadable update, and with egcs from July 18, 1999. Cleaned up some code files that were not being automatically compiled because they didn't have **main( )**s.

TICA17: June 27, 1999. Rewrote chapter 6 and added exercises. Rewrote chapter 7 and added exercises.

TICA16: June 1, 1999. Rewrote chapter 5 and added exercises. Modifications to chapter 19 before and after presentations at the SD conference. Added "Factories" section to design patterns chapter. Rechecked book code under May 24 build of egcs compiler.

TICA15: April 22, 1999. Rewrote chapter 4 and added exercises.

TICA14, March 28, 1999. Rewrote Chapter 2 and 3. I think they're both finished. Chapter 3 is rather big since it covers C syntax fundamentals, along with some C++ basics. Added many exercises to Chapters 2 & 3, to complete them both. Chapter 3 was a "hump" chapter; I think the others in section one shouldn't be as hard. Tried to conform all code in the book to the convention of "type names start with uppercase letters, functions and variables start with lowercase letters".

TICA13, March 9, 1999. Thorough rewrite of chapter one, including the addition of UML diagrams. I think chapter one is finished, now. Reorganized material elsewhere in the book, but that is still in transit. My goal right now is to move through all the chapters in section one, in order.

TICA12, January 15, 1999. Lots of work done on the Design Patterns chapter. All the exsting programs are now modified and redesigned (significantly!) to compile under C++. Added several new examples. Much of the prose in this chapter still needs work, and more patterns and

examples are forthcoming. Changed ExtractCode.cpp so that it generates "bugs" targets for each makefile, containing all the files that won't compile with a particular compiler so they can be re-checked with new compilers. Generates a master in the book's root directory called **makefile.bugs** which descends into each subdirectory and executes **make** with "bugs" as a target and the **–i** flag so you'll see all the errors.

TICA11, January 7, 1999. Completed the STL Algorithms chapter (significant additions and changes), edited and added examples the STL containers chapter. Added many exercises at the ends of both chapters. I consider these both completed now. Added an example or two to the strings chapter.

TICA10, December 28, 1998. Complete rewrite of the **ExtractCode.cpp** program to automatically generate makefiles for each compiler that the book tests, excluding files that the compiler can't handle (these are in a special list in the appendices, so you can see what breaks a compiler, and you can create your own). You now don't need to extract the files yourself (although you still can, for special cases) but instead you just download and unzip a file. All the files in the book (with the exception of the files that are still in Java) now compile with at least one Standard C++ compiler. Added the **trim.h**, **SiteMapConvert.cpp** and **StringCharReplace.cpp** examples to the strings chapter. Added the ProgVals example to chapter 20. Removed all the **strlwr( )** uses (it's a non-standard function).

TICA9, December 15, 1998. Massive work completed on the STL Algorithms chapter; it's quite close to being finished. The long delay was because (1) This chapter took a lot of research and thinking, including other research such as templates; you'll notice the "advanced templates" chapter has more in it's outline (2) I was traveling and giving seminars, etc. I'm entering a two-month hiatus where I'm primarily working on the book and should get a lot accomplished.

TICA8, September 26, 1998. Completed the STL containers chapter.

TICA7, August 14, 1998. Strings chapter modified. Other odds and ends.

TICA6, August 6, 1998. Strings chapter added, still needs some work but it's in fairly good shape. The basic structure for the STL Algorithms chapter is in place and "just" needs to be filled out. Reorganized the chapters; this should be very close to the final organization (unless I discover I've left something out).

TICA5, August 2, 1998: Lots of work done on this version. Everything compiles (except for the design patterns chapter with the Java code)

under Borland C++ 5.3. This is the only compiler that even comes close, but I have high hopes for the next verison of egcs. The chapters and organization of the book is starting to take on more form. A lot of work and new material added in the “STL Containers” chapter (in preparation for my STL talks at the Borland and SD conferences), although that is far from finished. Also, replaced many of the situations in the first edition where I used my home-grown containers with STL containers (typically vector). Changed all header includes to new style (except for C programs): **<iostream>** instead of <**iostream.h**>, **<cstdlib>** instead of **<stdlib.h>**, etc. Adjustment of namespace issues (“using namespace std” in **cpp** files, full qualification of names in header files). Added appendix A to describe coding style (including namespaces). Added “**require.h**” error testing code and used it universally. Rearranged header include order to go from more general to more specific (consistency and style issue described in appendix A). Replaced ‘main( ) {}’ form with ‘int main( ) { }’ form (this relies on the default “return 0” behavior, although some compilers, notably VC++, give warnings). Went through and implemented the class naming policy (following the Java/Smalltalk policy of starting with uppercase etc.) but not the member functions/data members (starting with lowercase etc.). Added appendix A on coding style. Tested code with my modified version of Borland C++ 5.3 (cribbed a corrected ostream_iterator from egcs and <sstream> from elsewhere) so not all the programs will compile with your compiler (VC++ in particular has a lot of trouble with namespaces). On the web site, I added the broken-up versions of the files for easier downloads.

TICA4, July 22, 1998: More changes and additions to the “CGI Programming” section at the end of Chapter 23. I think that section is finished now, with the exception of corrections.

TICA3, July 14, 1998: First revision with content editing (instead of just being a posting to test the formatting and code extraction process). Changes in the end of Chapter 23, on the “CGI Programming” section. Minor tweaks elsewhere. RTF format should be fixed now.

TICA2, July 9, 1998: Changed all fonts to Times and Courier (which are universal); changed distribution format to RTF (readable by most PC and Mac Word Processors, and by at least one on Linux: StarOffice from www.caldera.com. Please let me know if you know about other RTF word processors under Linux).

ToDo: Fix autobuild of make test makefile (remove backslashes); add test arguments (what about some kind of autofill via redirection?). Differentiate copy-assignment operator= from other forms of operator=.

HorseRace game as example of random number generator in early chapter? Change header numbering scheme as suggested?

The instructions on the web site (http://www.BruceEckel.com/ThinkingInCPP2e.html) show you how to extract code for both Win32 systems and Linux (only Red Hat Linux 5.0/5.1 has been tested). The contents of the book, including the contents of the source-code files generated during automatic code extraction, are not intended to indicate any accurate or finished form of the book or source code.

Please only add comments/corrections using the form found on http://www.BruceEckel.com/ThinkingInCPP2e.html

Please note that the book files are only available in Rich Text Format (RTF) or plain ASCII text without line breaks (that is, each paragraph is on a single line, so if you bring it into a typical text editor that does line wrapping, it will read decently). Please see the Web page for information about word processors that support RTF. The only fonts used are Times and Courier (so there should be no font difficulties); if you find any other fonts please report the location.

Thanks for your participation in this project.

Bruce Eckel

“This book is a tremendous achievement. You owe it to yourself to have a copy on your shelf. The chapter on iostreams is the most comprehensive and understandable treatment of that subject I’ve seen to date.”

Al Stevens
Contributing Editor, Doctor Dobbs Journal

“Eckel’s book is the only one to so clearly explain how to rethink program construction for object orientation. That the book is also an excellent tutorial on the ins and outs of C++ is an added bonus.”

Andrew Binstock
Editor, Unix Review

“Bruce continues to amaze me with his insight into C++, and *Thinking in C++* is his best collection of ideas yet. If you want clear answers to difficult questions about C++, buy this outstanding book.”

Gary Entsminger
Author, *The Tao of Objects*

“*Thinking in C++* patiently and methodically explores the issues of when and how to use inlines, references, operator overloading, inheritance and dynamic objects, as well as advanced topics such as the proper use of templates, exceptions and multiple inheritance. The entire effort is woven in a fabric that includes Eckel’s own philosophy of object and program design. A must for every C++ developer’s bookshelf, *Thinking in C++* is the one C++ book you must have if you’re doing serious development with C++.”

Richard Hale Shaw
Contributing Editor, PC Magazine

Thinking In C++

Bruce Eckel
President, MindView Inc.

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
http://www.phptr.com

Publisher: Alan Apt
Production Editor: Mona Pompilli
Development Editor: Sondra Chavez
Book Design, Cover Design and Cover Photo:
Daniel Will-Harris, daniel@will-harris.com
Copy Editor: Shirley Michaels
*Production Coordinator:*Lori Bulwin
Editorial Assistant: Shirley McGuire

Published by Prentice Hall Inc.
A Paramount Communications Company
Englewood Cliffs, New Jersey 07632

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-917709-4

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada, Inc., *Toronto*

Prentice-Hall Hisapnoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

dedication

to the scholar, the healer, and the muse

What's inside...

Preface

> Like any human language, C++ provides a way to express concepts. If successful, this medium of expression will be significantly easier and more flexible than the alternatives as problems grow larger and more complex.

You can't just look at C++ as a collection of features; some of the features make no sense in isolation. You can only use the sum of the parts if you are thinking about *design*, not simply coding. And to understand C++ in this way, you must understand the problems with C and with programming in general. This book discusses programming problems, why they are problems, and the approach C++ has taken to solve such problems. Thus, the set of features I explain in each chapter will be based on the way I see a particular type of problem being solved with the language. In this way I hope to move you, a little at a time, from understanding C to the point where the C++ mindset becomes your native tongue.

Throughout, I'll be taking the attitude that you want to build a model in your head that allows you to understand the language all the way down to the bare metal; if you encounter a puzzle you'll be able to feed it to your model and deduce the answer. I will try to convey to you the insights which have rearranged my brain to make me start "thinking in C++."

Prerequisites

In the first edition of this book, I decided to assume that someone else had taught you C and that you have at least a reading level of comfort with it. My primary focus was on simplifying what I found difficult – the C++ language. In this edition I have added a chapter that is a very rapid introduction to C, assuming that you have some kind of programming experience already. In addition, just as you learn many new words intuitively by seeing them in context in a novel, it's possible to learn a

great deal about C from the context in which it is used in the rest of the book.

Thinking in C

For those of you who need a gentler introduction to C than the chapter in this book, I have created with Chuck Allison a CD ROM called “Thinking in C: foundations for Java and C++” which will introduce you to the aspects of C that are necessary for you to move on to C++ or Java (leaving out the nasty bits that C programmers must deal with on a day-to-day basis but that the C++ and Java languages steer you away from). This CD can be ordered at http://www.BruceEckel.com. [Note: the CD will not be available until late Fall 98, at the earliest – watch the Web site for updates]

Learning C++

I clawed my way into C++ from exactly the same position as I expect the readers of this book will: As a C programmer with a very no-nonsense, nuts-and-bolts attitude about programming. Worse, my background and experience was in hardware-level embedded programming, where C has often been considered a high-level language and an inefficient overkill for pushing bits around. I discovered later that I wasn’t even a very good C programmer, hiding my ignorance of structures, **malloc( )** & **free( )**, **setjmp( )** & **longjmp( )**, and other “sophisticated” concepts, scuttling away in shame when the subjects came up in conversation rather than reaching out for new knowledge.

When I began my struggle to understand C++, the only decent book was Stroustrup’s self-professed “expert’s guide,[1] ” so I was left to simplify the basic concepts on my own. This resulted in my first C++ book,[2] which was essentially a brain dump of my experience. That was designed as a reader’s guide, to bring programmers into C and C++ at the same time. Both editions[3] of the book garnered an enthusiastic response and I still feel it is a valuable resource.

---

[1] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986 (first edition).

[2] *Using C++*, ibid.

[3] *Using C++* and *C++ Inside & Out*, ibid.

At about the same time that *Using C++* came out, I began teaching the language. Teaching C++ has become my profession; I've seen nodding heads, blank faces, and puzzled expressions in audiences all over the world since 1989. As I began giving in-house training with smaller groups of people, I discovered something during the exercises. Even those people who were smiling and nodding were confused about many issues. I found out, by chairing the C++ track at the Software Development Conference for the last three years, that I and other speakers tended to give the typical audience too many topics, too fast. So eventually, through both variety in the audience level and the way that I presented the material, I would end up losing some portion of the audience. Maybe it's asking too much, but because I am one of those people resistant to traditional lecturing (and for most people, I believe, such resistance results from boredom), I wanted to try to keep everyone up to speed.

For a time, I was creating a number of different presentations in fairly short order. Thus, I ended up learning by experiment and iteration (a technique that also works well in C++ program design). Eventually I developed a course using everything I had learned from my teaching experience, one I would be happy giving for a long time. It tackles the learning problem in discrete, easy-to-digest steps and for a hands-on seminar (the ideal learning situation), there are exercises following each of the short lessons.

This book developed over the course of two years, and the material in this book has been road-tested in many forms in many different seminars. The feedback that I've gotten from each seminar has helped me change and refocus the material until I feel it works well as a teaching medium. But it isn't just a seminar handout – I tried to pack as much information as I could within these pages, and structure it to draw you through, onto the next subject. More than anything, the book is designed to serve the solitary reader, struggling with a new programming language.

Goals

My goals in this book are to:

1. Present the material a simple step at a time, so the reader can easily digest each concept before moving on.

2. Use examples that are as simple and short as possible. This sometimes prevents me from tackling "real-world" problems,

but I've found that beginners are usually happier when they can understand every detail of an example rather than being impressed by the scope of the problem it solves. Also, there's a severe limit to the amount of code that can be absorbed in a classroom situation. For this I will no doubt receive criticism for using "toy examples," but I'm willing to accept that in favor of producing something pedagogically useful. Those who want more complex examples can refer to the later chapters of *C++ Inside & Out*.[4]

3. Carefully sequence the presentation of features so that you aren't seeing something you haven't been exposed to. Of course, this isn't always possible; in those situations, a brief introductory description will be given.

4. Give you what I think is important for you to understand about the language, rather than everything I know. I believe there is an "information importance hierarchy," and there are some facts that 95% of programmers will never need to know, but would just confuse people and add to their perception of the complexity of the language – and C++ is now considered to be more complex than ADA! To take an example from C, if you memorize the operator precedence table (I never did) you can write clever code. But if you have to think about it, it will confuse the reader/maintainer of that code. So forget about precedence, and use parentheses when things aren't clear. This same attitude will be taken with some information in the C++ language, which I think is more important for compiler writers than for programmers.

5. Keep each section focused enough so the lecture time – and the time between exercise periods – is small. Not only does this keep the audience' minds more active and involved during a hands-on seminar, but it gives the reader a greater sense of accomplishment.

6. Provide the reader with a solid foundation so they can understand the issues well enough to move on to more difficult coursework and books.

---

[4] Ibid.

7. I've endeavored not to use any particular vendor's version of C++ because, for learning the language, I don't feel like the details of a particular implementation are as important as the language itself. Most vendors' documentation concerning their own implementation specifics is adequate.

Chapters

C++ is a language where new and different features are built on top of an existing syntax. (Because of this it is referred to as a *hybrid* object-oriented programming language.) As more people have passed through the learning curve, we've begun to get a feel for the way C programmers move through the stages of the C++ language features. Because it appears to be the natural progression of the C-trained mind, I decided to understand and follow this same path, and accelerate the process by posing and answering the questions that came to me as I learned the language and that came from audiences as I taught it.

This course was designed with one thing in mind: the way people learn the C++ language. Audience feedback helped me understand which parts were difficult and needed extra illumination. In the areas where I got ambitious and included too many features all at once, I came to know – through the process of presenting the material – that if you include a lot of new features, you have to explain them all, and the student's confusion is easily compounded. As a result, I've taken a great deal of trouble to introduce the features as few at a time as possible; ideally, only one at a time per chapter.

The goal, then, is for each chapter to teach a single feature, or a small group of associated features, in such a way that no additional features are relied upon. That way you can digest each piece in the context of your current knowledge before moving on. To accomplish this, I leave many C features in place much longer than I would prefer. For example, I would like to be using the C++ iostreams IO library right away, instead of using the **printf( )** family of functions so familiar to C programmers, but that would require introducing the subject prematurely, and so many of the early chapters carry the C library functions with them. This is also true with many other features in the language. The benefit is that you, the C programmer, will not be confused by seeing all the C++ features used before they are explained, so your introduction to the language will be gentle and will mirror the way you will assimilate the features if left to your own devices.

Here is a brief description of the chapters contained in this book [[Please note this section will not be updated until all the chapters are in place]]

(0) The evolution of objects. When projects became too big and too complicated to easily maintain, the "software crisis" was born, saying, "We can't get projects done, and if we can they're too expensive!" This precipitated a number of responses, which are discussed in this chapter along with the ideas of object-oriented programming (OOP) and how it attempts to solve the software crisis. You'll also learn about the benefits and concerns of adopting the language and suggestions for moving into the world of C++.

(1) Data abstraction. Most features in C++ revolve around this key concept: the ability to create new data types. Not only does this provide superior code organization, but it lays the ground for more powerful OOP abilities. You'll see how this idea is facilitated by the simple act of putting functions inside structures, the details of how to do it, and what kind of code it creates.

(2) Hiding the implementation. You can decide that some of the data and functions in your structure are unavailable to the user of the new type by making them **private**. This means you can separate the underlying implementation from the interface that the client programmer sees, and thus allow that implementation to be easily changed without affecting client code. The keyword **class** is also introduced as a fancier way to describe a new data type, and the meaning of the word "object" is demystified (it's a variable on steroids).

(3) Initialization & cleanup. One of the most common C errors results from uninitialized variables. The *constructor* in C++ allows you to guarantee that variables of your new data type ("objects of your class") will always be properly initialized. If your objects also require some sort of cleanup, you can guarantee that this cleanup will always happen with the C++ *destructor*.

(4) Function overloading & default arguments. C++ is intended to help you build big, complex projects. While doing this, you may bring in multiple libraries that use the same function name, and you may also choose to use the same name with different meanings within a single library. C++ makes this easy with *function overloading*, which allows you to reuse the same function name as long as the argument lists are different. Default arguments allow you to call the same function in different ways by automatically providing default values for some of your arguments.

(5) Introduction to iostreams. One of the original C++ libraries – the one that provides the essential I/O facility – is called iostreams. Iostreams is intended to replace C's **stdio.h** with an I/O library that is easier to use, more flexible, and extensible – you can adapt it to work with your new classes. This chapter teaches you the ins and outs of how to make the best use of the existing iostream library for standard I/O, file I/O, and in-memory formatting.

(6) Constants. This chapter covers the **const** and **volatile** keywords that have additional meaning in C++, especially inside classes. It also shows how the meaning of **const** varies inside and outside classes and how to create compile-time constants in classes.

(7) Inline functions. Preprocessor macros eliminate function call overhead, but the preprocessor also eliminates valuable C++ type checking. The inline function gives you all the benefits of a preprocessor macro plus all the benefits of a real function call.

(8) Name control. Creating names is a fundamental activity in programming, and when a project gets large, the number of names can be overwhelming. C++ allows you a great deal of control over names: creation, visibility, placement of storage, and linkage. This chapter shows how names are controlled using two techniques. First, the **static** keyword is used to control visibility and linkage, and its special meaning with classes is explored. A far more useful technique for controlling names at the global scope is C++'s **namespace** feature, which allows you to break up the global name space into distinct regions.

(9) References & the copy-constructor. C++ pointers work like C pointers with the additional benefit of stronger C++ type checking. There's a new way to handle addresses; from Algol and Pascal, C++ lifts the *reference* which lets the compiler handle the address manipulation while you use ordinary notation. You'll also meet the copy-constructor, which controls the way objects are passed into and out of functions by value. Finally, the C++ pointer-to-member is illuminated.

(10) Operator overloading. This feature is sometimes called "syntactic sugar." It lets you sweeten the syntax for using your type by allowing operators as well as function calls. In this chapter you'll learn that operator overloading is just a different type of function call and how to write your own, especially the sometimes-confusing uses of arguments, return types, and making an operator a member or friend.

(11) Dynamic object creation. How many planes will an air-traffic system have to handle? How many shapes will a CAD system need? In the

general programming problem, you can't know the quantity, lifetime or type of the objects needed by your running program. In this chapter, you'll learn how C++'s **new** and **delete** elegantly solve this problem by safely creating objects on the heap.

(12) Inheritance & composition. Data abstraction allows you to create new types from scratch; with composition and inheritance, you can create new types from existing types. With composition you assemble a new type using other types as pieces, and with inheritance you create a more specific version of an existing type. In this chapter you'll learn the syntax, how to redefine functions, and the importance of construction and destruction for inheritance & composition.

(13) Polymorphism & virtual functions. On your own, you might take nine months to discover and understand this cornerstone of OOP. Through small, simple examples you'll see how to create a family of types with inheritance and manipulate objects in that family through their common base class. The **virtual** keyword allows you to treat all objects in this family generically, which means the bulk of your code doesn't rely on specific type information. This makes your programs extensible, so building programs and code maintenance is easier and cheaper.

(14) Templates & container classes. Inheritance and composition allow you to reuse object code, but that doesn't solve all your reuse needs. Templates allow you to reuse *source* code by providing the compiler with a way to substitute type names in the body of a class or function. This supports the use of *container class* libraries, which are important tools for the rapid, robust development of object-oriented programs. This extensive chapter gives you a thorough grounding in this essential subject.

(15) Multiple inheritance. This sounds simple at first: A new class is inherited from more than one existing class. However, you can end up with ambiguities and multiple copies of base-class objects. That problem is solved with virtual base classes, but the bigger issue remains: When do you use it? Multiple inheritance is only essential when you need to manipulate an object through more than one common base class. This chapter explains the syntax for multiple inheritance, and shows alternative approaches – in particular, how templates solve one common problem. The use of multiple inheritance to repair a "damaged" class interface is demonstrated as a genuinely valuable use of this feature.

(16) Exception handling. Error handling has always been a problem in programming. Even if you dutifully return error information or set a flag, the function caller may simply ignore it. Exception handling is a primary

feature in C++ that solves this problem by allowing you to “throw” an object out of your function when a critical error happens. You throw different types of objects for different errors, and the function caller “catches” these objects in separate error handling routines. If you throw an exception, it cannot be ignored, so you can guarantee that *something* will happen in response to your error.

(17) Run-time type identification. Run-time type identification (RTTI) lets you find the exact type of an object when you only have a pointer or reference to the base type. Normally, you’ll want to intentionally ignore the exact type of an object and let the virtual function mechanism implement the correct behavior for that type. But occasionally it is very helpful to know the exact type of an object for which you only have a base pointer; often this information allows you to perform a special-case operation more efficiently. This chapter explains what RTTI is for and how to use it.

Exercises

I’ve discovered that simple exercises are exceptionally useful during a seminar to complete a student’s understanding, so you’ll find a set at the end of each chapter.

These are fairly simple, so they can be finished in a reasonable amount of time in a classroom situation while the instructor observes, making sure all the students are absorbing the material. Some exercises are a bit more challenging to keep advanced students entertained. They’re all designed to be solved in a short time and are only there to test and polish your knowledge rather than present major challenges (presumably, you’ll find those on your own – or more likely they’ll find you).

Source code

The source code for this book is copyrighted freeware, distributed via the web site http://www.BruceEckel.com. The copyright prevents you from republishing the code in print media without permission.

To unpack the code, you download the text version of the book and run the program **ExtractCode** (from chapter 23), the source for which is also provided on the Web site. The program will create a directory for each chapter and unpack the code into those directories. In the starting

directory where you unpacked the code you will find the following copyright notice:

```
//:! :CopyRight.txt
Copyright (c) Bruce Eckel, 1999
Source code file from the book "Thinking in C++"
All rights reserved EXCEPT as allowed by the
following statements: You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C++" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
```

```
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

You may use the code in your projects and in the classroom as long as the copyright notice is retained.

Coding standards

In the text of this book, identifiers (function, variable, and class names) will be set in **bold**. Most keywords will also be set in bold, except for those keywords which are used so much that the bolding can become tedious, like class and virtual.

I use a particular coding style for the examples in this book. It was developed over a number of years, and was inspired by Bjarne Stroustrup's style in his original *The C++ Programming Language*.[5] The subject of formatting style is good for hours of hot debate, so I'll just say I'm not trying to dictate correct style via my examples; I have my own motivation for using the style that I do. Because C++ is a free-form programming language, you can continue to use whatever style you're comfortable with.

The programs in this book are files that are automatically extracted from the text of the book, which allows them to be tested to ensure they work correctly. (I use a special format on the first line of each file to facilitate this extraction; the line begins with the characters '**/**' '**/**' '**:**' and the file name and path information.) Thus, the code files printed in the book should all work without compiler errors when compiled with an implementation that conforms to Standard C++ (note that not all compilers support all language features). The errors that *should* cause compile-time error messages are commented out with the comment **//!** so they can be easily discovered and tested using automatic means. Errors discovered and reported to the author will appear first in the electronic version of the book (at **www.BruceEckel.com**) and later in updates of the book.

[5] Ibid.

One of the standards in this book is that all programs will compile and link without errors (although they will sometimes cause warnings). To this end, some of the programs, which only demonstrate a coding example and don't represent stand-alone programs, will have empty **main( )** functions, like this

```
int main() {}
```

This allows the linker to complete without an error.

The standard for **main( )** is to return an **int**, but Standard C++ states that if there is no **return** statement inside **main( )**, the compiler will automatically generate code to **return 0**. This option will be used in this book (although some compilers may still generate warnings for this).

Language standards

Throughout this book, when referring to conformance to the ANSI/ISO C standard, I will generally just say '**C**.' Only if it is necessary to distinguish between Standard C and older, pre-Standard versions of C will I make the distinction.

At this writing the ANSI/ISO C++ committee was finished working on the language. Thus, I will use the term *Standard C++* to refer to the standardized language. If I simply refer to C++ you should assume I mean "Standard C++."

Language support

Your compiler may not support all the features discussed in this book, especially if you don't have the newest version of your compiler. Implementing a language like C++ is a Herculean task, and you can expect that the features will appear in pieces rather than all at once. But if you attempt one of the examples in the book and get a lot of errors from the compiler, it's not necessarily a bug in the code or the compiler – it may simply not be implemented in your particular compiler yet.

Seminars & CD Roms

My company provides public hands-on training seminars based on the material in this book. Selected material from each chapter represents a lesson, which is followed by a monitored exercise period so each student

receives personal attention. Information and sign-up forms for upcoming seminars can be found at **http://www.BruceEckel.com**. If you have specific questions, you may direct them to **Bruce@EckelObjects.com**.

Errors

No matter how many tricks a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader. If you discover anything you believe to be an error, please use the correction form you will find at **http://www.BruceEckel.com**. Your help is appreciated.

Acknowledgements

The ideas and understanding in this book have come from many sources: friends like Dan Saks, Scott Meyers, Charles Petzold, and Michael Wilk; pioneers of the language like Bjarne Stroustrup, Andrew Koenig, and Rob Murray; members of the C++ Standards Committee like Nathan Myers (who was particularly helpful and generous with his insights), Tom Plum, Reg Charney, Tom Penello, Chuck Allison, Sam Druker, and Uwe Steinmueller; people who have spoken in my C++ track at the Software Development Conference; and very often students in my seminars, who ask the questions I need to hear in order to make the material clearer.

I have been presenting this material on tours produced by Miller Freeman Inc. with my friend Richard Hale Shaw. Richard's insights and support have been very helpful (and Kim's, too). Thanks also to KoAnn Vikoren, Eric Faurot, Jennifer Jessup, Nicole Freeman, Barbara Hanscome, Regina Ridley, Alex Dunne, and the rest of the cast and crew at MFI.

The book design, cover design, and cover photo were created by my friend Daniel Will-Harris, noted author and designer, who used to play with rub-on letters in junior high school while he awaited the invention of computers and desktop publishing. However, I produced the camera-ready pages myself, so the typesetting errors are mine. Microsoft® Word for Windows 97 was used to write the book and to create camera-ready pages. The body typeface is [Times for the electronic distribution] and the headlines are in [Times for the electronic distribution].

The people at Prentice Hall were wonderful. Thanks to Alan Apt, Sondra Chavez, Mona Pompili, Shirley McGuire, and everyone else there who made life easy for me.

A special thanks to all my teachers, and all my students (who are my teachers as well).

Personal thanks to my friends Gen Kiyooka and Kraig Brockschmidt. The supporting cast of friends includes, but is not limited to: Zack Urlocker, Andrew Binstock, Neil Rubenking, Steve Sinofsky, JD Hildebrandt, Brian McElhinney, Brinkley Barr, Larry O'Brien, Bill Gates at Midnight Engineering Magazine, Larry Constantine & Lucy Lockwood, Tom Keffer, Greg Perry, Dan Putterman, Christi Westphal, Gene Wang, Dave Mayer, David Intersimone, Claire Sawyers, Claire Jones, The Italians (Andrea Provaglio, Laura Fallai, Marco Cantu, Corrado, Ilsa and Christina Giustozzi), Chris & Laura Strand, The Almquists, Brad Jerbic, Marilyn Cvitanic, The Mabrys, The Haflingers, The Pollocks, Peter Vinci, The Robbins Families, The Moelter Families (& the McMillans), The Wilks, Dave Stoner, Laurie Adams, The Penneys, The Cranstons, Larry Fogg, Mike & Karen Sequeira, Gary Entsminger & Allison Brody, Chester Andersen, Joe Lordi, Dave & Brenda Bartlett, The Rentschlers, The Sudeks, Lynn & Todd, and their families. And of course, Mom & Dad.

1: Introduction to objects

> The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.

But computers are not so much machines as they are mind amplification tools (“bicycles for the mind,” as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other expressive mediums such as writing, painting, sculpture, animation and filmmaking. Object-oriented programming is part of this movement toward the computer as an expressive medium.

This chapter will introduce you to the basic concepts of object-oriented programming (OOP), including an overview of OOP development methods. This chapter, and this book, assume you have had experience in some programming language, although not necessarily C. If you feel you need more preparation in programming and the syntax of C before tackling this book, you may want to consider MindView’s “Thinking in C: Foundations for C++ and Java” training CD ROM, available at http://www.MindView.net.

This chapter is background and supplementary material. Many people do not feel comfortable wading into object-oriented programming without understanding the big picture first. Thus, there are many concepts that are introduced here to give you a solid overview of OOP. However, many other people don’t get the big picture concepts until they’ve seen some of the mechanics first; these people may become bogged down and lost without some code to get their hands on. If you’re part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter – skipping it at this point will not prevent you from writing programs or learning the language. However, you will want to come back here eventually, to fill in your knowledge so that you can understand why objects are important and how to design with them.

The progress of abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean "what is it that you are abstracting?" Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (such as Fortran, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space," which is the place where you're modeling that problem, such as a computer) and the model of the problem that is actually being solved (in the "problem space," which is the place where the problem actually exists). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("all problems are ultimately lists" or "all problems are algorithmic"). PROLOG casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. (The latter proved to be too restrictive.) Each of these approaches is a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (Of course, you will also need other objects that don't have problem-space analogs.) The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction

than what we've had before. Thus OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer, though. Each object looks quite a bit like a little computer; it has a state, and it has operations that you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world; they all have characteristics and behaviors.

Some language designers have decided that object-oriented programming itself is not adequate to easily solve all programming problems, and advocate the combination of various approaches into *multiparadigm* programming languages.[6]

Alan Kay summarized five basic characteristics of Smalltalk, the first successful object-oriented language and one of the languages upon which C++ is based. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object.** Think of an object as a fancy variable; it stores data, but you can "make requests" to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a function that belongs to a particular object.
3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity in a program while hiding it behind the simplicity of objects.
4. **Every object has a type**. Using the parlance, each object is an *instance* of a *class*, where "class" is synonymous with "type." The most important distinguishing characteristic of a class is "what messages can you send to it?"
5. **All objects of a particular type can receive the same messages**. This is actually a very loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to receive shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the

[6] See *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

description of a shape. This *substitutability* is one of the most powerful concepts in OOP.

An object has an interface

Aristotle was probably the first to begin a careful study of the concept of *type*; he spoke of things such as "the class of fishes and the class of birds." The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was directly used in the first object-oriented language, Simula-67, with its fundamental keyword **class** that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem[7]." In this, you have a bunch of tellers, customers, accounts, transactions, units of money – a lot of "objects." Objects that are identical except for their state during a program's execution are grouped together into "classes of objects" and that's where the keyword **class** came from. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You can create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests*; you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state, each account has a different balance, each teller has a name. Thus the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versa[8].

---

[7] You can find an interesting implementation of this problem later in the book.

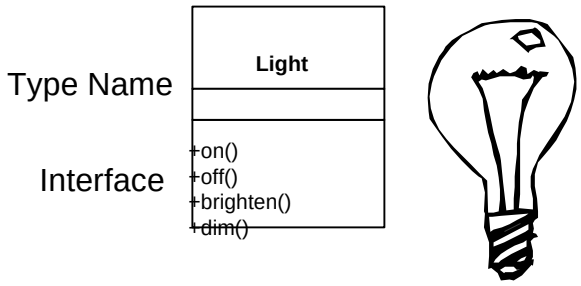
[8] Some people make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

Since a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system welcomes the new classes and gives them all the care and type-checking that it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution.

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space.

But how do you get an object to do useful work for you? There must be a way to make a request of that object so it will do something, such as complete a transaction, draw something on the screen or turn on a switch. And each object can satisfy only certain requests. The requests you can make of an object are defined by its *interface*, and the type is what determines the interface. A simple example might be a representation of a light bulb:




```
Light lt;
lt.on();
```

The interface establishes *what* requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. From a procedural programming standpoint, it's not that complicated. A type has

a function associated with each possible request, and when you make a particular request to an object, that function is called. This process is usually summarized by saying that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the type/class is **Light**, the name of this particular **Light** object is **lt**, and the requests that you can make of a **Light** object are to turn it on, turn it off, make it brighter or make it dimmer. You create a **Light** object by simply declaring a name (**lt**) for that identifier. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a pre-defined class, that's pretty much all there is to programming with objects.

The diagram shown above follows the format of the *Unified Modeling Language* (UML). Each class is represented by a box, with the type name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the *member functions* (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. The '+' signs before the member functions indicate they are public. Very often, only the name of the class and the public member functions are shown in UML design diagrams, and so the middle portion is not shown. If you're only interested in the class name, then the bottom portion doesn't need to be shown, either.

The hidden implementation

It is helpful to break up the playing field into *class creators* (those who create new data types) and *client programmers*[9] (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact to anyone else. The hidden portions usually represent the tender insides of an object

[9] I'm indebted to my friend Scott Meyers for this term.

that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs. The concept of implementation hiding cannot be overemphasized.

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library.

If all the members of a class are available to everyone, then the client programmer can do anything with that class and there's no way to enforce any rules. Even though you might really prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch – parts that are necessary for the internal machinations of the data type but not part of the interface that users need to solve their particular problems. This is actually a service to users because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, and then later discover you need to rewrite it in order to make it run faster. If the interface and implementation are clearly separated and protected, you can easily accomplish this and require only a relink by the user.

C++ uses three explicit keywords to set the boundaries in a class: **public**, **private**, **protected**. Their use and meaning are quite straightforward. These *access specifiers* determine who can use the definitions that follow. **public** means the following definitions are available to everyone. The **private** keyword, on the other hand, means that no one can access those definitions except you, the creator of the type, inside function members of that type. **private** is a brick wall between you and the client programmer. If someone tries to access a **private** member, they'll get a compile-time error. **protected** acts just like **private**, with the exception that an inheriting class has access to **protected** members, but not **private** members. Inheritance will be introduced shortly.

Reusing the implementation

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a good design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, whatever is necessary to achieve the functionality desired in your new class. This concept is called *composition* (or more generally, *aggregation*), since you are composing a new class from existing classes. Sometimes composition is referred to as a "has-a" relationship, as in "a car has an engine."

(The above UML diagram indicates composition with the filled diamond, which states there is one car.)

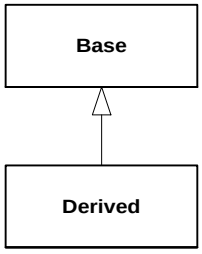
Composition comes with a great deal of flexibility. The member objects of your new class are usually private, making them inaccessible to client programmers using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at run time, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Because inheritance is so important in object-oriented programming it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overcomplicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will stay cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

Inheritance: reusing the interface

By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the **class** keyword.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it and make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base* or *super* or *parent* class) is changed, the modified "clone" (called the *derived* or *inherited* or *sub* or *child* class) also reflects those changes.



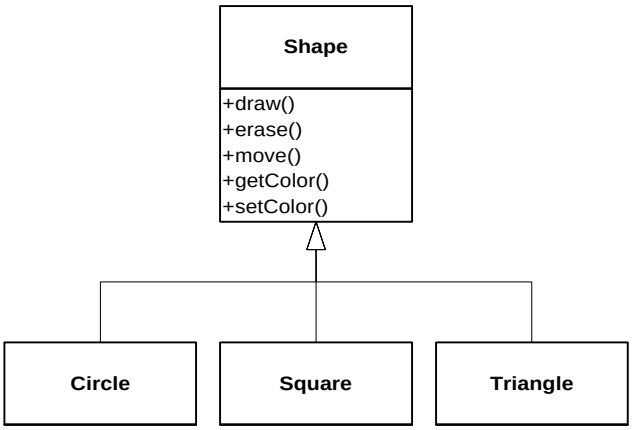
(The arrow in the above UML diagram points from the derived class to the base class. As you shall see, there can be more than one derived class.)

A type does more than describe the constraints on a set of objects; it also has a relationship with other types. Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently). Inheritance expresses this similarity between types with the concept of base types and derived types. A base type contains all the characteristics and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash," and each piece of trash has a weight, a value, and so on

and can be shredded, melted, or decomposed. From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic). In addition, some behaviors may be different (the value of paper depends on its type and condition). Using inheritance, you can build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the classic shape problem, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited): circle, square, triangle, and so on, each of which may have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors may be different (calculating the area of a shape). The type hierarchy embodies both the



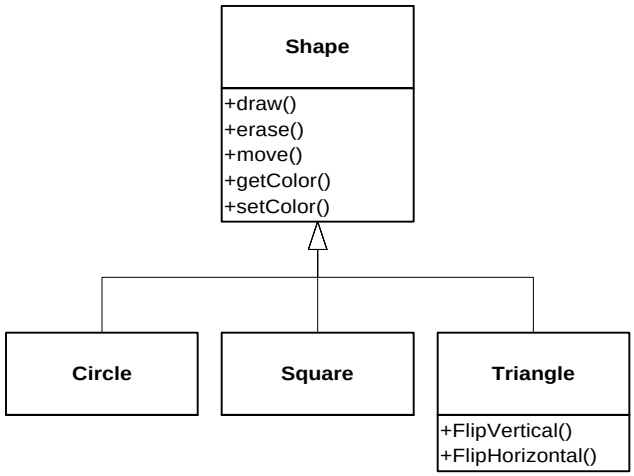
similarities and differences between the shapes.

Casting the solution in the same terms as the problem is tremendously beneficial because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is the primary model, so you go directly from the description of the system in the real world to the description of the system in code. Indeed, one of the difficulties people have with object-oriented design is that it's too simple to get from the beginning to the end. A mind trained to look for complex solutions is often stumped by this simplicity at first.

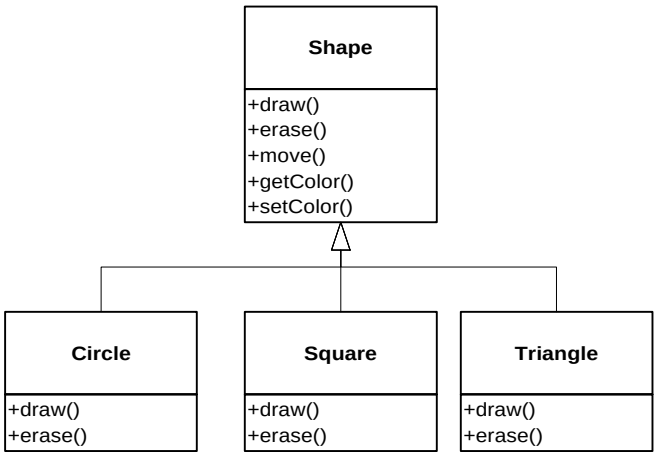
When you inherit from an existing type, you create a new type. This new type contains not only all the members of the existing type (although the **private** ones are hidden away and inaccessible), but more importantly it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived class *is the same type as the base class*. In the above example, "a circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Since both the base class and derived class have the same interface, there must be some implementation to go along with that interface. That is, there must be some code to execute when an object receives a particular message. If you simply inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

You have two ways to differentiate your new derived class from the original base class. The first is quite straightforward: you simply add brand new functions to the derived class. These new functions are not part of the base class interface. This means that the base class simply didn't do as much as you wanted it to, so you added more functions. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class might also need these additional functions. This process of discovery and iteration of your design happens regularly in object-oriented programming.



Although inheritance may sometimes imply that you are going to add new functions to the interface, that's not necessarily true. The second way to differentiate your new class is to *change* the behavior of an existing base-



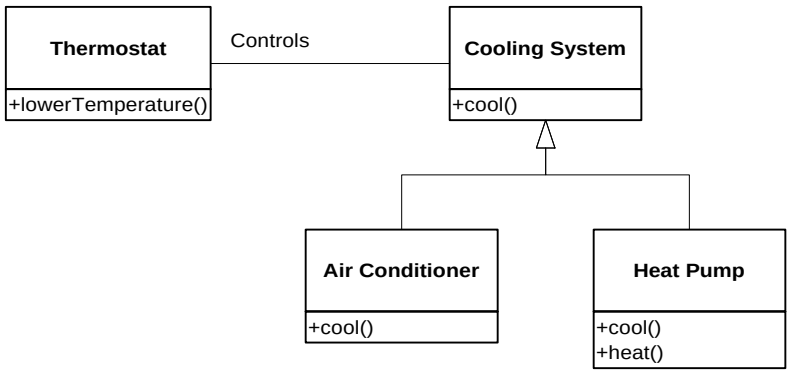
class function. This is referred to as *overriding* that function.

To override a function, you simply create a new definition for the function in the derived class. You're saying "I'm using the same interface function here, but I want it to do something different for my new type."

Is-a vs. is-like-a relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class functions (and not add new member functions that aren't in the base class)? This would mean that the derived type is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often referred to as the *substitution principle*. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say "a circle *is a* shape." A test for inheritance is whether you can state the is-a relationship about the classes and have it make sense.

There are times when you must add new interface elements to a derived type, thus extending the interface and creating a new type. The new type can still be substituted for the base type, but the substitution isn't perfect because your new functions are not accessible from the base type. This can be described as an *is-like-a* relationship; the new type has the interface of the old type but it also contains other functions, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that allows you to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object has been extended, and the existing system doesn't know about anything except the original interface.



Of course, once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so that it can also include heating – at which point the substitution principle will work. However, the above diagram is an example of what happens in design and in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it *is* nice if your design works out that way. But you'll find that there are times when it's equally clear that you must add new functions to the interface of a derived class. With inspection both cases should be reasonably obvious.

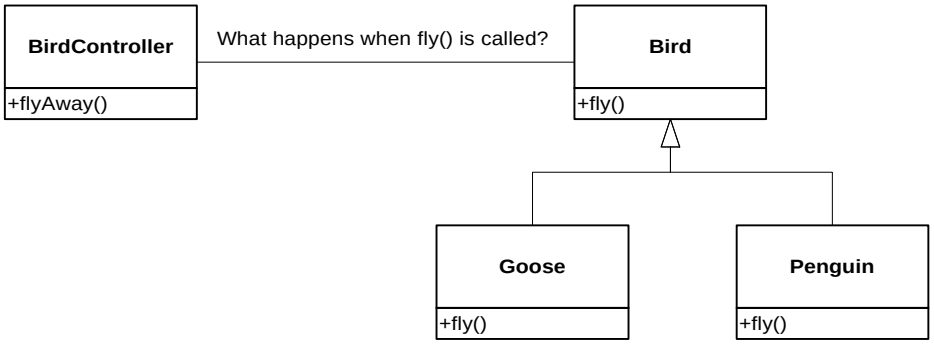
Interchangeable objects with polymorphism

When dealing with type hierarchies, you often want to treat an object not as the specific type that it is but instead as its base type. This allows you to write code that doesn't depend on specific types. In the shape example, functions manipulate generic shapes without respect to whether they're circles, squares, triangles, and so on. All shapes can be drawn, erased, and moved, so these functions simply send a message to a shape object; they don't worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the functions that deal only with generic shapes. This ability to extend a program easily by deriving new

subtypes is important because it greatly improves designs while reducing the cost of software maintenance.

There's a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a function is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to fly, the compiler cannot know at compile-time precisely what piece of code will be executed. That's the whole point – when the message is sent, the programmer doesn't w*ant* to know what piece of code will be executed; the draw function can be applied equally to a circle, square, or triangle, and the object will execute the proper code depending on its specific type. If you don't have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without changes to the function call. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do? For example, in the following diagram the **BirdController** object just works with generic **Bird** objects, and does not know what exact type they are. This is convenient from **BirdController**'s perspective, because it doesn't have to write special code to determine the exact type of **Bird** it's working with, or that **Bird**'s behavior. So how does it happen that, when **fly( )** is called



while ignoring the specific type of **Bird**, the right behavior will occur?

The answer is the primary twist in object-oriented programming: The compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler causes what is called *early binding*, a term you may not have heard before because you've never thought about it any other way. It means the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed. In OOP, the program cannot

determine the address of the code until runtime, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code being called isn't determined until runtime. The compiler does ensure that the function exists and it performs type checking on the arguments and return value (a language where this isn't true is called *weakly typed*), but it doesn't know the exact code to execute.

To perform late binding, the compiler inserts a special bit of code in lieu of the absolute call. This code calculates the address of the function body, using information stored in the object itself (this process is covered in great detail in Chapter XX). Thus, each object can behave differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

You state that you want a function to have the flexibility of late-binding properties using the keyword **virtual**. You don't need to understand the mechanics of **virtual** to use it, but without it you can't do object-oriented programming in C++. In C++, you must remember to add the **virtual** keyword because by default member functions are *not* dynamically bound. Virtual functions allow you to express the differences in behavior of classes in the same family. Those differences are what cause polymorphic behavior.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in the chapter.

To demonstrate polymorphism, we want to write a single piece of code that ignores the specific details of type and talks only to the base class. That code is *decoupled* from type-specific information, and thus is simpler to write and easier to understand. And, if a new type – a **Hexagon**, for example – is added through inheritance, the code you write will work just as well for the new type of **Shape** as it did on the existing types. Thus the program is *extensible*.

If you write a function in C++ (as you will soon learn how to do):

```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

This function speaks to any **Shape**, so it is independent of the specific type of object it's drawing and erasing (the '**&**' means "take the address of

the object that's passed to **doStuff( )**, but it's not important that you understand the details of that right now). If in some other part of the program we use the **doStuff( )** function:

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

The calls to **doStuff( )** automatically work right, regardless of the exact type of the object.

This is actually a pretty amazing trick. Consider the line:

```
doStuff(c);
```

What's happening here is that a **Circle** is being passed into a function that's expecting a **Shape**. Since a **Circle** *is* a **Shape** it can be treated as one by **doStuff( )**. That is, any message that **doStuff( )** can send to a **Shape**, a **Circle** can accept. So it is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance

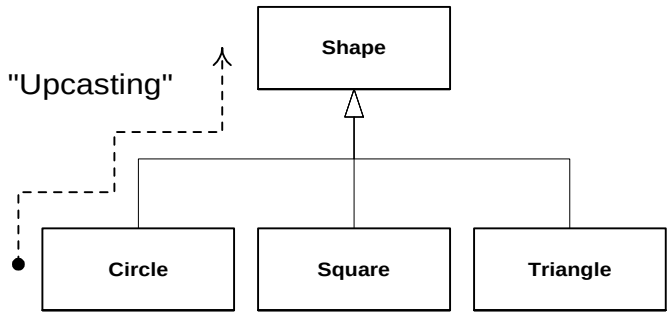


diagram: "upcasting."

An object-oriented program contains some upcasting somewhere, because that's how you decouple yourself from knowing about the exact type you're working with. Look at the code in **doStuff( )**:

```
    s.erase();
    // ...
    s.draw();
```

Notice that it doesn't say "If you're a **Circle**, do this, if you're a **Square**, do that, etc." If you write that kind of code, which checks for all the possible types that a **Shape** can actually be, it's messy and you need to change it every time you add a new kind of **Shape**. Here, you just say "You're a shape, I know you can **erase( )** yourself, do it and take care of the details correctly."

What's amazing about the code in **doStuff( )** is that somehow the right thing happens. Calling **draw( )** for **Circle** causes different code to be executed than when calling **draw( )** for a **Square** or a **Line**, but when the **draw( )** message is sent to an anonymous **Shape**, the correct behavior occurs based on the actual type that the **Shape** is. This is amazing because, as mentioned earlier, when the C++ compiler is compiling the code for **doStuff( )**, it cannot know exactly what types it is dealing with. So ordinarily, you'd expect it to end up calling the version of **erase( )** and **draw( )** for **Shape**, and not for the specific **Circle**, **Square**, or **Line**. And yet the right thing happens, because of polymorphism. The compiler and runtime system handle the details; all you need to know is that it happens and more importantly how to design with it. If a member function is **virtual**, then when you send a message to an object, the object will do the right thing, even when upcasting is involved.

Creating and destroying objects

Technically, the domain of OOP is abstract data typing, inheritance and polymorphism, but other issues can be at least as important. This section gives an overview of these issues.

Especially important is the way objects are created and destroyed. Where is the data for an object and how is the lifetime of that object controlled? Different programming languages use different philosophies here. C++ takes the approach that control of efficiency is the most important issue, so it gives the programmer a choice. For maximum runtime speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack or in static storage. The stack is an area in memory that is used directly by the microprocessor to store data during program execution. Variables on the stack are sometimes called *automatic* or *scoped* variables. The static storage area is simply a

fixed patch of memory that is allocated before the program begins to run. Using the stack or static storage places a priority on the speed of storage allocation and release, which can be very valuable in some situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime and type of objects *while* you're writing the program. If you are trying to solve a more general problem such as computer-aided design, warehouse management or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the *heap*. In this approach you don't know until run time how many objects you need, what their lifetime is or what their exact type is. Those decisions are made at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap when you need it, using the **new** keyword. When you're finished with the storage, you must release it, using the **delete** keyword.

Because the storage is managed dynamically, at run time, the amount of time required to allocate storage on the heap is significantly longer than the time to create storage on the stack. (Creating storage on the stack is often a single microprocessor instruction to move the stack pointer down, and another to move it back up.) The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve general programming problems.

There's another issue, however, and that's the lifetime of an object. If you create an object on the stack or in static storage, the compiler determines how long the object lasts and can automatically destroy it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In C++, the programmer must determine programmatically when to destroy the object, and then perform the destruction using the **delete** keyword. As an alternative, the environment can provide a feature called a *garbage collector* that automatically discovers when an object is no longer in use and destroys it. Of course, a garbage collector is much more convenient, but it requires that all applications must be able to tolerate the existence of the garbage collector and the overhead for garbage collection. This does not meet the design requirements of the C++ language and so it was not included, although third-party garbage collectors exist for C++.

Exception handling: dealing with errors

Ever since the beginning of programming languages, error handling has been one of the most difficult issues. Because it's so hard to design a good error-handling scheme, many languages simply ignore the issue, passing the problem on to library designers who come up with halfway measures that can work in many situations but can easily be circumvented, generally by just ignoring them. A major problem with most error-handling schemes is that they rely on programmer vigilance in following an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant, which often occurs when they are in a hurry, these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate *exception handler* designed to handle that particular type of error. It's as if exception handling is a different, parallel path of execution that can be taken when things go wrong. And because it uses a separate execution path, it doesn't need to interfere with your normally-executing code. This makes that code simpler to write since you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value that's returned from a function or a flag that's set by a function in order to indicate an error condition – these can be ignored. An exception cannot be ignored so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting the program, you are often able to set things right and restore the execution of a program, which produces much more robust systems.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented with an object. Exception handling existed before object-oriented languages.

Analysis and design

The object-oriented paradigm is a new and different way of thinking about programming and many folks have trouble at first knowing how to approach a project. Now that you know that everything is supposed to be

an object, and as you learn to think more in an object-oriented style, you can begin to create "good" designs, ones that will take advantage of all the benefits that OOP has to offer.

A *method* (also often called a *methodology*) is a set of processes and heuristics used to break down the complexity of a programming problem. Many OOP methods have been formulated since the dawn of object-oriented programming, and this section will give you a feel for what you're trying to accomplish when using a method.

Especially in OOP, methodology is a field of many experiments, so it is important to understand what problem the method is trying to solve before you consider adopting one. This is particularly true with C++, where the programming language itself is intended to reduce the complexity involved in expressing a program. This may in fact alleviate the need for ever-more-complex methodologies. Instead, simpler ones may suffice in C++ for a much larger class of problems than you could handle with simple methods for procedural languages.

It's also important to realize that the term "methodology" is often too grand and promises too much. Whatever you do now when you design and write a program is a method. It may be your own method, and you may not be conscious of doing it, but it is a process you go through as you create. If it is an effective process, it may need only a small tune-up to work with C++. If you are not satisfied with your productivity and the way your programs turn out, you may want to consider adopting a formal method, or choosing pieces from among the many formal methods.

While you're going through the development process, the most important issue is this: don't get lost. It's easy to do. Most of the analysis and design methods are intended to solve the largest of problems. Remember that most projects don't fit into that category, so you can usually have successful analysis and design with a relatively small subset of what a method recommends. But some sort of process, no matter how limited, will generally get you on your way in a much better fashion than simply beginning to code.

It's also easy to get stuck, to fall into "analysis paralysis," where you feel like you can't move forward because you haven't nailed down every little detail at the current stage. Remember that, no matter how much analysis you do, there are some things about a system that won't reveal themselves until design time, and more things that won't reveal themselves until you're coding, or not even until a program is up and running. Because of this, it's critical to move fairly quickly through analysis and design to implement a test of the proposed system.

This point is worth emphasizing. Because of the history we've had with procedural languages, it is commendable that a team will want to proceed carefully and understand every minute detail before moving to design and implementation. Certainly, when creating a DBMS, it pays to understand a customer's needs thoroughly. But a DBMS is in a class of problems that is very well-posed and well-understood. The class of programming problem discussed in this chapter is of the "wild-card" variety, where it isn't simply re-forming a well-known solution, but instead involves one or more "wild-card factors" – elements where there is no well-understood previous solution, and where research is necessary.[10] Attempting to thoroughly analyze a wild-card problem before moving into design and implementation results in analysis paralysis because you don't have enough information to solve this kind of problem during the analysis phase. Solving such a problem requires iteration through the whole cycle, and that requires risk-taking behavior (which makes sense, because you're trying to do something new and the potential rewards are higher). It may seem like the risk is compounded by "rushing" into a preliminary implementation, but it can instead reduce the risk in a wild-card project because you're finding out early whether a particular design is viable.

It's often proposed that you "build one to throw away." With OOP, you may still throw *part* of it away, but because code is encapsulated into classes, you will inevitably produce some useful class designs and develop some worthwhile ideas about the system design during the first iteration that do not need to be thrown away. Thus, the first rapid pass at a problem not only produces critical information for the next analysis, design, and implementation iteration, it also creates a code foundation for that iteration.

That said, if you're looking at a methodology that contains tremendous detail and suggests many steps and documents, it's still difficult to know when to stop. Keep in mind what you're trying to discover:

1. What are the objects? (How do you partition your project into its component parts?)
2. What are their interfaces? (What messages do you need to be able to send to each object?)

If you come up with nothing more than the objects and their interfaces then you can write a program. For various reasons you might need more

---

[10] My rule of thumb for estimating such projects: If there's more than one wild card, don't even try to plan how long it's going to take or how much it will cost. There are too many degrees of freedom.

descriptions and documents than this, but you can't really get away with any less.

The process can be undertaken in four phases, and a phase 0 which is just the initial commitment to using some kind of structure.

Phase 0: Make a plan

The first step is to decide what steps you're going to have in your process. It sounds simple (in fact, *all* of this sounds simple) and yet people often don't even get around to phase one before they start coding. If your plan is "let's jump in and start coding," fine. (Sometimes that's appropriate when you have a well-understood problem.) At least agree that this is the plan.

You might also decide at this phase that some additional process structure is necessary but not the whole nine yards. Understandably enough, some programmers like to work in "vacation mode" in which no structure is imposed on the process of developing their work: "It will be done when it's done." This can be appealing for awhile, but I've found that having a few milestones along the way helps to focus and galvanize your efforts around those milestones instead of being stuck with the single goal of "finish the project." In addition, it divides the project into more bite-sized pieces and make it seem less threatening (plus the milestones offer more opportunities for celebrating).

When I began to study story structure (so that I will someday write a novel) I was initially resistant to the idea of structure, feeling that when I wrote I simply let it flow onto the page. What I found was that when I wrote about computers the structure was simple enough so that I didn't need to think much about it, but I was still structuring my work, albeit only semi-consciously in my head. So even if you think that your plan is to just start coding, you still go through the following phases while asking and answering certain questions.

The mission statement

Any system you build, no matter how complicated, has a fundamental purpose, the business that it's in, the basic need that it satisfies. If you can look past the user interface, the hardware- or system-specific details, the coding algorithms and the efficiency problems, you will eventually find the core of its being, simple and straightforward. Like the so-called *high concept* from a Hollywood movie, you can describe it in one or two sentences. This pure description is the starting point.

The high concept is quite important because it sets the tone for your project; it's a mission statement. You won't necessarily get it right the first time (you may be in a later phase of the project before it becomes completely clear), but keep trying until it feels right. For example, in an air-traffic control system you may start out with a high concept focused on the system that you're building: "The tower program keeps track of the aircraft." But consider what happens when you shrink the system to a very small airfield; perhaps there's only a human controller or none at all. A more useful model won't concern the solution you're creating as much as it describes the problem: "Aircraft arrive, unload, service and reload, and depart."

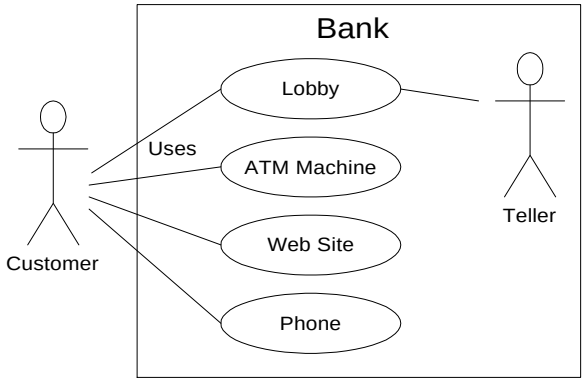
Phase 1: What are we making?

In the previous generation of program design (called *procedural design*), this is called "creating the *requirements analysis* and *system specification*." These, of course, were places to get lost: intimidatingly-named documents that could become big projects in their own right. Their intention was good, however. The requirements analysis says "Make a list of the guidelines we will use to know when the job is done and the customer is satisfied." The system specification says "Here's a description of *what* the program will do (not *how*) to satisfy the requirements." The requirements analysis is really a contract between you and the customer (even if the customer works within your company or is some other object or system). The system specification is a top-level exploration into the problem and in some sense a discovery of whether it can be done and how long it will take. Since both of these will require consensus among people, I think it's best to keep them as bare as possible – ideally, to lists and basic diagrams – to save time. You might have other constraints that require you to expand them into bigger documents, but by keeping the initial document small and concise, it can be created in a few sessions of group brainstorming with a leader who dynamically creates the description. This not only solicits input from everyone, it also fosters initial buy-in and agreement by everyone on the team. Perhaps most importantly, it can kick off a project with a lot of enthusiasm.

It's necessary to stay focused on the heart of what you're trying to accomplish in this phase: determine what the system is supposed to do. The most valuable tool for this is a collection of what are called "use-cases." These are essentially descriptive answers to questions that start with "What does the system do if …" For example, "What does the auto-teller do if a customer has just deposited a check within 24 hours and there's not enough in the account without the check to provide the desired

withdrawal?" The use-case then describes what the auto-teller does in that situation.

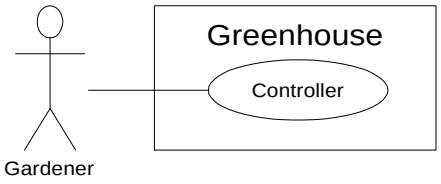
Use-case diagrams are intentionally very simple, to prevent you from



getting bogged down in system implementation details prematurely:

Each stick person represents an "actor," which is typically a human or some other kind of free agent (these can even be other computer systems). The box represents the boundary of your system. The ellipses represent the use cases themselves, which are units of functionality as they are perceived from outside of the system. That is, it doesn't matter how the system is actually implemented, as long as it looks like this to the user.

A use-case does not need to be terribly complex, even if the underlying system is complex. It is only intended to show the system as it appears to the user. For example:



The use cases produce the requirements specifications, by determining all the interactions that the user may have with the system. You try to discover a full set of use-cases for your system, and once you've done that you have the core of what the system is supposed to do. The nice thing about focusing on use-cases is that they always bring you back to the essentials and keep you from drifting off into issues that aren't critical

for getting the job done. That is, if you have a full set of use-cases you can describe your system and move onto the next phase. You probably won't get it all figured out perfectly at this phase, but that's OK. Everything will reveal itself in the fullness of time, and if you demand a perfect system specification at this point you'll get stuck.

If you get stuck, you can kick-start this phase by describing the system in a few paragraphs and then looking for nouns and verbs. The nouns become either actors or parts of use cases (or even entire use cases by themselves), and the verbs become the interactions between the two. You'll be surprised at how useful a tool this can be; sometimes it will accomplish the lion's share of the work for you.

Use-cases will identify key features in the system that will reveal some of the fundamental classes you'll be using. For example, if you're in the fireworks business, you may want to identify Workers, Firecrackers, and Customers; more specifically you'll need Chemists, Assemblers, and Handlers; AmateurFirecrackers and ProfessionalFirecrackers; Buyers and Spectators. Even more specifically, you could identify YoungSpectators, OldSpectators, TeenageSpectators, and ParentSpectators.

Although it's a black art, at this point some kind of scheduling can be quite useful. You now have an overview of what you're building so you'll probably be able to get some idea of how long it will take. A lot of factors come into play here: if you estimate a long schedule then the company might not decide to build it, or a manager might have already decided how long the project should take and will try to influence your estimate. But it's best to have an honest schedule from the beginning and deal with the tough decisions early. There have been a lot of attempts to come up with accurate scheduling techniques (like techniques to predict the stock market), but probably the best approach is to rely on your experience and intuition. Get a gut feeling for how long it will really take, then double that and add 10 percent. Your gut feeling is probably correct; you *can* get something working in that time. The "doubling" will turn that into something decent, and the 10 percent will deal with final polishing and details[11]. However you want to explain it, and regardless of the moans and manipulations that happen when you reveal such a schedule, it just seems to work out that way.

---

[11] My personal take on this has changed lately. Doubling and adding 10 percent will give you a reasonably accurate estimate (assuming there are not too many wild-card factors), but you still have to work quite dilligently to finish in that time. If you actually want time to really make it elegant and to enjoy yourself in the process, the correct multiplier is more like three or four times, I believe.

Phase 2: How will we build it?

In this phase you must come up with a design that describes what the classes look like and how they will interact. An excellent tool in determining classes and interactions is the *Class-Responsibility-Collaboration* (CRC) card. Part of the value of this technique is that it's so low-tech: you start out with a set of blank 3" by 5" cards, and you write on them. Each card represents a single class, and on the card you write:

1. The name of the class. It's important that this name capture the essence of what the class does, so that it makes sense at a glance.
2. The "responsibilities" of the class: what it should do. This can typically be summarized by just stating the names of the member functions (since those names should be descriptive in a good design), but it does not preclude other notes. If you need to seed the process, look at the problem from a lazy programmer's standpoint: What objects would you like to magically appear to solve your problem?
3. The "collaborations" of the class: what other classes does it interact with? "Interact" is an intentionally broad term; it could mean aggregation or simply that some other object exists that will perform services for an object of the class. Collaborations should also consider the audience for this class. For example, if you create a class Firecracker, who is going to observe it, a Chemist or a Spectator? The former will want to know what chemicals go into the construction, and the latter will respond to the colors and shapes released when it explodes.

You may feel like the cards should be bigger because of all the information you'd like to get on them, but they are intentionally small, not only to keep your classes small but also to keep you from getting into too much detail too early. If you can't fit all you need to know about a class on a small card, the class is too complex (either you're getting too detailed, or you should create more than one class). The ideal class should be understood at a glance. The idea of CRC cards is to assist you in coming up with a first cut on the design, so that you can get the big picture and refine the design.

One of the great benefits of CRC cards is in communication. It's best done real-time, in a group, without computers. Each person takes responsibility for several classes (which at first have no names or other information), and you run a live simulation by going through your use-cases and deciding what messages go to which objects to satisfy each use case. As you go through this process, you discover the classes you need along with their responsibilities and collaborations, and you fill out the cards as you

do this. When you've moved through all the use cases, you should have a fairly complete first cut of your design.

Before I began using CRC cards, the most successful consulting experiences I had when coming up with an initial design involved standing in front of a team, who hadn't built an OOP project before, and drawing objects on a whiteboard. We talked about how the objects should communicate with each other, and erased some of them and replaced them with other objects (effectively, I was managing all the "CRC cards" on the whiteboard). The team (who knew what the project was supposed to do) actually created the design; they "owned" the design rather than having it given to them. All I was doing was guiding the process by asking the right questions, trying out the assumptions and taking the feedback from the team to modify those assumptions. The true beauty of the process was that the team learned how to do object-oriented design not by reviewing abstract examples, but by working on the one design that was most interesting to them at that moment: theirs.

Once you've come up with a set of CRC cards, you may want to create a more formal description of your design using UML. There are a fair number of books on UML, and you can get the specification at *http://www.rational.com*. You don't need to use UML, but it can be helpful, especially if you want to put a diagram up on the wall for everyone to ponder, which is a good idea. An alternative to UML is a textual description of the objects and their interfaces, but this can be limiting.

UML also provides a diagramming notation for describing the dynamic model of your system, for situations where the state transitions of a system or subsystem are dominant enough that they need their own diagrams (such as in a control system), and for describing the data structures, for systems or subsystems where data is a dominant factor (such as a database).

You'll know you're done with phase 2 when you have described the objects and their interfaces. Well, most of them – there are usually a few that slip through the cracks and don't make themselves known until phase 3. But that's OK. All you are concerned with is that you eventually discover all of your objects. It's nice to discover them early in the process but OOP provides enough structure so that it's not so bad if you discover them later. In fact, the design of an object tends to happen in five stages, throughout the process of program development.

Five stages of object design

The design life of an object is not limited to the period of time when you're writing the program. Instead, the design of an object appears over a sequence of stages. It's helpful to have this perspective because you stop expecting perfection right away; instead, you realize that the understanding of what an object does and what it should look like happens over time. This view also applies to the design of various types of programs; the pattern for a particular type of program emerges through struggling again and again with that problem (design patterns are covered in Chapter XX). Objects, too, have their patterns that emerge through understanding, use, and reuse.

1. Object discovery. This stage occurs during the initial analysis of a program. Objects may be discovered by looking for external factors and boundaries, duplication of elements in the system, and the smallest conceptual units. Some objects are obvious if you already have a set of class libraries. Commonality between classes suggesting base classes and inheritance may appear right away, or later in the design process.

2. Object assembly. As you're building an object you'll discover the need for new members that didn't appear during discovery. The internal needs of the object may require new classes to support it.

3. System construction. Once again, more requirements for an object may appear at this later stage. As you learn, you evolve your objects. The need for communication and interconnection with other objects in the system may change the needs of your classes or require new classes. For example, here you may discover the need for facilitator or helper classes, such as a linked list, that contain little or no state information and simply help other classes to function.

4. System extension. As you add new features to a system you may discover that your previous design doesn't support easy system extension. With this new information, you can restructure parts of the system, very possibly adding new classes or class hierarchies.

5. Object reuse. This is the real stress test for a class. If someone tries to reuse it in an entirely new situation, they'll probably discover some shortcomings. As you change a class to adapt to more new programs, the general principles of the class will become clearer, until you have a truly reusable type.

Guidelines for object development

These stages suggest some guidelines when thinking about developing your classes:

1. Let a specific problem generate a class, then let the class grow and mature during the solution of other problems.
2. Remember, discovering the classes you need (and their interfaces) is the majority of the system design. If you already had those classes, this would be an easy project.
3. Don't force yourself to know everything at the beginning; learn as you go. That's the way it will happen anyway.
4. Start programming; get something working so you can prove or disprove your design. Don't fear procedural-style spaghetti code – classes partition the problem and help control anarchy and entropy. Bad classes do not break good classes.
5. Always keep it simple. Little clean objects with obvious utility are better than big complicated interfaces. When decision points come up, use a modified Occam's Razor approach: Consider the choices and select the one that is simplest, because simple classes are almost always best. You can always start small and simple and expand the class interface when you understand it better, but as time goes on, it's difficult to remove elements from a class.

Phase 3: Build it

This is the initial conversion from the rough design to a compiling body of code that can be tested, and especially that will prove or disprove your design. This is not a one-pass process, but rather the beginning of a series of writes and rewrites, as you'll see in phase 4.

If you're reading this book you're probably a programmer, so now we're at the part you've been trying to get to. By following a plan – no matter how simple and brief – and coming up with design structure before coding, you'll discover that things fall together far more easily than if you dive in and start hacking, and you'll also realize a great deal of satisfaction. Getting code to run and do what you want is fulfilling, and can easily become an obsession. But it's my experience that coming up with an elegant solution is deeply satisfying at an entirely different level; it feels closer to art than technology. And elegance always pays off; it's not a frivolous pursuit. Not only does it give you a program that's easier to build and debug, but it's also easier to understand and maintain, and that's where the financial value lies.

After you build the system and get it running, it's important to do a reality check, and here's where the requirements analysis and system specification comes in. Go through your program and make sure that all the requirements are checked off, and that all the use-cases work the way

they're described (an even better approach is to use the requirements analysis and use-cases to generate test code). Now you're done. Or are you?

Phase 4: Iteration

This is the point in the development cycle that has traditionally been called "maintenance," a catch-all term that can mean everything from "getting it to work the way it was really supposed to in the first place" to "adding features that the customer forgot to mention" to the more traditional "fixing the bugs that show up" and "adding new features as the need arises." So many misconceptions have been applied to the term "maintenance" that it has taken on a slightly deceiving quality, partly because it suggests that you've actually built a pristine program and all you need to do is change parts, oil it and keep it from rusting. Perhaps there's a better term to describe what's going on.

The term is *iteration*. That is, "You won't get it right the first time, so give yourself the latitude to learn and to go back and make changes." You might need to make a lot of changes as you learn and understand the problem more deeply. The elegance you'll produce if you iterate until you get it right will pay off, both in the short and the long term. Iteration is where your program goes from good to great, and where those issues that you didn't really understand in the first pass become clear. It's also where your classes can evolve from single-project usage to reusable resources.

What it means to "get it right" isn't just that the program works according to the requirements and the use-cases. It also means that the internal structure of the code makes sense to you, and feels like it fits together well, with no awkward syntax, oversized objects or ungainly exposed bits of code. In addition, you must have some sense that the program structure will survive the changes that it will inevitably go through during its lifetime, and that those changes can be made easily and cleanly. This is no small feat. You must not only understand what you're building, but also how the program will evolve (what I call the *vector of change*). Fortunately, object-oriented programming languages are particularly adept at supporting this kind of continuing modification – the boundaries created by the objects are what tend to keep the structure from breaking down. They are also what allow you to make changes – ones that would seem drastic in a procedural program – without causing earthquakes throughout your code. In fact, support for iteration might be the most important benefit of OOP.

With iteration, you create something that at least approximates what you think you're building, and then you kick the tires, compare it to your

requirements and see where it falls short. Then you can go back and fix it by redesigning and re-implementing the portions of the program that didn't work right.[12] You might actually need to solve the problem, or an aspect of the problem, several times before you hit on the right solution. (A study of *Design Patterns*, described in Chapter XX, is usually helpful here.)

Iteration also occurs when you build a system, see that it matches your requirements and then discover it wasn't actually what you *wanted*. When you see the system in operation, you find that you really wanted to solve a different problem. If you think this kind of iteration is going to happen, then you owe it to yourself to build your first version as quickly as possible so you can find out if it's what you want.

Iteration is closely tied to *incremental development*. Incremental development means that you start with the core of your system and implement it as a framework upon which to build the rest of the system piece by piece. Then you start adding features one at a time. The trick to this is in designing a framework that will accommodate all the features you plan to add to it. (See Chapter XX for more insight into this issue.) The advantage is that once you get the core framework working, each feature you add is like a small project in itself rather than part of a big project. Also, new features that are incorporated later in the development or maintenance phases can be added more easily. OOP supports incremental development because if your program is designed well, your increments will turn out to be discrete objects or groups of objects.

Perhaps the most important thing to remember is that by default – by definition, really – if you modify a class its super- and subclasses will still function. You need not fear modification; it won't necessarily break the program, and any change in the outcome will be limited to subclasses and/or specific collaborators of the class you change.

You have to know when to stop iterating the design. Ideally, you achieve target functionality and are in the process of refinement and addition of new features when the deadline comes along and forces you to stop and ship that version. (Remember, software is a subscription business.)

---

[12] This is something like "rapid prototyping," where you were supposed to build a quick-and-dirty version so that you could learn about the system, and then throw away your prototype and build it right. The trouble with rapid prototyping is that people didn't throw away the prototype, but instead built upon it. Combined with the lack of structure in procedural programming, this often leads to messy systems that are expensive to maintain.

Plans pay off

Of course you wouldn't build a house without a lot of carefully-drawn plans. If you build a deck or a dog house, your plans won't be so elaborate but you'll still probably start with some kind of sketches to guide you on your way. Software development has gone to extremes. For a long time, people didn't have much structure in their development, but then big projects began failing. In reaction, we ended up with methodologies that had an intimidating amount of structure and detail, primarily intended for those big projects. These methodologies were too scary to use – it looked like you'd spend all your time writing documents and no time programming. (This was often the case.) I hope that what I've shown you here suggests a middle path – a sliding scale. Use an approach that fits your needs (and your personality). No matter how minimal you choose to make it, *some* kind of plan will make a big improvement in your project as opposed to no plan at all. Remember that, by most estimates, over 50 percent of projects fail (some estimates go up to 70 percent!).

Why C++ succeeds

Part of the reason C++ has been so successful is that the goal was not just to turn C into an OOP language (although it started that way), but also to solve many other problems facing developers today, especially those who have large investments in C. Traditionally, OOP languages have suffered from the attitude that you should abandon everything you know and start from scratch with a new set of concepts and a new syntax, arguing that it's better in the long run to lose all the old baggage that comes with procedural languages. This may be true, in the long run. But in the short run, a lot of that baggage was valuable. The most valuable elements may not be the existing code base (which, given adequate tools, could be translated), but instead the existing *mind base*. If you're a functioning C programmer and must drop everything you know about C in order to adopt a new language, you immediately become nonproductive for many months, until your mind fits around the new paradigm. Whereas if you can leverage off of your existing C knowledge and expand upon it, you can continue to be productive with what you already know while moving into the world of object-oriented programming. As everyone has his or her own mental model of programming, this move is messy enough as it is without the added expense of starting with a new language model from square one. So the reason for the success of C++, in a nutshell, is economic: It still costs to move to OOP, but C++ may cost less.

The goal of C++ is improved productivity. This productivity comes in many ways, but the language is designed to aid you as much as possible, while hindering you as little as possible with arbitrary rules or any requirement that you use a particular set of features. C++ is designed to be practical; language design decisions were based on providing the maximum benefits to the programmer (at least, from the world view of C).

A better C

You get an instant win even if you continue to write C code because C++ has closed many holes in the C language and provides better type checking and compile-time analysis. You're forced to declare functions so the compiler can check their use. The need for the preprocessor has virtually been eliminated for value substitution and macros, which removes a set of difficult-to-find bugs. C++ has a feature called *references* that allows more convenient handling of addresses for function arguments and return values. The handling of names is improved through a feature called *function overloading*, which allows you to use the same name for different functions. A feature called *namespaces* also improves the control of names. There are numerous other small features that improve the safety of C.

You're already on the learning curve

The problem with learning a new language is productivity: No company can afford to suddenly lose a productive software engineer because he or she is learning a new language. C++ is an extension to C, not a complete new syntax and programming model. It allows you to continue creating useful code, applying the features gradually as you learn and understand them. This may be one of the most important reasons for the success of C++.

In addition, all your existing C code is still viable in C++, but because the C++ compiler is pickier, you'll often find hidden errors when recompiling the code.

Efficiency

Sometimes it is appropriate to trade execution speed for programmer productivity. A financial model, for example, may be useful for only a short period of time, so it's more important to create the model rapidly

than to execute it rapidly. However, most applications require some degree of efficiency, so C++ always errs on the side of greater efficiency. Because C programmers tend to be very efficiency-conscious, this is also a way to ensure they won't be able to argue that the language is too fat and slow. A number of features in C++ are intended to allow you to tune for performance when the generated code isn't efficient enough.

Not only do you have the same low-level control as in C (and the ability to directly write assembly language within a C++ program), but anecdotal evidence suggests that the program speed for an object-oriented C++ program tends to be within ±10% of a program written in C, and often much closer. The design produced for an OOP program may actually be more efficient than the C counterpart.

Systems are easier to express and understand

Classes designed to fit the problem tend to express it better. This means that when you write the code, you're describing your solution in the terms of the problem space ("put the grommet in the bin") rather than the terms of the computer, which is the solution space ("set the bit in the chip that means that the relay will close"). You deal with higher-level concepts and can do much more with a single line of code.

The other benefit of this ease of expression is maintenance, which (if reports can be believed) takes a huge portion of the cost over a program's lifetime. If a program is easier to understand, then it's easier to maintain. This can also reduce the cost of creating and maintaining the documentation.

Maximal leverage with libraries

The fastest way to create a program is to use code that's already written: a library. A major goal in C++ is to make library use easier. This is accomplished by casting libraries into new data types (classes), so that bringing in a library means adding a new types to the language. Because the C++ compiler takes care of how the library is used – guaranteeing proper initialization and cleanup, and ensuring functions are called properly – you can focus on what you want the library to do, not how you have to do it.

Because names can be sequestered to portions of your program via C++ namespaces, you can use as many libraries as you want without the kinds of name clashes you'd run into with C.

Source-code reuse with templates

There is a significant class of types that require source-code modification in order to reuse them effectively. The *template* feature in C++ performs the source code modification automatically, making it an especially powerful tool for reusing library code. A type you design using templates will work effortlessly with many other types. Templates are especially nice because they hide the complexity of this type of code reuse from the client programmer.

Error handling

Error handling in C is a notorious problem, and one that is often ignored – finger-crossing is usually involved. If you're building a large, complex program, there's nothing worse than having an error buried somewhere with no clue of where it came from. C++ *exception handling* (the subject of Chapter XX) is a way to guarantee that an error is noticed and that something happens as a result.

Programming in the large

Many traditional languages have built-in limitations to program size and complexity. BASIC, for example, can be great for pulling together quick solutions for certain classes of problems, but if the program gets more than a few pages long or ventures out of the normal problem domain of that language, it's like trying to run through an ever-more viscous fluid. C, too, has these limitations. For example, when a program gets beyond perhaps 50,000 lines of code, name collisions start to become a problem – effectively, you run out of function and variable names. Another particularly bad problem is the little holes in the C language – errors can get buried in a large program that are extremely difficult to find.

There's no clear line that tells when your language is failing you, and even if there were, you'd ignore it. You don't say, "My BASIC program just got too big; I'll have to rewrite it in C!" Instead, you try to shoehorn a few more lines in to add that one extra feature. So the extra costs come creeping up on you.

C++ is designed to aid *programming in the large*, that is, to erase those creeping-complexity boundaries between a small program and a large one. You certainly don't need to use OOP, templates, namespaces, and exception handling when you're writing a hello-world style utility program,

but those features are there when you need them. And the compiler is aggressive about ferreting out bug-producing errors for small and large programs alike.

Strategies for transition

If you buy into OOP, your next question is probably, “How can I get my manager/colleagues/department/peers to start using objects?” Think about how you – one independent programmer – would go about learning to use a new language and a new programming paradigm. You’ve done it before. First comes education and examples; then comes a trial project to give you a feel for the basics without doing anything too confusing; then try a “real world” project that actually does something useful. Throughout your first projects you continue your education by reading, asking questions of experts, and trading hints with friends. This is the approach many experienced programmers suggest for the switch from C to C++. Switching an entire company will of course introduce certain group dynamics, but it will help at each step to remember how one person would do it.

Guidelines

Here are some guidelines to consider when making the transition to OOP and C++:

1. Training

The first step is some form of education. Remember the company’s investment in plain C code, and try not to throw everything into disarray for 6 to 9 months while everyone puzzles over how multiple inheritance works. Pick a small group for indoctrination, preferably one composed of people who are curious, work well together, and can function as their own support network while they’re learning C++.

An alternative approach that is sometimes suggested is the education of all company levels at once, including overview courses for strategic managers as well as design and programming courses for project builders. This is especially good for smaller companies making fundamental shifts in the way they do things, or at the division level of larger companies. Because the cost is higher, however, some may choose to start with project-level training, do a pilot project (possibly with an outside mentor), and let the project team become the teachers for the rest of the company.

2. Low-risk project

Try a low-risk project first and allow for mistakes. Once you've gained some experience, you can either seed other projects from members of this first team or use the team members as an OOP technical support staff. This first project may not work right the first time, so it should not be mission-critical for the company. It should be simple, self-contained, and instructive; this means that it should involve creating classes that will be meaningful to the other programmers in the company when they get their turn to learn C++.

3. Model from success

Seek out examples of good object-oriented design before starting from scratch. There's a good probability that someone has solved your problem already, and if they haven't solved it exactly you can probably apply what you've learned about abstraction to modify an existing design to fit your needs. This is the general concept of *design patterns,* covered in Chapter XX.

4. Use existing class libraries

The primary economic motivation for switching to C++ is the easy use of existing code in the form of class libraries (in particular, the Standard C++ libraries, which are covered later in this book). The shortest application development cycle will result when you don't have to write anything but **main( )**. However, some new programmers don't understand this, are unaware of existing class libraries, or through fascination with the language desire to write classes that may already exist. Your success with OOP and C++ will be optimized if you make an effort to seek out and reuse other people's code early in the transition process.

5. Don't rewrite existing code in C++

Although *compiling* your C code in C++ usually produces (sometimes great) benefits by finding problems in the old code, it is not usually the best use of your time to take existing, functional code and rewrite it in C++ (if you must turn it into objects, you can "wrap" the C code in C++ classes). There are incremental benefits, especially if the code is slated for reuse. But chances are you aren't going to see the dramatic increases in productivity that you hope for in your first few projects unless that project is a new one. C++ and OOP shine best when taking a project from concept to reality.

Management obstacles

If you're a manager, your job is to acquire resources for your team, to overcome barriers to your team's success, and in general to try to provide the most productive and enjoyable environment so your team is most likely to perform those miracles that are always being asked of you. Moving to C++ falls in all three of these categories, and it would be wonderful if it didn't cost you anything as well. Although moving to C++ may be cheaper – depending on your constraints[13] – than the OOP alternatives for team of C programmers (and probably for programmers in other procedural languages), it isn't free, and there are obstacles you should be aware of before trying to sell the move to C++ within your company and embarking on the move itself.

Startup costs

The cost of moving to C++ is more than just the acquisition of C++ compilers (the GNU C++ compiler, one of the very best, is free). Your medium- and long-term costs will be minimized if you invest in training (and possibly mentoring for your first project) and also if you identify and purchase class libraries that solve your problem rather than trying to build those libraries yourself. These are hard-money costs that must be factored into a realistic proposal. In addition, there are the hidden costs in loss of productivity while learning a new language and possibly a new programming environment. Training and mentoring can certainly minimize these but team members must overcome their own struggles to understand the issues. During this process they will make more mistakes (this is a feature, because acknowledged mistakes are the fastest path to learning) and be less productive. Even then, with some types of programming problems, the right classes, and the right development environment, it's possible to be more productive while you're learning C++ (even considering that you're making more mistakes and writing fewer lines of code per day) than if you'd stayed with C.

Performance issues

A common question is, "Doesn't OOP automatically make my programs a lot bigger and slower?" The answer is, "It depends." Most traditional OOP languages were designed with experimentation and rapid prototyping in mind rather than lean-and-mean operation. Thus, they virtually

[13] Because of its productivity improvements, the Java language should also be considered here.

guaranteed a significant increase in size and decrease in speed. C++, however, is designed with production programming in mind. When your focus is on rapid prototyping, you can throw together components as fast as possible while ignoring efficiency issues. If you're using any third-party libraries, these are usually already optimized by their vendors; in any case it's not an issue while you're in rapid-development mode. When you have a system you like, if it's small and fast enough, then you're done. If not, you begin tuning with a profiling tool, looking first for speedups that can be done with simple applications of built-in C++ features. If that doesn't help, you look for modifications that can be made in the underlying implementation so no code that uses a particular class needs to be changed. Only if nothing else solves the problem do you need to change the design. The fact that performance is so critical in that portion of the design is an indicator that it must be part of the primary design criteria. You have the benefit of finding this out early through rapid prototyping.

As mentioned earlier, the number that is most often given for the difference in size and speed between C and C++ is ±10%, and often much closer to par. You may actually get a significant improvement in size and speed when using C++ rather than C because the design you make for C++ could be quite different from the one you'd make for C.

The evidence for size and speed comparisons between C and C++ tends to be anecdotal and is likely to remain so. Regardless of the number of people who suggest that a company try the same project using C and C++, no company is likely to waste money that way unless it's very big and interested in such research projects. Even then, it seems like the money could be better spent. Almost universally, programmers who have moved from C (or some other procedural language) to C++ have had the personal experience of a great acceleration in their programming productivity, and that's the most compelling argument you can find.

Common design errors

When starting your team into OOP and C++, programmers will typically go through a series of common design errors. This often happens because of too little feedback from experts during the design and implementation of early projects, because no experts have been developed within the company. It's easy to feel that you understand OOP too early in the cycle and go off on a bad tangent; something that's obvious to someone experienced with the language may be a subject of great internal debate for a novice. Much of this trauma can be skipped by using an outside expert for training and mentoring.

On the other hand, the fact that it is easy to make these design errors points to C++'s main drawback: its backwards-compatibility with C (of

course, that's also its main strength). To accomplish the feat of being able to compile C code, the language had to make some compromises which have resulted in a number of "dark corners." These are a reality, and comprise much of the learning curve for the language. In this book (and in others; see the "Recommended Reading" appendix) I try to reveal most of the pitfalls you are likely to encounter when working with C++, but you should always be aware that there are some holes in the safety net.

Summary

This chapter attempts to give you a feel for the broad issues of object-oriented programming and C++, including why OOP is different, and why C++ in particular is different, concepts of OOP methodologies, and finally the kinds of issues you will encounter when moving your own company to OOP and C++.

OOP and C++ may not be for everyone. It's important to evaluate your own needs and decide whether C++ will optimally satisfy those needs, or if you might be better off with another programming system. If you know that your needs will be very specialized for the foreseeable future and if you have specific constraints that may not be satisfied by C++, then you owe it to yourself to investigate the alternatives. Even if you eventually choose C++ as your language, you'll at least understand what the options were and have a clear vision of why you took that direction.

You know what a procedural program looks like: data definitions and function calls. To find the meaning of such a program you have to work a little, looking through the function calls and low-level concepts to create a model in your mind. This is the reason we need intermediate representations when designing procedural programs – by themselves, these programs tend to be confusing because the terms of expression are oriented more toward the computer than the problem you're solving.

Because C++ adds many new concepts to the C language, your natural assumption may be that, of course, the **main( )** in a C++ program will be far more complicated than the equivalent C program. Here, you'll be pleasantly surprised: A well-written C++ program is generally far simpler and much easier to understand than the equivalent C program. What you'll see are the definitions of the objects that represent concepts in your problem space (rather than the issues of the computer representation) and messages sent to those objects to represent the activities in that space. One of the delights of object-oriented programming is that, with a well-designed program, it's very easy to understand the code by reading

it. Usually there's a lot less code, as well, because many of your problems will be solved by reusing existing library code.

2: Making & using objects

This chapter will introduce enough C++ syntax and program construction concepts to allow you to write and run some simple object-oriented programs. In the subsequent chapter we will cover the basic syntax of C and C++ in detail.

By seeing this chapter first, you'll get the basic flavor of what it is like to program with objects in C++, and you'll also discover some of the reasons for the enthusiasm surrounding this language. This should be enough to carry you through Chapter 3, which can be a bit exhausting since it contains most of the details of the C language.

The user-defined data type, or *class*, is what distinguishes C++ from traditional procedural languages. A class is a new data type that you or someone else creates to solve a particular kind of problem. Once a class is created, anyone can use it without knowing the specifics of how it works, or even how classes are built. This chapter treats classes as if they are just another built-in data type available for use in programs.

Classes that someone else has created are typically packaged into a library. This chapter uses several of the class libraries that come with all C++ implementations. An especially important standard library is iostreams, which (among other things) allows you to read from files and the keyboard, and to write to files and the display. You'll also see the very handy **string** class, and the **vector** container from the Standard Template Library (STL). By the end of the chapter, you'll see how easy it is to utilize a pre-defined library of classes.

In order to create your first program you must understand the tools used to build applications.

The process of language translation

All computer languages are translated from something that tends to be easy for a human to understand (*source code*) into something that is executed on a computer (*machine instructions*). Traditionally, translators fall into two classes: *interpreters* and *compilers*.

Interpreters

An interpreter translates source code into activities (which may comprise groups of machine instructions) and immediately executes those activities. BASIC, for example, has been a popular interpreted language. Traditional BASIC interpreters translate and execute one line at a time, and then forget that the line has been translated. This makes them slow, since they must re-translate any repeated code. BASIC has also been compiled, for speed. More modern interpreters, such as those for the Perl language, translate the entire program into an intermediate language which is then executed by a much faster interpreter[14].

Interpreters have many advantages. The transition from writing code to executing code is almost immediate, and the source code is always available so the interpreter can be much more specific when an error occurs. The benefits often cited for interpreters are ease of interaction and rapid development (but not necessarily execution) of programs.

Interpreted languages often have severe limitations when building large projects (Perl seems to be an exception to this). The interpreter (or a reduced version) must always be in memory to execute the code, and even the fastest interpreter may introduce unacceptable speed restrictions. Most interpreters require that the complete source code be brought into the interpreter all at once. Not only does this introduce a space limitation, it can also cause more difficult bugs if the language doesn't provide facilities to localize the effect of different pieces of code.

---

[14] The boundary between compilers and interpreters can tend to become a bit fuzzy, especially with Perl, which has many of the features and power of a compiled language but the quick turnaround of an interpreted language.

Compilers

A compiler translates source code directly into assembly language or machine instructions. The eventual end product is a file or files containing machine code. This is an involved process, and usually takes several steps. The transition from writing code to executing code is significantly longer with a compiler.

Depending on the acumen of the compiler writer, programs generated by a compiler tend to require much less space to run, and they run much more quickly. Although size and speed are probably the most often cited reasons for using a compiler, in many situations they aren't the most important reasons. Some languages (such as C) are designed to allow pieces of a program to be compiled independently. These pieces are eventually combined into a final *executable* program by a tool called the *linker*. This process is called *separate compilation*.

Separate compilation has many benefits. A program that, taken all at once, would exceed the limits of the compiler or the compiling environment can be compiled in pieces. Programs can be built and tested a piece at a time. Once a piece is working, it can be saved and treated as a building block. Collections of tested and working pieces can be combined into *libraries* for use by other programmers. As each piece is created, the complexity of the other pieces is hidden. All these features support the creation of large programs[15].

Compiler debugging features have improved significantly over time. Early compilers only generated machine code, and the programmer inserted print statements to see what was going on. This is not always effective. Modern compilers can insert information about the source code into the executable program. This information is used by powerful *source-level debuggers* to show exactly what is happening in a program by tracing its progress through the source code.

Some compilers tackle the compilation-speed problem by performing *in-memory compilation*. Most compilers work with files, reading and writing them in each step of the compilation process. In-memory compilers keep the program in RAM. For small programs, this can seem as responsive as an interpreter.

---

[15] Perl is again an exception, since it also provides separate compilation.

The compilation process

To program in C and C++, you need to understand the steps and tools in the compilation process. Some languages (C and C++, in particular) start compilation by running a *preprocessor* on the source code. The preprocessor is a simple program that replaces patterns in the source code with other patterns the programmer has defined (using *preprocessor directives*). Preprocessor directives are used to save typing and to increase the readability of the code (Later in the book, you'll learn how the design of C++ is meant to discourage much of the use of the preprocessor, since it can cause subtle bugs). The pre-processed code is often written to an intermediate file.

Compilers usually do their work in two passes. The first pass *parses* the pre-processed code. The compiler breaks the source code into small units and organizes it into a structure called a *tree*. In the expression "**A + B**" the elements '**A**', '**+**' and '**B**' are leaves on the parse tree.

A *global optimizer* is sometimes used between the first and second passes to produce smaller, faster code.

In the second pass, the *code generator* walks through the parse tree and generates either assembly language code or machine code for the nodes of the tree. If the code generator creates assembly code, the assembler must then be run. The end result in both cases is an object module (a file that typically has an extension of **.o** or **.obj**). A *peephole optimizer* is sometimes used in the second pass to look for pieces of code containing redundant assembly-language statements.

The use of the word "object" to describe chunks of machine code is an unfortunate artifact. The word came into use before object-oriented programming was in general use. "Object" is used in the same sense as "goal" when discussing compilation, while in object-oriented programming it means "**a thing with boundaries.**"

The *linker* combines a list of object modules into an executable program that can be loaded and run by the operating system. When a function in one object module makes a reference to a function or variable in another object module, the linker resolves these references – it makes sure that all the external functions and data you claimed existed during compilation actually do exist. The linker also adds a special object module to perform start-up activities.

The linker can search through special files called *libraries* in order to resolve all its references. A library contains a collection of object modules

in a single file. A library is created and maintained by a program called a *librarian*.

Static type checking

The compiler performs *type checking* during the first pass. Type checking tests for the proper use of arguments in functions, and prevents many kinds of programming errors. Since type checking occurs during compilation rather than when the program is running, it is called *static type checking*.

Some object-oriented languages (notably Java) perform some type checking at runtime (*dynamic type checking*). If combined with static type checking, dynamic type checking is more powerful than static type checking alone. However, it also adds overhead to program execution.

C++ uses static type checking because the language cannot assume any particular runtime support for bad operations. Static type checking notifies the programmer about misuses of types during compilation, and thus maximizes execution speed. As you learn C++ you will see that most of the language design decisions favor the same kind of high-speed, production-oriented programming the C language is famous for.

You can disable static type checking in C++. You can also do your own dynamic type checking – you just need to write the code.

Tools for separate compilation

Separate compilation is particularly important when building large projects. In C and C++, a program can be created in small, manageable, independently tested pieces. The most fundamental tool for breaking a program up into pieces is the ability to create named subroutines or subprograms. In C and C++, a subprogram is called a *function*, and functions are the pieces of code that can be placed in different files, enabling separate compilation. Put another way, the function is the atomic unit of code, since you cannot have part of a function in one file and another part in a different file – the entire function must be placed in a single file (although files can and do contain more than one function).

When you call a function, you typically pass it some *arguments*, which are values you'd like the function to work with during its execution. When the

function is finished, you typically get back a *return value*, a value that the function hands back to you as a result. It's also possible to write functions that take no arguments and return no values.

To create a program with multiple files, functions in one file must access functions and data in other files. When compiling a file, the C or C++ compiler must know about the functions and data in the other files, in particular their names and proper usage. The compiler ensures that functions and data are used correctly. This process of "telling the compiler" the names of external functions and data and what they should look like is called *declaration*. Once you declare a function or variable, the compiler knows how to check to make sure it is used properly.

Declarations vs. definitions

It's important to understand the difference between *declarations* and *definitions* because these terms will be used precisely throughout the book. Essentially all C and C++ programs require declarations. Before you can write your first program, you need to understand the proper way to write a declaration.

A *declaration* introduces a name – an identifier – to the compiler. It tells the compiler "this function or this variable exists somewhere, and here is what it should look like." A *definition*, on the other hand, says: "make this variable here" or "make this function here." It allocates storage for the name. This meaning works whether you're talking about a variable or a function; in either case, at the point of definition the compiler allocates storage. For a variable, the compiler determines how big that variable is and causes space to be generated in memory to hold the data for that variable. For a function, the compiler generates code, which ends up occupying storage in memory.

You can declare a variable or a function in many different places, but there must only be one definition in C and C++ (this is sometimes called the ODR: *one-definition rule*). When the linker is uniting all the object modules, it will usually complain if it finds more than one definition for the same function or variable.

A definition can also be a declaration. If the compiler hasn't seen the name **x** before and you define **int x;**, the compiler sees the name as a declaration and allocates storage for it all at once.

Function declaration syntax

A function declaration in C and C++ gives the function name, the argument types passed to the function, and the return value of the function. For example, here is a declaration for a function called **func1( )** that takes two integer arguments (integers are denoted in C/C++ with the keyword **int**) and returns an integer:

```
int func1(int,int);
```

The first keyword you see is the return value, all by itself: **int**. The arguments are enclosed in parentheses after the function name, in the order they are used. The semicolon indicates the end of a statement; in this case, it tells the compiler "that's all – there is no function definition here!"

C and C++ declarations attempt to mimic the form of the item's use. For example, if **a** is another integer the above function might be used this way:

```
a = func1(2,3);
```

Since **func1**( ) returns an integer, the C or C++ compiler will check the use of **func1( )** to make sure that **a** can accept the return value and that the arguments are appropriate.

Arguments in function declarations may have names. The compiler ignores the names but they can be helpful as mnemonic devices for the user. For example, we can declare **func1( )** in a different fashion that has the same meaning:

```
int func1(int length, int width);
```

A gotcha

There is a significant difference between C and C++ for functions with empty argument lists. In C, the declaration:

```
int func2();
```

means "a function with any number and type of argument." This prevents type-checking, so in C++ it means "a function with no arguments."

Function definitions

Function definitions look like function declarations except they have bodies. A body is a collection of statements enclosed in braces. Braces

denote the beginning and ending of a block of code. To give **func1( )** a definition which is an empty body (a body containing no code), write this:

```
int func1(int length, int width) { }
```

Notice that in the function definition, the braces replace the semicolon. Since braces surround a statement or group of statements, you don't need a semicolon. Notice also that the arguments in the function definition must have names if you want to use the arguments in the function body (since they are never used here, they are optional).

Variable declaration syntax

The meaning attributed to the phrase "variable declaration" has historically been confusing and contradictory, and it's important that you understand the correct definition so you can read code properly. A variable declaration tells the compiler what a variable looks like. It says "I know you haven't seen this name before, but I promise it exists someplace, and it's a variable of X type."

In a function declaration, you give a type (the return value), the function name, the argument list, and a semicolon. That's enough for the compiler to figure out that it's a declaration, and what the function should look like. By inference, a variable declaration might be a type followed by a name. For example:

```
int a;
```

could declare the variable **a** as an integer, using the above logic. Here's the conflict: there is enough information in the above code for the compiler to create space for an integer called **a**, and that's what happens. To resolve this dilemma, a keyword was necessary for C and C++ to say "this is only a declaration; it's defined elsewhere." The keyword is **extern**. It can mean the definition is **extern**al to the file, or that the definition occurs later in the file.

Declaring a variable without defining it means using the **extern** keyword before a description of the variable, like this:

```
extern int a;
```

extern can also apply to function declarations. For **func1( )**, it looks like this:

```
extern int func1(int length, int width);
```

This statement is equivalent to the previous **func1( )** declarations. Since there is no function body, the compiler must treat it as a function

declaration rather than a function definition. The **extern** keyword is thus superfluous and optional for function declarations. It is probably unfortunate that the designers of C did not require the use of **extern** for function declarations; it would have been more consistent and less confusing (but would have required more typing, which probably explains the decision).

Here are some more examples of declarations:

```
//: C03:Declare.cpp
// Declaration & definition examples
extern int i; // Declaration without definition
extern float f(float); // Function declaration

float b;  // Declaration & definition
float f(float a) {  // Definition
  return a + 1.0;
}

int i; // Definition
int h(int x) { // Declaration & definition
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

In the function declarations, the argument identifiers are optional. In the definitions, they are required. This is true only in C, not C++.

Including headers

Most libraries contain significant numbers of functions and variables. To save work and ensure consistency when making the external declarations for these items, C and C++ use a device called the *header file*. A header file is a file containing the external declarations for a library; it conventionally has a file name extension of 'h', such as **headerfile.h**. (You may also see some older code using different extensions like **.hxx** or **.hpp**, but this is becoming very rare.)

The programmer who creates the library provides the header file. To declare the functions and external variables in the library, the user simply includes the header file. To include a header file, use the **#include** preprocessor directive. This tells the preprocessor to open the named header file and insert its contents where the **include** statement appears. Files may be named in an **include** statement in two ways: in angle brackets (**< >**) or in double quotes.

File names in angle brackets, such as:

```
#include <header>
```

causes the preprocessor to search for the file in a way that is particular to your implementation, but typically there's some kind of "include search path" that you specify in your environment or on the compiler command line. The mechanism for setting the search path varies between machines, operating systems and C++ implementations, and may require some investigation on your part.

File names in double quotes, such as:

```
#include "local.h"
```

tell the preprocessor to search for the file in (according to the specification) an "implementation-defined way." What this typically means is to search the current directory for the file. If the file is not found, then the include directive is reprocessed as if it had angle brackets instead of quotes.

To include the iostream header file, you say:

```
#include <iostream>
```

The preprocessor will find the iostream header file (often in a subdirectory called "include") and insert it.

Standard C++ include format

As C++ evolved, different compiler vendors chose different extensions for file names. In addition, various operating systems have different restrictions on file names, in particular on name length. These issues caused source-code portability problems. To smooth over these rough edges, the standard uses a format that allows file names longer than the notorious eight characters and eliminates the extension. For example, instead of the old style of including **iostream.h**, which looks like this:

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

you can now say:

```
#include <iostream>
```

The translator can implement the include statements in a way to suit the needs of that particular compiler and operating system, if necessary truncating the name and adding an extension. Of course, you can also copy the headers given you by your compiler vendor to ones without extensions if you want to use this style before a vendor has provided support for it.

The libraries that have been inherited from C are still available with the traditional '**.h**' extension. However, you can also use them with the more modern C++ include style by prepending a "**c**" before the name. Thus:

```
#include <stdio.h>
#include <stdlib.h>
```

Become:

```
#include <cstdio>
#include <cstdlib>
```

And so on, for all the Standard C headers. This provides a nice distinction to the reader indicating when you're using C versus C++ libraries.

Linking

The linker collects object modules (which often use file name extensions like **.o** or **.obj**), generated by the compiler, into an executable program the operating system can load and run. It is the last phase of the compilation process.

Linker characteristics vary from system to system. Generally, you just tell the linker the names of the object modules and libraries you want linked together, and the name of the executable, and it goes to work. Some systems require you to invoke the linker yourself. With most C++ packages you invoke the linker through the C++ compiler. In many situations, the linker is invoked for you, invisibly.

Some older linkers won't search object files and libraries more than once, and they search through the list you give them from left to right. This means that the order of object files and libraries can be important. If you have a mysterious problem that doesn't show up until link time, one possibility is the order in which the files are given to the linker.

Using libraries

Now that you know the basic terminology, you can understand how to use a library. To use a library:

1. Include the library's header file
2. Use the functions and variables in the library
3. Link the library into the executable program

These steps also apply when the object modules aren't combined into a library. Including a header file and linking the object modules are the basic steps for separate compilation in both C and C++.

How the linker searches a library

When you make an external reference to a function or variable in C or C++, the linker, upon encountering this reference, can do one of two things. If it has not already encountered the definition for the function or variable, it adds the identifier to its list of "unresolved references." If the linker has already encountered the definition, the reference is resolved.

If the linker cannot find the definition in the list of object modules, it searches the libraries. Libraries have some sort of indexing so the linker doesn't need to look through all the object modules in the library – it just looks in the index. When the linker finds a definition in a library, the entire object module, not just the function definition, is linked into the executable program. Note that the whole library isn't linked, just the object module in the library that contains the definition you want (otherwise programs would be unnecessarily large). If you want to minimize executable program size, you might consider putting a single function in each source code file when you build your own libraries. This requires more editing[16], but it can be helpful to the user.

Because the linker searches files in the order you give them, you can pre-empt the use of a library function by inserting a file with your own function, using the same function name, into the list before the library name appears. Since the linker will resolve any references to this function by using your function before it searches the library, your function is used instead of the library function.

---

[16] I would recommend using Perl to automate this task as part of your library-packaging process.

Secret additions

When a C or C++ executable program is created, certain items are secretly linked in. One of these is the startup module, which contains initialization routines that must be run any time a C or C++ program begins to execute. These routines set up the stack and initialize certain variables in the program.

The linker always searches the standard library for the compiled versions of any “standard” functions called in the program. Because the standard library is always searched, you can use anything in that library by simply including the appropriate header file in your program – you don’t have tell it to search the standard library. The iostream functions, for example, are in the Standard C++ library. To use them, you just include the **<iostream>** header file.

If you are using an add-on library, you must explicitly add the library name to the list of files handed to the linker.

Using plain C libraries

Just because you are writing code in C++, you are not prevented from using C library functions. In fact, the entire C library is included by default into Standard C++. There has been a tremendous amount of work done for you in these functions, so they can save you a lot of time.

This book will use Standard C++ (and thus also Standard C) library functions when convenient, but only *standard* library functions will be used, to ensure the portability of programs. In the few cases where library functions must be used that are not in the C++ standard, all attempts will be made to use POSIX-compliant functions. POSIX is a standard based on a Unix standardization effort which includes functions that go beyond the scope of the C++ library. You can generally expect to find POSIX functions on Unix (in particular, Linux) platforms, and often under DOS/Windows.

Your first C++ program

You now know almost enough of the basics to create and compile a program. The program will use the Standard C++ iostream classes. These read from and write to files and “standard” input and output (which normally comes from and goes to the console, but may be redirected to files or devices). In this very simple program, a stream object will be used to print a message on the screen.

Using the iostreams class

To declare the functions and external data in the iostreams class, include the header file with the statement

```
#include <iostream>
```

The first program uses the concept of standard output, which means "a general-purpose place to send output." You will see other examples using standard output in different ways, but here it will just go to the console. The iostream package automatically defines a variable (an object) called **cout** that accepts all data bound for standard output.

To send data to standard output, you use the operator **<<**. C programmers know this operator as the "bitwise left shift," which will be described in the next chapter. Suffice it to say that a bitwise left shift has nothing to do with output. However, C++ allows operators to be *overloaded*. When you overload an operator, you give it a new meaning when that operator is used with an object of a particular type. With iostream objects, the operator **<<** means "send to." For example:

```
cout << "howdy!";
```

sends the string "howdy!" to the object called **cout** (which is short for "console output").

That's enough operator overloading to get you started. Chapter XX covers operator overloading in detail.

Namespaces

As mentioned in the previous chapter, one of the problems encountered in the C language is that you "run out of names" for functions and identifiers when your programs reach a certain size. Of course, you don't really run out of names – however, it becomes harder to think of new ones after awhile. More importantly, when a program reaches a certain size it's typically broken up into pieces, each of which is built and maintained by a different person or group. Since C effectively has a single arena where all the identifier and function names live, this means that all the developers must be careful not to accidentally use the same names in situations where they can conflict. This rapidly becomes tedious, time-wasting and, ultimately, expensive.

Standard C++ has a mechanism to prevent this collison: the **namespace** keyword. Each set of C++ definitions in a library or program is "wrapped"

in a namespace, and if some other definition has an identical name, but is in a different namespace, then there is no collision.

Namespaces are a convenient and helpful tool, but their presence means you must be aware of them before you can write any programs at all. If you simply include a header file and use some functions or objects from that header, you'll probably get strange-sounding errors when you try to compile the program, to the effect that the compiler cannot find any of the declarations for the items that you just included in the header file! After you see this message a few times you'll become familiar with its meaning (which is: "you included the header file but all the declarations are within a namespace and you didn't tell the compiler that you wanted to use the declarations in that namespace").

There's a keyword that allows you to say "I want to use the declarations and/or definitions in this namespace." This keyword, appropriately enough, is **using**. All of the Standard C++ libraries are wrapped in a single namespace, which is **std** (for "standard"). As this book uses the standard libraries almost exclusively, you'll see the following *using directive* in almost every program:

```
using namespace std;
```

This means that you want to expose all the elements from the namespace called **std**. After this statement, you don't have to worry that your particular library component is inside a namespace, since the **using** directive makes that namespace available throughout the file where the **using** directive was written.

Exposing all the elements from a namespace after someone has gone to the trouble to hide them may seem a bit counterproductive, and in fact you should be careful about thoughtlessly doing this (as you'll learn later in the book). However, the **using** directive only exposes those names for the current file, so it is not quite so drastic as it first sounds (but think twice about doing it in a header file – that *is* reckless).

There's a relationship between namespaces and the way header files are included. Before the current header file inclusion style of **<iostream>** (that is, no trailing '**.h**') was standardized, the typical way to include a header file was with the '**.h**', such as **<iostream.h>**. At that time, namespaces were not part of the language, either. So to provide backwards compatibility with existing code, if you say

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

It means

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

However, in this book the standard include format will be used (without the '**.h**') and so the **using** directive must be explicit.

For now, that's all you need to know about namespaces, but in Chapter XX the subject is covered much more thoroughly.

Fundamentals of program structure

A C or C++ program is a collection of variables, function definitions and function calls. When the program starts, it executes initialization code and calls a special function, "**main( )**." You put the primary code for the program here.

As mentioned earlier, a function definition consists of a return type (which must be specified in C++), a function name, an argument list in parentheses, and the function code contained in braces. Here is a sample function definition:

```
int function() {
  // Function code here (this is a comment)
}
```

The above function has an empty argument list, and a body that contains only a comment.

There can be many sets of braces within a function definition, but there must always be at least one set surrounding the function body. Since **main( )** is a function, it must follow these rules. In C++, **main( )** always has return type of **int**.

C and C++ are free form languages. With few exceptions, the compiler ignores newlines and white space, so it must have some way to determine the end of a statement. Statements are delimited by semicolons.

C comments start with **/*** and end with ***/**. They can include newlines. C++ uses C-style comments and has an additional type of comment: **//**. The **//** starts a comment that terminates with a newline. It is more convenient than **/* */** for one-line comments, and is used extensively in this book.

"Hello, world!"

And now, finally, the first program:

```
//: C02:Hello.cpp
// Saying Hello with C++
#include <iostream> // Stream declarations
using namespace std;

int main() {
  cout << "Hello, World! I am " << 8 << " Today!" << endl;
} ///:~
```

The **cout** object is handed a series of arguments via the '**<<**' operators. It prints out these arguments in left-to-right order. The special iostream function **endl** outputs the line and a newline. With iostreams, you can string together a series of arguments like this, which makes the class easy to use.

In C, text inside double quotes is traditionally called a "string." However, the Standard C++ library has a powerful class called **string** for manipulating text, and so I shall use the more precise term *character array* for text inside double quotes.

The compiler creates storage for character arrays and stores the ASCII equivalent for each character in this storage. The compiler automatically terminates this array of characters with an extra piece of storage containing the value 0, to indicate the end of the character array.

Inside a character array, you can insert special characters by using *escape sequences*. These consist of a backslash (****) followed by a special code. For example **\n** means newline. Your compiler manual or local C guide gives a complete set of escape sequences; others include **\t** (tab), **** (backslash) and **\b** (backspace).

Notice that the entire statement terminates with a semicolon.

Character array arguments and constant numbers are mixed together in the above **cout** statement. Because the operator **<<** is overloaded with a variety of meanings when used with **cout**, you can send **cout** a variety of different arguments, and it will "figure out what to do with the message."

Throughout this book you'll notice that the first line of each file will be a comment that starts with the characters that start a comment (typically **//**), followed by a colon. This is a technique I use to allow easy extraction of information from code files (the program to do this is in the last chapter

of the book). The first line also has the name and location of the file, so it can be referred to in text and in other files, and so you can easily locate it in the source code for this book (which is freely downloadable from http://www.BruceEckel.com, where you'll find instructions on how to unpack it).

Running the compiler

After downloading and unpacking the book's source code, find the program in the subdirectory **CO2**. Invoke the compiler with **Hello.cpp** as the argument. For simple, one-file programs like this one, most compilers will take you all the way through the process. For example, to use the Gnu C++ compiler (which is freely available on the Internet), you say:

g++ Hello.cpp

Other compilers will have a similar syntax; consult your compiler's documenation for details.

More about iostreams

So far you have seen only the most rudimentary aspect of the iostreams class. The output formatting available with iostreams also includes features like number formatting in decimal, octal and hex. Here's another example of the use of iostreams:

```
//: C02:Stream2.cpp
// More streams features
#include <iostream>
using namespace std;

int main() {
  // Specifying formats with manipulators:
  cout << "a number in decimal: "
       << dec << 15 << endl;
  cout << "in octal: " << oct << 15 << endl;
  cout << "in hex: " << hex << 15 << endl;
  cout << "a floating-point number: "
       << 3.14159 << endl;
  cout << "non-printing char (escape): "
       << char(27) << endl;
} ///:~
```

This example shows the iostreams class printing numbers in decimal, octal and hexadecimal using iostream *manipulators* (which don't print anything, but change the state of the output stream). The formatting of floating-point numbers is determined automatically, by the compiler. In addition, any character can be sent to a stream object using a *cast* to a **char** (a **char** is a data type which holds single characters). This *cast* looks like a function call: **char( )**, along with the character's ASCII value. In the above program, the **char(27)** sends an "escape" to **cout**.

Character array concatenation

An important feature of the C preprocessor is *character array concatenation*. This feature is used in some of the examples in this book. If two quoted character arrays are adjacent, and no punctuation is between them, the compiler will paste the character arrays together into a single character array. This is particularly useful when code listings have width restrictions:

```
//: C02:Concat.cpp
// Character array Concatenation
#include <iostream>
using namespace std;

int main() {
  cout << "This is far too long to put on a single "
    "line but it can be broken up with no ill effects\n"
    "as long as there is no punctuation separating "
    "adjacent character arrays.\n";
} ///:~
```

At first, the above code can look like an error because there's no familiar semicolon at the end of each line. Remember that C and C++ are free-form languages, and although you'll usually see a semicolon at the end of each line, the actual requirement is for a semicolon at the end of each statement, and it's possible for a statement to continue over several lines.

Reading input

The iostreams classes provide the ability to read input. The object used for standard input is **cin** (for "console input"). **cin** normally expects input from the console, but this input can be redirected from other sources. An example of redirection is shown later in this chapter.

The iostreams operator used with **cin** is **>>**. This operator waits for the same kind of input as its argument. For example, if you give it an integer argument, it waits for an integer from the console. Here's an example:

```
//: C02:Numconv.cpp
// Converts decimal to octal and hex
#include <iostream>
using namespace std;

int main() {
  int number;
  cout << "Enter a decimal number: ";
  cin >> number;
  cout << "value in octal = 0" << oct << number << endl;
  cout << "value in hex = 0x" << hex << number << endl;
} ///:~
```

This program converts a number typed in by the user into octal and hexadecimal representations.

Simple file manipulation

Standard I/O provides a very simple way to read and write files, called I/O redirection. If a program takes input from standard input (**cin** for iostreams) and sends its output to standard output (**cout** for iostreams), that input and output can be redirected. Input can be taken from a file, and output can be sent to a file. To re-direct I/O on the command line of some operating systems (Unix/Linux & DOS, in particular), use the '**<**' sign to redirect input and the '**>**' sign to redirect output. For example, if we have a fictitious program called **fiction** that reads from standard input and writes to standard output, you can redirect standard input from the file **stuff** and redirect the output to the file **such** with the command:

fiction < stuff > such

Since the files are opened for you, the job is much easier (although you'll see later that iostreams has a very simple mechanism for opening files).

As a useful example, suppose you want to record the number of times you perform an activity, but the program that records the incidents must be loaded and run many times, and the machine may be turned off, etc. To keep a permanent record of the incidents, you must store the data in a file. This file will be called **incident.dat** and will initially contain the character 0. For easy reading, it will always contain ASCII digits representing the number of incidents.

The program to increment the number is very simple:

```
//: C02:Incr.cpp
// Read a number, add one and write it
#include <iostream>
using namespace std;

int main() {
  int num;
  cin >> num;
  cout << num + 1;
} ///:~
```

To test the program, run it, type a number and press the “Enter” key. The program should print a number one larger than the one you type.

While the typical way to use a program that reads from standard input and writes to standard output is within a Unix shell script or DOS batch file, any program can be called from inside a C or C++ program using the Standard C **system( )** function, which is declared in the header file **<cstdlib>**:

```
//: C02:Incident.cpp
// Records an incident using INCR
#include <cstdlib> // Declare "system()"
using namespace std;

int main() {
  // Other code here...
  system("incr < incident.dat > incident.dat");
} ///:~
```

To use the **system( )** function, you give it a character array that you would normally type at the operating system command prompt. The command executes and control returns to the program[17].

Notice that the file **incident.dat** is read and written using I/O redirection. Since the single ‘**>**’ is used, the file is overwritten. Although it works fine here, reading and writing the same file isn’t always a safe thing to do – if you aren’t careful you can end up with garbage in the file.

[17] If you find yourself writing programs that use a lot of **system( )** calls, you might be more productive using the Perl language instead.

If a double '**>>**' is used instead of a single '**>**', the output is appended to the file (and this program wouldn't work correctly).

This program shows you how easy it is to use plain C library functions in C++: just include the header file and call the function. This upward compatibility from C to C++ is a big advantage if you are learning the language starting from a background in C.

Introducing **string**s

While a character array can be fairly useful, it is quite limited. It's simply a group of characters in memory, but if you want to do anything with it you must manage all the little details. For example, the size of a quoted character array is fixed at compile time. If you have a character array and you want to add some more characters to it, you'll need to understand quite a lot (inluding dynamic memory management, character array copying and concatenation) before you can get your wish. This is exactly the kind of thing we'd like to have an object do for us.

The Standard C++ **string** class is designed to take care of (and hide) all the low-level manipulations of character arrays that were previously required of the C++ programmer. These manipulations have been a constant source of time-wasting and errors since the inception of the C language. So, although an entire chapter is devoted to the **string** class later in the book, the **string** is so important and it makes life so much easier that it will be introduced here and used in much of the early part of the book.

To use **string**s you include the C++ header file **<string>**. The **string** class is in the namespace **std** so a **using** directive is necessary. Because of operator overloading, the syntax for using **string**s is quite intuitive:

```
//: C02:HelloStrings.cpp
// The basics of the Standard C++ string class
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // Empty strings
  string s3 = "Hello, World."; // Initialized
  string s4("I am"); // Also initialized
  s2 = "Today"; // Assigning to a string
```

```
  s1 = s3 + " " + s4; // Combining strings
  s1 += " 8 "; // Appending to a string
  cout << s1 + s2 + "!" << endl;
} ///:~
```

The first two **string**s, **s1** and **s2**, start out empty, while **s3** and **s4** show two equivalent ways to initialize **string** objects from character arrays (you can as easily initialize **string** objects from other **string** objects).

You can assign to any **string** object using '**=**'. This replaces the previous contents of the string with whatever is on the right-hand side, and you don't have to worry about what happens to the previous contents – that's handled automatically for you. To combine **string**s you simply use the '**+**' operator, which also allows you to combine character arrays with **string**s. If you want to append either a **string** or a character array to another **string**, you can use the operator '**+=**'. Finally, note that iostreams already know what to do with **string**s, so you can just send a **string** (or an expression that produces a **string**, which happens with **s1 + s2 + "!"**) directly to **cout** in order to print it.

Reading and writing files

We could use IO redirection in order to read files and write files, as shown previously. In C, the process of opening and manipulating files requires a lot of language background to prepare you for the complexity of the operations. However, the C++ iostream library provides a very simple way to manipulate files, and so this functionality can be introduced much earlier than it would be in C.

To open files for reading and writing, you must include **<fstream>**. Although this will automatically include **<iostream>**, it's generally prudent to explicitly include **<iostream>** if you're planning to use **cin**, **cout**, etc.

To open a file for reading, you create an **ifstream** object, which then behaves like **cin**. To open a file for writing, you create an **ofstream** object, which then behaves like **cout**. Once you've opened the file, you can read from it or write to it just as you would with any other iostream object. It's that simple (which is, of course, the whole point).

One of the most useful functions in the iostream library is **getline( )**, which allows you to read one line (terminated by a newline) into a **string**

object[18]. The first argument is the **ifstream** object you're reading from, and the second argument is the **string** object. When the function call is finished, the **string** object will contain the line.

Here's a simple example, which copies the contents of one file into another:

```
//: C02:Scopy.cpp
// Copy one file to another, a line at a time
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Open for reading
  ofstream out("Scopy2.cpp"); // Open for writing
  string s;
  while(getline(in, s)) // Discards newline char
    out << s << "\n"; // ... must add it back
} ///:~
```

To open the files, you just hand the **ifstream** and **ofstream** objects the file names you want to create, as seen above.

There is a new concept introduced here, which is the **while** loop. Although this will be explained in detail in the next chapter, the basic idea is that the expression in parentheses following the **while** controls the execution of the subsequent statement (which can also be multiple statements, wrapped inside curly braces). As long as the expression in parentheses (in this case, **getline(in, s)**) produces a "true" result, then the statement controlled by the **while** will continue to execute. It turns out that **getline( )** will return a value that can be interpreted as "true" if another line has been successfully read, and "false" upon reaching the end of the input. Thus, the above **while** loop reads every line in the input file and sends each line to the output file.

getline( ) reads each line until it discovers a newline (the termination character can be changed, but that won't be an issue until Chapter XX). However, it discards the newline and doesn't store it in the resulting **string** object. Thus, if we want the copied file to look just like the source file, we must add the newline back in, as shown.

[18] There are actually a number of variants of **getline( )**, which will be discussed thoroughly in the iostreams chapter, Chapter XX.

Another interesting example is to copy the entire file into a single **string** object:

```
//: C02:FillString.cpp
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, line;
  while(getline(in, line))
    s += line + "\n";
  cout << s;
} ///:~
```

Because of the dynamic nature of **string**s, you don't have to worry about how much storage to allocate for a **string** – you can just keep adding things and the **string** will keep expanding to hold whatever you put into it.

One of the nice things about putting an entire file into a **string** is that the **string** class has many functions for searching and manipulation which would then allow you to modify the file as a single string. However, this has its limitations. For one thing, it is often convenient to treat a file as a collection of lines instead of just a big blob of text. For example, if you want to add line numbering it's much easier if you have each line as a separate **string** object. To accomplish this, we'll need another approach.

Introducing **vector**

With **string**s, we can fill up a **string** object without knowing how much storage we're going to need. The problem with reading lines from a file into individual **string** objects is that you don't know up front how many **string**s you're going to need – you only know after you've read the entire file. To solve this problem, we need some sort of holder that will automatically expand to contain as many **string** objects as we care to put into it.

In fact, why limit ourselves to holding **string** objects? It turns out that this kind of problem – not knowing how many of something you have

while you're writing a program – happens a lot. And this "container" object sounds like it would be more useful if it would hold *any kind of object at all!* Fortunately, the Standard C++ library has a ready-made solution: the STL container classes. STL stands for "Standard Template Library," and it's one of the real powerhouses of Standard C++. Even though the implementation of the STL uses some advanced concepts, and the full coverage of the STL is given two large chapters later in this book, this library can also be very potent without knowing a lot about it. It's so useful that the most basic of the STL containers, the **vector**, is introduced in this early chapter and used throughout the introductory part of the book. You'll find that you can do a tremendous amount just by using the basics of **vector** and not worrying about the underlying implementation (again, an important goal of OOP). Since you'll learn much more about this and the other containers when you reach the STL chapters, it seems forgiveable if the programs that use **vector** in the early portion of the book aren't exactly what an experienced C++ programmer would do. You'll find that in most cases, the usage shown here is adequate.

The **vector** class is a *template*, which means that it can be efficiently applied to different types. That is, we can create a **vector** of **shape**s, a **vector** of **cat**s, a **vector** of **string**s, etc. Basically, with a template you can create a "class of anything." To tell the compiler what it is that the class will work with (in this case, what the **vector** will hold), you put the name of the desired type in "angle brackets," which means 'less-than' and 'greater-than' signs. So a **vector** of **string** would be denoted **vector<string>**. When you do this, you end up with a customized vector that will only hold **string** objects, and you'll get an error message from the compiler if you try to put anything else into it.

Since **vector** expresses the concept of a "container," there must be a way to put things into the container and get things back out of the container. To add a brand-new element on the end of a **vector**, you use a the member function **push_back( )** (remember that, since it's a member function, you use a '**.**' to call it for a particular object). The reason the name of this member function might seem a bit verbose – **push_back( )** instead of something simpler like "put" – is because there are other containers and other member functions for putting new elements into containers. For example, the there is an **insert( )** member function to put something in the middle of a container. **vector** supports this but its use is more complicated and we won't need to explore it until later in the book. There's also a **push_front( )** (not part of **vector**) to put things at the beginning. There are many more member functions in **vector** and many

more containers in the STL, but you'll be surprised at how much you can do just knowing about a few simple features.

So you can put new elements into a **vector** with **push_back( )**, but how do you get these elements back out again? This solution is more clever and elegant – operator overloading is used to make the **vector** look like an *array*. The array (which will be described more fully in the next chapter) is a data type which is available in virtually every programming language so you should already be somewhat familiar with it. Arrays are *aggregates*, which mean they consist of a number of elements clumped together. The distinguishing characteristic of an array is that these elements are the same size and are arranged to be one right after the other. Most importantly, these elements can be selected by "indexing," which means you can say "I want element number n" and that element will be produced, usually very quickly. Although there are exceptions in programming languages, the indexing is normally achieved using square brackets, so if you have an array **a** and you want to produce element 5, you say **a[5]**.

This very compact and powerful indexing notation is incorporated into the **vector** using operator overloading, just like '**<<**' and '**>>**' were incorporated into iostreams. Again, you don't need to know how the overloading was implemented – that's saved for a later chapter – but it's helpful if you're aware that there's some magic going on under the covers in order to make the **[ ]** work with **vector**.

With that in mind, you can now see a program that uses **vector**. To use a **vector**, you include the header file **<vector>**:

```
//: C02:Fillvector.cpp
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line); // Add the line to the end
  // Add line numbers:
```

```
  for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

Much of this program is similar to the previous one: a file is opened and lines are read into **string** objects one at a time. However, these **string** objects are pushed onto the back of the **vector v**. Once the **while** loop completes, the entire file is resident in memory, inside **v**.

The next statement in the program is called a **for** loop. It is similar to a **while** loop except that it adds some extra control. After the **for**, there is a "control expression" inside of parentheses, just like the **while** loop. However, this control expression is in three parts: a part which initializes, one that tests to see if we should exit the loop, and one which changes something, typically to step through a sequence of items. This program shows the **for** loop in the way you'll see it most commonly used: the initialization part **int i = 0** creates an integer **i** to use as a loop counter and gives it an initial value of zero. The testing portion says that to stay in the loop, **i** should be less than the number of elements in the **vector v** (this is produced using the member function **size( )** which I just sort of slipped in here, but you must admit it has a fairly obvious meaning). The final portion uses a shorthand for C and C++, the "auto-increment" operator, to add one to the value of **i**. Effectively, **i++** says "get the value of **i**, add one to it, and put the result back into **i**. Thus, the total effect of the **for** loop is to take a variable **i** and march it through the values from zero to one less than the size of the **vector**. For each value of **i**, the **cout** statement is executed and this builds a line that consists of the value of **i** (magically converted to a character array by **cout**), a colon and a space, the line from the file, and a newline provided by **endl**. When you compile and run it you'll see the effect is to add line numbers to the file.

Because of the way that the '**>>**' operator works with iostreams, you can easily modify the above program so that it breaks up the input into whitespace-separated words instead of lines:

```
//: C02:GetWords.cpp
// Break a file into whitespace-separated words
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
```

```
  vector<string> words;
  ifstream in("GetWords.cpp");
  string word;
  while(in >> word)
    words.push_back(word);
  for(int i = 0; i < words.size(); i++)
    cout << words[i] << endl;
} ///:~
```

The expression

```
while(in >> word)
```

is what gets the input one “word” at a time, and when this expression evaluates to false it means the end of the file has been reached. Of course, delimiting words by whitespace is quite crude, but it makes for a simple example. Later in the book you’ll see more sophisticated examples that let you break up input just about any way you’d like.

To demonstrate how easy it is to use a **vector** with any type, here’s an example that creates a **vector<int>**:

```
//: C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // Assignment
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

To create a **vector** that holds a different type, you just put that type in as the template argument (the argument in angle brackets). Templates and

well-designed template libraries are intended to be exactly this easy to use.

This example goes on to demonstrate another essential feature of **vector**. In the expression

```
v[i] = v[i] * 10;
```

you can see that the **vector** is not limited to only putting things in and getting things out. You also have the ability to *assign* (and thus to change) to any element of a **vector**, also through the use of the square-brackets indexing operator. This means that **vector** is a very general-purpose, very flexible "scratchpad" for working with collections of objects, and we will definitely make use of it in coming chapters.

Summary

The intent of this chapter is to show you how easy object-oriented programming can be – *if* someone else has gone to the work of defining the objects for you. In that case, you include a header file, create the objects, and send messages to them. If the types you are using are very powerful and well-designed, then you won't have to do very much work and your resulting program will also be powerful.

In the process of showing the ease of OOP when using library classes, this chapter also introduced some of the most basic and useful types in the Standard C++ library: the family of iostreams (in particular, those that read from and write to the console and files), the **string** class, and the **vector** template. You've seen how straightforward it is to use these and can now probably imagine many things you can accomplish with them, but there's actually a lot more that they're capable of[19]. Even though we'll only be using a limited subset of the functionality of these tools in the early part of the book, they nonetheless provide a very large step up from the primitiveness of learning a low-level language like C – and while learning the low-level aspects of C is very educational, it's also time consuming. In the end, you'll be much more productive if you've got objects to manage the low-level issues. After all, the whole *point* of OOP is to hide the details so you can "paint with a bigger brush."

---

[19] If you're particularly eager to see all the things that can be done with these and other Standard library components, see [[location of $20 C++ standard]] or [[www.dinkumware.com]]

However, as high-level as OOP tries to be, there are some fundamental aspects of C that you can't avoid knowing, and these will be covered in the next chapter.

Exercises

1. Modify **Hello.cpp** so that it prints out your name and age (or shoe size, or your dog's age, if that makes you feel better). Compile and run the program.
2. Starting with **Stream2.cpp** and **Numconv.cpp**, create a program that asks for the radius of a circle and prints the area of that circle. You can just use the '*****' operator to square the radius.
3. Create a program that opens a file and counts the whitespace-separated words in that file.
4. Create a program that counts the occurrence of a particular word in a file (use the **string** class' operator '**==**' to find the word).
5. Modify **FillString.cpp** so that it adds line numbers to each of the input lines as they are added to **s**.
6. Change **Fillvector.cpp** so it prints the lines (backwards) from last to first.
7. Change **Fillvector.cpp** so it concatenates all the elements in the **vector** into a single string before printing it out, but don't try to add line numbering.
8. Display a file a line at a time, waiting for the user to press the "Enter" key after each line.
9. Create a **vector<float>** and put 25 floating-point numbers into it using a **for** loop. Display the **vector**.
10. Create three **vector<float>** objects and fill the first two as in the previous exercise. Write a **for** loop that adds each corresponding element in the first two **vector**s and puts the result in the corresponding element of the third **vector**. Display all three **vector**s.
11. Create a **vector<float>** and put 25 numbers into it as in the previous exercises. Now square each number and put the result back into the same location in the **vector**. Display the **vector** before and after the multiplication.

3: The C in C++

Since C++ is based on C, you must be familiar with the syntax of C in order to program in C++, just as you must be reasonably fluent in algebra in order to tackle calculus.

If you've never seen C before, this chapter will give you a decent background in the style of C used in C++. If you are familiar with the style of C described in the first edition of Kernighan & Ritchie (often called K&R C) you will find some new and different features in C++ as well as in Standard C. If you are familiar with Standard C, you should skim through this chapter looking for features that are particular to C++. Note that there are some fundamental C++ features introduced here, although they are basic ideas that are akin to the features in C. The more sophisticated C++ features will not be introduced until later chapters.

This chapter is a fairly fast coverage of C constructs, with the understanding that you've had some experience programming in another language. If after reading the chapter you still don't feel comfortable with the fundamentals, you may want to consider purchasing *Thinking in C: Foundations for Java & C++* by Chuck Allison (published by MindView, Inc., and available at **http://www.MindView.net**, where you'll find the introductory lecture as a free demonstration). This is a seminar on a CD-ROM, much like the CD packaged with this book, and its goal is to take you carefully through the fundamentals of the C language, but focusing on the knowledge necessary for you to be able to move on to the C++ or Java languages rather than trying to make you an expert in all the dark corners of C (one of the reasons for using a higher-level language like C++ or Java is precisely so we can avoid many of these dark corners). It also contains excercises and guided solutions.

Creating functions

In old (pre-Standard) C, you could call a function with any number or type of arguments, and the compiler wouldn't complain. Everything seemed fine until you ran the program. You got mysterious results (or worse, the program crashed) with no hints as to why. The lack of help with argument passing and the enigmatic bugs that resulted is probably one reason why C was dubbed a "high-level assembly language." Pre-Standard C programmers just adapted to it.

Standard C and C++ use a feature called *function prototyping*. With function prototyping, you must use a description of the types of arguments when declaring and defining a function. This description is the "prototype." When the function is called, the compiler uses the prototype to ensure the proper arguments are passed in, and that the return value is treated correctly. If the programmer makes a mistake when calling the function, the compiler catches the mistake.

Essentially, you learned about function prototyping (without naming it as such) in the previous chapter, since the form of function declaration in C++ requires proper prototyping. In a function prototype, the argument list contains the types of arguments that must be passed to the function and (optionally for the declaration) identifiers for the arguments. The order and type of the arguments must match in the declaration, definition and function call. Here's an example of a function prototype in a declaration:

```
int translate(float x, float y, float z);
```

You do not use the same form when declaring variables in function prototypes as you do in ordinary variable definitions. That is, you cannot say: **float x, y, z**. You must indicate the type of *each* argument. In a function declaration, the following form is also acceptable:

```
int translate(float, float, float);
```

Since the compiler doesn't do anything but check for types when the function is called, the identifiers are only included for clarity, when someone is reading the code.

In the function definition, names are required because the arguments are referenced inside the function:

```
int translate(float x, float y, float z) {
  x = y = z;
```

```
  // ...
}
```

It turns out this rule only applies to C. In C++, an argument may be unnamed in the argument list of the function definition. Since it is unnamed, you cannot use it in the function body, of course. The reason unnamed arguments are allowed is to give the programmer a way to "reserve space in the argument list." Whoever uses the function must still call the function with the proper arguments. However, the person creating the function can then use the argument in the future without forcing modification of code that calls the function. This option of ignoring an argument in the list is also possible if you leave the name in, but you will get an obnoxious warning message about the value being unused every time you compile the function. The warning is eliminated if you remove the name.

C and C++ have two other ways to declare an argument list. If you have an empty argument list you can declare it as **func( )** in C++, which tells the compiler there are exactly zero arguments. You should be aware that this only means an empty argument list in C++. In C it means "an indeterminate number of arguments (which is a "hole" in C since it disables type checking in that case). In both C and C++, the declaration **func(void);** means an empty argument list. The **void** keyword means "nothing" in this case (it can also mean "no type" in the case of pointers, as you'll see later in this chapter).

The other option for argument lists occurs when you don't know how many arguments or what type of arguments you will have; this is called a variable argument list. This "uncertain argument list" is represented by ellipses (**...**). Defining a function with a variable argument list is significantly more complicated than defining a regular function. You can use a variable argument list for a function that has a fixed set of arguments if (for some reason) you want to disable the error checks of function prototyping. Because of this, you should restrict your use of variable argument lists to C, and avoid them in C++ (where, as you'll learn, there are much better alternatives). Handling variable argument lists is described in the library section of your local C guide.

Function return values

A C++ function prototype must specify the return value type of the function (in C, if you leave off the return value type it defaults to **int**). The return type specification precedes the function name. To specify that no value is returned, use the **void** keyword. This will generate an error if you

try to return a value from the function. Here are some complete function prototypes:

```
int f1(void); // Returns an int, takes no arguments
int f2(); // Like f1() in C++ but not in Standard C!
float f3(float, int, char, double); // Returns a float
void f4(void); // Takes no arguments, returns nothing
```

To return a value from a function, you use the **return** statement. **return** exits the function, back to the point right after the function call. If **return** has an argument, that argument becomes the return value of the function. If a function says that it will return a particular type, then each **return** statement must return that type. You can have more than one **return** statement in a function definition:

```
//: C03:Return.cpp
// Use of "return"
#include <iostream>
using namespace std;

char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}

int main() {
  cout << "type an integer: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

In **cfunc( )**, the first **if** that evaluates to **true** exits the function via the **return** statement. Notice that a function declaration is not necessary because the function definition appears before it is used in **main( )**, so the compiler knows about it from that function definition.

Using the C function library

All the functions in your local C function library are available while you are programming in C++. You should look hard at the function library before defining your own function – there's a good chance that someone has already solved your problem for you, and probably given it a lot more thought and debugging.

A word of caution, though: many compilers include a lot of extra functions that make life even easier and are very tempting to use, but are not part of the Standard C library. If you are certain you will never want to move the application to another platform (and who is certain of that?), go ahead –use those functions and make your life easier. If you want your application to be portable, you should restrict yourself to Standard library functions. If you must perform platform-specific activities, try to isolate that code in one spot so it can easily be changed when porting to another platform. In C++, platform-specific activities are often encapsulated in a class, which is the ideal solution.

The formula for using a library function is as follows: first, find the function in your programming reference (many programming references will index the function by category as well as alphabetically). The description of the function should include a section that demonstrates the syntax of the code. The top of this section usually has at least one **#include** line, showing you the header file containing the function prototype. Duplicate this **#include** line in your file, so the function is properly declared. Now you can call the function in the same way it appears in the syntax section. If you make a mistake, the compiler will discover it by comparing your function call to the function prototype in the header, and tell you about your error. The linker searches the standard library by default, so that's all you need to do: include the header file, and call the function.

Creating your own libraries with the librarian

You can collect your own functions together into a library. Most programming packages come with a librarian that manages groups of object modules. Each librarian has its own commands, but the general idea is this: if you want to create a library, make a header file containing the function prototypes for all the functions in your library. Put this header file somewhere in the preprocessor's search path, either in the local

directory (so it can be found by **#include "header"**) or in the include directory (so it can be found by **#include <header>**). Now take all the object modules and hand them to the librarian along with a name for the finished library (most librarians require a common extension, such as **.lib** or **.a**). Place the finished library where the other libraries reside, so the linker can find it. When you use your library, you will have to add something to the command line so the linker knows to search the library for the functions you call. You must find all the details in your local manual, since they vary from system to system.

Controlling execution

This section covers the execution control statements in C++. You must be familiar with these statements before you can read and write C or C++ code.

C++ uses all C's execution control statements. These include **if-else**, **while**, **do-while**, **for**, and a selection statement called **switch**. C++ also allows the infamous **goto**, which will be avoided in this book.

True and false

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **A == B**. This uses the conditional operator **==** to see if the variable **A** is equivalent to the variable B**.** The expression produces a boolean **true** or **false** (these are keywords only in C++; in C an expression is "true" if it evaluates to a nonzero value). Other conditional operators are **>**, **<**, **>=**, etc. Conditional statements are covered more fully later in this chapter.

if-else

The **if-else** statement can exist in two forms: with or without the **else**. The two forms are:

if(expression)
  statement

or

if(expression)
  statement

else
statement

The “expression” evaluates to **true** or **false**. The “statement” means either a simple statement terminated by a semicolon or compound statement, which is a group of simple statements enclosed in braces. Any time the word “statement” is used, it always implies that the statement is simple or compound. Note this statement can also be another **if**, so they can be strung together.

```
//: C03:Ifthen.cpp
// Demonstration of if and if-else conditionals
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "It's greater than 5" << endl;
  else
    if(i < 5)
      cout << "It's less than 5 " << endl;
    else
      cout << "It's equal to 5 " << endl;

  cout << "type a number and 'Enter'" << endl;
  cin >> i;
  if(i < 10)
    if(i > 5)  // "if" is just another statement
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
  else // Matches "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

It is conventional to indent the body of a control flow statement so the reader may easily determine where it begins and ends[20].

while

while, **do-while** and **for** control looping. A statement repeats until the controlling expression evaluates to **false**. The form of a **while** loop is

while(expression)
statement

The expression is evaluated once at the beginning of the loop, and again before each further iteration of the statement.

This example stays in the body of the **while** loop until you type the secret number or press control-C.

```
//: C03:Guess.cpp
// Guess a number (demonstrates "while")
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess = 0;
  // "!=" is the "not-equal" conditional:
  while(guess != secret) { // Compound statement
    cout << "guess the number: ";
    cin >> guess;
  }
  cout << "You guessed it!" << endl;
} ///:~
```

The **while**'s conditional expression is not restricted to a simple test as in the above example; it can be as complicated as you like as long as it produces a **true** or **false** result. You will even see code where the loop has no body, just a bare semicolon:

```
while(/* Do a lot here */)
  ;
```

[20] Note that all conventions seem to end after the agreement that some sort of indentation take place. The feud between styles of code formatting is unending. See appendix A for the description of this book's coding style.

In these cases the programmer has written the conditional expression not only to perform the test but also to do the work.

do-while

The form of **do-while** is

```
do
  statement
while(expression);
```

The **do-while** is different from the while because the statement always executes at least once, even if the expression evaluates to false the first time. In a regular **while**, if the conditional is false the first time the statement never executes.

If a **do-while** is used in **Guess.cpp**, the variable **guess** does not need an initial dummy value, since it is initialized by the **cin** statement before it is tested:

```
//: C03:Guess2.cpp
// The guess program using do-while
#include <iostream>
using namespace std;

int main() {
  int secret = 15;
  int guess; // No initialization needed here
  do {
    cout << "guess the number: ";
    cin >> guess; // Initialization happens
  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

For some reason, most programmers tend to avoid **do-while** and just work with **while**.

for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of “stepping.” The form of the **for** loop is:

for(*initialization***;** *conditional***;** *step***)**
  statement

Any of the expressions *initialization*, *conditional* or *step* may be empty. The *initialization* code executes once at the very beginning. The *conditional* is tested before each iteration (if it evaluates to false at the beginning, the statement never executes). At the end of each loop, the *step* executes.

for loops are usually used for "counting" tasks:

```
//: C03:Charlist.cpp
// Display all the ASCII characters
// Demonstrates "for"
#include <iostream>
using namespace std;

int main() {
  for(int i = 0; i < 128; i = i + 1)
    if (i != 26)  // ANSI Terminal Clear screen
      cout << " value: " << i <<
      " character: " <<
      char(i) << endl; // Type conversion
} ///:~
```

You may notice that the variable **i** is defined at the point where it is used, instead of at the beginning of the block denoted by the open curly brace '**{**'. This is in contrast to traditional procedural languages (including C), which require that all variables be defined at the beginning of the block. This will be discussed later in this chapter.

The **break** and **continue** Keywords

Inside the body of any of the looping constructs **while**, **do-while** or **for**, you can control the flow of the loop using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin a new iteration.

As an example of the use of **break** and **continue**, this program is a very simple menu system:

```
//: C03:Menu.cpp
```

```
// Simple menu program demonstrating
// the use of "break" and "continue"
#include <iostream>
using namespace std;

int main() {
  char c; // To hold response
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: left, r: right, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // Out of "while(1)"
    if(c == 'l') {
      cout << "LEFT MENU:" << endl;
      cout << "select a or b: ";
      cin >> c;
      if(c == 'a') {
        cout << "you chose 'a'" << endl;
        continue; // Back to main menu
      }
      if(c == 'b') {
        cout << "you chose 'b'" << endl;
        continue; // Back to main menu
      }
      else {
        cout << "you didn't choose a or b!"
             << endl;
        continue; // Back to main menu
      }
    }
    if(c == 'r') {
      cout << "RIGHT MENU:" << endl;
      cout << "select c or d: ";
      cin >> c;
      if(c == 'c') {
        cout << "you chose 'c'" << endl;
        continue; // Back to main menu
      }
      if(c == 'd') {
        cout << "you chose 'd'" << endl;
        continue; // Back to main menu
```

```
      }
      else {
        cout << "you didn't choose c or d!"
             << endl;
        continue; // Back to main menu
      }
    }
    cout << "you must type l or r or q!" << endl;
  }
  cout << "quitting menu..." << endl;
} ///:~
```

If the user selects 'q' in the main menu, the **break** keyword is used to quit, otherwise the program just continues to execute indefinitely. After each of the sub-menu selections, the **continue** keyword is used to pop back up to the beginning of the while loop.

The **while(true)** statement is the equivalent of saying "do this loop forever." The **break** statement allows you to break out of this infinite while loop when the user types a 'q.'

switch

A **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
       (...)
  default: statement;
}
```

Selector is an expression that produces an integral value. The **switch** compares the result of *selector* to each *integral-value*. If it finds a match, the corresponding statement (simple or compound) executes. If no match occurs, the **default** statement executes.

You will notice in the above definition that each **case** ends with a **break**, which causes execution to jump to the end of the **switch** body (the closing brace that completes the **switch**). This is the conventional way to build a **switch** statement, but the **break** is optional. If it is missing, your

case “drops through” to the one after it. That is, the code for the following **case** statements execute until a **break** is encountered. Although you don’t usually want this kind of behavior, it can be useful to an experienced programmer.

The **switch** statement is a very clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value at compile-time. If you want to use, for example, a **string** object as a selector, it won’t work in a **switch** statement. For a **string** selector, you must instead use a series of **if** statements and compare the **string** inside the conditional.

The menu example shown previously provides a particularly nice example of a **switch**:

```
//: C03:Menu2.cpp
// A menu using a switch statement
#include <iostream>
using namespace std;

int main() {
  bool quit = false;  // Flag for quitting
  while(quit == false) {
    cout << "Select a, b, c or q to quit: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "you chose 'a'" << endl;
                 break;
      case 'b' : cout << "you chose 'b'" << endl;
                 break;
      case 'c' : cout << "you chose 'c'" << endl;
                 break;
      case 'q' : cout << "quitting menu" << endl;
                 quit = true;
                 break;
      default  : cout << "Please use a,b,c or q!"
                 << endl;
    }
  }
} ///:~
```

The **quit** flag is a **bool**, short for "Boolean," which is a type you'll only find in C++. It can only have the keyword values **true** or **false**. Selecting 'q' sets the **quit** flag to **true**. The next time the selector is evaluated, **quit == false** returns **false** so the body of the **while** does not execute.

Recursion

Recursion is an interesting and sometimes useful programming technique whereby you call the function that you're in. Of course, if this is all you do you'll keep calling the function you're in until you run out of memory, so there must be some way to "bottom out" the recursive call. In the following example, this "bottoming out" is accomplished by simply saying that the recursion will only go until the **cat** exceeds 'Z':[21]

```
//: C03:CatsInHats.cpp
// Simple demonstration of recursion
#include <iostream>
using namespace std;

void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
    removeHat(cat + 1); // Recursive call
  } else
    cout << "VOOM!!!" << endl;
}

int main() {
  removeHat('A');
} ///:~
```

In **removeHat( )**, you can see that as long as **cat** is less than 'Z', **removeHat( )** will be called from *within* **removeHat( )**, thus effecting the recursion. Each time **removeHat( )** is called, its argument is one greater than the current **cat** so the argument keeps increasing.

Recursion is often used when evaluating some sort of arbitrarily complex problem, since you aren't restricted to a particular "size" for the solution –

[21] Thanks to Kris C. Matson for suggesting this exercise topic.

the function can just keep recursing until it's reached the end of the problem.

Introduction to operators

You can think of operators as a special type of function (you'll learn that C++ operator overloading treats operators precisely that way). An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary function calls, but the effect is the same.

From your previous programming experience, you should be reasonably comfortable with the operators that have been used so far. The concepts of addition (**+**), subtraction and unary minus (**-**), multiplication (*****), division (**/**) and assignment(**=**) all have essentialy the same meaning in any programming language. The full set of operators are enumerated later in this chapter.

Precedence

Operator precedence defines the order in which an expression evaluates when several different operators are present. C and C++ have specific rules to determine the order of evaluation. The easiest to remember is that multiplication and division happen before addition and subtraction. After that, if an expression isn't transparent to you it probably won't be for anyone reading the code, so you should use parentheses to make the order of evaluation explicit. For example:

```
A = X + Y - 2/2 + Z;
```

has a very different meaning from the same statement with a particular grouping of parentheses:

```
A = X + (Y - 2)/(2 + Z);
```

(Try evaluating the result with X = 1, Y = 2 and Z = 3.)

Auto increment and decrement

C, and therefore C++, is full of shortcuts. Shortcuts can make code much easier to type, and sometimes much harder to read. Perhaps the C

language designers thought it would be easier to understand a tricky piece of code if your eyes didn't have to scan as large an area of print.

One of the nicer shortcuts is the auto-increment and auto-decrement operators. You often use these to change loop variables, which control the number of times a loop executes.

The auto-decrement operator is **--** and means "decrease by one unit." The auto-increment operator is **++** and means "increase by one unit." If **A** is an **int**, for example, the expression **++A** is equivalent to (**A = A + 1**). Auto-increment and auto-decrement operators produce the value of the variable as a result. If the operator appears before the variable, (i.e., **++A**), the operation is first performed and the resulting value is produced. If the operator appears after the variable (i.e. **A++**), the current value is produced, and then the operation is performed. For example:

```
//: C03:AutoIncrement.cpp
// Shows use of auto-increment
// and auto-decrement operators.
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
} ///:~
```

If you've been wondering about the name "C++," now you understand. It implies "one step beyond C."

Introduction to data types

Data types define the way you utilize storage (memory) in the programs that you write. By specifying a data type, you tell the compiler how to create a particular piece of storage, and also how to manipulate that storage.

Data types can be built-in or abstract. A built-in data type is one that the compiler intrinsically understands, one that is wired directly into the compiler. The types of built-in data are almost identical in C and C++. In contrast, a user-defined data type is one that you or another programmer create as a class. These are commonly referred to as abstract data types. The compiler knows how to handle built-in types when it starts up; it "learns" how to handle abstract data types by reading header files containing class declarations (you'll learn about this in later chapters).

Basic built-in types

The Standard C specification for built-in types (which C++ inherits) doesn't say how many bits each of the built-in types must contain. Instead, it stipulates the minimum and maximum values that the built-in type must be able to hold. When a machine is based on binary, this maximum value can be directly translated into a minimum number of bits necessary to hold that value. However, if a machine uses, for instance, binary-coded decimal (BCD) to represent numbers then the amount of space in the machine required to hold the maximum numbers for each data type will be different. The minimum and maximum values that can be stored in the various data types are defined in the system header files **limits.h** and **float.h** (in C++ you will generally **#include <climits>** and **<cfloat>** instead).

C and C++ have four basic built-in data types, described here for binary-based machines. A **char** is for character storage and uses a minimum of 8 bits (one byte) of storage. An **int** stores an integral number and uses a minimum of two bytes of storage. The **float** and **double** types store floating-point numbers, usually in IEEE floating-point format. **float** is for single-precision floating point and **double** is for double-precision floating point.

As previously mentioned, you can define variables anywhere in a scope, and you can define and initialize them at the same time. Here's how to define variables using the four basic data types:

```
//: C03:Basic.cpp
// Defining the four basic data
// types in C and C++

int main() {
  // Definition without initialization:
  char protein;
  int carbohydrates;
```

```
  float fiber;
  double fat;
  // Simultaneous definition & initialization:
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
  // Exponential notation:
  double fudge_ripple = 6e-4;
} ///:~
```

The first part of the program defines variables of the four basic data types without initializing them. If you don't initialize a variable, the Standard says that its contents are undefined (usually, this means they contain garbage). The second part of the program defines and initializes variables at the same time (it's always best, if possible, to provide an initialization value at the point of definition). Notice the use of exponential notation in the constant 6e-4, meaning: "6 times 10 to the minus fourth power."

bool, true, & false

Before **bool** became part of Standard C++, everyone tended to use different techniques in order to produce Boolean-like behavior. These produced portability problems and could introduce subtle errors.

The Standard C++ **bool** type can have two states expressed by the built-in constants **true** (which converts to an integral one) and **false** (which converts to an integral zero). All three names are keywords. In addition, some language elements have been adapted:

Element	Usage with bool
&& || !	Take **bool** arguments and return **bool**.
< > <= >= == !=	Produce **bool** results
if, **for**, **while**, **do**	Conditional expressions convert to **bool** values
? :	First operand converts to **bool** value

Because there's a lot of existing code that uses an **int** to represent a flag, the compiler will implicitly convert from an **int** to a **bool**. Ideally, the compiler will give you a warning as a suggestion to correct the situation.

An idiom that falls under "poor programming style" is the use of **++** to set a flag to true. This is still allowed, but *deprecated*, which means that at some time in the future it will be made illegal. The problem is that you're making an implicit type conversion from **bool** to **int**, incrementing the value (perhaps beyond the range of the normal **bool** values of zero and one), and then implicitly converting it back again.

Pointers (which will be introduced later in this chapter) will also be automatically converted to **bool** when necessary.

Specifiers

Specifiers modify the meanings of the basic built-in types, and expand the built-in types to a much larger set. There are four specifiers: **long**, **short**, **signed** and **unsigned**.

long and **short** modify the maximum and minimum values that a data type will hold. A plain **int** must be at least the size of a **short**. The size hierarchy for integral types is: **short int**, **int**, **long int**. All the sizes could conceivably be the same, as long as they satisfy the minimum/maximum value requirements. On a machine with a 64-bit word, for instance, all the data types might be 64 bits.

The size hierarchy for floating point numbers is: **float**, **double**, and **long double**. "Long float" is not a legal type. There are no **short** floating-point numbers.

The **signed** and **unsigned** specifiers tell the compiler how to use the sign bit with integral types and characters (floating-point numbers always contain a sign). An **unsigned** number does not keep track of the sign and thus has an extra bit available, so it can store positive numbers twice as large as the positive numbers that can be stored in a **signed** number. **signed** is the default and is only necessary with **char**; **char** may or may not default to **signed**. By specifying **signed char**, you force the sign bit to be used.

The following example shows the size of the data types in bytes by using the **sizeof( )** operator, introduced later in this chapter:

```
//: C03:Specify.cpp
// Demonstrates the use of specifiers
#include <iostream>
```

```
using namespace std;

int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
  short iis; // Same as short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // Same as long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

Be aware that the results you get by running this program will probably be different from one machine/operating system/compiler to the next, since (as previously mentioned) the only thing that must be consistent is that each different type hold the minimum and maximum values specified in the Standard.

When you are modifying an **int** with **short** or **long**, the keyword **int** is optional, as shown above.

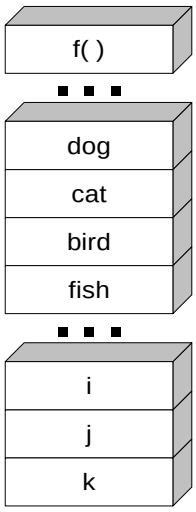
Introduction to Pointers

Whenever you run a program, it is first loaded (typically from disk) into the computer's memory. Thus, all elements of your program are located somewhere in memory. Memory is typically laid out as a sequential series of memory locations; we usually refer to these locations as eight-bit *bytes* but actually the size of each space depends on the architecture of that particular machine and is usually called that machine's *word size*. Each space can be uniquely distinguished from all other spaces by its *address*. For the purposes of this discussion, we'll just say that all machines use bytes which have sequential addresses starting at zero and going up to however much memory you have in your computer.

Since your program lives in memory while it's being run, every element of your program has an address. Suppose we start with a simple program:

```
//: C03:YourPets1.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
} ///:~
```

Each of the elements in this program has a location in storage when the program is running. You can visualize it like this:



All the variables, and even the function, occupy storage. As you'll see, it turns out that what an element is and the way you define it usually determines the area of memory where that element is placed.

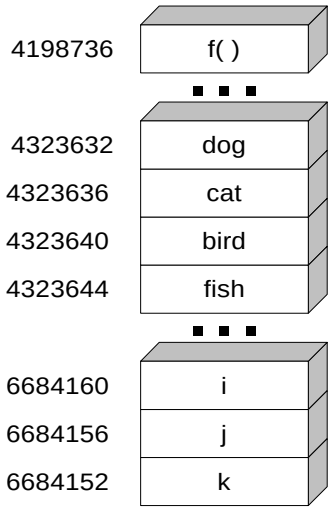
There is an operator in C and C++ that will tell you the address of an element. This is the '**&**' operator. All you do is precede the identifier name with '**&**' and it will produce the address of that identifier. **YourPets1.cpp** can be modified to print out the addresses of all its elements, like this:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;

int dog, cat, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
```

```
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

The **(long)** is a *cast*. It says "don't treat this as it's normal type, instead treat it as a **long**." The cast isn't essential, but if it wasn't there the addresses would have been printed out in hexadecimal instead, so casting to a **long** makes things a little more readable.

The results of this program will vary depending on your computer, OS and all sorts of other factors, but it will still give you some interesting insights. For a single run on my computer, the results looked like this:



You can see how the variables that are defined inside **main( )** are in a different area than the variables defined outside of **main( )**; you'll understand why as you learn more about the language. Also, **f( )** appears to be in its own area; code is typically separated from data in memory.

Another interesting thing to note is that variables defined one right after the other appear to be placed contiguously in memory. They are separated by the number of bytes that are required by their data type. Here, the only data type used is **int**, and **cat** is four bytes away from **dog**, **bird** is four bytes away from **cat**, etc. So it would appear that, on this machine, an **int** is four bytes long.

Other than this interesting experiment showing how memory is mapped out, what can you do with an address? The most important thing you can do is store it inside another variable for later use. C and C++ have a

special type of variable that holds an address. This variable is called a *pointer*.

The operator that defines a pointer is the same as the one used for multiplication: '*****'. The compiler knows that it isn't multiplication because of the context in which it is used, as you shall see.

When you define a pointer, you must specify the type of variable it points to. You start out by giving the type name, then instead of immediately giving an identifier for the variable, you say "wait, it's a pointer" by inserting a star between the type and the identifier. So a pointer to an **int** looks like this:

```
int* ip; // ip points to an int variable
```

The association of the '*****' with the type looks sensible and reads easily, but it can actually be a bit deceiving. Your inclination might be to say "intpointer" as if it is a single discrete type. However, with an **int** or other basic data type, it's possible to say:

```
int a, b, c;
```

whereas with a pointer, you'd *like* to say:

```
int* ipa, ipb, ipc;
```

C syntax (and by inheritance, C++ syntax) does not allow such sensible expressions. In the above definitions, only **ipa** is a pointer, but **ipb** and **ipc** are ordinary **int**s (you can say that "* binds more tightly to the identifer"). Consequently, the best results can be achieved by using only one definition per line: you still get the sensible syntax without the confusion:

```
int* ipa;
int* ipb;
int* ipc;
```

Since a general guideline for C++ programming is that you should always initialize a variable at the point of definition, this form actually works better. For example, the above variables are not initialized to any particular value; they hold garbage. It's much better to say something like:

```
int a = 47;
int* ipa = &a;
```

Now both **a** and **ipa** have been initialized, and **ipa** holds the address of **a**.

Once you have an initialized pointer, the most basic thing you can do with it is to use it to modify the value it points to. To access a variable through a pointer, you *dereference* the pointer using the same operator that you used to define it, like this:

```
*ipa = 100;
```

Now **a** contains the value 100 instead of 47.

These are the basics of pointers: you can hold an address, and you can use that address to modify the original variable. But the question still remains: why do you want to modify one variable using another variable as a proxy?

For this introductory view of pointers, we can put the answer into two broad categories:

1. To change “outside objects” from within a function. This is perhaps the most basic use of pointers, and it will be examined here.
2. To achieve many other clever programming techniques, which you’ll learn about in portions of the rest of the book.

Modifying the outside object

Ordinarily, when you pass an argument to a function, a copy of that argument is made inside the function. This is referred to as *pass-by-value*. You can see the effect of pass-by-value in the following program:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

In **f( )**, **a** is a *local variable*, so it only exists for the duration of the function call to **f( )**. Because it's a function argument, the value of **a** is initialized by the arguments that are passed when the function is called; in **main( )** the argument is **x** which has a value of 47, so this value is copied into **a** when **f( )** is called.

When you run this program you'll see:

```
x = 47
a = 47
a = 5
x = 47
```

Initially, of course, **x** is 47. When **f( )** is called, temporary space is created to hold the variable **a** for the duration of the function call, and **a** is initialized by copying the value of **x**, which is verified by printing it out. Of course, you can change the value of **a** and show that it is changed. But when **f( )** is completed, the temporary space that was created for **a** disappears, and we see that the only connection that ever existed between **a** and **x** happened when the value of **x** was copied into **a**.

When you're inside **f( )**, **x** is the *outside object* (my terminology), and changing the local variable does not affect the outside object, naturally enough, since they are two separate locations in storage. But what if you *do* want to modify the outside object? This is where pointers come in handy. In a sense, a pointer is an alias for another variable. So if we pass a *pointer* into a function instead of an ordinary value, we are actually passing an alias to the outside object, enabling the function to modify that outside object, like this:

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;

void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
```

```
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

Now **f( )** takes a pointer as an argument, and dereferences the pointer during assignment, and this causes the outside object **x** to be modified. The output is:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

Notice that the value contained in **p** is the same as address of **x** – the pointer **p** does indeed point to **x**. If that isn't convincing enough, when **p** is dereferenced to assign the value 5, we see that the value of **x** is now changed to 5 as well.

Thus, passing a pointer into a function will allow that function to modify the outside object. You'll see plenty of other uses for pointers later, but this is arguably the most basic and possibly the most common use.

Introduction to C++ references

Pointers work roughly the same in C and in C++, but C++ adds an additional way to pass an address into a function. This is *pass-by-reference* and it exists in several other programming languages so it was not a C++ invention.

Your initial perception of references may be that they are unnecessary – that you could write all your programs without references. In general, this is true, with the exception of a few important places which you'll learn about later in the book. You'll also learn more about references later in the book, but the basic idea is the same as the above demonstration of pointer use: you can pass the address of an argument using a reference. The difference between references and pointers is that *calling* a function that takes references is cleaner, syntactically, than calling a function that takes pointers (and it is exactly this syntactic difference that makes references essential in certain situations). If **PassAddress.cpp** is modified to use references, you can see the difference in the function call in **main( )**:

```
//: C03:PassReference.cpp
```

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

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value,
        // is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```

In **f( )**'s argument list, instead of saying **int*** to pass a pointer, you say **int&** to pass a reference. Inside **f( )**, if you just say '**r**' (which would produce the address if **r** were a pointer) you get *the value in the variable that **r** references*. If you assign to **r**, you actually assign to the variable that **r** references. In fact, the only way to get the address that's held inside **r** is with the '**&**' operator.

In **main( )**, you can see the key effect of references in the syntax of the call to **f( )**, which is just **f(x)**. Even though this looks like an ordinary pass-by-value, the effect of the reference is that it actually takes the address and passes it in, rather than making a copy of the value. The output is:

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

So you can see that pass-by-reference allows a function to modify the outside object, just like passing a pointer does (you can also observe that the reference obscures the fact that an address is being passed – this will be examined later in the book). Thus, for this simple introduction you can assume that references are just a syntactically different way (sometimes

referred to as “syntactic sugar”) to accomplish the same thing that pointers do: allow functions to change outside objects.

Pointers and references as modifiers

So far, you’ve seen the basic data types **char**, **int**, **float** and **double**, along with the specifiers **signed**, **unsigned**, **short** and **long**, which can be used with the basic data types in almost any combination. Now we’ve added pointers and references which are orthogonal to the basic data types and specifiers, so the possible combinations have just tripled:

```
//: C03:AllDefinitions.cpp
// All possible combinations of basic data types,
// specifiers, pointers and references
#include <iostream>
using namespace std;

void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);

int main() {} ///:~
```

Pointers and references also work when passing objects into and out of functions; you’ll learn about this in a later chapter.

There’s one other type that works with pointers: **void**. If you state that a pointer is a **void***, it means that any type of address at all can be

assigned to that pointer (whereas if you have an **int***, you can only assign the address of an **int** variable to that pointer). For example:

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
  double d;
  // The address of ANY type can be
  // assigned to a void pointer:
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

Once you assign to a **void*** you lose any information about what type it is. This means that before you can use the pointer, you must cast it to the correct type:

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
  // Can't dereference a void pointer:
  // *vp = 3; // Compile-time error
  // Must cast back to int before dereferencing:
  *((int*)vp) = 3;
} ///:~
```

The cast **(int*)vp** takes the **void*** and tells the compiler to treat it as an **int***, and thus it can be successfully dereferenced. You might observe that this syntax is ugly, and it is, but it's worse than that – the **void*** introduces a hole in the language's type system. That is, it allows, or even promotes the treatment of one type as another type. In the above example, I treat an **int** as an **int** by casting **vp** to an **int***, but there's nothing that says I can't cast it to a **char*** or **double***, which would modify a different amount of storage that had been allocated for the **int**, possibly crashing your program. In general, **void** pointers should be avoided, and only used in rare special cases, the likes of which you won't be ready to consider until significantly later in the book.

You cannot have a **void** reference, for reasons that will be explained in a future chapter.

Scoping

Scoping rules tell you where a variable is valid, where it is created and where it gets destroyed (i.e., goes out of scope). The scope of a variable extends from the point where it is defined to the first closing brace that matches the closest opening brace before the variable was defined. To illustrate:

```
//: C03:Scope.cpp
// How variables are scoped
int main() {
  int scp1;
  // scp1 visible here
  {
    // scp1 still visible here
    //.....
    int scp2;
    // scp2 visible here
    //.....
    {
      // scp1 & scp2 still visible here
      //..
      int scp3;
      // scp1, scp2 & scp3 visible here
      // ...
    } // <-- scp3 destroyed here
    // scp3 not available here
    // scp1 & scp2 still visible here
    // ...
  } // <-- scp2 destroyed here
  // scp3 & scp2 not available here
  // scp1 still visible here
  //..
} // <-- scp1 destroyed here
///:~
```

The above example shows when variables are visible, and when they are unavailable (that is, when they *go out of scope*). A variable can only be used when inside its scope. Scopes can be nested, indicated by matched

pairs of braces inside other matched pairs of braces. Nesting means that you can access a variable in a scope that encloses the scope you are in. In the above example, the variable **scp1** is available inside all of the other scopes, while **scp3** is only available in the innermost scope.

Defining variables on the fly

As noted earlier in this chapter, there is a significant difference between C and C++ when defining variables. Both languages require that variables be defined before they are used, but C (and many other traditional procedural languages) forces you to define all the variables at the beginning of a scope, so that when the compiler creates a block it can allocate space for those variables.

While reading C code, a block of variable definitions is usually the first thing you see when entering a scope. Declaring all variables at the beginning of the block requires the programmer to write in a particular way because of the implementation details of the language. Most people don't know all the variables they are going to use before they write the code, so they must keep jumping back to the beginning of the block to insert new variables, which is awkward and causes errors. These variable definitions don't usually mean much to the reader, and actually tend to be confusing because they appear apart from the context in which they are used.

C++ (not C) allows you to define variables anywhere in a scope, so you can define a variable right before you use it. In addition, you can initialize the variable at the point you define it, which prevents a certain class of errors. Defining variables this way makes the code much easier to write and reduces the errors you get from being forced to jump back and forth within a scope. It makes the code easier to understand because you see a variable defined in the context of its use. This is especially important when you are defining and initializing a variable at the same time – you can see the meaning of the initialization value by the way the variable is used.

You can also define variables inside the control expressions of **for** loops and **while** loops, inside the conditional of an **if** statement, and inside the selector statement of a **switch**. Here's an example showing on-the-fly variable definitions:

```
//: C03:OnTheFly.cpp
// On-the-fly variable definitions
#include <iostream>
using namespace std;
```

```
int main() {
  //..
  { // Begin a new scope
    int q = 0; // C requires definitions here
    //..
    // Define at point of use:
    for(int i = 0; i < 100; i++) {
      q++; // q comes from a larger scope
      // Definition at the end of the scope:
      int p = 12;
    }
    int p = 1;  // A different p
  } // End scope containing q & outer p
  cout << "Type characters:" << endl;
  while(char c = cin.get() != 'q') {
    cout << c << " wasn't it" << endl;
    if(char x = c == 'a' || c == 'b')
      cout << "You typed a or b" << endl;
    else
      cout << "You typed " << x << endl;
  }
  cout << "Type A, B, or C" << endl;
  switch(int i = cin.get()) {
    case 'A': cout << "Snap" << endl; break;
    case 'B': cout << "Crackle" << endl; break;
    case 'C': cout << "Pop" << endl; break;
    default: cout << "Not A, B or C!" << endl;
  }
} ///:~
```

In the innermost scope, **p** is defined right before the scope ends, so it is really a useless gesture (but it shows you can define a variable anywhere). The **p** in the outer scope is in the same situation.

The definition of **i** in the control expression of the **for** loop is an example of being able to define a variable *exactly* at the point you need it (you can only do this in C++). The scope of **i** is the scope of the expression controlled by the **for** loop, so you can turn around and re-use **i** in the next **for** loop. This is a convenient and commonly-used idiom in C++; **i** is the classic name for a loop counter and you don't have to keep inventing new names.

Although the example also shows variables defined within **while**, **if** and **switch** statements, this kind of definition is much less common than those in **for** expressions, possibly because the syntax is so constrained. For example, you cannot have any parentheses. That is, you cannot say:

```
while((char c = cin.get()) != 'q')
```

The addition of the extra parentheses would seem like a very innocent and useful thing to do, and because you cannot use them, the results are not what you might like. The problem occurs because '**!=**' has a higher precedence than '**=**', so the **char c** ends up containing a **bool** converted to **char**. When that's printed, on many terminals you'll see a smiley-face character.

In general, you can consider the ability to define variables within **while**, **if** and **switch** statements as being there for completeness, but the only place you're likely to use this kind of variable definition is in a **for** loop (where you'll use it quite often).

Specifying storage allocation

When creating a variable, you have a number of options to specify the lifetime of the variable, how the storage is allocated for that variable, and how the variable is treated by the compiler.

Global variables

Global variables are defined outside all function bodies and are available to all parts of the program (even code in other files). Global variables are unaffected by scopes and are always available (i.e., the lifetime of a global variable lasts until the program ends). If the existence of a global variable in one file is declared using the **extern** keyword in another file, the data is available for use by the second file. Here's an example of the use of global variables:

```
//: C03:Global.cpp
//{L} Global2
// Demonstration of global variables
#include <iostream>
using namespace std;
```

```
int globe;
void func();
int main() {
  globe = 12;
  cout << globe << endl;
  func(); // Modifies globe
  cout << globe << endl;
} ///:~
```

Here's a file that accesses **globe** as an extern:

```
//: C03:Global2.cpp {O}
// Accessing external global variables
extern int globe;
// (The linker resolves the reference)
void func() {
  globe = 47;
} ///:~
```

Storage for the variable **globe** is created by the definition in **Global.cpp**, and that same variable is accessed by the code in **Global2.cpp**. Since the code in **Global2.cpp** is compiled separately from the code in **Global.cpp**, the compiler must be informed that the variable exists elsewhere by the declaration

```
extern int globe;
```

When you run the program, you'll see that the call to **func( )** does indeed affect the single global instance of **globe**.

In **Global.cpp**, you can see the special comment tag (which is my own design):

```
//{L} Global2
```

This says that to create the final program, the object file with the name **Global2** must be linked in (there is no extension because the extension names of object files differ from one system to the next). In **Global2.cpp**, the first line has another special comment tag **{O}** which says "don't try to create an executable out of this file, it's being compiled so that it can be linked into some other executable." The **ExtractCode.cpp** program at the end of this book reads these tags and creates the appropriate **makefile** so everything compiles properly (you'll learn about **makefile**s at the end of this chapter).

Local variables

Local variables occur within a scope; they are “local” to a function. They are often called *automatic* variables because they automatically come into being when the scope is entered, and automatically go away when the scope closes. The keyword **auto** makes this explicit, but local variables default to **auto** so it is never necessary to declare something as an **auto**.

Register variables

A register variable is a type of local variable. The **register** keyword tells the compiler “make accesses to this variable as fast as possible.” Increasing the access speed is implementation dependent but, as the name suggests, it is often done by placing the variable in a register. There is no guarantee that the variable will be placed in a register or even that the access speed will increase. It is a hint to the compiler.

There are restrictions to the use of **register** variables. You cannot take or compute the address of a **register** variable. A **register** variable can only be declared within a block (you cannot have global or **static register** variables). You can, however, use a **register** variable as a formal argument in a function (i.e., in the argument list).

Generally, you shouldn’t try to second-guess the compiler’s optimizer, since it will probably do a better job than you can. Thus, the **register** keyword is best avoided.

static

The **static** keyword has several distinct meanings. Normally, variables defined local to a function disappear at the end of the function scope. When you call the function again, storage for the variables is created anew and the values are re-initialized. If you want a value to be extant throughout the life of a program, you can define a function’s local variable to be **static** and give it an initial value. The initialization is only performed the first time the function is called, and the data retains its value between function calls. This way, a function can “remember” some piece of information between function calls.

You may wonder why a global variable isn’t used instead. The beauty of a **static** variable is that it is unavailable outside the scope of the function, so it can’t be inadvertently changed. This localizes errors.

Here’s an example of the use of **static** variables:

```
//: C03:Static.cpp
// Using a static variable in a function
#include <iostream>
using namespace std;

void func() {
  static int i = 0;
  cout << "i = " << ++i << endl;
}

int main() {
  for(int x = 0; x < 10; x++)
    func();
} ///:~
```

Each time **func**( ) is called in the for loop, it prints a different value. If the keyword **static** is not used, the value printed will always be '1'.

The second meaning of **static** is related to the first in the "unavailable outside a certain scope" sense. When **static** is applied to a function name or to a variable that is outside of all functions, it means "this name is unavailable outside of this file." The function name or variable is local to the file; we say it *has file scope*. As a demonstration, compiling and linking the following two files will cause a linker error:

```
//: C03:FileStatic.cpp
// File scope demonstration. Compiling and
// linking this file with FileStatic2.cpp
// will cause a linker error

// File scope means only available in this file:
static int fs;

int main() {
  fs = 1;
} ///:~
```

Even though the variable **fs** is claimed to exist as an **extern** in the following file, the linker won't find it because it has been declared **static** in **FileStatic.cpp**.

```
//: C03:FileStatic2.cpp {O}
// Trying to reference fs
extern int fs;
void func() {
```

```
  fs = 100;
} ///:~
```

The **static** specifier may also be used inside a **class**. This explanation will be delayed until you learn to create classes, later in the book.

extern

The **extern** keyword has already been briefly described and demonstrated. It tells the compiler that a variable or a function exists, even if the compiler hasn't yet seen it in the file currently being compiled. This variable or function may be defined in another file or further down in the current file. As an example of the latter:

```
//: C03:Forward.cpp
// Forward function & data declarations
#include <iostream>
using namespace std;

// This is not actually external, but the
// compiler must be told it exists somewhere:
extern int i;
extern void func();
int main() {
  i = 0;
  func();
}
int i; // The data definition
void func() {
  i++;
  cout << i;
} ///:~
```

When the compiler encounters the declaration '**extern int i**' it knows that the definition for **i** must exist somewhere as a global variable. When the compiler reaches the definition of **i**, no other declaration is visible so it knows it has found the same **i** declared earlier in the file. If you were to define **i** as **static**, you would be telling the compiler that **i** is defined globally (via the **extern**), but it also has file scope (via the **static**), so the compiler will generate an error.

Linkage

To understand the behavior of C and C++ programs, you need to know about *linkage*. In an executing program, an identifier is represented by storage in memory that holds a variable or a compiled function body. Linkage describes this storage it is seen by the linker. There are two types of linkage: *internal linkage* and *external linkage*.

Internal linkage means that storage is created to represent the identifier only for the file being compiled. Other files may use the same identifier name with internal linkage, or for a global variable, and no conflicts will be found by the linker – separate storage is created for each identifier. Internal linkage is specified by the keyword **static** in C and C++.

External linkage means that a single piece of storage is created to represent the identifier for all files being compiled. The storage is created once, and the linker must resolve all other references to that storage. Global variables and function names have external linkage. These are accessed from other files by declaring them with the keyword **extern**. Variables defined outside all functions (with the exception of **const** in C++) and function definitions default to external linkage. You can specifically force them to have internal linkage using the **static** keyword. You can explicitly state that an identifier has external linkage by defining it with the **extern** keyword. Defining a variable or function with **extern** is not necessary in C, but it is sometimes necessary for **const** in C++.

Automatic (local) variables exist only temporarily, on the stack, while a function is being called. The linker doesn't know about automatic variables, and they have *no linkage*.

Constants

In old (pre-Standard) C, if you wanted to make a constant, you had to use the preprocessor:

```
#define PI 3.14159
```

Everywhere you used **PI**, the value 3.14159 was substituted by the preprocessor (you can still use this method in C and C++).

When you use the preprocessor to create constants, you place control of those constants outside the scope of the compiler. No type checking is performed on the name **PI** and you can't take the address of **PI** (so you can't pass a pointer or a reference to **PI**). **PI** cannot be a variable of a

user-defined type. The meaning of **PI** lasts from the point it is defined to the end of the file; the preprocessor doesn't recognize scoping.

C++ introduces the concept of a named constant that is just like a variable, except its value cannot be changed. The modifier **const** tells the compiler that a name represents a constant. Any data type, built-in or user-defined, may be defined as **const**. If you define something as **const** and then attempt to modify it, the compiler will generate an error.

You must specify the type of a **const**, like this:

```
const int x = 10;
```

In Standard C and C++, you can use a named constant in an argument list, even if the argument it fills is a pointer or a reference (i.e., you can take the address of a **const**). A **const** has a scope, just like a regular variable, so you can "hide" a **const** inside a function and be sure that the name will not affect the rest of the program.

The **const** was taken from C++ and incorporated into Standard C, albeit quite differently. In C, the compiler treats a **const** just like a variable that has a special tag attached that says "don't change me." When you define a **const** in C, the compiler creates storage for it, so if you define more than one **const** with the same name in two different files (or put the definition in a header file), the linker will generate error messages about conflicts. The intended use of **const** in C is quite different from its intended use in C++ (in short, it's nicer in C++).

Constant values

In C++, a **const** must always have an initialization value (in C, this is not true). Constant values for built-in types are expressed as decimal, octal, hexadecimal, or floating-point numbers (sadly, binary numbers were not considered important), or as characters.

In the absence of any other clues, the compiler assumes a constant value is a decimal number. The numbers 47, 0 and 1101 are all treated as decimal numbers.

A constant value with a leading 0 is treated as an octal number (base 8). Base 8 numbers can only contain digits 0-7; the compiler flags other digits as an error. A legitimate octal number is 017 (15 in base 10).

A constant value with a leading 0x is treated as a hexadecimal number (base 16). Base 16 numbers contain the digits 0-9 and a-f or A-F. A legitimate hexadecimal number is 0x1fe (510 in base 10).

Floating point numbers can contain decimal points and exponential powers (represented by e, which means “10 to the power”). Both the decimal point and the **e** are optional. If you assign a constant to a floating-point variable, the compiler will take the constant value and convert it to a floating-point number (this process is one form of what’s called *implicit type conversion*). However, it is a good idea to use either a decimal point or an **e** to remind the reader you are using a floating-point number; some older compilers also need the hint.

Legitimate floating-point constant values are: 1e4, 1.0001, 47.0, 0.0 and -1.159e-77. You can add suffixes to force the type of floating-point number: **f** or **F** forces a **float**, **L** or **l** forces a **long double**, otherwise the number will be a **double**.

Character constants are characters surrounded by single quotes, as: ‘**A**’, ‘**0**’, ‘ ‘. Notice there is a big difference between the character ‘**0**’ (ASCII 96) and the value **0**. Special characters are represented with the “backslash escape”: ‘**\n**’ (newline), ‘**\t**’ (tab), ‘****’ (backslash), ‘**\r**’ (carriage return), ‘**\"**’ (double quotes), ‘**\'**’ (single quote), etc. You can also express char constants in octal: ‘**\17**’ or hexadecimal: ‘**\xff**’.

volatile

Whereas the qualifier **const** tells the compiler “this never changes” (which allows the compiler to perform extra optimizations) the qualifier **volatile** tells the compiler “you never know when this will change,” and prevents the compiler from performing any optimizations. Use this keyword when you read some value outside the control of your code, such as a register in a piece of communication hardware. A **volatile** variable is always read whenever its value is required, even if it was just read the line before.

A special case of some storage being “outside the control of your code” is in a multithreaded program. If you’re watching a particular flag that is modified by another thread or process, that flag should be **volatile** so the compiler doesn’t make the assumption that it can optimize away multiple reads of the flag.

Note that **volatile** may have no effect when a compiler is not optimizing, but may prevent critical bugs when you start optimizing the code (which is when the compiler will begin looking for reduntant reads).

The **const** and **volatile** keywords will be further illuminated in a later chapter.

Operators and their use

This section covers all the operators in C and C++.

All operators produce a value from their operands. This value is produced without modifying the operands, except with the assignment, increment and decrement operators. Modifying an operand is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value produced is available for your use just as in operators without side effects.

Assignment

Assignment is performed with the operator **=**. It means "take the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*). An rvalue is any constant, variable, or expression that can produce a value, but an lvalue must be a distinct, named variable (that is, there must be a physical space in which to store data). For instance, you can assign a constant value to a variable (**A = 4;**), but you cannot assign anything to constant value – it cannot be an lvalue (you can't say **4 = A;**).

Mathematical operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (**+**), subtraction (**-**), division (**/**), multiplication (*****) and modulus (**%**, this produces the remainder from integer division). Integer division truncates the result (it doesn't round). The modulus operator cannot be used with floating-point numbers.

C & C++ also use a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable **x** and assign **x** to the result, you say: **x += 4;**.

This example shows the use of the mathematical operators:

```
//: C03:Mathops.cpp
// Mathematical operators
#include <iostream>
using namespace std;
```

```
// A macro to display a string and a value.
#define PRINT(STR, VAR) \
  cout << STR " = " << VAR << endl

int main() {
  int i, j, k;
  float u,v,w;  // Applies to doubles, too
  cout << "enter an integer: ";
  cin >> j;
  cout << "enter another integer: ";
  cin >> k;
  PRINT("j",j);  PRINT("k",k);
  i = j + k; PRINT("j + k",i);
  i = j - k; PRINT("j - k",i);
  i = k / j; PRINT("k / j",i);
  i = k * j; PRINT("k * j",i);
  i = k % j; PRINT("k % j",i);
  // The following only works with integers:
  j %= k; PRINT("j %= k", j);
  cout << "Enter a floating-point number: ";
  cin >> v;
  cout << "Enter another floating-point number:";
  cin >> w;
  PRINT("v",v); PRINT("w",w);
  u = v + w; PRINT("v + w", u);
  u = v - w; PRINT("v - w", u);
  u = v * w; PRINT("v * w", u);
  u = v / w; PRINT("v / w", u);
  // The following works for ints, chars,
  // and doubles too:
  u += v; PRINT("u += v", u);
  u -= v; PRINT("u -= v", u);
  u *= v; PRINT("u *= v", u);
  u /= v; PRINT("u /= v", u);
} ///:~
```

The rvalues of all the assignments can, of course, be much more complex.

Introduction to preprocessor macros

Notice the use of the macro **PRINT( )** to save typing (and typing errors!). Preprocessor macros are traditionally named with all uppercase letters, so

they stand out – you'll learn later that macros can quickly become dangerous (and they can also be very useful).

The arguments in the parenthesized list following the macro name are substituted in all the code following the closing parenthesis. The preprocessor removes the name **PRINT** and substitutes the code wherever the macro is called, so the compiler cannot generate any error messages using the macro name, and it doesn't do any type checking on the arguments (the latter can be beneficial, as shown in the debugging macros at the end of the chapter).

Relational operators

Relational operators establish a relationship between the values of the operands. They produce a Boolean (specified with the **bool** keyword in C++) **true** if the relationship is true, and **false** if the relationship is false. The relational operators are: less than (**<**), greater than (**>**), less than or equal to (**<=**), greater than or equal to (**>=**), equivalent (**==**) and not equivalent (**!=**). They may be used with all built-in data types in C and C++. They may be given special definitions for user-defined data types in C++ (you'll learn about this in Chapter XX, on operator overloading).

Logical operators

The logical operators *and* (**&&**) and *or* (**||**) produce a **true** or **false** based on the logical relationship of its arguments. Remember that in C and C++, a statement is **true** if it has a non-zero value, and **false** if it has a value of zero. If you print a **bool**, you'll typically see a '**1**' for **true** and '**0**' for **false**.

This example uses the relational and logical operators:

```
//: C03:Boolean.cpp
// Relational and logical operators.
#include <iostream>
using namespace std;

int main() {
  int i,j;
  cout << "enter an integer: ";
  cin >> i;
  cout << "enter another integer: ";
  cin >> j;
```

```
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
      << ((i < 10) && (j < 10))  << endl;
} ///:~
```

You can replace the definition for **int** with **float** or **double** in the above program. Be aware, however, that the comparison of a floating-point number with the value of zero is very strict: a number that is the tiniest fraction different from another number is still "not equal." A floating-point number that is the tiniest bit above zero is still true.

Bitwise operators

The bitwise operators allow you to manipulate individual bits in a number (since floating point values use a special internal format, the bitwise operators only work with integral numbers). Bitwise operators perform boolean algebra on the corresponding bits in the arguments to produce the result.

The bitwise *and* operator (**&**) produces a one in the output bit if both input bits are one; otherwise it produces a zero. The bitwise *or* operator (**|**) produces a one in the output bit if either input bit is a one and only produces a zero if both input bits are zero. The bitwise *exclusive or*, or *xor* (**^**) produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise *not* (**~**, also called the *ones complement* operator) is a unary operator – it only takes one argument (all other bitwise operators are binary operators). Bitwise *not* produces the opposite of the input bit – a one if the input bit is zero, a zero if the input bit is one.

Bitwise operators can be combined with the **=** sign to unite the operation and assignment: **&=**, **|=** and **^=** are all legitimate operations (since **~** is a unary operator it cannot be combined with the **=** sign).

Shift operators

The shift operators also manipulate bits. The left-shift operator (**<<**) produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator. The right-shift operator (**>>**) produces the operand to the left of the operator shifted to the right by the number of bits specified after the operator. If the value after the shift operator is greater than the number of bits in the left-hand operand, the result is undefined. If the left-hand operand is unsigned, the right shift is a logical shift so the upper bits will be filled with zeros. If the left-hand operand is signed, the right shift may or may not be a logical shift (that is, the behavior is undefined).

Shifts can be combined with the equal sign (**<<=** and **>>=**). The lvalue is replaced by the lvalue shifted by the rvalue.

Here's an example that demonstrates the use of all the operators involving bits:

```
//: C03:Bitwise.cpp
// Demonstration of bit manipulation
#include <iostream>
using namespace std;

// Display a byte in binary:
void printBinary(const unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      cout << "1";
    else
      cout << "0";
}

// A macro to save typing:
#define PR(STR, EXPR) \
  cout << STR; printBinary(EXPR); cout << endl;

int main() {
  unsigned int getval;
  unsigned char a, b;
  cout << "Enter a number between 0 and 255: ";
  cin >> getval; a = getval;
  PR("a in binary: ", a);
  cout << "Enter a number between 0 and 255: ";
```

```
  cin >> getval; b = getval;
  PR("b in binary: ", b);
  PR("a | b = ", a | b);
  PR("a & b = ", a & b);
  PR("a ^ b = ", a ^ b);
  PR("~a = ", ~a);
  PR("~b = ", ~b);
  // An interesting bit pattern:
  unsigned char c = 0x5A;
  PR("c in binary: ", c);
  a |= c;
  PR("a |= c; a = ", a);
  b &= c;
  PR("b &= c; b = ", b);
  b ^= a;
  PR("b ^= a; b = ", b);
} ///:~
```

The **printBinary( )** function takes a single byte and displays it bit-by-bit. The expression **(1 << i)** produces a one in each successive bit position; in binary: 00000001, 00000010, etc. If this bit is bitwise *and*ed with **val** and the result is nonzero, it means there was a one in that position in **val**.

Once again, a preprocessor macro is used to save typing. It prints the string of your choice, then the binary representation of an expression, then a newline.

In **main( )**, the variables are **unsigned**. This is because, generally, you don't want signs when you are working with bytes. An **int** must be used instead of a **char** for **getval** because the "**cin >>**" statement will otherwise treat the first digit as a character. By assigning **getval** to **a** and **b**, the value is converted to a single byte (by truncating it).

The **<<** and **>>** provide bit-shifting behavior, but when they shift bits off the end of the number, those bits are lost (it's commonly said that they fall into the mythical *bit bucket*, a place where discarded bits end up, presumably so they can be reused…). When manipulating bits you can also perform *rotation*, which means that the bits that fall off one end are inserted back at the other end, as if they're being rotated around a loop. Even though most computer processors provide a machine-level rotate command (so you'll see it in the assembly language for that processor), there is no direct support for "rotate" in C or C++. Presumably the designers of C felt justified in leaving "rotate" off (aiming, as they said, for

a minimal language) because you can build your own rotate command. For example, here are functions to perform left and right rotations:

```
//: C03:Rotation.cpp {O}
// Perform left and right rotations

unsigned char rol(unsigned char val) {
  int highbit;
  if(val & 0x80) // 0x80 is the high bit only
    highbit = 1;
  else
    highbit = 0;
  // Left shift (bottom bit becomes 0):
  val <<= 1;
  // Rotate the high bit onto the bottom:
  val |= highbit;
  return val;
}

unsigned char ror(unsigned char val) {
  int lowbit;
  if(val & 1) // Check the low bit
    lowbit = 1;
  else
    lowbit = 0;
  val >>= 1; // Right shift by one position
  // Rotate the low bit onto the top:
  val |= (lowbit << 7);
  return val;
} ///:~
```

Try using these functions in **Bitwise.cpp**. Notice the definitions (or at least declarations) of **rol( )** and **ror( )** must be seen by the compiler in **Bitwise.cpp** before the functions are used.

The bitwise functions are generally extremely efficient to use because they translate directly into assembly language statements. Sometimes a single C or C++ statement will generate a single line of assembly code.

Unary operators

Bitwise *not* isn't the only operator that takes a single argument. Its companion, the *logical not* (**!**), will take a **true** value and produce a **false** value. The unary minus (**-**) and unary plus (**+**) are the same operators as

binary minus and plus – the compiler figures out which usage is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler can figure out:

```
x = a * -b;
```

but the reader might get confused, so it is safer to say:

```
x = a * (-b);
```

The unary minus produces the negative of the value. Unary plus provides symmetry with unary minus, although it doesn't actually do anything.

The increment and decrement operators (**++** and **--**) were introduced earlier in this chapter. These are the only operators other than those involving assignment that have side effects. These operators increase or decrease the variable by one unit, although "unit" can have different meanings according to the data type – this is especially true with pointers.

The last unary operators are the address-of (**&**), dereference (***** and **->**) and cast operators in C and C++, and **new** and **delete** in C++. Address-of and dereference are used with pointers, described in this chapter. Casting is described later in this chapter, and **new** and **delete** are described in Chapter XX.

The ternary operator

The ternary **if-else** is unusual because it has 3 operands. It is truly an operator because it produces a value, unlike the ordinary **if-else** statement. It consists of three expressions: if the first expression (followed by a **?**) evaluates to **true**, the expression following the **?** is evaluated and its result becomes the value produced by the operator. If the first expression is **false**, the third expression (following a **:**) is executed and its result becomes the value produced by the operator.

The conditional operator can be used for its side effects or for the value it produces. Here's a code fragment that demonstrates both:

```
A = --B ? B : (B = -99);
```

Here, the conditional produces the rvalue. **A** is assigned to the value of **B** if the result of decrementing **B** is nonzero. If **B** became zero, **A** and **B** are both assigned to -99. **B** is always decremented, but it is only assigned to -99 if the decrement causes **B** to become 0. A similar statement can be used without the "**A =**" just for its side effects:

```
--B ? B : (B = -99);
```

Here the second B is superfluous, since the value produced by the operator is unused. An expression is required between the **?** and **:**. In this case the expression could simply be a constant that might make the code run a bit faster.

The comma operator

The comma is not restricted to separating variable names in multiple definitions such as

```
int i, j, k;
```

Of course, it's also used in function argument lists. However, it can also be used as an operator to separate expressions – in this case it produces only the value of the last expression. All the rest of the expressions in the comma-separated list are only evaluated for their side effects. This code fragment increments a list of variables and uses the last one as the rvalue:

```
A = (B++,C++,D++,E++);
```

The parentheses are critical here. Without them, the statement will evaluate to:

```
(A = B++), C++, D++, E++;
```

In general, it's best to avoid using the comma as anything other than a separator, since people are not used to seeing it as an operator.

Common pitfalls when using operators

As illustrated above, one of the pitfalls when using operators is trying to get away without parentheses when you are even the least bit uncertain about how an expression will evaluate (consult your local C manual for the order of expression evaluation).

Another extremely common error looks like this:

```
//: C03:Pitfall.cpp
// Operator mistakes

int main() {
```

```
  int a = 1, b = 1;
  while(a = b) {
    // ....
  }
} ///:~
```

The statement **a = b** will always evaluate to true when **b** is non-zero. The variable **a** is assigned to the value of **b**, and the value of **b** is also produced by the operator **=**. Generally you want to use the equivalence operator **==** inside a conditional statement, not assignment. This one bites a lot of programmers (however, some compilers will point out the problem to you, which is very helpful).

A similar problem is using bitwise *and* and *or* instead of their logical counterparts. Bitwise *and* and *or* use one of the characters (**&** or **|**) while logical *and* and *or* use two (**&&** and **||**). Just as with **=** and **==**, it's easy to just type one character instead of two. A useful mnemonic device is to observe that "bits are smaller, so they don't need as many characters in their operators."

Casting operators

The word *cast* is used in the sense of "casting into a mold." The compiler will automatically change one type of data into another if it makes sense. For instance, if you assign an integral value to a floating-point variable, the compiler will secretly call a function (or more probably, insert code) to convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

To perform a cast, put the desired data type (including all modifiers) inside parentheses to the left of the value. This value can be a variable, a constant, the value produced by an expression or the return value of a function. Here's an example:

```
int b = 200;
a = (unsigned long int)b;
```

Casting is very powerful, but it can cause headaches because in some situations it forces the compiler to treat data as if it were (for instance) larger than it really is, so it will occupy more space in memory – this can trample over other data. This usually occurs when casting pointers, not when making simple casts like the one shown above.

C++ has an additional kind of casting syntax, which follows the function call syntax. This syntax puts the parentheses around the argument, like a function call, rather than around the data type:

```
float a = float(200);
```

This is equivalent to:

```
float a = (float)200;
```

Of course in the above case you wouldn't really need a cast; you could just say **200f** (and that's typically what the compiler will do for the above expression, in effect). Casts are generally used instead with variables, rather than constants.

sizeof – an operator by itself

The **sizeof( )** operator stands alone because it satisfies an unusual need. **sizeof( )** gives you information about the amount of memory allocated for data items. As described earlier in this chapter, **sizeof( )** tells you the number of bytes used by any particular variable. It can also give the size of a data type (with no variable name):

```
cout << "sizeof(double) = " << sizeof(double);
```

sizeof( ) can also give you the sizes of user-defined data types. This is used later in the book.

The **asm** keyword

This is an escape mechanism that allows you to write assembly code for your hardware within a C++ program. Often you're able to reference C++ variables within the assembly code, which means you can easily communicate with your C++ code and limit the assembly code to that necessary for efficiency tuning or to utilize special processor instructions. The exact syntax of the assembly language is compiler-dependent and can be discovered in your compiler's documentation.

Explicit operators

These are keywords for bitwise and logical operators. Non-U.S. programmers without keyboard characters like **&**, **|**, **^**, and so on, were forced to use C's horrible *trigraphs*, which were not only annoying to type, but obscure when reading. This is repaired in C++ with additional

keywords:

Keyword	Meaning
and	&& (logical *and*)
or	|| (logical *or*)
not	! (logical NOT)
not_eq	!= (logical not-equivalent)
bitand	& (bitwise *and*)
and_eq	&= (bitwise *and*-assignment)
bitor	| (bitwise *or*)
or_eq	|= (bitwise *or*-assignment)
xor	^ (bitwise exclusive-*or*)
xor_eq	^= (bitwise exclusive-*or*-assignment)
compl	~ (ones complement)

If your compiler complies with Standard C++, it will support these keywords.

Composite type creation

The fundamental data types and their variations are essential, but rather primitive. C and C++ provide tools that allow you to compose more sophisticated data types from the fundamental data types. As you'll see, the most important of these is **struct**, which is the foundation for **class** in C++. However, the simplest way to create more sophisticated types is simply to alias a name to another name via **typedef**.

Aliasing names with **typedef**

This keyword promises more than it delivers: **typedef** suggests "type definition" when "alias" would probably have been a more accurate description, since that's all it really does. The syntax is:

typedef existing-type-description alias-name

People often use **typedef** when data types get slightly complicated, just to prevent extra keystrokes. Here is a commonly-used **typedef**:

```
typedef unsigned long ulong;
```

Now if you say **ulong** the compiler knows that you mean **unsigned long**. You might think that this could as easily be accomplished using preprocessor substitution, but there are key situations where the compiler must be aware that you're treating a name as if it were a type, so **typedef** is essential.

You can argue that it's more explicit and therefore more readable to avoid **typedef**s for primitive types, and indeed programs rapidly become difficult to read when many **typedef**s are used. However, **typedef**s become especially important in C when used with **struct**.

Combining variables with **struct**

A **struct** is a way to collect a group of variables into a structure. Once you create a **struct**, then you can make many instances of this "new" type of variable you've invented. For example:

```
//: C03:SimpleStruct.cpp

struct Structure1 {
  char c;
  int i;
  float f;
  double d;
};

int main() {
  struct Structure1 s1, s2;
  s1.c = 'a'; // Select an element using a '.'
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

The **struct** declaration must end with a semicolon. In **main( )**, two instances of **Structure1** are created: **s1** and **s2**. Each of these have their

own separate versions of **c**, **i**, **f** and **d**. So **s1** and **s2** represent clumps of completely independent variables. To select one of the elements within **s1** or **s2**, you use a '**.**', syntax you've seen in the previous chapter when using C++ **class** objects – since **class**es evolved from **struct**s, this is where that syntax arose.

One thing you'll notice is the awkwardness of the use of **Structure1**. You can't just say **Structure1** when you're defining variables, you must say **struct Structure1**. This is where **typedef** becomes especially handy:

```
//: C03:SimpleStruct2.cpp
// Using typedef with struct
typedef struct {
  char c;
  int i;
  float f;
  double d;
} Structure2;

int main() {
  Structure2 s1, s2;
  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14;
  s1.d = 0.00093;
  s2.c = 'a';
  s2.i = 1;
  s2.f = 3.14;
  s2.d = 0.00093;
} ///:~
```

By using **typedef** in this way, you can pretend that **Structure2** is a built-in type, like **int** or **float**, when you define **s1** and **s2** (but notice it only has data – characteristics – and does not include behavior, which is what we get with real objects in C++). You'll notice that the **struct** name has been left off at the beginning, because the goal is to create the **typedef**. However, there are times when you might need to refer to the **struct** during its definition. In those cases, you can actually repeat the name of the **struct** as the **struct** name and as the **typedef**:

```
//: C03:SelfReferential.cpp
// Allowing a struct to refer to itself

typedef struct SelfReferential {
```

```
  int i;
  SelfReferential* sr; // Head spinning yet?
} SelfReferential;

int main() {
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
} ///:~
```

If you look at this for awhile, you'll see that **sr1** and **sr2** point to each other, as well as each holding a piece of data.

Actually, the **struct** name does not have to be the same as the **typedef** name, but it is usually done this way as it tends to keep things simpler.

Pointers and **struct**s

In the above examples, all the **struct**s are manipulated as objects. However, like any piece of storage you can take the address of a **struct** object (as seen in **SelfReferential.cpp** above). To select the elements of a particular **struct** object, you use a '**.**', as seen above. However, if you have a pointer to a **struct** object, you must select an element of that object using a different operator: the '**->**'. Here's an example:

```
//: C03:SimpleStruct3.cpp
// Using pointers to structs
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;

int main() {
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Point to a different struct object
```

```
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
} ///:~
```

In **main( )**, the **struct** pointer **sp** is initially pointing to **s1**, and the members of **s1** are initialized by selecting them with the '**->**' (and you use this same operator in order to read those members). But then **sp** is pointed to **s2**, and those variables are initialized the same way. So you can see that another benefit of pointers is that they can be dynamically redirected to point to different objects; this provides more flexibility in your programming, as you shall learn.

For now, that's all you need to know about **struct**s, but you'll become much more comfortable with **struct**s (and especially their more potent successors, **class**es) as the book progresses.

Clarifying programs with **enum**

An enumerated data type is a way of attaching names to numbers, thereby giving more meaning to anyone reading the code. The **enum** keyword (from C) automatically enumerates any list of words you give it by assigning them values of 0, 1, 2, etc. You can declare **enum** variables (which are always **int**s). The declaration of an **enum** looks similar to a **struct** declaration.

An enumerated data type is very useful when you want to keep track of some sort of feature:

```
//: C03:Enum.cpp
// Keeping track of shapes.

enum ShapeType {
  circle,
  square,
  rectangle
};  // Must end with a semicolon like a struct

int main() {
  ShapeType shape = circle;
  // Activities here....
  // Now do something based on what the shape is:
  switch(shape) {
```

```
    case circle:  /* circle stuff */ break;
    case square:  /* square stuff */ break;
    case rectangle:  /* rectangle stuff */ break;
   }
} ///:~
```

shape is a variable of the **ShapeType** enumerated data type, and its value is compared with the value in the enumeration. Since **shape** is really just an **int**, however, it can be any value an **int** can hold (including a negative number). You can also compare an **int** variable with a value in the enumeration.

If you don't like the way the compiler assigns values, you can do it yourself, like this:

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

If you give values to some names and not to others, the compiler will use the next integral value. For example,

```
enum snap { crackle = 25, pop };
```

The compiler gives **pop** the value 26.

You can see how much more readable the code is when you use enumerated data types. However, to some degree this is still an attempt (in C) to accomplish the things that we can do with a **class** in C++, so you'll see **enum** used less in C++.

Type checking for enumerations

C's enumerations are fairly primitive, simply associating integral values with names, but they provide no type checking. In C++, as you may have come to expect by now, the concept of type is fundamental, and this is true with enumerations. When you create a named enumeration, you effectively create a new type just as you do with a class: The name of your enumeration becomes a reserved word for the duration of that translation unit.

In addition, there's stricter type checking for enumerations in C++ than in C. You'll notice this in particular if you have an instance of an enumeration **color** called **a**. In C you can say **a++** but in C++ you can't. This is because incrementing an enumeration is performing two type conversions, one of them legal in C++ and one of them illegal. First, the value of the enumeration is implicitly cast from a **color** to an **int**, then the value is

incremented, then the **int** is cast back into a **color**. In C++ this isn't allowed, because **color** is a distinct type and not equivalent to an **int**. This makes sense, because how do you know the increment of **blue** will even be in the list of colors? If you want to increment a **color**, then it should be a class (with an increment operation) and not an **enum**, because the class can be made to be much safer. Any time you write code that assumes an implicit conversion to an **enum** type, the compiler will flag this inherently dangerous activity.

Unions (described next) have similar additional type checking in C++.

Saving memory with **union**

Sometimes a program will handle different types of data using the same variable. In this situation, you have two choices: you can create a **struct** containing all the possible different types you might need to store, or you can use a **union**. A **union** piles all the data into a single space; it figures out the amount of space necessary for the largest item you've put in the **union**, and makes that the size of the **union**. Use a **union** to save memory.

Anytime you place a value in a **union**, the value always starts in the same place at the beginning of the **union**, but only uses as much space as is necessary. Thus, you create a "super-variable," capable of holding any of the **union** variables. All the addresses of the **union** variables are the same (in a class or **struct**, the addresses are different).

Here's a simple use of a **union**. Try removing various elements and see what effect it has on the size of the **union**. Notice that it makes no sense to declare more than one instance of a single data type in a **union** (unless you're just doing it to use a different name).

```
//: C03:Union.cpp
// The size and simple use of a union
#include <iostream>
using namespace std;

union packed { // Declaration similar to a class
  char i;
  short j;
  int k;
  long l;
  float f;
  double d;
```

```
  // The union will be the size of a
  // double, since that's the largest element
};  // Semicolon ends a union, like a struct

int main() {
  cout << "sizeof(packed) = "
    << sizeof(packed) << endl;
  packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

The compiler performs the proper assignment according to the union member you select.

Once you perform an assignment, the compiler doesn't care what you do with the union. In the above example, you could assign a floating-point value to **x**:

```
x.f = 2.222;
```

and then send it to the output as if it were an **int**:

```
cout << x.i;
```

This would produce garbage.

Arrays

Arrays are a kind of composite type because they allow you to clump a lot of variables together, one right after the other, under a single identifier name. If you say:

```
int a[10];
```

You create storage for 10 **int** variables stacked on top of each other, but without unique identifier names for each variable. Instead, they are all lumped under the name **a**.

To access one of these *array elements*, you use the same square-bracket syntax that you use to define an array:

```
a[5] = 47;
```

However, you must remember that even though the *size* of **a** is 10, you select array elements starting at zero (this is sometimes called *zero indexing*), so you can only select the array elements 0-9, like this:

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

Array access is extremely fast. However, if you index past the end of the array, there is no safety net – you'll step on other variables. The other drawback is the fact that you must define the size of the array at compile time; if you want to change the size at runtime you can't do it with the above syntax (C does have a way to create an array dynamically, but it's significantly messier). The C++ **vector**, introduced in the previous chapter, provides an array-like object that automatically resizes itself, so it is usually a much better solution if your array size cannot be known at compile time.

You can make an array of any type, even of **struct**s:

```
//: C03:StructArray.cpp
// An array of struct

typedef struct {
  int i, j, k;
} ThreeDpoint;

int main() {
  ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

Notice how the **struct** identifier **i** is independent of the **for** loop's **i**.

To see that each element of an array is contiguous with the next, you can print out the addresses like this:

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = " << sizeof(int) <<endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
      << (long)&a[i] << endl;
} ///:~
```

When you run this program, you'll see that each element is one **int** size away from the previous one. That is, they are stacked one on top of the other.

Pointers and arrays

The identifier of an array is unlike the identifiers for ordinary variables. For one thing, an array identifier is not an lvalue – you cannot assign to it. It's really just a hook into the square-bracket syntax, and when you give the name of an array, without square brackets, what you get is the starting address of the array:

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

When you run this program you'll see that the two addresses (which will be printed in hexadecimal, since there is no cast to **long**) are the same.

So one way to look at the array identifier is as a read-only pointer to the beginning of an array. And although we can't change the array identifier to point somewhere else, we *can* create another pointer and use that to move around in the array. In fact, the square-bracket syntax works with regular pointers, as well:

```
//: C03:PointersAndBrackets.cpp

int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

The fact that naming an array produces its starting address turns out to be quite important when you want to pass an array to a function. If you declare an array as a function argument, what you're really declaring is a pointer. So in the following example, **func1( )** and **func2( )** effectively have the same argument lists:

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;

void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}

void func2(int* a, int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}

void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
    cout << name << "[" << i << "] = "
      << a[i] << endl;
}

int main() {
  int a[5], b[5];
  // Probably garbage values:
  print(a, "a", 5);
  print(b, "b", 5);
  // Initialize the arrays:
  func1(a, 5);
```

```
  func1(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
  // Notice the arrays are always modified:
  func2(a, 5);
  func2(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
} ///:~
```

Even though **func1( )** and **func2( )** declare their arguments differently, the usage is the same inside the function. There are some other issues that this example reveals: arrays cannot be passed by value[22], that is, you never automatically get a local copy of the array that you pass into a function. Thus, when you modify an array, you're always modifying the outside object. This can be a bit confusing at first, if you're expecting the pass-by-value provided with ordinary arguments.

You'll notice that **print( )** uses the square-bracket syntax for array arguments. Even though the pointer syntax and the square-bracket syntax are effectively the same when passing arrays as arguments, the square-bracket syntax makes it clearer to the reader that you mean for this argument to be an array.

Also note that the **size** argument is passed in each case. Just passing the address of an array isn't enough information; you must always be able to know how big the array is inside your function, so you don't run off the end of that array.

Arrays can be of any type, including arrays of pointers. In fact, when you want to pass command-line arguments into your program, C & C++ have a special argument list for **main( )** which looks like this:

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

The first argument is the number of elements in the array which is the second argument. The second argument is always an array of **char***,

---

[22] Unless you take the very strict approach that "all argument passing in C/C++ is by value, and the 'value' of an array is what is produced by the array identifier: it's address." This can be seen as true from the assembly-language standpoint, but I don't think it helps when trying to work with higher-level concepts. The addition of references in C++ makes the "all passing is by value" argument more confusing, to the point where I feel it's more helpful to think in terms of "passing by value" vs. "passing addresses."

because the arguments are passed from the command line as character arrays (and remember, an array can only be passed as a pointer). Each whitespace-delimited cluster of characters on the command line is turned into a separate array argument. The following program prints out all its command-line arguments by stepping through the array:

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
      << argv[i] << endl;
} ///:~
```

You'll notice that **argv[0]** is the path and name of the program itself. This allows the program to discover information about itself. It also adds one more to the array of program arguments, so a common error when fetching command-line arguments is to grab **argv[0]** when you want **argv[1]**.

You are not forced to use **argc** and **argv** as identifiers in **main( )**; those identifiers are only conventions (but it will confuse people if you don't use them). Also, there are alternate ways to declare **argv**:

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

All three forms are equivalent, but I find the form used in this book to be the most intuitive when reading the code, since it says, directly, "this is an array of character pointers."

All you get from the command-line is character arrays; if you want to treat an argument as some other type, you are responsible for converting it, inside your program. To facilitate the conversion to numbers, there are some helper functions in the Standard C library, declared in **<cstdlib>**. The simplest ones to use are **atoi( )**, **atol( )** and **atof( )**, to convert an ASCII character array to an **int**, **long** and **double** floating-point value, respectively. Here's an example using **atoi( )** (the other two functions are called the same way):

```
//: C03:ArgsToInts.cpp
// Converting command-line arguments to ints
#include <iostream>
```

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

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

In this program, you can put any number of arguments on the command line. You'll notice that the **for** loop starts at the value **1** to skip over the program name at **argv[0]**. Also, if you put a floating-point number containing a decimal point on the command line, **atoi( )** only takes the digits up to the decimal point. If you put non-numbers on the command line, these come back from **atoi( )** as zero

Pointer arithmetic

If all you could do with a pointer that points at an array is treat it as if it were an alias for that array, pointers into arrays wouldn't be very interesting. However, pointers are more flexible than this, since they can be moved around (and remember, the array identifier cannot be moved).

Pointer arithmetic refers to the application of some of the arithmetic operators to pointers. The reason pointer arithmetic is a separate subject from ordinary arithmetic is that pointers must conform to special constraints in order to make them behave properly. For example, a common operator to use with pointers is **++**, which "adds one to the pointer." What this actually means is that the pointer is changed to move to "the next value," whatever that means. Here's an example:

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
  cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
```

```
  cout << "dp = " << (long)dp << endl;
} ///:~
```

For one run on my machine, the output is:

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

What's interesting here is that even though the operation **++** appears to be the same operation for both the **int*** and the **double***, you can see that the pointer has been changed only 4 bytes for the **int*** but 8 bytes for the **double***. Not coincidently, these are the sizes of **int** and **double** on my machine. And that's the trick of pointer arithmetic: the compiler figures out the right amount to change the pointer so that it's pointing to the next element in the array (pointer arithmetic is only meaningful within arrays). This even works with arrays of **struct**s:

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;

typedef struct {
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;

int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
    << sizeof(Primitives) << endl;
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

The output for one run on my machine was:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

So you can see the compiler also does the right thing for pointers to **struct**s (and **class**es and **union**s).

Pointer arithmetic also works with the operators **--**, **+** and **-**, but the latter two operators are limited: you cannot add or subtract two pointers. Instead, you must add or subtract an integral value. Here's an example demonstrating the use of pointer arithmetic:

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

#define P(EXP) \
  cout << #EXP << ": " << EXP << endl;

int main() {
  int a[10];
  for(int i = 0; i < 10; i++)
    a[i] = i; // Give it index values
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
} ///:~
```

It begins with another macro, but this one uses a preprocessor feature called *stringizing* (implemented with the '**#**' sign before an expression) which takes any expression and turns it into a character array. This is quite convenient, since it allows the expression to be printed, followed by a colon, followed by the value of the expression. In **main( )** you can see the useful shorthand that is produced.

Although pre- and postfix versions of **++** and **--** are valid with pointers, only the prefix versions are used in this example because they are applied before the pointers are dereferenced in the above expressions, so they allow us to see the effects of the operations. Note that only integral values

are being added and subtracted; if two pointers were combined this way the compiler would not allow it.

Here is the output of the above program:

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

In all cases, the pointer arithmetic results in the pointer being adjusted to point to the "right place," based on the size of the elements being pointed to.

If pointer arithmetic seems a bit overwhelming at first, don't worry. Most of the time you'll only need to create arrays and index into them with **[ ]**, and the most sophisticated pointer arithmetic you'll usually need is **++** and **--**. Pointer arithmetic is generally reserved for more clever and complex programs, and many of the containers in the Standard C++ library hide most of these clever details so you don't have to worry about them.

Debugging hints

In an ideal environment, you have an excellent debugger available that easily makes the behavior of your program transparent so you can quickly discover errors. However, most debuggers have blind spots, and these will require you to embed code snippets in your program to help you understand what's going on. In addition, you may be developing in an environment (such as an embedded system, which is where I spent my formative years) that has no debugger available, and perhaps very limited feedback (i.e. a one-line LED display). In these cases you become creative in the ways you discover and display information about the execution of your program. This section suggests some techniques for doing this.

Debugging flags

If you hard-wire your debugging code into a program, you can run into problems. You start to get too much information, which makes the bugs difficult to isolate. When you think you've found the bug you start tearing out debugging code, only to find you need to put it back in again. You can

solve these problems with two types of flags: preprocessor debugging flags and runtime debugging flags.

Preprocessor debugging flags

By using the preprocessor to **#define** one or more debugging flags (preferably in a header file), you can test a flag using an **#ifdef** statement and conditionally include debugging code. When you think your debugging is finished, you can simply **#undef** the flag(s) and the code will automatically be removed (and you'll reduce the size and runtime overhead of your executable file).

It is best to decide on names for debugging flags before you begin building your project so the names will be consistent. Preprocessor flags are traditionally distinguished from variables by writing them in all upper case. A common flag name is simply DEBUG (but be careful you don't use NDEBUG, which is reserved in C). The sequence of statements might be:

```
#define DEBUG // Probably in a header file
//...
#ifdef DEBUG // Check to see if flag is defined
/* debugging code here */
#endif // DEBUG
```

Most C and C++ implementations will also let you **#define** and **#undef** flags from the compiler command line, so you can re-compile code and insert debugging information with a single command (preferably via the makefile, a tool that will be described next). Check your local documentation for details.

Runtime debugging flags

In some situations it is more convenient to turn debugging flags on and off during program execution, especially by setting them when the program starts up using the command line. Large programs are tedious to recompile just to insert debugging code.

To turn the debugger on and off dynamically, create **bool** flags:

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// Debug flags aren't necessarily global:
bool debug = false;
```

```
int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {
      // Debugging code here
      cout << "Debugger is now on!" << endl;
    } else {
      cout << "Debugger is now off." << endl;
    }
    cout << "Turn debugger [on/off/quit]: ";
    string reply;
    cin >> reply;
    if(reply == "on") debug = true; // Turn it on
    if(reply == "off") debug = false; // Off
    if(reply == "quit") break; // Out of 'while'
  }
} ///:~
```

This program continues to allow you to turn the debugging flag on and off until you type “quit” to tell it you want to exit. Notice that it requires that full words be typed in, not just letters (you can shorten it to letter if you wish). Also, a command-line argument can optionally be used to turn debugging on a startup – this argument can appear anyplace in the command line, since the startup code in **main( )** looks at all the arguments. The testing is quite simple because of the expression:

```
string(argv[i])
```

This takes the **argv[i]** character array and creates a **string**, which then can be easily compared to the right-hand side of the **==**. The above program searches for the entire string **--debug=on**. You can also look for **--debug=** and then see what’s after that, to provide more options. The chapter on the **string** class later in the book will show you how to do that.

Although a debugging flag is one of the relatively few areas where it makes a lot of sense to use a global variable, there’s nothing that says it must be that way. Notice that the variable is in lower case letters to remind the reader it isn’t a preprocessor flag.

Turning variables and expressions into strings

When writing debugging code, it is tedious to write print expressions consisting of a character array containing the variable name, followed by the variable. Fortunately, Standard C includes the *stringize* operator '**#**', which was used earlier in this chapter. When you put a **#** before an argument in a preprocessor macro, the preprocessor turns that argument into a character array. This, combined with the fact that character arrays with no intervening punctuation are concatenated into a single character array, allows you to make a very convenient macro for printing the values of variables during debugging:

```
#define PR(x) cout << #x " = " << x << "\n";
```

If you print the variable **a** by calling the macro **PR(a)**, it will have the same effect as the code:

```
cout << "a = " << a << "\n";
```

This same process works with entire expressions. The following program uses a macro to create a shorthand that prints the stringized expression and then evaluates the expression and prints the result:

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;

#define P(A) cout << #A << ": " << (A) << endl;

int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

You can see how a technique like this can quickly become indispensible, especially if you have no debugger (or must use multiple development environments). You can also insert an **#ifdef** to cause **P(A)** to be defined as "nothing" when you want to strip out debugging.

The C **assert( )** macro

In the standard header file **<cassert>** you'll find **assert( )**, which is a convenient debugging macro. When you use **assert( )**, you give it an argument that is an expression you are "asserting to be true." The preprocessor generates code that will test the assertion. If the assertion isn't true, the program will stop after issuing an error message telling you what the assertion was and that it failed. Here's a trivial example:

```
//: C03:Assert.cpp
// Use of the assert() debugging macro
#include <cassert>  // Contains the macro
using namespace std;

int main() {
  int i = 100;
  assert(i != 100);
} ///:~
```

The macro originated in Standard C, so it's also available in the header file **assert.h**.

When you are finished debugging, you can remove the code generated by the macro by placing the line:

```
#define NDEBUG
```

in the program before the inclusion of **<cassert>**, or by defining NDEBUG on the compiler command line. NDEBUG is a flag used in **<cassert>** to change the way code is generated by the macros.

Later in this book, you'll see some more sophisticated alternatives to **assert( )**.

Make: an essential tool for separate compilation

When using *separate compilation* (breaking code into a number of translation units), you need some way to automatically compile each file and to tell the linker to build all the pieces – along with the appropriate

libraries and startup code – into an executable file. Most compilers allow you to do this with a single command-line statement. For the Gnu C++ compiler, for example, you might say

```
g++ SourceFile1.cpp SourceFile2.cpp
```

The problem with this approach is that the compiler will first compile each individual file, regardless of whether that file *needs* to be rebuilt or not. With many files in a project, it can become prohibitive to recompile everything if you've only changed a single file.

The solution to this problem, developed on Unix but available everywhere in some form, is a program called **make**. The **make** utility manages all the individual files in a project by following the instructions in a text file called a **makefile**. When you edit some of the files in a project and type **make**, the **make** program follows the guidelines in the **makefile** to compare the dates on the source code files to the dates on the corresponding target files, and if a source code file date is more recent than its target file, **make** invokes the compiler on the source code file. **make** only recompiles the source code files that were changed, and any other source-code files that are affected by the modified files. By using **make**, you don't have to re-compile all the files in your project every time you make a change, nor do you have to check to see that everything was built properly. The **makefile** contains all the commands to put your project together. Learning to use **make** will save you a lot of time and frustration. You'll also discover that **make** is the typical way that you install new software on a Linux/Unix machine (although those **makefile**s tend to be far more complicated than the ones presented in this book, and you'll often automatically generate a **makefile** for your particular machine as part of the installation process).

Because **make** is available in some form for virtually all C++ compilers (and even if it isn't, you can use freely-available **make**s with any compiler), it will be the tool used throughout this book. However, compiler vendors have also created their own project building tools. These tools ask you which files are in your project, and determine all the relationships themselves. These tools use something similar to a **makefile**, generally called a *project file*, but the programming environment maintains this file so you don't have to worry about it. The configuration and use of project files varies from one development environment to another, so you must find the appropriate documentation on how to use them (although project file tools provided by compiler vendors are usually so simple to use that you can learn them by playing around – my favorite form of education).

The **makefile**s used within this book should work even if you are also using a specific vendor's project-building tool.

Make activities

When you type **make** (or whatever the name of your "make" program happens to be), the **make** program looks in the current directory for a file named **makefile**, which you've created if it's your project. This file lists dependencies between source code files. **make** looks at the dates on files. If a dependent file has an older date than a file it depends on, **make** executes the *rule* given after the dependency.

All comments in **makefile**s start with a **#** and continue to the end of the line.

As a simple example, the **makefile** for a program called "hello" might contain:

```
# A comment
hello.exe: hello.cpp
	mycompiler hello.cpp
```

This says that **hello.exe** (the target) depends on **hello.cpp**. When **hello.cpp** has a newer date than **hello.exe**, **make** executes the "rule" **mycompiler hello.cpp**. There may be multiple dependencies and multiple rules. Many **make** programs require that all the rules begin with a tab. Other than that, whitespace is generally ignored so you can format for readability.

The rules are not restricted to being calls to the compiler; you can call any program you'd like from within **make**. By creating groups of interdependent dependency-rule sets, you can modify your source code files, type **make** and be certain that all the affected files will be rebuilt correctly.

Macros

A **makefile** may contain *macros* (note that these are completely different from C/C++ preprocessor macros). Macros allow convenient string replacement. The **makefile**s in this book use a macro to invoke the C++ compiler. For example,

```
CPP = mycompiler
hello.exe: hello.cpp
	$(CPP) hello.cpp
```

The **=** is used to identify **CPP** as a macro, and the **$** and parentheses expand the macro. In this case, the expansion means that the macro call **$(**CPP) will be replaced with the string **mycompiler**. With the above macro, if you want to change to a different compiler called **cpp**, you just change the macro to:

```
CPP = cpp
```

You can also add compiler flags, etc., to the macro, or use separate macros to add compiler flags.

Suffix Rules

It becomes tedious to tell **make** how to invoke the compiler for every single **cpp** file in your project, when you know it's the same basic process each time. Since **make** is designed to be a time-saver, it also has a way to abbreviate actions, as long as they depend on file name suffixes. These abbreviations are called *suffix rules*. A suffix rule is the way to teach **make** how to convert a file with one type of extension (**.cpp**, for example) into a file with another type of extension (**.obj** or **.exe**). Once you teach **make** the rules for producing one kind of file from another, all you have to do is tell **make** which files depend on which other files. When **make** finds a file with a date earlier than the file it depends on, it uses the rule to create a new file.

The suffix rule tells **make** that it doesn't need explicit rules to build everything, but instead it can figure out how to build things based on their file extension. In this case it says: "to build a file that ends in **exe** from one that ends in **cpp**, invoke the following command." Here's what it looks like for the above example:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
	$(CPP) $<
```

The **.SUFFIXES** directive tells **make** that it should watch out for any of the following file-name extensions because they have special meaning. Next you see the suffix rule **.cpp.exe** which says: "here's how to convert any file with an extension of **cpp** to one with an extension of **exe**" (when the **cpp** file is more recent than the **exe** file). As before, the **$(CPP)** macro is used, but then you see something new: **$<**. Because this begins with a '**$**' it's a macro, but this is one of **make**'s special built-in macros. The **$<** can only be used in suffix rules, and it means "whatever

prerequisite triggered the rule" (sometimes called the *dependent*), which in this case translates to: "the **cpp** file that needs to be compiled."

Once the suffix rules have been set up, you can simply say, for example, "**make Union.exe**," and the suffix rule will kick in, even though there's no mention of "Union" anywhere in the **makefile**.

Default targets

After the macros and suffix rules, **make** looks for the first "target" in a file, and builds that, unless you specify differently. So for the following **makefile**:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
	$(CPP) $<
target1.exe:
target2.exe:
```

If you just type '**make**', **target1.exe** will be built (using the default suffix rule) because that's the first target that **make** encounters. To build **target2.exe** you'd have to explicitly say '**make target2.exe**'. This becomes tedious, so you normally create a default "dummy" target which depends on all the rest of the targets, like this:

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
	$(CPP) $<
all: target1.exe target2.exe
```

Here, '**all**' does not exist, and there's no file called '**all**', so every time you type **make**, the program sees '**all**' as the first target in the list (and thus the default target), then it sees that '**all**' does not exist so it had better make it by checking all the dependencies. So it looks at **target1.exe** and (using the suffix rule) sees whether (1) **target1.exe** exists and (2) whether **target1.cpp** is more recent than **target1.exe**, and if so runs the suffix rule (if you provide an explicit rule for a particular target, that rule is used instead). Then it moves on to the next file in the default target list. Thus, by creating a default target list (typically called '**all**' by convention, but you can call it anything) you can cause every executable in your project to be made simply by typing '**make**'. In addition, you can have other non-default target lists that do other things – for example, you

could set it up so that typing '**make debug**' rebuilds all your files with debugging wired in.

Makefiles in this book

Using the program **ExtractCode.cpp** which is shown in Chapter XX, all the code listings in this book are automatically extracted from the ASCII text version of this book and placed in subdirectories according to their chapters. In addition, **ExtractCode.cpp** creates several **makefile**s in each subdirectory (with different names) so that you can simply move into that subdirectory and type **make -f mycompiler.makefile** (substituting the name of your compiler for '**mycompiler**', the '**-f**' flag says "use what follows as the **makefile**"). Finally, **ExtractCode.cpp** creates a "master" **makefile** in the root directory where the book's files have been expanded, and this **makefile** descends into each subdirectory and calls **make** with the appropriate **makefile**. This way you can compile all the code in the book by invoking a single **make** command, and the process will stop whenever your compiler is unable to handle a particular file (note that a Standard C++ conforming compiler should be able to compile all the files in this book). Because implementations of **make** vary from system to system, only the most basic, common features are used in the generated **makefile**s.

An example **makefile**

As mentioned, the code-extraction tool **ExtractCode.cpp** automatically generates **makefile**s for each chapter. Because of this, the **makefile**s for each chapter will not be placed in the book (all the makefiles are packaged with the source code, which is freely available at **http://www.BruceEckel.com**). However, it's useful to see an example of a **makefile**. What follows is a very shortened version of the one that was automatically generated for this chapter by the book's extraction tool. You'll find more than one **makefile** in each subdirectory (they have different names – you invoke a specific one with '**make -f**'). This one is for Gnu C++:

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
    $(CPP) $(CPPFLAGS) -c $<
.c.o :
```

```
    $(CPP) $(CPPFLAGS) -c $<

all: \
    Return \
    Declare \
    Ifthen \
    Guess \
    Guess2
# Rest of the files for this chapter not shown

Return: Return.o
    $(CPP) $(OFLAG)Return Return.o

Declare: Declare.o
    $(CPP) $(OFLAG)Declare Declare.o

Ifthen: Ifthen.o
    $(CPP) $(OFLAG)Ifthen Ifthen.o

Guess: Guess.o
    $(CPP) $(OFLAG)Guess Guess.o

Guess2: Guess2.o
    $(CPP) $(OFLAG)Guess2 Guess2.o

Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

The macro CPP is set to the name of the compiler. To use a different compiler, you can either edit the **makefile** or change the value of the macro on the command line, like this:

```
make CPP=cpp
```

Note, however, that **ExtractCode.cpp** has an automatic scheme to automatically build **makefile**s for additional compilers.

The second macro **OFLAG** is the flag that's used to indicate the name of the output file. Although many compilers automatically assume the output file has the same base name as the input file, others don't (such as Linux/Unix compilers, which default to creating a file called **a.out**).

You can see there are two suffix rules here, one for **cpp** files and one for **.c** files (in case any C source code needs to be compiled). The default target is **all**, and each line for this target is "continued" by using the backslash, up until **Guess2**, which is the last one in the list and thus has no backslash. There are many more files in this chapter, but only these are shown here for the sake of brevity.

The suffix rules take care of creating object files (with a **.o** extension) from **cpp** files, but in general you need to explicitly state rules for creating the executable, because normally an executable is created by linking many different object files and **make** cannot guess what those are. Also, in this case (Linux/Unix) there is no standard extension for executables so a suffix rule won't work for these simple situation. Thus, you see all the rules for building the final executables explicitly stated.

This **makefile** takes the absolute safest route of using as few **make** features as possible – it only uses the basic **make** concepts of targets and dependencies, as well as macros. This way it is virtually assured of working with as many **make** programs as possible. It tends to produce a larger **makefile**, but that's not so bad since it's automatically generated by **ExtractCode.cpp**.

There are lots of other **make** features that this book will not use, as well as newer and cleverer versions and variations of **make** with advanced shortcuts that can save a lot of time. Your local documentation may describe the further features of your particular **make**, and you can learn more about **make** from *Managing Projects with Make* by Oram & Talbott (O'Reilly, 1993). Also, if your compiler vendor does not supply a **make** or they use a non-standard **make**, you can find Gnu make for virtually any platform in existence by searching the Internet for Gnu archives (of which there are many).

Summary

This chapter was a fairly intense tour through all the fundamental features of C++ syntax, most of which are inherited from and in common with C (and result in C++'s vaunted backwards compatibility with C). Although some C++ features were introduced here, this tour is primarily intended for people who are conversant in programming, and simply need to be given an introduction to the syntax basics of C and C++. If you're already a C programmer, you may have even seen one or two things about C here that were unfamiliar, aside from the C++ features that were most likely new to you. However, if this chapter has still seemed a bit overwhelming,

you may want to consider going through the CD-ROM course *Thinking in C: Foundations for C++ & Java* (which contains lectures, excercises and guided solutions) available at http://www.MindView.net.

Exercises

1. Create a header file (with an extension of '**.h**'). In this file, declare a group of functions by varying the argument lists and return values from among the following: **void**, **char**, **int**, and **float**. Now create a **.cpp** file which includes your header file and creates definitions for all these functions. Each definition should simply print out the function name, argument list and return type so you know it's been called. Create a second **.cpp** file which includes your header file and defines **int main( )**, containing calls to all your functions. Compile and run your program.
2. Write a program that uses two nested **for** loops and the modulus operator (**%**) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1).
3. Write a program that uses a **while** loop to read words from standard input (**cin**) into a **string**. This is an "infinite" **while** loop, which you break out of (and exit the program) using a **break** statement. For each word that is read, evaluate it by first using a sequence of **if** statements to "map" an integral value to the word, and then use a **switch** statement that uses that integral value as its selector (this sequence of events is not meant to be good programming style; it's just supposed to give you exercise with control flow). Inside each **case**, print something meaningful. You must decide what the "interesting" words are and what the meaning is. You must also decide what word will signal the end of the program. Test the program by redirecting a file into the program's standard input (if you want to save typing, this file can be your program's source file).
4. Modify **Menu.cpp** to use **switch** statements instead of **if** statements.
5. Write a program that evaluates the two expressions in the section labeled "precedence."
6. Modify **YourPets2.cpp** so that it uses various different data types (**char**, **int**, **float**, **double** and their variants). Run the

program and create a map of the resulting memory layout. If you have access to more than one kind of machine or operating system or compiler, try this experiment with as many variations as you can manage.

7. Create two functions, one that takes a **string*** and one that takes a **string&**. Each of these functions should modify the outside **string** object in their own unique way. In **main( )**, create and initialize a **string** object, print it, then pass it to each of the two functions, printing the results.
8. Compile and run **Static.cpp**. Remove the **static** keyword from the code, compile and run it again and explain what happens.
9. Try to compile and link **FileStatic.cpp** with **FileStatic2.cpp**. What does the resulting error message mean?
10. Modify **Boolean.cpp** so that it works with **double** values instead of **int**s.
11. Modify **Boolean.cpp** and **Bitwise.cpp** so they use the explicit operators (if your compiler is conformant to the C++ Standard it will support these).
12. Modify **Bitwise.cpp** to use the functions from **Rotation.cpp**. Make sure you display the results in such a way that it's clear what's happening during rotations.
13. Modify **Ifthen.cpp** to use the ternary **if-else** operator (**?:**).
14. Create a **struct** that holds two **string** objects and one **int**. Use a **typedef** for the **struct** name. Create an instance of the **struct**, initialize all three values in your instance, and print them out. Take the address of your instance and assign it to a pointer to your **struct** type. Change the three values in your instance and print them out, all using the pointer.
15. Create a program that uses an enumeration of colors. Create a variable of this **enum** type and print out all the numbers that correspond with the color names, using a **for** loop.
16. Experiment with **Union.cpp** by removing various **union** elements to see the effects on the size of the resulting **union**. Try assigning to one element (thus one type) of the **union** and printing out a via a different element (thus a different type) to see what happens.

17. Create a program that defines two **int** arrays, one right after the other. Index off the end of the first array into the second, and make an assignment. Print out the second array to see the changes cause by this. Now try defining a **char** variable between the first array definition and the second, and repeat the experiment. You may want to create an array printing function to simplify your coding.
18. Modify **ArrayAddresses.cpp** to work with the data types **char**, **long int**, **float** and **double**.
19. Apply the technique in **ArrayAddresses.cpp** to print out the size of the **struct** and the addresses of the array elements in **StructArray.cpp**.
20. Create an array of **string** objects and assign a string to each element. Print out the array using a **for** loop.
21. Create two new programs starting from **ArgsToInts.cpp** so they use **atol( )** and **atof( )**, respectively.
22. Modify **PointerIncrement2.cpp** so it uses a **union** instead of a **struct**.
23. Modify **PointerArithmetic.cpp** to work with **long** and **long double**.
24. Define a **float** variable. Take its address, cast that address to an **unsigned char**, and assign it to an **unsigned char** pointer. Using this pointer and **[ ]**, index into the **float** variable and use the **printBinary( )** function defined in this chapter to print out a map of the **float** (go from 0 to **sizeof(float)**). Change the value of the **float** and see if you can figure out what's going on (the **float** contains encoded data).
25. Create a makefile that not only compiles **YourPets1.cpp** and **YourPets2.cpp** (for your particular compiler) but also executes both programs as part of the default target behavior. Make sure you use suffix rules.
26. Modify **StringizingExpressions.cpp** so that **P(A)** is conditionally **#ifdef**ed to allow the debugging code to be automatically stripped out by setting a command-line flag. You will need to consult your compiler's documentation to see how to define and undefine preprocessor values on the compiler command line.
27. Create a **makefile** for the previous exercise that allows you to type **make** for a production build of the program, and

make debug for a build of the program including debugging information.

4: Data abstraction

> C++ is a productivity enhancement tool. Why else would you make the effort (and it is an effort, regardless of how easy we attempt to make the transition)

to switch from some language that you already know and are productive with to a new language where you're going to be *less* productive for a while, until you get the hang of it? It's because you've become convinced that you're going to get big gains by using this new tool.

Productivity, in computer programming terms, means that fewer people can make much more complex and impressive programs in less time. There are certainly other issues when it comes to choosing a language, like efficiency (does the nature of the language cause slowdown and code bloat?), safety (does the language help you ensure that your program will always do what you plan, and handle errors gracefully?), and maintenance (does the language help you create code that is easy to understand, modify and extend?). These are certainly important factors that will be examined in this book.

But raw productivity means a program that formerly took three of you a week to write now takes one of you a day or two. This touches several levels of economics. You're happy because you get the rush of power that comes from building something, your client (or boss) is happy because products are produced faster and with fewer people, and the customers are happy because they get products more cheaply. The only way to get massive increases in productivity is to leverage off other people's code. That is, to use libraries.

A library is simply a bunch of code that someone else has written and packaged together. Often, the most minimal package is a file with an

extension like **lib** and one or more header files to tell your compiler what's in the library. The linker knows how to search through the library file and extract the appropriate compiled code. But that's only one way to deliver a library. On platforms that span many architectures, like Linux/Unix, often the only sensible way to deliver a library is with source code, so it can be reconfigured and recompiled on the new target.

Thus, libraries are probably the most important way to improve productivity, and one of the primary design goals of C++ is to make library use easier. This implies that there's something hard about using libraries in C. Understanding this factor will give you a first insight into the design of C++, and thus insight into how to use it.

A tiny C-like library

A library usually starts out as a collection of functions, but if you have used third-party C libraries you know there's usually more to it than that because there's more to life than behavior, actions and functions. There are also characteristics (blue, pounds, texture, luminance), which are represented by data. And when you start to deal with a set of characteristics in C, it is very convenient to clump them together into a **struct**, especially if you want to represent more than one similar thing in your problem space. Then you can make a variable of this **struct** for each thing.

Thus, most C libraries have a set of **struct**s and a set of functions that act on those **struct**s. As an example of what such a system looks like, consider a programming tool that acts like an array, but whose size can be established at runtime, when it is created. I'll call it a **CStash**. Although it's written in C++, it has the style of what you'd write in C:

```
//: C04:CLib.h
// Header file for a C-like library
// An array-like entity created at runtime

typedef struct CStashTag {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
} CStash;
```

```
void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

A tag name like **CStashTag** is generally used for a **struct** in case you need to reference the **struct** inside itself. For example, when creating a *linked list* (each element in your list contains a pointer to the next element), you need a pointer to the next **struct** variable, so you need a way to identify the type of that pointer within the **struct** body. Also, you'll almost universally see the **typedef** as shown above for every **struct** in a C library. This is done so you can treat the **struct** as if it were a new type and define variables of that **struct** like this:

```
CStash A, B, C;
```

The **storage** pointer is an **unsigned char***. An **unsigned char** is the smallest piece of storage a C compiler supports, although on some machines it can be the same size as the largest. It's implementation dependent, but is often one byte long. You might think that because the **CStash** is designed to hold any type of variable, a **void*** would be more appropriate here. However, the purpose is not to treat this storage as a block of some unknown type, but rather as a block of contiguous bytes.

The source code for the implementation file (which you may not get if you buy a library commercially – you might get only a compiled **obj** or **lib** or **dll**, etc.) looks like this:

```
//: C04:CLib.cpp {O}
// Implementation of example C-like library
// Declare structure and functions:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
```

```
  s->storage = 0;
  s->next = 0;
}

int add(CStash* s, const void* element) {
  if(s->next >= s->quantity) //Enough space left?
    inflate(s, increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1); // Index number
}

void* fetch(CStash* s, int index) {
  // Check index boundaries:
  assert(0 <= index && index < s->next);
  // Produce pointer to desired element:
  return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next;  // Elements in CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Copy old to new
  delete [](s->storage); // Old storage
  s->storage = b; // Point to new memory
  s->quantity = newQuantity;
}

void cleanup(CStash* s) {
```

```
  if(s->storage != 0) {
   cout << "freeing storage" << endl;
   delete []s->storage;
  }
} ///:~
```

initialize( ) performs the necessary setup for **struct CStash** by setting the internal variables to appropriate values. Initially, the **storage** pointer is set to zero, and the **size** indicator is also zero – no initial storage is allocated.

The **add( )** function inserts an element into the **CStash** at the next available location. First, it checks to see if there is any available space left. If not, it expands the storage using the **inflate( )** function, described later.

Because the compiler doesn't know the specific type of the variable being stored (all the function gets is a **void***), you can't just do an assignment, which would certainly be the convenient thing. Instead, you must copy the variable byte-by-byte. The most straightforward way to perform the copying is with array indexing. Typically, there are already data bytes in **storage**, and this is indicated by the value of **next**. To start with the right byte offset, **next** is multiplied by the size of each element (in bytes) to produce **startBytes**. Then the argument **element** is cast to an **unsigned char*** so that it can be addressed byte-by-byte and copied into the available **storage** space. **next** is incremented so that it indicates the next available piece of storage, and the "index number" where the value was stored so that value can be retrieved using this index number with **fetch( )**.

fetch( ) checks to see that the index isn't out of bounds and then returns the address of the desired variable, calculated using the **index** argument. Since **index** indicates the number of elements to offset into the **Cstash**, it must be multiplied by the number of bytes occupied by each piece to produce the numerical offset in bytes. When this offset is used to index into **storage** using array indexing, you don't get the address, but instead the byte at the address. To produce the address, you must use the address-of operator **&**.

count( ) may look a bit strange at first to a seasoned C programmer. It seems like a lot of trouble to go through to do something that would probably be a lot easier to do by hand. If you have a **struct CStash** called **intStash**, for example, it would seem much more straightforward to find out how many elements it has by saying **intStash.next** instead of making a function call (which has overhead) like **count(&intStash)**. However, if

you wanted to change the internal representation of **CStash** and thus the way the count was calculated, the function call interface allows the necessary flexibility. But alas, most programmers won't bother to find out about your "better" design for the library. They'll look at the **struct** and grab the **next** value directly, and possibly even change **next** without your permission. If only there were some way for the library designer to have better control over things like this! (Yes, that's foreshadowing.)

Dynamic storage allocation

You never know the maximum amount of storage you might need for a **CStash**, so the memory pointed to by **storage** is allocated from the *heap*. The heap is a big block of memory used for allocating smaller pieces at runtime. You use the heap when you don't know the size of the memory you'll need while you're writing a program. That is, only at runtime will you find out that you need space to hold 200 **Airplane** variables instead of 20. In Standard C, dynamic-memory allocation functions are part of and include **malloc( )**, **calloc( )**, **realloc( )**, and **free( )**. Instead of library calls, however, C++ has a more sophisticated (albeit simpler to use) approach to dynamic memory which is integrated into the language via the keywords **new** and **delete**.

The **inflate( )** function uses **new** to get a bigger chunk of space for the **CStash**. In this situation, we will only expand memory and not shrink it, and the **assert( )** will guarantee that a negative number is not passed to **inflate( )** as the **increase** value. The new number of elements that can be held (after **inflate( )** completes) is calculated as **newQuantity**, and this is multiplied by the number of bytes per element to produce **newBytes**, which will be the number of bytes in the allocation. So that we know how many bytes to copy over from the old location, **oldBytes** is calculated using the old **quantity**.

The actual storage allocation occurs in the *new-expression*, which is the expression involving the **new** keyword:

```
new unsigned char[newBytes];
```

The general form of the new-expression is:

new Type;

where **Type** describes the type of variable you want allocated on the heap. In this case, we want an array of **unsigned char** that is **newBytes** long, so that is what appears as the **Type**. You can also allocate something as simple as an **int** by saying:

```
new int;
```

and although this is rarely done, you can see that the form is consistent.

A new-expression returns a *pointer* to an object of the exact type that you asked for. So if you say **new Type**, you get back a pointer to a **Type**. If you say **new int**, you get back a pointer to an **int**. If you want a **new unsigned char** array, you get back a pointer to the first element of that array. The compiler will ensure that you assign the return value of the new-expression to a pointer of the correct type.

Of course, any time you request memory it's possible for the request to fail, if there is no more memory. As you will learn, C++ has mechanisms that come into play if the memory-allocation operation is unsuccessful.

Once the new storage is allocated, the data in the old storage must be copied to the new storage – this is again accomplished with array indexing, copying one byte at a time in a loop. After the data is copied, the old storage must be released so that it can be used by other parts of the program if they need new storage. The **delete** keyword is the complement of **new**, and must be applied to release any storage that is allocated with **new** (if you forget to use **delete**, that storage remains unavailable, and if this so-called *memory leak* happens enough, you'll run out of memory). In addition, there's a special syntax when you're deleting an array. It's as if you must remind the compiler that this pointer is not just pointing to one object, but to an array of objects: you put a set of empty square brackets in front of the pointer to be deleted:

```
delete []myArray;
```

Once the old storage has been deleted, the pointer to the new storage can be assigned to the **storage** pointer, the quantity is adjusted, and **inflate( )** has completed its job.

Note that the heap manager is fairly primitive. It gives you chunks of memory and takes them back when you **delete** them. There's no inherent facility for *heap compaction*, which compresses the heap to provide bigger free chunks. If a program allocates and frees heap storage for a while, you can end up with a *fragmented* heap that has lots of memory free, but without any pieces that are big enough to allocate the size you're looking for at the moment. A heap compactor complicates a program because it moves memory chunks around, so your pointers won't retain their proper values. Some operating environments have heap compaction built in, but they require you to use special memory *handles* (which can be temporarily converted to pointers, after locking the memory so the heap compactor

can't move it) instead of pointers. You can also build your own heap-compaction scheme, but this is not a task to be undertaken lightly.

When you create a variable on the stack at compile-time, the storage for that variable is automatically created and freed by the compiler. The compiler knows exactly how much storage is needed, and it knows the lifetime of the variables because of scoping. With dynamic memory allocation, however, the compiler doesn't know how much storage you're going to need, *and* it doesn't know the lifetime of that storage. That is, the storage doesn't get cleaned up automatically. Therefore, you're responsible for releasing the storage using **delete**, which tells the heap manager that storage can be used by the next call to **new**. The logical place for this to happen in the library is in the **cleanup( )** function because that is where all the closing-up housekeeping is done.

To test the library, two **CStash**es are created. The first holds **int**s and the second holds arrays of 80 **char**s:

```
//: C04:CLibTest.cpp
//{L} CLib
// Test the C-like library
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Define variables at the beginning
  // of the block, as in C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
  const int bufsize = 80;
  // Now remember to initialize the variables:
  initialize(&intStash, sizeof(int));
  for(i = 0; i < 100; i++)
    add(&intStash, &i);
  for(i = 0; i < count(&intStash); i++)
    cout << "fetch(&intStash, " << i << ") = "
         << *(int*)fetch(&intStash, i)
```

```
        << endl;
  // Holds 80-character strings:
  initialize(&stringStash, sizeof(char)*bufsize);
  in.open("CLibTest.cpp");
  assert(in);
  while(getline(in, line))
    add(&stringStash, line.c_str());
  while((cp = (char*)fetch(&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
        << cp << endl;
  cleanup(&intStash);
  cleanup(&stringStash);
} ///:~
```

Following the form required by C, all the variables are created at the beginning of the scope of **main( )**. Of course, you must remember to initialize the **CStash** variables later in the block, by calling **initialize( )**. One of the problems with C libraries is that you must carefully convey to the user the importance of the initialization and cleanup functions. If these functions aren't called, there will be a lot of trouble. Unfortunately, the user doesn't always wonder if initialization and cleanup are mandatory. They know what *they* want to accomplish, and they're not as concerned about you jumping up and down saying, "Hey, wait, you have to do *this* first!" Some users have even been known to initialize the elements of a structure themselves. There's certainly no mechanism in C to prevent it (more foreshadowing).

The **intStash** is filled up with integers, and the **stringStash** is filled with character arrays. These character arrays are produced by opening the source code file, **CLibTest.cpp**, and reading the lines from it into a **string** called **line**, and then producing a pointer to the character representation of **line** using the member function **c_str( )**.

After each **Stash** is loaded, it is displayed. The **intStash** is printed using a **for** loop, which uses **count( )** to establish its limit. The **stringStash** is printed with a **while**, which breaks out when **fetch( )** returns zero to indicate it is out of bounds.

Bad guesses

There is one more important issue you should understand before we look at the general problems in creating a C library. Note that the **CLib.h** header file *must* be included in any file that refers to **CStash** because the compiler can't even guess at what that structure looks like. However, it

can guess at what a function looks like – this sounds like a feature but it turns out to be a major C pitfall.

Although you should always declare functions by including a header file, function declarations aren't essential in C. It's possible in C (but *not* in C++) to call a function that you haven't declared. A good compiler will warn you that you probably ought to declare a function first, but it isn't enforced by the C language standard. This is a dangerous practice, because the C compiler can assume that a function that you call with an **int** argument has an argument list containing **int**, even if it may actually contain a **float**. This can produce bugs that are very difficult to find, as you will see.

Each separate C file is a *translation unit*. That is, the compiler is run separately on each translation unit, and when it is running it is aware of only that unit. Thus, any information you provide by including header files is quite important because it provides the compiler's understanding of the rest of your program. Declarations in header files are particularly important, because everywhere the header is included, the compiler will know exactly what to do. If, for example, you have a declaration in a header file that says **void func(float)**, the compiler knows that if you call that function with an integer argument, it should convert the **int** to a **float** as it passes the argument (this is called *promotion*). Without the declaration, the C compiler would simply assume that a function **func(int)** existed, it wouldn't do the promotion, and the wrong data would quietly be passed into **func( )**.

For each translation unit, the compiler creates an object file, with an extension of **o** or **obj** or something similar. These object files, along with the necessary start-up code, must be collected by the linker into the executable program. During linking, all the external references must be resolved. For example, in **CLibTest.cpp**, functions like **initialize( )** and **fetch( )** are declared (that is, the compiler is told what they look like) and used, but not defined. They are defined elsewhere, in **CLib.cpp**. Thus, the calls in **CLib.cpp** are external references. The linker must, when it puts all the object files together, take the unresolved external references and find the addresses they actually refer to. Those addresses are put into the executable program to replace the external references.

It's important to realize that in C, the external references that the linker searches for are simply function names, generally with an underscore in front of them. So all the linker has to do is match up the function name where it is called and the function body in the object file, and it's done. If you accidentally made a call that the compiler interpreted as **func(int)**

and there's a function body for **func(float)** in some other object file, the linker will see **_func** in one place and **_func** in another, and it will think everything's OK. The **func( )** at the calling location will push an **int** onto the stack, and the **func( )** function body will expect a **float** to be on the stack. If the function only reads the value and doesn't write to it, it won't blow up the stack. In fact, the **float** value it reads off the stack might even make some kind of sense. That's worse because it's harder to find the bug.

What's wrong?

We are remarkably adaptable, even in situations where perhaps we *shouldn't* adapt. The style of the **CStash** library has been a staple for C programmers, but if you look at it for a while, you might notice that it's rather . . . awkward. When you use it, you have to pass the address of the structure to every single function in the library. When reading the code, the mechanism of the library gets mixed with the meaning of the function calls, which is confusing when you're trying to understand what's going on.

One of the biggest obstacles, however, to using libraries in C is the problem of *name clashes*. C has a single name space for functions; that is, when the linker looks for a function name, it looks in a single master list. In addition, when the compiler is working on a translation unit, it can only work with a single function with a given name.

Now suppose you decide to buy two libraries from two different vendors, and each library has a structure that must be initialized and cleaned up. Both vendors decided that **initialize( )** and **cleanup( )** are good names. If you include both their header files in a single translation unit, what does the C compiler do? Fortunately, C gives you an error, telling you there's a type mismatch in the two different argument lists of the declared functions. But even if you don't include them in the same translation unit, the linker will still have problems. A good linker will detect that there's a name clash, but some linkers take the first function name they find, by searching through the list of object files in the order you give them in the link list. (Indeed, this can be thought of as a feature because it allows you to replace a library function with your own version.)

In either event, you can't use two C libraries that contain a function with the identical name. To solve this problem, C library vendors will often prepend a sequence of unique characters to the beginning of all their function names. So **initialize( )** and **cleanup( )** might become

CStash_initialize( ) and **CStash_cleanup( )**. This is a logical thing to do because it "decorates" the name of the **struct** the function works on with the name of the function.

Now it's time to take the first step toward creating classes in C++. Variable names inside a **struct** do not clash with global variable names. So why not take advantage of this for function names, when those functions operate on a particular **struct**? That is, why not make functions members of **struct**s?

The basic object

Step one is exactly that. C++ functions can be placed inside **struct**s as "member functions." Here's what it looks like after converting the C version of **CStash** to the C++ **Stash**:

```
//: C04:CppLib.h
// C-like library converted to C++

struct Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
   // Dynamically allocated array of bytes:
  unsigned char* storage;
  // Functions!
  void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```

First, notice there is no **typedef**. Instead of requiring you to create a **typedef**, the C++ compiler turns the name of the structure into a new type name for the program (just as **int**, **char**, **float** and **double** are type names).

All the data members are exactly the same as before, but now the functions are inside the body of the **struct**. In addition, notice that the first argument from the C version of the library has been removed. In C++, instead of forcing you to pass the address of the structure as the

first argument to all the functions that operate on that structure, the compiler secretly does this for you. Now the only arguments for the functions are concerned with what the function *does*, not the mechanism of the function's operation.

It's important to realize that the function code is effectively the same as it was with the C version of the library. The number of arguments are the same (even though you don't see the structure address being passed in, it's still there), and there's only one function body for each function. That is, just because you say

```
Stash A, B, C;
```

doesn't mean you get a different **add( )** function for each variable.

So the code that's generated is almost identical to what you would have written for the C version of the library. Interestingly enough, this includes the "name decoration" you probably would have done to produce **Stash_initialize( )**, **Stash_cleanup( )**, and so on. When the function name is inside the **struct**, the compiler effectively does the same thing. Therefore, **initialize( )** inside the structure **Stash** will not collide with a function named **initialize( )** inside any other structure, or even a global function named **initialize( )**. Most of the time you don't have to worry about the function name decoration – you use the undecorated name. But sometimes you do need to be able to specify that this **initialize( )** belongs to the **struct Stash**, and not to any other **struct**. In particular, when you're defining the function you need to fully specify which one it is. To accomplish this full specification, C++ has an operator (**::**) called the *scope resolution operator* (named so because names can now be in different scopes: at global scope, or within the scope of a **struct**). For example, if you want to specify **initialize( )** which belongs to **Stash**, you say **Stash::initialize(int size)**. You can see how the scope resolution operator is used in the function definitions:

```
//: C04:CppLib.cpp {O}
// C library converted to C++
// Declare structure and functions:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantity of elements to add
// when increasing storage:
const int increment = 100;
```

```
void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index && index < next);
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete []storage; // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
```

```
}

void Stash::cleanup() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
} ///:~
```

There are several other things that are different between C and C++. First, the declarations in the header files are *required* by the compiler. In C++ you cannot call a function without declaring it first. The compiler will issue an error message otherwise. This is an important way to ensure that function calls are consistent between the point where they are called and the point where they are defined. By forcing you to declare the function before you call it, the C++ compiler virtually ensures you will perform this declaration by including the header file. If you also include the same header file in the place where the functions are defined, then the compiler checks to make sure that the declaration in the header and the function definition match up. This means that the header file becomes a validated repository for function declarations and ensures that functions are used consistently throughout all translation units in the project.

Of course, global functions can still be declared by hand every place where they are defined and used. (This is so tedious that it becomes very unlikely.) However, structures must always be declared before they are defined or used, and the most convenient place to put a structure definition is in a header file, except for those you intentionally hide in a file.

You can see that all the member functions look almost the same as when they were C functions, except for the scope resolution and the fact that the first argument from the C version of the library is no longer explicit. It's still there, of course, because the function has to be able to work on a particular **struct** variable. But notice, inside the member function, that the member selection is also gone! Thus, instead of saying **s->size = sz;** you say **size = sz;** and eliminate the tedious **s->**, which didn't really add anything to the meaning of what you were doing anyway. The C++ compiler is apparently doing this for you. Indeed, it is taking the "secret" first argument (the address of the structure that we were previously passing in by hand) and applying the member selector whenever you refer to one of the data members of a **struct**. This means that whenever you are inside the member function of another **struct**, you can refer to any member (including another member function) by simply giving its name.

The compiler will search through the local structure's names before looking for a global version of that name. You'll find that this feature means that not only is your code easier to write, it's a lot easier to read.

But what if, for some reason, you *want* to be able to get your hands on the address of the structure? In the C version of the library it was easy because each function's first argument was a **CStash*** called **s**. In C++, things are even more consistent. There's a special keyword, called **this**, which produces the address of the **struct**. It's the equivalent of the '**s**' in the C version of the library. So we can revert to the C style of things by saying

```
this->size = Size;
```

The code generated by the compiler is exactly the same, so you don't need to use **this** in such a fashion – occasionally, you'll see code where people explicitly use **this->** everywhere but it doesn't add anything to the meaning of the code and often indicates an inexperienced programmer. Usually, you don't use **this** very often, but when you need it, it's there (some of the examples later in the book will use **this**).

There's one last item to mention. In C, you could assign a **void*** to any other pointer like this:

```
int i = 10;
void* vp = &i; // OK in both C and C++
int* ip = vp; // Only acceptable in C
```

and there was no complaint from the compiler. But in C++, this statement is not allowed. Why? Because C is not so particular about type information, so it allows you to assign a pointer with an unspecified type to a pointer with a specified type. Not so with C++. Type is critical in C++, and the compiler stamps its foot when there are any violations of type information. This has always been important, but it is especially important in C++ because you have member functions in **struct**s. If you could pass pointers to **struct**s around with impunity in C++, then you could end up calling a member function for a **struct** that doesn't even logically exist for that **struct**! A real recipe for disaster. Therefore, while C++ allows the assignment of any type of pointer to a **void*** (this was the original intent of **void***, which is required to be large enough to hold a pointer to any type), it will *not* allow you to assign a **void** pointer to any other type of pointer. A cast is always required, to tell the reader and the compiler that you really do want to treat it as the destination type.

This brings up an interesting issue. One of the important goals for C++ is to compile as much existing C code as possible to allow for an easy

transition to the new language. However, this doesn't mean any code that C allows will automatically be allowed in C++. There are a number of things the C compiler lets you get away with that are dangerous and error-prone. (We'll look at them as the book progresses.) The C++ compiler generates warnings and errors for these situations. This is often much more of an advantage than a hindrance. In fact, there are many situations where you are trying to run down an error in C and just can't find it, but as soon as you recompile the program in C++, the compiler points out the problem! In C, you'll often find that you can get the program to compile, but then you have to get it to work. In C++, when the program compiles correctly, it often works, too! This is because the language is a lot stricter about type.

You can see a number of new things in the way the C++ version of **Stash** is used, in the following test program:

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Test of C++ library
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  // Holds 80-character strings:
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
```

```
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

One thing you'll notice is that the variables are all defined "on the fly" (as introduced in the previous chapter). That is, they are defined at any point in the scope, rather than being restricted – as in C – to the beginning of the scope.

The code is quite similar to **CLibTest.cpp**, but when a member function is called, the call occurs using the member selection operator '**.**' preceded by the name of the variable. This is a convenient syntax because it mimics the selection of a data member of the structure. The difference is that this is a function member, so it has an argument list.

Of course, the call that the compiler *actually* generates looks much more like the original C library function. Thus, considering name decoration and the passing of **this**, the C++ function call **intStash.initialize(sizeof(int), 100)** becomes something like **Stash_initialize(&intStash, sizeof(int), 100)**. If you ever wonder what's going on underneath the covers, remember that the original C++ compiler **cfront** from AT&T produced C code as its output, which was then compiled by the underlying C compiler. This approach meant that **cfront** could be quickly ported to any machine that had a C compiler, and it helped to rapidly disseminate C++ compiler technology. But because the C++ compiler had to generate C, you know that there must be some way to represent C++ syntax in C.

You'll also notice an additional cast in

```
while(cp = (char*)stringStash.fetch(k++))
```

This is due again to the stricter type checking in C++.

There's one other change from **ClibTest.cpp**, which is the introduction of the **require.h** header file. This is a header file which I created for this book to perform more sophisticated error checking than that provided by **assert( )**. It contains several functions, including the one used here called **assure( )** which is used for files. This function checks to see if the file has successfully been opened, and if not it reports to standard error that the

file could not be opened (thus it needs the name of the file as the second argument) and exits the program. The **require.h** functions will be used throughout the book, in particular to ensure that there are the right number of command-line arguments and that files are opened properly. The **require.h** functions replace repetetive and distracting error-checking code, and yet they provide essentially useful error messages. These functions will be fully explained later in the book.

What's an object?

Now that you've seen an initial example, it's time to step back and take a look at some terminology. The act of bringing functions inside structures is the root of what C++ adds to C, and it introduces a new way of thinking about structures: as concepts. In C, a structure is an agglomeration of data, a way to package data so you can treat it in a clump. But it's hard to think about it as anything but a programming convenience. The functions that operate on those structures are elsewhere. However, with functions in the package, the structure becomes a new creature, capable of describing both characteristics (like a C **struct** does) *and* behaviors. The concept of an object, a free-standing, bounded entity that can remember *and* act, suggests itself.

In C++, an object is just a variable, and the purest definition is "a region of storage" (this is a more specific way of saying "an object must have a unique identifier," which in the case of C++ is a unique memory address). It's a place where you can store data, and it's implied that there are also operations that can be performed on this data.

Unfortunately there's not complete consistency across languages when it comes to these terms, although they are fairly well-accepted. You will also sometimes encounter disagreement about what an object-oriented language is, although that seems to be reasonably well sorted out by now. There are languages that are *object-based*, which means they have objects like the C++ structures-with-functions that you've seen so far. This, however, is only part of the picture when it comes to an object-oriented language, and languages that stop at packaging functions inside data structures are object-based, not object-oriented.

Abstract data typing

The ability to package data with functions allows you to create a new data type. This is often called *encapsulation*[23]. An existing data type may have several pieces of data packaged together. For example a **float** has an exponent, a mantissa, and a sign bit. You can tell it to do things: add to another **float** or to an **int**, and so on. It has characteristics and behavior.

The definition of **Stash** creates a new data type. You can **add( )** and **fetch( )** and **inflate( )**. You create one by saying **Stash s**, just as you create a **float** by saying **float f**. A **Stash** also has characteristics and behavior. Even though it acts like a real, built-in data type, we refer to it as an *abstract data type*, perhaps because it allows us to abstract a concept from the problem space into the solution space. In addition, the C++ compiler treats it like a new data type, and if you say a function expects a **Stash**, the compiler makes sure you pass a **Stash** to that function. So the same level of type checking happens with abstract data types (sometimes called *user-defined types*) as with built-in types.

You can immediately see a difference, however, in the way you perform operations on objects. You say **object.memberFunction(arglist)**. This is "calling a member function for an object." But in object-oriented parlance, this is also referred to as "sending a message to an object." So for a **Stash s**, the statement **s.add(&i)** "sends a message to **s**" saying "**add( )** this to yourself." In fact, object-oriented programming can be summed up in a single phrase: *sending messages to objects*. Really, that's all you do – create a bunch of objects and send messages to them. The trick, of course, is figuring out what your objects and messages *are*, but once you accomplish this the implementation in C++ is surprisingly straightforward.

Object details

A question that comes up a lot in seminars is "How big is an object, and what does it look like?" The answer is "about what you expect from a C **struct**." In fact, the code the C compiler produces for a C **struct** (with no C++ adornments) will usually look *exactly* the same as the code produced by a C++ compiler. This is reassuring to those C programmers who

[23] This term can cause debate. Some people use it as defined here; others use it to describe *implementation hiding*, discussed in the following chapter.

depend on the details of size and layout in their code, and for some reason directly access structure bytes instead of using identifiers (relying on a particular size and layout for a structure is a nonportable activity).

The size of a **struct** is the combined size of all its members. Sometimes when the compiler lays out a **struct**, it adds extra bytes to make the boundaries come out neatly – this may increase execution efficiency. In Chapters XX and XX, you'll see how in some cases "secret" pointers are added to the structure, but you don't need to worry about that right now.

You can determine the size of a **struct** using the **sizeof** operator. Here's a small example:

```
//: C04:Sizeof.cpp
// Sizes of structs
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;

struct A {
  int i[100];
};

struct B {
  void f();
};

void B::f() {}

int main() {
  cout << "sizeof struct A = " << sizeof(A)
    << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B)
    << " bytes" << endl;
  cout << "sizeof CStash in C = "
    << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = "
    << sizeof(Stash) << " bytes" << endl;
} ///:~
```

On my machine (your results may vary) the first print statement produces 200 because each **int** occupies two bytes. **struct B** is something of an anomaly because it is a **struct** with no data members. In C, this is illegal,

but in C++ we need the option of creating a **struct** whose sole task is to scope function names, so it is allowed. Still, the result produced by the second print statement is a somewhat surprising nonzero value. In early versions of the language, the size was zero, but an awkward situation arises when you create such objects: They have the same address as the object created directly after them, and so are not distinct. One of the fundamental rules of objects is that each object must have a unique address, so structures with no data members will always have some minimum nonzero size.

The last two **sizeof** statements show you that the size of the structure in C++ is the same as the size of the equivalent version in C. C++ tries not to add any unnecessary overhead.

Header file etiquette

When you create a **struct** containing member functions, you are creating a new data type. Generally, you want this type to be easily accessible to yourself and others. In addition, you want to separate the interface (the declaration) from the implementation (the definition of the member functions) so the implementation can be changed without forcing a re-compile of the entire system. You achieve this end by putting the declaration for your new type in a header file.

When I first learned to program in C, the header file was a mystery to me. Many C books don't seem to emphasize it, and the compiler didn't enforce function declarations, so it seemed optional most of the time, except when structures were declared. In C++ the use of header files becomes crystal clear. They are almost mandatory for easy program development, and you put very specific information in them: declarations. The header file tells the compiler what is available in your library. You can use the library even if you only posess the header file along with the object file or library file – you don't need the source code for the **cpp** file. The header file is where the interface specification is stored.

Although it is not enforced by the compiler, the best approach to building large projects in C is to use libraries: collect associated functions into the same object module or library, and use a header file to hold all the declarations for the functions. It is *de rigueur* in C++: you could throw any function into a C library, but the C++ abstract data type determines the functions that are associated by dint of their common access to the data in a **struct**. Any member function must be declared in the **struct**

declaration; you cannot put it elsewhere. The use of function libraries was encouraged in C and institutionalized in C++.

Importance of header files

When using a function from a library, C allows you the option of ignoring the header file and simply declaring the function by hand. In the past, people would sometimes do this to speed up the compiler just a bit by avoiding the task of opening and including the file (this is usually not an issue with modern compilers). For example, here's an extremely lazy declaration of the C function **printf( )** (from **<stdio.h>**):

```
printf(...);
```

The ellipses specify a *variable argument list*[24], which says: **printf( )** has some arguments, each of which has a type, but ignore that. Just take whatever arguments you see and accept them. By using this kind of declaration, you suspend all error checking on the arguments.

This practice can cause subtle problems. If you declare functions by hand, in one file you may make a mistake. Since the compiler only sees your hand-declaration in that file, it may be able to adapt to your mistake. The program will then link correctly, but the use of the function in that one file will be faulty. This is a tough error to find, and is easily avoided by using a header file.

If you place all your function declarations in a header file, and include that header everywhere you use the function and where you define the function, you ensure a consistent declaration across the whole system. You also ensure that the declaration and the definition match by including the header in the definition file.

If a **struct** is declared in a header file in C++, you *must* include the header file everywhere a **struct** is used and where **struct** member functions are defined. The C++ compiler will give an error message if you try to call a regular function, or call or define a member function, without declaring it first. By enforcing the proper use of header files, the language ensures consistency in libraries, and reduces bugs by forcing the same interface to be used everywhere.

---

[24] To write a function definition for a function that takes a true variable argument list, you must use *varargs*. This is avoided in C++; you can find details about the use of varargs in your C manual.

The header is a contract between you and the user of your library. The contract describes your data structures, and states the arguments and return values for the function calls. It says, "Here's what my library does." The user needs some of this information to develop the application and the compiler needs all of it to generate proper code. The user of the **struct** simply includes the header file, creates objects (instances) of that **struct**, and links in the object module or library (i.e.: the compiled code).

The compiler enforces the contract by requiring you to declare all structures and functions before they are used and, in the case of member functions, before they are defined. Thus, you're forced to put the declarations in the header and to include the header in the file where the member functions are defined and the file(s) where they are used. Because a single header file describing your library is included throughout the system, the compiler can ensure consistency and prevent errors.

There are certain issues that you must be aware of in order to organize your code properly and write effective header files. The first issue concerns what you can put into header files. The basic rule is "only declarations," that is, only information to the compiler but nothing that allocates storage by generating code or creating variables. This is because the header file will typically be included in several translation units in a project, and if storage for one identifier is allocated in more than one place, the linker will come up with a multiple definition error (this is C++'s *one definition rule*: you can declare things as many times as you want, but there can be only one actual definition for each thing).

This rule isn't completely hard and fast. If you define a variable that is "file static" (has visibility only within a file) inside a header file, there will be multiple instances of that data across the project, but the linker won't have a collision[25]. Basically, you don't want to do anything in the header file that will cause an ambiguity at link time.

The multiple-declaration problem

The second header-file issue is this: when you put a **struct** declaration in a header file, it is possible for the file to be included more than once in a complicated program. Iostreams are a good example. Any time a **struct**

[25] However, in Standard C++ file static is a deprecated feature.

does I/O it may include one of the iostream headers. If the **cpp** file you are working on uses more than one kind of **struct** (typically including a header file for each one), you run the risk of including the istreams header more than once and re-declaring streams.

The compiler considers the redeclaration of a structure (this includes both **struct**s and **class**es) to be an error, since it would otherwise allow you to use the same name for different types. To prevent this error when multiple header files are included, you need to build some intelligence into your header files using the preprocessor (Standard C++ header files like **<iostream>** already have this "intelligence").

Both C and C++ allow you to redeclare a function, as long as the two declarations match, but neither will allow the redeclaration of a structure. In C++ this rule is especially important because if the compiler allowed you to redeclare a structure and the two declarations differed, which one would it use?

The problem of redeclaration comes up quite a bit in C++ because each data type (structure with functions) generally has its own header file, and you have to include one header in another if you want to create another data type that uses the first one. In any **cpp** file in your project, it's very likely that you'll include several files that include the same header file. During a single compilation, the compiler can see the same header file several times. Unless you do something about it, the compiler will see the redeclaration of your structure and report a compile-time error. To solve the problem, you need to know a bit more about the preprocessor.

The preprocessor directives **#define**, **#ifdef** and **#endif**

The preprocessor directive **#define** can be used to create compile-time flags. You have two choices: you can simply tell the preprocessor that the flag is defined, without specifying a value:

```
#define FLAG
```

or you can give it a value (which is the typical C way to define a constant):

```
#define PI 3.14159
```

In either case, the label can now be tested by the preprocessor to see if it has been defined:

```
#ifdef FLAG
```

will yield a true result, and the code following the **#ifdef** will be included in the package sent to the compiler. This inclusion stops when the preprocessor encounters the statement

```
#endif
```

or

```
#endif // FLAG
```

Any non-comment after the **#endif** on the same line is illegal, even though some compilers may accept it. The **#ifdef**/**#endif** pairs may be nested within each other.

The complement of **#define** is **#undef** (short for "un-define"), which will make an **#ifdef** statement using the same variable yield a false result. **#undef** will also cause the preprocessor to stop using a macro. The complement of **#ifdef** is **#ifndef**, which will yield a true if the label has not been defined (this is the one we will use in header files).

There are other useful features in the C preprocessor. You should check your local documentation for the full set.

A standard for header files

In each header file that contains a structure, you should first check to see if this header has already been included in this particular **cpp** file. You do this by testing a preprocessor flag. If the flag isn't set, the file wasn't included and you should set the flag (so the structure can't get re-declared) and declare the structure. If the flag was set then that type has already been declared so you should just ignore the code that declares it. Here's how the header file should look:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// Type declaration here...
#endif // HEADER_FLAG
```

As you can see, the first time the header file is included, the contents of the header file (including your type declaration) will be included by the preprocessor. All the subsequent times it is included – in a single compilation unit – the type declaration will be ignored. The name HEADER_FLAG can be any unique name, but a reliable standard to follow is to capitalize the name of the header file and replace periods with

underscores (leading underscores are reserved for system names). Here's an example:

```
//: C04:Simple.h
// Simple header that prevents re-definition
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

Although the **SIMPLE_H** after the **#endif** is commented out and thus ignored by the preprocessor, it is useful for documentation.

These preprocessor statements that prevent multiple inclusion are often referred to as *include guards*.

Namespaces in headers

You'll notice that *using directives* are present in nearly all the **cpp** files in this book, usually in the form:

```
using namespace std;
```

Since **std** is the namespace that surrounds the entire Standard C++ library, this particular using directive allows the names in the Standard C++ library to be used without qualification. However, you'll virtually never see a using directive in a header file (at least, not outside of a scope). The reason is that the using directive eliminates the protection of that particular namespace, and the effect lasts until the end of the current compilation unit. If you put a using directive (outside of a scope) in a header file, it means that this loss of "namespace protection" will occur with any file that includes this header, which often means other header files. Thus, if you start putting using directives in header files, it's very easy to end up "turning off" namespaces practically everywhere, and thereby neutralizing the beneficial effects of namespaces.

In short: don't do it.

Using headers in projects

When building a project in C++, you'll usually create it by bringing together a lot of different types (data structures with associated functions). You'll usually put the declaration for each type or group of associated types in a separate header file, then define the functions for that type in a translation unit. When you use that type, you must include the header file to perform the declarations properly.

Sometimes that pattern will be followed in this book, but more often the examples will be very small, so everything – the structure declarations, function definitions, and the **main( )** function – may appear in a single file. However, keep in mind that you'll want to use separate files and header files in practice.

Nested structures

The convenience of taking data and function names out of the global name space extends to structures. You can nest a structure within another structure, and therefore keep associated elements together. The declaration syntax is what you would expect, as you can see in the following structure, which implements a push-down stack as a very simple linked list so it "never" runs out of memory:

```
//: C04:Stack.h
// Nested struct in linked list
#ifndef STACK_H
#define STACK_H

struct Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

The nested **struct** is called **Link**, and it contains a pointer to the next **Link** in the list and a pointer to the data stored in the **Link**. If the **next** pointer is zero, it means you're at the end of the list.

Notice that the **head** pointer is defined right after the declaration for **struct Link**, instead of a separate definition **Link* head**. This is a syntax that came from C, but it emphasizes the importance of the semicolon after the structure declaration – the semicolon indicates the end of the list of definitions of that structure type. (Usually the list is empty.)

The nested structure has its own **initialize( )** function, like all the structures presented so far, to ensure proper initialization. **Stack** has both an **initialize( )** and **cleanup( )** function, as well as **push( )**, which takes a pointer to the data you wish to store (it assumes this has been allocated on the heap), and **pop( )**, which returns the **data** pointer from the top of the **Stack** and removes the top element. (If you **pop( )** an element, then *you* are responsible for destroying the object pointed to by the **data**.) The **peek( )** function also returns the **data** pointer from the top element, but it leaves the top element on the **Stack**.

cleanup( ) goes through the **Stack** and removes each element *and* frees the **data** pointer (thus the **data** objects *must* be on the heap). Notice there's something you have to keep track of, a bit: if you **pop( )** the element, you must call **delete**, but if you don't, then **cleanup( )** will call **delete**. The subject of "who's responsible for the memory" is not even that simple, as we'll see in later chapters.

Here are the definitions for the member functions:

```
//: C04:Nested.cpp {O}
// Linked list with nesting
#include "Stack.h"
#include "../require.h"
using namespace std;

void
Stack::Link::initialize(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link();
```

```
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() { return head->data; }

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  Link* cursor = head;
  while(head) {
    cursor = cursor->next;
    delete head->data; // Assumes a 'new'!
    delete head;
    head = cursor;
  }
  head = 0; // Officially empty
} ///:~
```

The first definition is particularly interesting because it shows you how to define a member of a nested structure. You simply use an additional level of scope resolution, to specify the name of the enclosing **struct**. **Stack::Link::initialize( )** takes the arguments and assigns them to its members.

Stack::initialize( ) sets **head** to zero, so the object knows it has an empty list.

Stack::push( ) takes the argument, which is a pointer to the variable you want to keep track of, and pushes it on the **Stack**. First, it uses **new** to allocate storage for the **Link** it will insert at the top. Then it calls **Link**'s **initialize( )** function to assign the appropriate values to the members of the **Link**. Notice that the **next** pointer is assigned to the current **head**; then **head** is assigned to the new **Link** pointer. This effectively pushes the **Link** in at the top of the list.

Stack::pop( ) captures the **data** pointer at the current top of the **Stack**; then it moves the **head** pointer down and deletes the old top of the **Stack**, finally returning the captured pointer.

Stack::cleanup( ) creates a **cursor** to move through the **Stack** and **delete** both the **data** in each link and the link itself. After it's finished destroying all the links, **head** is set to zero. This not only indicates that the **Stack** is empty, but if **cleanup( )** is called a second time it will not wander off and try to **delete** inappropriate storage (which would be a run-time error, and might cause difficult-to-find bugs).

Here's an example to test the **Stack**:

```
//: C04:StackTest.cpp
//{L} Nested
//{T} NestTest.cpp
// Test of nested linked list
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
  // Read file and store lines in the Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the Stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

This is very similar to the earlier example, but it pushes lines from a file (as **string** pointers) on the **Stack** and then pops them off, which results in the file being printed out in reverse order. Note that the **pop( )** member function returns a **void*** and this must be cast back to a **string*** before it can be used. To print the **string**, the pointer is dereferenced.

As **textlines** is being filled, the contents of **line** is "cloned" for each **push( )** by making a **new string(line)**. The value returned from the new-expression is a pointer to the new **string** that was created and that copied the information from **line**. If you had simply passed the address of **line** to **push( )**, you would end up with a **Stack** filled with identical addresses, all pointing to **line**. You'll learn more about this "cloning" process later in the book.

The file name is taken from the command line. To guarantee that there are enough arguments on the command line, you see a second function used from the **require.h** header file: **requireArgs( )**, which compares **argc** to the desired number of arguments and prints an appropriate error message and exits the program if there aren't enough arguments.

Global scope resolution

The scope resolution operator gets you out of situations where the name the compiler chooses by default (the "nearest" name) isn't what you want. For example, suppose you have a structure with a local identifier **a**, and you want to select a global identifier **a** from inside a member function. The compiler would default to choosing the local one, so you must tell it to do otherwise. When you want to specify a global name using scope resolution, you use the operator with nothing in front of it. Here's an example that shows global scope resolution for both a variable and a function:

```
//: C04:Scoperes.cpp
// Global scope resolution
int a;
void f() {}

struct S {
  int a;
  void f();
};

void S::f() {
  ::f();  // Would be recursive otherwise!
```

```
  ::a++;  // Select the global a
  a--;    // The a at struct scope
}
int main() { S s; f(); } ///:~
```

Without scope resolution in **S::f( )**, the compiler would default to selecting the member versions of **f( )** and **A**.

Summary

In this chapter, you've learned the fundamental "twist" of C++: that you can place functions inside of structures. This new type of structure is called an *abstract data type*, and variables you create using this structure are called *objects*, or *instances*, of that type. Calling a member function for an object is called *sending a message* to that object. The primary action in object-oriented programming is sending messages to objects.

Although packaging data and functions together is a significant benefit for code organization and makes library use easier because it prevents name clashes by hiding the names, there's a lot more you can do to make programming safer in C++. In the next chapter, you'll learn how to protect some members of a **struct** so that only you can manipulate them. This establishes a clear boundary between what the user of the structure can change and what only the programmer may change.

Exercises

1. In the Standard C library, the function **puts( )** prints a char array to the console (so you can say **puts("hello")**). Write a C program that uses **puts( )** but does not include **<stdio.h>** or otherwise declare the function. Compile this program with your C compiler (Some C++ compilers are not distinct from their C compilers; in this case you may need to discover a command-line flag that forces a C compilation). Now compile it with the C++ compiler and note the difference.
2. Create a **struct** declaration with a single member function; then create a definition for that member function. Create an object of your new data type, and call the member function.

3. Change your solution to the previous exercise so the **struct** is declared in a properly "guarded" header file, the definition is in one **cpp** file and your **main( )** is in another.
4. Create a **struct** with a single **int** data member, and two global functions, each of which takes a pointer to that **struct**. The first function has a second **int** argument and sets the **struct**'s **int** to the argument value, the second displays the **int** from the **struct**. Test the functions.
5. Repeat the previous exercise but move the functions so they are member functions of the **struct**, and test again.
6. Write and compile a piece of code that performs data member selection and a function call using the **this** keyword (which refers to the address of the current object).
7. Make a **Stash** that holds **double**s. Fill it with 25 **double** values, then print them out to the console.
8. Repeat the previous exercise with **Stack**.
9. Create a file containing a function **f( )** which takes an **int** argument and prints it to the console using the **printf( )** function in **<stdio.h>** by saying: **printf("%d\n", i)** where **i** is the **int** you wish to print. Create a separate file containing **main( )**, and in this file declare **f( )** to take a **float** argument. Call **f( )** from inside **main( )**. Try to compile and link your program with the C++ compiler and see what happens. Now compile and link the program using the C compiler, and see what happens when it runs. Explain the behavior.
10. Find out how to produce assembly language from your C and C++ compilers. Write a function in C, and a **struct** with a single member function in C++, and produce assembly language from each and find the function names that are produced by your C function and your C++ member function, so you can see what sort of name decoration occurs inside the compiler.
11. Write a program with conditionally-compiled code in **main( )**, so that when a preprocessor value is defined one message is printed, but when it is not defined another message is printed. Compile this code experimenting with a **#define** within the program, then discover the way your compiler takes preprocessor definitions on the command line and experiment with that.

12. Write a program that uses **assert( )** with an argument that is always true (nonzero) to see what happens when you run it. Now compile it with **#define NDEBUG** and run it again to see the difference.
13. Create an abstract data type that represents a video tape in a video rental store. Try to consider all the data and operations that may be necessary for the **Video** type to work well within the video rental management system. Include a **print( )** member function that displays information about the **Video**.
14. Create a **Stack** object to hold the **Video** objects from the previous exercise. Create several **Video** objects, store them in the **Stack**, then display them using **Video::print( )**.
15. Write a program that prints out all the sizes for the fundamental data types on your computer, using **sizeof( )**.
16. Modify **Stash** to use a **vector<char>** as its underlying data structure.
17. Dynamically create pieces of storage of the following types, using **new**: **int**, **long**, an array of 100 **char**s, an array of 100 **float**s. Print the addresses of these and then free the storage using **delete**.
18. Write a function that takes a **char*** argument. Using **new**, dynamically allocate an array of **char** which is the size of the **char** array that's passed to the function. Using array indexing, copy the characters from the argument to the dynamically allocated array (don't forget the null terminator) and return the pointer to the copy. In your **main( )**, test the function by passing a static quoted character array, then take the result of that and pass it back into the function. Print both strings and both pointers so you can see they are different storage. Using **delete**, clean up all the dynamic storage.
19. Show an example of a structure declared within another structure (a *nested structure*). Declare data members in both **struct**s, and declare and define member functions in both **struct**s. Write a **main( )** that tests your new types.
20. How big is a structure? Write a piece of code that prints the size of various structures. Create structures that have data members only and ones that have data members and function members. Then create a structure that has no members at all. Print out the sizes of all these. Explain the

reason for the result of the structure with no data members at all.

21. C++ automatically creates the equivalent of a **typedef** for **struct**s, as you've seen in this chapter. It also does this for enumerations and unions. Write a small program that demonstrates this.
22. Create a **Stack** that holds **Stash**es. Each **Stash** will hold 5 lines from an input file. Create the **Stash**es using **new**. Read a file into your **Stack**, then reprint it in its original form by extracting it from the **Stack**.
23. Modify the previous exercise so that you create a **struct** that encapsulates the **Stack** of **Stash**es. The user should only add and get lines via member functions, but under the covers the **struct** happens to use a **Stack** of **Stash**es.
24. Create a **struct** that holds an **int** and a pointer to another instance of the same **struct**. Write a function that takes the address of one of these **struct**s and an **int** indicating the length of the list you want created. This function will make makes a whole chain of these **struct**s (a *linked list*), starting from the argument (the *head* of the list), with each one pointing to the next. Make the new **struct**s using **new**, and put the count (which object number this is) in the **int**. In the last **struct** in the list, put a zero value in the pointer to indicate that it's the end. Write a second function that takes the head of your list and moves through to the end, printing out both the pointer value and the **int** value for each one.
25. Repeat the previous exercise, but put the functions inside a **struct** instead of using "raw" **struct**s and functions.
26. Write a function which takes two **int** arguments, **rows** and **columns**. This function returns a pointer to a dynamically-allocated two-dimensional array of **float**. As a hint, the first call to **new** is:
new float[rows][]. This must be followed by multiple calls to **new** in order to create all the storage for the **rows**. Write a second function that takes this "matrix" and frees the storage using **delete**. Now convert the code into a **struct** called **matrix**.

5: Hiding the implementation

A typical C library contains a **struct** and some associated functions to act on that **struct**. So far, you've seen how C++ takes functions that are *conceptually* associated and makes them *literally* associated, by

putting the function declarations inside the scope of the **struct**, changing the way functions are called for the **struct**, eliminating the passing of the structure address as the first argument, and adding a new type name to the program (so you don't have to create a **typedef** for the **struct** tag).

These are all convenient – they help you organize your code and make it easier to write and read. However, there are other important issues when making libraries easier in C++, especially the issues of safety and control. This chapter looks at the subject of boundaries in structures.

Setting limits

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the *client programmer* who uses that library to build an application or another library.

In a C **struct**, as with most things in C, there are no rules. Client programmers can do anything they want with that **struct**, and there's no

way to force any particular behaviors. For example, even though you saw in the last chapter the importance of the functions named **initialize( )** and **cleanup( )**, the client programmer has the option not to call those functions. (We'll look at a better approach in the next chapter.) And even though you would really prefer that the client programmer not directly manipulate some of the members of your **struct**, in C there's no way to prevent it. Everything's naked to the world.

There are two reasons for controlling access to members. The first is to keep the client programmer's hands off tools they shouldn't touch, tools that are necessary for the internal machinations of the data type, but not part of the interface the client programmer needs to solve their particular problems. This is actually a service to client programmers because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the structure without worrying about how it will affect the client programmer. In the **Stack** example in the last chapter, you might want to allocate the storage in big chunks, for speed, rather than creating new storage each time an element is added. If the interface and implementation are clearly separated and protected, you can accomplish this and require only a relink by the client programmer.

C++ access control

C++ introduces three new keywords to set the boundaries in a structure: **public**, **private**, and **protected**. Their use and meaning are remarkably straightforward. These *access specifiers* are used only in a structure declaration, and they change the boundary for all the declarations that follow them. Whenever you use an access specifier, it must be followed by a colon.

public means all member declarations that follow are available to everyone. **public** members are like **struct** members. For example, the following **struct** declarations are identical:

```
//: C05:Public.cpp
// Public is just like C's struct

struct A {
  int i;
  char j;
  float f;
```

```
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

The **private** keyword, on the other hand, means that no one can access that member except you, the creator of the type, inside function members of that type. **private** is a brick wall between you and the client programmer; if someone tries to access a **private** member, they'll get a compile-time error. In **struct B** in the above example, you may want to make portions of the representation (that is, the data members) hidden, accessible only to you:

```
//: C05:Private.cpp
// Setting the boundary

struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};
```

```
void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;    // OK, public
//!  b.j = '1';  // Illegal, private
//!  b.f = 1.0;  // Illegal, private
} ///:~
```

Although **func( )** can access any member of **B** (because **func( )** is itself a member of **B**, thus automatically granting it permission), an ordinary global function like **main( )** cannot. Of course, neither can member functions of other structures. Only the functions that are clearly stated in the structure declaration (the "contract") can have access to **private** members.

There is no required order for access specifiers, and they may appear more than once. They affect all the members declared after them and before the next access specifier.

protected

The last access specifier is **protected**. **protected** acts just like **private**, with one exception that we can't really talk about right now: "Inherited" structures (which cannot access **private** members) are granted access to **protected** members. But inheritance won't be introduced until Chapter XX, so this doesn't have any meaning to you. For the current purposes, consider **protected** to be just like **private**; it will be clarified when inheritance is introduced.

Friends

What if you want to explicitly grant access to a function that isn't a member of the current structure? This is accomplished by declaring that function a **friend** *inside* the structure declaration. It's important that the **friend** declaration occurs inside the structure declaration because you (and the compiler) must be able to read the structure declaration and see

every rule about the size and behavior of that data type. And a very important rule in any relationship is “who can access my private implementation?”

The class controls which code has access to its members. There’s no magic way to “break in” from the outside if you aren’t a **friend**; you can’t declare a new class and say “hi, I’m a friend of **Bob**!” and expect to see the **private** and **protected** members of **Bob**.

You can declare a global function as a **friend**, and you can also declare a member function of another structure, or even an entire structure, as a **friend**. Here’s an example :

```
//: C05:Friend.cpp
// Friend allows special access

// Declaration (incomplete type specification):
struct X;

struct Y {
  void f(X*);
};

struct X { // Definition
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h();
};

void X::initialize() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
```

```
}

struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};

void Z::initialize() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
  x.i = 100; // Direct data manipulation
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

struct Y has a member function **f( )** that will modify an object of type **X**. This is a bit of a conundrum because the C++ compiler requires you to declare everything before you can refer to it, so **struct Y** must be declared before its member **Y::f(X*)** can be declared as a friend in **struct X**. But for **Y::f(X*)** to be declared, **struct X** must be declared first!

Here's the solution. Notice that **Y::f(X*)** takes the *address* of an **X** object. This is critical because the compiler always knows how to pass an address, which is of a fixed size regardless of the object being passed, even if it doesn't have full information about the size of the type. If you try to pass the whole object, however, the compiler must see the entire structure definition of **X**, to know the size and how to pass it, before it allows you to declare a function such as **Y::g(X)**.

By passing the address of an **X**, the compiler allows you to make an *incomplete type specification* of **X** prior to declaring **Y::f(X*)**. This is accomplished in the declaration

```
struct X;
```

The declaration simply tells the compiler there's a **struct** by that name, so it's OK to refer to it as long as you don't require any more knowledge than the name.

Now, in **struct X**, the function **Y::f(X*)** can be declared as a **friend** with no problem. If you tried to declare it before the compiler had seen the full specification for **Y**, it would have given you an error. This is a safety feature to ensure consistency and eliminate bugs.

Notice the two other **friend** functions. The first declares an ordinary global function **g( )** as a **friend**. But **g( )** has not been previously declared at the global scope! It turns out that **friend** can be used this way to simultaneously declare the function *and* give it **friend** status. This extends to entire structures:

```
friend struct Z;
```

is an incomplete type specification for **Z**, and it gives the entire structure **friend** status.

Nested friends

Making a structure nested doesn't automatically give it access to **private** members. To accomplish this you must follow a particular form: first define the nested structure, then declare it as a **friend** using full scoping. The structure definition must be separate from the **friend** declaration, otherwise it would be seen by the compiler as a nonmember. Here's an example:

```
//: C05:NestFriend.cpp
// Nested friends
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

struct Holder {
private:
  int a[sz];
public:
```

```
  void initialize();
  struct Pointer {
  private:
    Holder* h;
    int* p;
  public:
    void initialize(Holder* h);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    // Access values:
    int read();
    void set(int i);
  };
  friend Holder::Pointer;
};

void Holder::initialize() {
  memset(a, 0, sz * sizeof(int));
}

void Holder::Pointer::initialize(Holder* h) {
  h = h;
  p = h->a;
}

void Holder::Pointer::next() {
  if(p < &(h->a[sz - 1])) p++;
}

void Holder::Pointer::previous() {
  if(p > &(h->a[0])) p--;
}

void Holder::Pointer::top() {
  p = &(h->a[0]);
}

void Holder::Pointer::end() {
  p = &(h->a[sz - 1]);
```

```
}

int Holder::Pointer::read() {
  return *p;
}

void Holder::Pointer::set(int i) {
  *p = i;
}

int main() {
  Holder h;
  Holder::Pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < sz; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.read()
      << ", hp2 = " << hp2.read() << endl;
    hp.next();
    hp2.previous();
  }
} ///:~
```

The **struct Holder** contains an array of **int**s and the **Pointer** allows you to access them. Because **Pointer** is strongly associated with **Holder**, it's sensible to make it a member structure of **Holder**. Once **Pointer** is defined, it is granted access to the private members of **Holder** by saying:

```
friend Holder::Pointer;
```

Notice that the **struct** keyword is not necessary because the compiler already knows what **Pointer** is.

Because **Pointer** is a separate class from **Holder**, you can make more than one of them in **main( )** and use them to select different parts of the

array. Because **Pointer** is a class instead of a raw C pointer, you can guarantee that it will always safely point inside the **Holder**.

The Standard C library function **memset( )** (in **<cstring>**) is used for convenience in the above program. It sets all memory starting at a particular address (the first argument) to a particular value (the second argument) for **n** bytes past the starting address (**n** is the third argument). Of course, you could have simply used a loop to iterate through all the memory, but **memset( )** is available, well-tested (so it's less likely you'll introduce an error) and probably more efficient than if you coded it by hand.

Is it pure?

The class definition gives you an audit trail, so you can see from looking at the class which functions have permission to modify the private parts of the class. If a function is a **friend**, it means that it isn't a member, but you want to give permission to modify private data anyway, and it must be listed in the class definition so everyone can see that it's one of the privileged functions.

C++ is a hybrid object-oriented language, not a pure one, and **friend** was added to get around practical problems that crop up. It's fine to point out that this makes the language less "pure," because C++ *is* designed to be pragmatic, not to aspire to an abstract ideal.

Object layout

Chapter XX stated that a **struct** written for a C compiler and later compiled with C++ would be unchanged. This referred primarily to the object layout of the **struct**, that is, where the storage for the individual variables is positioned in the memory allocated for the object. If the C++ compiler changed the layout of C **struct**s, then any C code you wrote that inadvisably took advantage of knowledge of the positions of variables in the **struct** would break.

When you start using access specifiers, however, you've moved completely into the C++ realm, and things change a bit. Within a particular "access block" (a group of declarations delimited by access specifiers), the variables are guaranteed to be laid out contiguously, as in C. However, the access blocks themselves may not appear in the object in the order that you declare them. Although the compiler will *usually* lay the blocks out exactly as you see them, there is no rule about it, because a

particular machine architecture and/or operating environment may have explicit support for **private** and **protected** that might require those blocks to be placed in special memory locations. The language specification doesn't want to restrict this kind of advantage.

Access specifiers are part of the structure and don't affect the objects created from the structure. All of the access specification information disappears before the program is run; generally this happens during compilation. In a running program, objects become "regions of storage" and nothing more. If you really want to, you can break all the rules and access the memory directly, as you can in C. C++ is not designed to prevent you from doing unwise things. It just provides you with a much easier, highly desirable alternative.

In general, it's not a good idea to depend on anything that's implementation-specific when you're writing a program. When you must, those specifics should be encapsulated inside a structure, so any porting changes are focused in one place.

The class

Access control is often referred to as *implementation hiding*. Including functions within structures (encapsulation) produces a data type with characteristics and behaviors, but access control puts boundaries within that data type, for two important reasons. The first is to establish what the client programmers can and can't use. You can build your internal mechanisms into the structure without worrying that client programmers will think it's part of the interface they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but the client programmers can't do anything but send messages to the **public** interface, then you can change anything that's **private** without requiring modifications to their code.

Encapsulation and implementation hiding, taken together, invent something more than a C **struct**. We're now in the world of object-oriented programming, where a structure is describing a class of objects, as you would describe a class of fishes or a class of birds: Any object belonging to this class will share these characteristics and behaviors. That's what the structure declaration has become, a description of the way all objects of this type will look and act.

In the original OOP language, Simula-67, the keyword **class** was used to describe a new data type. This apparently inspired Stroustrup to choose the same keyword for C++, to emphasize that this was the focal point of the whole language: the creation of new data types that are more than just C **struct**s with functions. This certainly seems like adequate justification for a new keyword.

However, the use of **class** in C++ comes close to being an unnecessary keyword. It's identical to the **struct** keyword in absolutely every way except one: **class** defaults to **private**, whereas **struct** defaults to **public**. Here are two structures that produce the same result:

```
//: C05:Class.cpp
// Similarity of struct and class

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}

void A::g() {
  i = j = k = 0;
}

// Identical results are produced with:

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i + j + k;
}
```

```
void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
} ///:~
```

The **class** is the fundamental OOP concept in C++. It is one of the keywords that will *not* be set in bold in this book – it becomes annoying with a word repeated as often as "class." The shift to classes is so important that I suspect Stroustrup's preference would have been to throw **struct** out altogether, but the need for backwards compatibility with C wouldn't allow that.

Many people prefer a style of creating classes that is more **struct**-like than class-like, because you override the "default-to-**private**" behavior of the class by starting out with **public** elements:

```
class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

The logic behind this is that it makes more sense for the reader to see the members of interest first, then they can ignore anything that says **private**. Indeed, the only reasons all the other members must be declared in the class at all are so the compiler knows how big the objects are and can allocate them properly, and so it can guarantee consistency.

The examples in this book, however, will put the **private** members first, like this:

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
```

```
};
```

Some people even go to the trouble of decorating their own private names:

```
class Y {
public:
  void f();
private:
  int mX;  // "Self-decorated" name
};
```

Because **mX** is already hidden in the scope of **Y**, the **m** is unnecessary. However, in projects with many global variables (something you should strive to avoid, but which is sometimes inevitable in existing projects) it is helpful to be able to distinguish, inside a member function definition, which data is global and which is a member.

Modifying **Stash** to use access control

It makes sense to take the examples from the previous chapter and modify them to use classes and access control. Notice how the client programmer portion of the interface is now clearly distinguished, so there's no possibility of client programmers accidentally manipulating a part of the class that they shouldn't.

```
//: C05:Stash.h
// Converted to use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
```

```
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

The **inflate( )** function has been made **private** because it is used only by the **add( )** function and is thus part of the underlying implementation, not the interface. This means that, sometime later, you can change the underlying implementation to use a different system for memory management.

Other than the name of the include file, the above header is the only thing that's been changed for this example. The implementation file and test file are the same.

Modifying **Stack** to use access control

As a second example, here's the **Stack** turned into a class. Now the nested data structure is **private**, which is nice because it ensures that the client programmer will neither have to look at it nor be able to depend on the internal representation of the **Stack**:

```
//: C05:Stack2.h
// Nested structs via linked list
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

As before, the implementation doesn't change and so is not repeated here. The test, too, is identical. The only thing that's been changed is the robustness of the class interface. The real value of access control is during development, to prevent you from crossing boundaries. In fact, the compiler is the only thing that knows about the protection level of class members. There is no access control information mangled into the member name that carries through to the linker. All the protection checking is done by the compiler; it has vanished by runtime.

Notice that the interface presented to the client programmer is now truly that of a push-down stack. It happens to be implemented as a linked list, but you can change that without affecting what the client programmer interacts with, or (more importantly) a single line of client code.

Handle classes

Access control in C++ allows you to separate interface from implementation, but the implementation hiding is only partial. The compiler must still see the declarations for all parts of an object in order to create and manipulate it properly. You could imagine a programming language that requires only the public interface of an object and allows the private implementation to be hidden, but C++ performs type checking statically (at compile time) as much as possible. This means that you'll learn as early as possible if there's an error. It also means your program is more efficient. However, including the private implementation has two effects: The implementation is visible even if you can't easily access it, and it can cause needless recompilation.

Hiding the implementation

Some projects cannot afford to have their implementation visible to the client programmer. It may show strategic information in a library header file that the company doesn't want available to competitors. You may be working on a system where security is an issue – an encryption algorithm, for example – and you don't want to expose any clues in a header file that might help people to crack the code. Or you may be putting your library in a "hostile" environment, where the programmers will directly access the private components anyway, using pointers and casting. In all these situations, it's valuable to have the actual structure compiled inside an implementation file rather than exposed in a header file.

Reducing recompilation

The project manager in your programming environment will cause a recompilation of a file if that file is touched (that is, modified) *or* if another file it's dependent upon – that is, an included header file – is touched. This means that any time you make a change to a class, whether it's to the public interface or the private member declarations, you'll force a recompilation of anything that includes that header file. For a large project in its early stages this can be very unwieldy because the underlying implementation may change often; if the project is very big, the time for compiles can prohibit rapid turnaround.

The technique to solve this is sometimes called *handle classes* or the "Cheshire Cat"[26] – everything about the implementation disappears except for a single pointer, the "smile." The pointer refers to a structure whose definition is in the implementation file along with all the member function definitions. Thus, as long as the interface is unchanged, the header file is untouched. The implementation can change at will, and only the implementation file needs to be recompiled and relinked with the project.

Here's a simple example demonstrating the technique. The header file contains only the public interface and a single pointer of an incompletely specified class:

```
//: C05:Handle.h
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // Class declaration only
  Cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
```

[26] This name is attributed to John Carolan, one of the early pioneers in C++, and of course, Lewis Carroll. This technique can also be seen as a form of the "proxy" design pattern, described in Chapter XX.

```
};
#endif // HANDLE_H ///:~
```

This is all the client programmer is able to see. The line

```
struct Cheshire;
```

is an *incomplete type specification* or a *class declaration* (A *class definition* includes the body of the class.) It tells the compiler that **Cheshire** is a structure name, but it doesn't give any details about the **struct**. This is only enough information to create a pointer to the **struct**; you can't create an object until the structure body has been provided. In this technique, that structure body is hidden away in the implementation file:

```
//: C05:Handle.cpp {O}
// Handle implementation
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire();
  require(smile != 0);
  smile->i = 0;
}

void Handle::cleanup() {
  delete smile;
}

int Handle::read() {
  return smile->i;
}

void Handle::change(int x) {
  smile->i = x;
} ///:~
```

Cheshire is a nested structure, so it must be defined with scope resolution:

```
struct Handle::Cheshire {
```

In **Handle::initialize( )**, storage is allocated for a **Cheshire** structure, and in **Handle::cleanup( )** this storage is released. This storage is used in lieu of all the data elements you'd normally put into the **private** section of the class. When you compile **Handle.cpp**, this structure definition is hidden away in the object file where no one can see it. If you change the elements of **Cheshire**, the only file that must be recompiled is **Handle.cpp** because the header file is untouched.

The use of **Handle** is like the use of any class: include the header, create objects, and send messages.

```
//: C05:UseHandle.cpp
//{L} Handle
// Use the Handle class
#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
} ///:~
```

The only thing the client programmer can access is the public interface, so as long as the implementation is the only thing that changes, the above file never needs recompilation. Thus, although this isn't perfect implementation hiding, it's a big improvement.

Summary

Access control in C++ gives valuable control to the creator of a class. The users of the class can clearly see exactly what they can use and what to ignore. More important, though, is the ability to ensure that no client programmer becomes dependent on any part of the underlying implementation of a class. If you know this as the creator of the class, you can change the underlying implementation with the knowledge that no client programmer will be affected by the changes because they can't access that part of the class.

When you have the ability to change the underlying implementation, you can not only improve your design at some later time, but you also have

the freedom to make mistakes. No matter how carefully you plan and design, you'll make mistakes. Knowing that it's relatively safe to make these mistakes means you'll be more experimental, you'll learn faster, and you'll finish your project sooner.

The public interface to a class is what the client programmer *does* see, so that is the most important part of the class to get "right" during analysis and design. But even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more functions, as long as you don't remove any that client programmers have already used in their code.

Exercises

1. Create a class with **public**, **private**, and **protected** data members and function members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members.
2. Write a **struct** called **Lib** which contains three **string** objects **a**, **b** and **c**. In **main( )** create a **Lib** object called **x** and assign to **x.a**, **x.b**, and **x.c**. Print out the values. Now replace **a**, **b** and **c** with an array of **string s[3]**. Show that your code in **main( )** breaks as a result of the change. Now create a **class** called **Libc**, with **private string** objects **a**, **b** and **c**, and member functions **seta( )**, **geta( )**, **setb( )**, **getb( )**, **setc( )**, **getc( )** to set and get the values. Write **main( )** as before. Now change the **private string** objects **a**, **b** and **c** to a **private** array of **string s[3]**. Show that the code in **main( )** does *not* break as a result of the change.
3. Create a class and a global **friend** function that manipulates the **private** data in the class.
4. Write two classes, each of which has a member function that takes a pointer to an object of the other class. Create instances of both objects in **main( )** and call the aforementioned member function in each class.
5. Create three classes. The first class contains **private** data, and grants friendship to the entire second class and to a member function of the third class. In **main( )**, demonstrate that all these work correctly.
6. Create a **Hen** class. Inside this, nest a **Nest** class. Inside **Nest**, place an **Egg** class. Each class should have a

display( ) member function. In **main( )**, create an instance of each class and call the **display( )** function for each one.

7. Modify the above example so that **Nest** and **Egg** each contain **private** data. Grant friendship to allow the enclosing classes access to this **private** data.
8. Create a class with data members distributed among numerous **public**, **private** and **protected** sections. Add a member function **showMap( )** which prints the names of each of these data members and their addresses. If possible, compile and run this program on more than one compiler and/or computer and/or operating system to see if there are layout differences in the object.
9. Copy the implementation and test files for **Stash** in the previous chapter so you can compile and test **Stash.h** in this chapter.
10. Place objects of the **Hen** class from the earlier exercise in a **Stash**. Fetch them out and print them (if you have not already done so, you will need to add **Hen::print( )**).
11. Copy the implementation and test files for **Stack** in the previous chapter so you can compile and test **Stack2.h** in this chapter.
12. Place objects of the **Hen** class from the earlier exercise in a **Stack**. Fetch them out and print them (if you have not already done so, you will need to add **Hen::print( )**).
13. Modify **Cheshire** in **Handle.cpp**, and verify that your project manager recompiles and relinks only this file, but doesn't recompile **UseHandle.cpp**.
14. Create a class using the "Cheshire cat" technique which represents an encryption algorithm that you want to hide as much as possible. The pointer that's in the handle class should point to an object that contains a member function which is the encryption algorithm, so that function should take a **string** object and produce an "encrypted" **string**. Use a trivial encryption algorithm, such as adding one to each letter.

6: Initialization & cleanup

Chapter 4 made a significant improvement in library use by taking all the scattered components of a typical C library and encapsulating them into a structure (an abstract data type, called a *class* from now on).

This not only provides a single unified point of entry into a library component, but it also hides the names of the functions within the class name. In Chapter 5, access control (implementation hiding) was introduced. This gives the class designer a way to establish clear boundaries for determining what the client programmer is allowed to manipulate and what is off limits. It means the internal mechanisms of a data type's operation are under the control and discretion of the class designer, and it's clear to client programmers what members they can and should pay attention to.

Together, encapsulation and implementation hiding make a significant step in improving the ease of library use. The concept of "new data type" they provide is better in some ways than the existing built-in data types from C. The C++ compiler can now provide type-checking guarantees for that data type and thus ensure a level of safety when that data type is being used.

When it comes to safety, however, there's a lot more the compiler can do for us than C provides. In this and future chapters, you'll see additional features that have been engineered into C++ that make the bugs in your program almost leap out and grab you, sometimes before you even compile the program, but usually in the form of compiler warnings and errors. For this reason, you will soon get used to the unlikely-sounding scenario that a C++ program that compiles usually runs right the first time.

Two of these safety issues are initialization and cleanup. A large segment of C bugs occur when the programmer forgets to initialize or clean up a variable. This is especially true with C libraries, when client programmers don't know how to initialize a **struct**, or even that they must. (Libraries often do not include an initialization function, so the client programmer is forced to initialize the **struct** by hand.) Cleanup is a special problem because C programmers are comfortable with forgetting about variables once they are finished, so any cleaning up that may be necessary for a library's **struct** is often missed.

In C++ the concept of initialization and cleanup is essential for easy library use and to eliminate the many subtle bugs that occur when the client programmer forgets to perform these activities. This chapter examines the features in C++ that help guarantee proper initialization and cleanup.

Guaranteed initialization with the constructor

Both the **Stash** and **Stack** classes have a function called **initialize( )**, which hints by its name that it should be called before using the object in any other way. Unfortunately, this means the client programmer must ensure proper initialization. Client programmers are prone to miss details like initialization in their headlong rush to make your amazing library solve their problem. In C++, initialization is too important to leave to the client programmer. The class designer can guarantee initialization of every object by providing a special function called the *constructor*. If a class has a constructor, the compiler automatically calls that constructor at the point an object is created, before client programmers can get their hands on the object. The constructor call isn't even an option for the client programmer; it is performed by the compiler at the point the object is defined.

The next challenge is what to name this function. There are two issues. The first is that any name you use is something that can potentially clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which function to call. The solution Stroustrup chose seems the easiest and most logical: The name of the constructor is the same as the name of the class. It makes sense that such a function will be called automatically on initialization.

Here's a simple class with a constructor:

```
class X {
  int i;
public:
  X();  // Constructor
};
```

Now, when an object is defined,

```
void f() {
  X a;
  // ...
}
```

the same thing happens as if **a** were an **int**: Storage is allocated for the object. But when the program reaches the *sequence point* (point of execution) where **a** is defined, the constructor is called automatically. That is, the compiler quietly inserts the call to **X::X( )** for the object **a** at the point of definition. Like any member function, the first (secret) argument to the constructor is the **this** pointer – the address of the object for which it is being called. In the case of the constructor, however, **this** is pointing to an un-initialized block of memory, and it's the job of the constructor to initialize this memory properly.

Like any function, the constructor can have arguments to allow you to specify how an object is created, give it initialization values, and so on. Constructor arguments provide you with a way to guarantee that all parts of your object are initialized to appropriate values. For example, if the class **Tree** has a constructor that takes a single integer argument denoting the height of the tree, then you must create a tree object like this:

```
Tree t(12);  // 12-foot tree
```

If **tree(int)** is your only constructor, the compiler won't let you create an object any other way. (We'll look at multiple constructors and different ways to call constructors in the next chapter.)

That's really all there is to a constructor: It's a specially named function that is called automatically by the compiler for every object, at the point of that object's creation. Despite it's simplicity, it is exceptionally valuable because it eliminates a large class of problems and makes the code easier to write and read. In the preceding code fragment, for example, you don't see an explicit function call to some **initialize( )** function that is

conceptually separate from definition. In C++, definition and initialization are unified concepts – you can't have one without the other.

Both the constructor and destructor are very unusual types of functions: They have no return value. This is distinctly different from a **void** return value, where the function returns nothing but you still have the option to make it something else. Constructors and destructors return nothing and you don't have an option. The acts of bringing an object into and out of the program are special, like birth and death, and the compiler always makes the function calls itself, to make sure they happen. If there were a return value, and if you could select your own, the compiler would somehow have to know what to do with the return value, or the client programmer would have to explicitly call constructors and destructors, which would eliminate their safety.

Guaranteed cleanup with the destructor

As a C programmer, you often think about the importance of initialization, but it's rarer to think about cleanup. After all, what do you need to do to clean up an **int**? Just forget about it. However, with libraries, just "letting go" of an object once you're done with it is not so safe. What if it modifies some piece of hardware, or puts something on the screen, or allocates storage on the heap? If you just forget about it, your object never achieves closure upon its exit from this world. In C++, cleanup is as important as initialization and is therefore guaranteed with the destructor.

The syntax for the destructor is similar to that for the constructor: The class name is used for the name of the function. However, the destructor is distinguished from the constructor by a leading tilde (**~**). In addition, the destructor never has any arguments because destruction never needs any options. Here's the declaration for a destructor:

```
class Y {
public:
  ~Y();
};
```

The destructor is called automatically by the compiler when the object goes out of scope. You can see where the constructor gets called by the point of definition of the object, but the only evidence for a destructor call is the closing brace of the scope that surrounds the object. Yet the

destructor is still called, even when you use **goto** to jump out of a scope. (**goto** still exists in C++, for backward compatibility with C and for the times when it comes in handy.) You should note that a *nonlocal goto*, implemented by the Standard C library functions **setjmp( )** and **longjmp( )**, doesn't cause destructors to be called. (This is the specification, even if your compiler doesn't implement it that way. Relying on a feature that isn't in the specification means your code is nonportable.)

Here's an example demonstrating the features of constructors and destructors you've seen so far:

```
//: C06:Constructor1.cpp
// Constructors & destructors
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();  // Destructor
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}
```

```
int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
} ///:~
```

Here's the output of the above program:

```
before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace
```

You can see that the destructor is automatically called at the closing brace of the scope that encloses it.

Elimination of the definition block

In C, you must always define all the variables at the beginning of a block, after the opening brace. This is not an uncommon requirement in programming languages, and the reason given has often been that it's "good programming style." On this point, I have my suspicions. It has always seemed inconvenient to me, as a programmer, to pop back to the beginning of a block every time I need a new variable. I also find code more readable when the variable definition is close to its point of use.

Perhaps these arguments are stylistic. In C++, however, there's a significant problem in being forced to define all objects at the beginning of a scope. If a constructor exists, it must be called when the object is created. However, if the constructor takes one or more initialization arguments, how do you know you will have that initialization information at the beginning of a scope? In the general programming situation, you

won't. Because C has no concept of **private**, this separation of definition and initialization is no problem. However, C++ guarantees that when an object is created, it is simultaneously initialized. This ensures you will have no uninitialized objects running around in your system. C doesn't care; in fact, C *encourages* this practice by requiring you to define variables at the beginning of a block before you necessarily have the initialization information.

Generally C++ will not allow you to create an object before you have the initialization information for the constructor. As a result, you can't be forced to define variables at the beginning of a scope. In fact, the style of the language would seem to encourage the definition of an object as close to its point of use as possible. In C++, any rule that applies to an "object" automatically refers to an object of a built-in type, as well. This means that any class object or variable of a built-in type can also be defined at any point in a scope. It also means that you can wait until you have the information for a variable before defining it, so you can always define and initialize at the same time:

```
//: C06:DefineInitialize.cpp
// Defining variables anywhere
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
  int i;
public:
  G(int ii);
};

G::G(int ii) { i = ii; }

int main() {
  cout << "initialization value? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

You can see that some code is executed, then **retval** is defined, initialized and used to capture user input, then **y** and **g** are defined. C, on the other hand, would never allow a variable to be defined anywhere except at the beginning of the scope.

Generally, you should define variables as close to their point of use as possible, and always initialize them when they are defined. (This is a stylistic suggestion for built-in types, where initialization is optional.) This is a safety issue. By reducing the duration of the variable's availability within the scope, you are reducing the chance it will be misused in some other part of the scope. In addition, readability is improved because the reader doesn't have to jump back and forth to the beginning of the scope to know the type of a variable.

for loops

In C++, you will often see a **for** loop counter defined right inside the **for** expression:

```
for(int j = 0; j < 100; j++) {
   cout << "j = " << j << endl;
}
for(int i = 0; i < 100; i++)
   cout << "i = " << i << endl;
```

The above statements are important special cases, which cause confusion to new C++ programmers.

The variables **i** and **j** are defined directly inside the **for** expression (which you cannot do in C). They are then available for use in the **for** loop. It's a very convenient syntax because the context removes all question about the purpose of **i** and **j**, so you don't need to use such ungainly names as **i_loop_counter** for clarity.

However, some confusion may result if you expect lifetime of the variables **i** and **j** to extend beyond the scope of the for loop – they do not[27].

Chapter 3 points out that **while** and **switch** statements also allow the definition of objects in their control expressions, although this usage seems far less important than with the **for** loop.

---

[27] An earlier iteration of the C++ draft standard said the variable lifetime extended to the end of the scope that enclosed the **for** loop. Some compilers still implement that, but it is not correct so your code will only be portable if you limit the scope to the **for** loop.

Watch out for local variables that hide variables in the enclosing scope. In general, using the same name for a nested variable as a varable global to that scope is confusing and error prone[28].

I find small scopes an indicator of good design. If you have several pages for a single function, perhaps you're trying to do too much with that function. More granular functions are not only more useful, but it's also easier to find bugs.

Storage allocation

A variable can now be defined at any point in a scope, so it might seem that the storage for a variable may not be defined until its point of definition. It's actually more likely that the compiler will follow the practice in C of allocating all the storage for a scope at the opening brace of that scope. It doesn't matter because, as a programmer, you can't access the storage (a.k.a. the object) until it has been defined[29]. Although the storage is allocated at the beginning of the block, the constructor call doesn't happen until the sequence point where the object is defined because the identifier isn't available until then. The compiler even checks to make sure you don't put the object definition (and thus the constructor call) where the sequence point only conditionally passes through it, such as in a **switch** statement or somewhere a **goto** can jump past it. Uncommenting the statements in the following code will generate a warning or an error:

```
//: C06:Nojump.cpp
// Can't jump past constructors

class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
```

[28] The Java language considered this such a bad idea that it flags such code as an error.

[29] OK, you probably could by fooling around with pointers, but you'd be very, very bad.

```
  //! goto jump1; // Error: goto bypasses init
 }
 X x1;  // Constructor called here
jump1:
 switch(i) {
  case 1 :
   X x2;  // Constructor called here
   break;
 //! case 2 : // Error: case bypasses init
   X x3;  // Constructor called here
   break;
 }
}

int main() {
 f(9);
 f(11);
}///:~
```

In the above code, both the **goto** and the **switch** can potentially jump past the sequence point where a constructor is called. That object will then be in scope even if the constructor hasn't been called, so the compiler gives an error message. This once again guarantees that an object cannot be created unless it is also initialized.

All the storage allocation discussed here happens, of course, on the stack. The storage is allocated by the compiler by moving the stack pointer "down" (a relative term, which may indicate an increase or decrease of the actual stack pointer value, depending on your machine). Objects can also be allocated on the heap using **new**, which is something we'll explore further in Chapter XX.

Stash with constructors and destructors

The examples from previous chapters have obvious functions that map to constructors and destructors: **initialize( )** and **cleanup( )**. Here's the **Stash** header using constructors and destructors:

```
//: C06:Stash2.h
// With constructors & destructors
```

```
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

The only member function definitions that are changed are **initialize( )** and **cleanup( )**, which have been replaced with a constructor and destructor:

```
//: C06:Stash2.cpp {O}
// Constructors & destructors
#include "Stash2.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
```

```
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  assert(0 <= index && index < next);
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Old storage
  storage = b; // Point to new memory
  quantity = newQuantity;
}

Stash::~Stash() {
  if(storage != 0) {
   cout << "freeing storage" << endl;
   delete []storage;
  }
} ///:~
```

Looking at **inflate( )**, you might ask why the "primitive" **assert( )** is still being used after the **require.h** functions have already been introduced. The distinction is important: in this book, **assert( )** will be used to watch for programmer errors. This makes sense because the output of a failed **assert( )** is not particularly end-user friendly and should only be seen by

programmers, while the **require.h** functions (which will be shown later in the book) are specifically designed to be reasonably useful for end-users.

Because **inflate( )** is private, the only way an **assert( )** could occur is if one of the other member functions accidentally passed an incorrect value to **inflate( )**. If you are certain this can't happen, you could consider removing the **assert( )**, but you might keep in mind that until the class is stable, there's always the possibility that new code might be added to the class which could cause errors. The cost of the **assert( )** is low (and can be removed by defining **NDEBUG**) and the value of code robustness is high.

Notice, in the following test program, how the definitions for **Stash** objects appear right before they are needed, and how the initialization appears as part of the definition, in the constructor argument list:

```
//: C06:Stash2Test.cpp
//{L} Stash2
// Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
```

```
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Also notice how the **cleanup( )** calls have been eliminated, but the destructors are still automatically called when **intStash** and **stringStash** go out of scope.

Stack with constructors & destructors

Reimplementing the linked list (inside **Stack**) with constructors and destructors shows up a significant problem. Here's the modified header file:

```
//: C06:Stack3.h
// With constructors/destructors
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
  void* pop();
};
#endif // STACK3_H ///:~
```

Not only does **Stack** have a constructor and destructor, but so does the nested class **Link**:

```
//: C06:Stack3.cpp {O}
// Constructors/destructors
#include "Stack3.h"
```

```
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() {
  delete data;
}

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() { return head->data; }

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  Link* cursor = head;
  while(head) {
    cursor = cursor->next;
    delete head;
    head = cursor;
  }
  head = 0; // Officially empty
} ///:~
```

The **Link::Link( )** constructor simply initializes the **data** and **next** pointers, so in **Stack::push( )** the line

```
head = new Link(dat,head);
```

not only allocates a new link (using dynamic object creation with the keyword **new**, introduced earlier in the book), but it also neatly initializes the pointers for that link.

Because the allocation and cleanup are hidden within **Stack** – it's part of the underlying implementation – you don't see the effect in the test program:

```
//: C06:Stack3Test.cpp
//{L} Stack3
// Constructors/destructors
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Read file and store lines in the stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Pop the lines from the stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << s << endl;
    delete s;
  }
} ///:~
```

The constructor and destructor for **textlines** are called automatically, so the user of the class can focus on what to do with the object and not worry about whether or not it will be properly initialized and cleaned up.

Aggregate initialization

An *aggregate* is just what it sounds like: a bunch of things clumped together. This definition includes aggregates of mixed types, like **struct**s and **class**es. An array is an aggregate of a single type.

Initializing aggregates can be error-prone and tedious. C++ *aggregate initialization* makes it much safer. When you create an object that's an aggregate, all you must do is make an assignment, and the initialization will be taken care of by the compiler. This assignment comes in several flavors, depending on the type of aggregate you're dealing with, but in all cases the elements in the assignment must be surrounded by curly braces. For an array of built-in types this is quite simple:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

If you try to give more initializers than there are array elements, the compiler gives an error message. But what happens if you give *fewer* initializers, such as

```
int b[6] = {0};
```

Here, the compiler will use the first initializer for the first array element, and then use zero for all the elements without initializers. Notice this initialization behavior doesn't occur if you define an array without a list of initializers. So the above expression is a very succinct way to initialize an array to zero, without using a **for** loop, and without any possibility of an off-by-one error (Depending on the compiler, it may also be more efficient than the **for** loop.)

A second shorthand for arrays is *automatic counting*, where you let the compiler determine the size of the array based on the number of initializers:

```
int c[] = { 1, 2, 3, 4 };
```

Now if you decide to add another element to the array, you simply add another initializer. If you can set your code up so it needs to be changed in only one spot, you reduce the chance of errors during modification. But how do you determine the size of the array? The expression **sizeof c / sizeof *c** (size of the entire array divided by the size of the first element)

does the trick in a way that doesn't need to be changed if the array size changes[30]:

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
  c[i]++;
```

Because structures are also aggregates, they can be initialized in a similar fashion. Because a C-style **struct** has all its members **public**, they can be assigned directly:

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c' };
```

If you have an array of such objects, you can initialize them by using a nested set of curly braces for each object:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

Here, the third object is initialized to zero.

If any of the data members are **private** (which is typically the case for a well-designed class in C++), or even if everything's **public** but there's a constructor, things are different. In the above examples, the initializers are assigned directly to the elements of the aggregate, but constructors are a way of forcing initialization to occur through a formal interface. Here, the constructors must be called to perform the initialization. So if you have a **struct** that looks like this,

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

You must indicate constructor calls. The best approach is the explicit one as follows:

```
Y y2[] = { Y(1), Y(2), Y(3) };
```

[30] In chapter XX, you'll see a more succinct calculation of an array size using templates.

You get three objects and three constructor calls. Any time you have a constructor, whether it's a **struct** with all members **public** or a **class** with **private** data members, all the initialization must go through the constructor, even if you're using aggregate initialization.

Here's a second example showing multiple constructor arguments:

```
//: C06:Multiarg.cpp
// Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

Notice that it looks like an explicit constructor is called for each object in the array.

Default constructors

A *default constructor* is one that can be called with no arguments. A default constructor is used to create a "vanilla object," but it's also very important when the compiler is told to create an object but isn't given any

details. For example, if you take the class **Y** defined previously and use it in a definition like this,

```
Y y4[2] = { Y(1) };
```

the compiler will complain that it cannot find a default constructor. The second object in the array wants to be created with no arguments, and that's where the compiler looks for a default constructor. In fact, if you simply define an array of **Y** objects,

```
Y y5[7];
```

or an individual object,

```
Y y;
```

the compiler will complain because it must have a default constructor to initialize every object in the array. (Remember, if you have a constructor the compiler ensures it is *always* called, regardless of the situation.)

The default constructor is so important that *if* (and only if) there are no constructors for a structure (**struct** or **class**), the compiler will automatically create one for you. So this works:

```
//: C06:AutoDefaultConstructor.cpp
// Automatically-generated default constructor

class V {
  int i;  // private
}; // No constructor

int main() {
  V v, v2[10];
} ///:~
```

If any constructors are defined, however, and there's no default constructor, the above object definitions will generate compile-time errors.

You might think that the default constructor should do some intelligent initialization, like setting all the memory for the object to zero. But it doesn't – that would add extra overhead but be out of the programmer's control. If you want the memory to be initialized to zero, you must do it yourself.

Although the compiler will create a default constructor for you, the behavior of the automatically-generated constructor is rarely what you want. You should treat this feature as a safety net, but use it sparingly –

in general, you should define your constructors explicitly and not allow the compiler to do it for you.

Summary

The seemingly elaborate mechanisms provided by C++ should give you a strong hint about the critical importance placed on initialization and cleanup in the language. As Stroustrup was designing C++, one of the first observations he made about productivity in C was that a significant portion of programming problems are caused by improper initialization of variables. These kinds of bugs are very hard to find, and similar issues apply to improper cleanup. Because constructors and destructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created and destroyed without the proper constructor and destructor calls), you get complete control and safety.

Aggregate initialization is included in a similar vein – it prevents you from making typical initialization mistakes with aggregates of built-in types and makes your code more succinct.

Safety during coding is a big issue in C++. Initialization and cleanup are an important part of this, but you'll also see other safety issues as the book progresses.

Exercises

1. Write a simple class called **Simple** with a constructor that prints something to tell you that it's been called. In **main( )** make an object of your class.
2. Add a destructor to the previous example that prints out a message to tell you that it's been called.
3. Modify the previous example so that the class contains an **int** member. Modify the constructor so that it takes an **int** argument which it stores in the class member. Both the constructor and destructor should print out the **int** value as part of their message, so you can see the objects as they are created and destroyed.
4. Demonstrate that destructors are still called even when **goto** is used to jump out of a loop.
5. Write two **for** loops that print out values from zero to 10. In the first, define the loop counter before the **for** loop, and in

the second define the loop counter in the control expression of the **for** loop. For the second part of this exercise, modify the identifier in the second **for** loop so that it as the same name as the loop counter for the first and see what your compiler does.

6. Modify the **Handle.h**, **Handle.cpp**, and **UseHandle.cpp** files at the end of Chapter 5 to use constructors and destructors.
7. Use aggregate initialization to create an array of **double** where you specify the size of the array but do not provide enough elements. Print out this array using **sizeof** to determine the size of the array. Now create an array of **double** using aggregate intialization *and* automatic counting. Print out the array.
8. Use aggregate initialization to create an array of **string** objects. Create a **Stack** to hold these **string**s and step through your array, pushing each **string** on your **Stack**. Finally, **pop** the **string**s off your **Stack** and print each one.
9. Demonstrate automatic counting and aggregate initialization with an array of objects of the class you created in Exercise 3. Add a member function to that class that prints a message. Calculate the size of the array and move through it, calling your new member function.
10. Create a class without any constructors, and show that you can create objects with the default constructor. Now create a nondefault constructor (one with an argument) for the class, and try compiling again. Explain what happened.

7: Function overloading & default arguments

One of the important features in any programming language is the convenient use of names.

When you create an object (a variable), you give a name to a region of storage. A function is a name for an action. By making up names to describe the system at hand, you create a program that is easier for people to understand and change. It's a lot like writing prose – the goal is to communicate with your readers.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings, depending on context. That is, a single word has multiple meanings – it's *overloaded*. This is very useful, especially when it comes to trivial differences. You say "wash the shirt, wash the car." It would be silly to be forced to say, "shirt_wash the shirt, car_wash the car" just so the hearer doesn't have to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. We don't need unique identifiers – we can deduce meaning from context.

Most programming languages, however, require that you have a unique identifier for each function. If you have three different types of data that you want to print: **int**, **char**, and **float**, you generally have to create three different function names, for example, **print_int( )**, **print_char( )**, and **print_float( )**. This loads extra work on you as you write the program, and on readers as they try to understand it.

In C++, another factor forces the overloading of function names: the constructor. Because the constructor's name is predetermined by the name of the class, it would seem that there can be only one constructor. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way and also by reading information from a file. You need two constructors, one that takes no arguments (the default constructor) and one that takes a **string** as an argument, which is the name of the file to initialize the object. Both are constructors, so they must have the same name: the name of the class. Thus function overloading is essential to allow the same function name – the constructor in this case – to be used with different argument types.

Although function overloading is a must for constructors, it's a general convenience and can be used with any function, not just class member functions. In addition, function overloading means that if you have two libraries that contain functions of the same name, they won't conflict as long as the argument lists are different. We'll look at all these factors in detail throughout this chapter.

The theme of this chapter is convenient use of function names. Function overloading allows you to use the same name for different functions, but there's a second way to make calling a function more convenient. What if you'd like to call the same function in different ways? When functions have long argument lists, it can become tedious to write (and confusing to read) the function calls when most of the arguments are the same for all the calls. A very commonly used feature in C++ is called *default arguments*. A default argument is one the compiler inserts if it isn't specified in the function call. Thus the calls **f("hello")**, **f("hi", 1)** and **f("howdy", 2, 'c')** can all be calls to the same function. They could also be calls to three overloaded functions, but when the argument lists are this similar, you'll usually want similar behavior, which calls for a single function.

Function overloading and default arguments really aren't very complicated. By the time you reach the end of this chapter, you'll

understand when to use them and the underlying mechanisms that implement them during compiling and linking.

More name decoration

In Chapter 4 the concept of *name decoration* was introduced. In the code

```
void f();
class X { void f(); };
```

the function **f( )** inside the scope of **class X** does not clash with the global version of **f( )**. The compiler performs this scoping by manufacturing different internal names for the global version of **f( )** and **X::f( )**. In Chapter 4 it was suggested that the names are simply the class name "decorated" together with the function name, so the internal names the compiler uses might be **_f** and **_X_f**. However, it turns out that function name decoration involves more than the class name.

Here's why. Suppose you want to overload two function names

```
void print(char);
void print(float);
```

It doesn't matter whether they are both inside a class or at the global scope. The compiler can't generate unique internal identifiers if it uses only the scope of the function names. You'd end up with **_print** in both cases. The idea of an overloaded function is that you use the same function name, but different argument lists. Thus, for overloading to work the compiler must decorate the function name with the names of the argument types. The above functions, defined at global scope, produce internal names that might look something like **_print_char** and **_print_float**. It's worth noting there is no standard for the way names must be decorated by the compiler, so you will see very different results from one compiler to another. (You can see what it looks like by telling the compiler to generate assembly-language output.) This, of course, causes problems if you want to buy compiled libraries for a particular compiler and linker – but even if name decoration were standardized, there would be other roadblocks because of the way different compilers generate code.

That's really all there is to function overloading: You can use the same function name for different functions, as long as the argument lists are different. The compiler decorates the name, the scope, and the argument lists to produce internal names for it and the linker to use.

Overloading on return values

It's common to wonder "why just scopes and argument lists? Why not return values?" It seems at first that it would make sense to also decorate the return value with the internal function name. Then you could overload on return values, as well:

```
void f();
int f();
```

This works fine when the compiler can unequivocally determine the meaning from the context, as in **int x = f( );**. However, in C you've always been able to call a function and ignore the return value. How can the compiler distinguish which call is meant in this case? Possibly worse is the difficulty the reader has in knowing which function call is meant. Overloading solely on return value is a bit too subtle, and thus isn't allowed in C++.

Type-safe linkage

There is an added benefit to all this name decoration. A particularly sticky problem in C occurs when the client programmer misdeclares a function, or, worse, a function is called without declaring it first, and the compiler infers the function declaration from the way it is called. Sometimes this function declaration is correct, but when it isn't, it can be a very difficult bug to find.

Because all functions *must* be declared before they are used in C++, the opportunity for this problem to pop up is greatly diminished. The C++ compiler refuses to declare a function automatically for you, so it's likely you will include the appropriate header file. However, if for some reason you still manage to misdeclare a function, either by declaring by hand or including the wrong header file (perhaps one that is out of date), the name decoration provides a safety net that is often referred to as *type-safe linkage*.

Consider the following scenario. In one file is the definition for a function:

```
//: C07:Def.cpp {O}
// Function definition
void f(int) {}
///:~
```

In the second file, the function is misdeclared and then called:

```
//: C07:Use.cpp
//{L} Def
// Function misdeclaration
void f(char);

int main() {
//!  f(1); // Causes a linker error
} ///:~
```

Even though you can see that the function is actually **f(int)**, the compiler doesn't know this because it was told – through an explicit declaration – that the function is **f(char)**. Thus, the compilation is successful. In C, the linker would also be successful, but *not* in C++. Because the compiler decorates the names, the definition becomes something like **f_int**, whereas the use of the function is **f_char**. When the linker tries to resolve the reference to **f_char**, it can only find **f_int**, and it gives you an error message. This is type-safe linkage. Although the problem doesn't occur all that often, when it does it can be incredibly difficult to find, especially in a large project. This is one of the cases where you can easily find a difficult error in a C program simply by running it through the C++ compiler.

Overloading example

We can now modify earlier examples to use function overloading. As stated before, an immediately useful place for overloading is in constructors. You can see this in the following version of the **Stash** class:

```
//: C07:Stash3.h
// Function overloading
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Size of each space
  int quantity;  // Number of storage spaces
  int next;      // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Zero quantity
  Stash(int size, int initQuant);
```

```
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

The first **Stash( )** constructor is the same as before, but the second one has a **Quantity** argument to indicate the initial number of storage places to be allocated. In the definition, you can see that the internal value of **quantity** is set to zero, along with the **storage** pointer. In the second constructor, the call to **inflate(initQuant)** increases **quantity** to the allocated size:

```
//: C07:Stash3.cpp {O}
// Function overloading
#include "Stash3.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuant) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuant);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
}
```

```
int Stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(increment);
  // Copy element into storage,
  // starting at next empty space:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Index number
}

void* Stash::fetch(int index) {
  assert(0 <= index && index < next);
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

When you use the first constructor no memory is allocated for **storage**. The allocation happens the first time you try to **add( )** an object and any time the current block of memory is exceeded inside **add( )**.

Both constructors are exercised in the test program:

```
//: C07:Stash3Test.cpp
```

```
//{L} Stash3
// Function overloading
#include "Stash3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

The constructor call for **stringStash** uses a second argument; presumably you know something special about the specific problem you're solving that allows you to choose an initial size for the **Stash**.

unions

As you've seen, the only difference between **struct** and **class** in C++ is that **struct** defaults to **public** and **class** defaults to **private**. A **struct** can also have constructors and destructors, as you might expect. But it turns out that a **union** can also have a constructor, destructor, member

functions and even access control. You can again see the use and benefit of overloading in the following example:

```
//: C07:UnionClass.cpp
// Unions with constructors and member functions
#include<iostream>
using namespace std;

union U {
private: // Access control too!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

You might think from the above code that the only difference between a **union** and a **class** is the way the data is stored (that is, the **int** and **float** are overlaid on the same piece of storage). However, a **union** cannot be used as a base class during inheritance, which is quite limiting from an object-oriented design standpoint (you'll learn about inheritance in Chapter XX).

Although the member functions civilize access to the **union** somewhat, there is still no way to prevent the client programmer from selecting the wrong element type once the **union** is initialized. In the above example, you could say **X.read_float( )** even though it is inappropriate. However, a "safe" **union** can be encapsulated in a class. In the following example, notice how the **enum** clarifies the code, and how overloading comes in handy with the constructors:

```
//: C07:SuperVar.cpp
// A super-variable
#include <iostream>
using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Define one
  union {  // Anonymous union
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar:: SuperVar(char ch) {
  vartype = character;
  c = ch;
}

SuperVar:: SuperVar(int ii) {
  vartype = integer;
  i = ii;
}

SuperVar:: SuperVar(float ff) {
  vartype = floating_point;
```

```
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
      cout << "float: " << f << endl;
      break;
  }
}

int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

In the above code, the **enum** has no type name (it is an untagged enumeration). This is acceptable if you are going to immediately define instances of the **enum**, as is done here. There is no need to refer to the **enum's** type name in the future, so the type name is optional.

The **union** has no type name and no variable name. This is called an *anonymous union*, and creates space for the **union** but doesn't require accessing the **union** elements with a variable name and the dot operator. For instance, if your anonymous **union** is:

```
union { int i, float f };
```

you access members by saying:

```
i = 12;
f = 1.22;
```

just like other variables. The only difference is that both variables occupy the same space. If the anonymous **union** is at file scope (outside all functions and classes) then it must be declared **static** so it has internal linkage.

Although **SuperVar** is now safe, its usefulness is a bit dubious because the reason for using a **union** in the first place is to save space, and the addition of **vartype** takes up quite a bit of space relative to the data in the **union**, so the savings are effectively eliminated. There are a couple of alternatives to make this scheme workable: if the **vartype** was controlling more than one **union** instance – if they were all the same type – then you'd only need one for the group and it wouldn't take up more space. A more useful approach is to have **#ifdef**s around all the **vartype** code which can then guarantee things are being used correctly during development and testing. For shipping code, the extra space and time overhead can be eliminated.

Default arguments

Examine the two constructors for **Stash( )**. They don't seem all that different, do they? In fact, the first constructor seems to be a special case of the second one with the initial **size** set to zero. It's a bit of a waste of effort to create and maintain two different versions of a similar function.

C++ provides a remedy with *default arguments*. A default argument is a value given in the declaration that the compiler automatically inserts if you don't provide a value in the function call. In the **Stash** example, we can replace the two functions:

```
Stash(int size); // Zero quantity
Stash(int size, int quantity);
```

with the single function:

```
Stash(int size, int quantity = 0);
```

The **Stash(int)** definition is simply removed – all that is necessary is the single **Stash(int, int)** definition.

Now, the two object definitions

```
Stash A(100), B(100, 0);
```

will produce exactly the same results. The identical constructor is called in both cases, but for **A**, the second argument is automatically substituted by the compiler when it sees the first argument is an **int** and that there is no second argument. The compiler has seen the default argument, so it knows it can still make the function call if it substitutes this second argument, which is what you've told it to do by making it a default.

Default arguments are a convenience, as function overloading is a convenience. Both features allow you to use a single function name in different situations. The difference is that with default arguments the compiler is substituting arguments when you don't want to put them in yourself. The preceding example is a good place to use default arguments instead of function overloading; otherwise you end up with two or more functions that have similar signatures and similar behaviors. If the functions have very different behaviors, it doesn't usually make sense to use default arguments (for that matter, you might want to question whether two functions with very different behaviors should have the same name).

There are two rules you must be aware of when using default arguments. First, only trailing arguments may be defaulted. That is, you can't have a default argument followed by a nondefault argument. Second, once you start using default arguments in a particular function call, all the subsequent arguments in that function's argument list must be defaulted (this follows from the first rule).

Default arguments are only placed in the declaration of a function (typically placed in a header file). The compiler must see the default value before it can use it. Sometimes people will place the commented values of the default arguments in the function definition, for documentation purposes

```
void fn(int x /* = 0 */) { // ...
```

Placeholder arguments

Arguments in a function declaration can be declared without identifiers. When these are used with default arguments, it can look a bit funny. You can end up with

```
void f(int x, int = 0, float = 1.1);
```

In C++ you don't need identifiers in the function definition, either:

```
void f(int x, int, float flt) { /* ... */ }
```

In the function body, **x** and **flt** can be referenced, but not the middle argument, because it has no name. Function calls must still provide a value for the placeholder, though: **f(1)** or **f(1,2,3.0)**. This syntax allows you to put the argument in as a placeholder without using it. The idea is that you might want to change the function definition to use the placeholder later, without changing all the code where the function is called. Of course, you can accomplish the same thing by using a named

argument, but if you define the argument for the function body without using it, most compilers will give you a warning message, assuming you've made a logical error. By intentionally leaving the argument name out, you suppress this warning.

More important, if you start out using a function argument and later decide that you don't need it, you can effectively remove it without generating warnings, and yet not disturb any client code that was calling the previous version of the function.

Choosing overloading vs. default arguments

Both function overloading and default arguments provide a convenience for calling function names. However, it can seem confusing at times to know which technique to use. For example, consider the following tool which is designed to automatically manage blocks of memory for you:

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

A **Mem** object holds a block of **byte**s, and makes sure that you have enough storage. The default constructor doesn't allocate any storage, and the second constructor ensures that there is **sz** storage in the **Mem** object. The destructor releases the storage, **msize( )** tells you how many

bytes there are currently in the **Mem** object, and **pointer( )** produces a pointer to the starting address of the storage (**Mem** is a fairly low-level tool). There's an overloaded version of **pointer( )** where the client programmer can say that they want a pointer to a block of bytes that is at least **minSize** large, and the member function ensures this.

Both the constructor and the **pointer( )** member function use the **private ensureMinSize( )** member function to increase the size of the memory block (notice that it's not safe to hold the result of **pointer( )** if the memory is resized).

Here's the implementation of the class:

```
//: C07:Mem.cpp {O}
#include "Mem.h"
#include <cstring>
using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
```

```
  ensureMinSize(minSize);
  return mem;
} ///:~
```

You can see that **ensureMinSize( )** is the only function responsible for allocating memory, and that it is used from the second constructor and the second overloaded form of **pointer( )**. Inside **ensureMinSize( )**, nothing needs to be done if the **size** is large enough. If new storage must be allocated in order to make the block bigger (which is also the case when the block is of size zero, after default construction), the new "extra" portion is set to zero using the Standard C library function **memset( )**, which was introduced earlier in the book. The subsequent function call is to the Standard C library function **memcpy( )**, which in this case copies the existing bytes from **mem** to **newmem** (typically in a very efficient fashion). Finally, the old memory is deleted and the new memory and sizes are assigned to the appropriate members.

The **Mem** class is designed to be used as a tool within other classes, to simplify their memory management (it could also be used to hide a more sophisticated memory-management system provided, for example, by the operating system). Appropriately, it is tested here by creating a very simple "string" class:

```
//: C07:MemTest.cpp
// Testing the Mem class
//{L} Mem
#include "Mem.h"
#include <cstring>
#include <iostream>
using namespace std;

class myString {
  Mem* buf;
public:
  myString();
  myString(char* str);
  ~myString();
  void concat(char* str);
  void print(ostream& os);
};

myString::myString() {  buf = 0; }

myString::myString(char* str) {
```

```
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void myString::concat(char* str) {
  if(!buf) buf = new Mem();
  strcat((char*)buf->pointer(
    buf->msize() + strlen(str) + 1), str);
}

void myString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

myString::~myString() { delete buf; }

int main() {
  myString s("My test string");
  s.print(cout);
  s.concat(" some additional stuff");
  s.print(cout);
  myString s2;
  s2.concat("Using default constructor");
  s2.print(cout);
} ///:~
```

All you can do with this class is to create a **myString**, concatenate text, and print to an **ostream**. The class only contains a pointer to a **Mem**, but note the distinction between the default constructor, which sets the pointer to zero, and the second constructor, which creates a **Mem** and copies data into it. The advantage of the default constructor is that you can create, for example, a large array of empty **myString** objects very cheaply, since the size of each object is only one pointer and the only overhead of the default constructor is that of assigning to zero. The cost of a **myString** only begins to accrue when you concatenate data; at that point the **Mem** object is created if it hasn't been already. However, if you use the default constructor and never concatenate any data, the destructor call is still safe because calling **delete** for zero is defined such that it does not try to release storage or otherwise cause problems.

If you look at these two constructors it might at first seem like this is a prime candidate for default arguments. However, if you drop the default constructor and write the remaining constructor with a default argument:

```
myString(char* str = "");
```

everything will work correctly, but you'll lose the previous efficiency benefit since a **Mem** object will always be created. To get the efficiency back, you must modify the constructor:

```
myString::myString(char* str) {
  if(!*str) { // Pointing at an empty string
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

This means, in effect, that the default value becomes a flag that causes a separate piece of code to be executed than if a non-default value is used. Although it seems innocent enough with a small constructor like this one, in general this practice can cause problems. If you have to *look* for the default rather than treating it as an ordinary value, that should be a clue that you will end up with effectively two different functions inside a single function body: one version for the normal case, and one for the default. You might as well split it up into two distinct function bodies and let the compiler do the selection. This results in a slight (but usually invisible) increase in efficiency, because the extra argument isn't passed and the extra code for the conditional isn't executed. More importantly, you are keeping the code for two separate functions *in* two separate functions rather than combining them into one using default arguments, which will result in easier maintainability, especially if the functions are large.

On the other hand, consider the **Mem** class. If you look at the definitions of the two constructors and the two **pointer( )** functions, you can see that using default arguments in both cases will not cause the member function definitions to to change at all. Thus the class could easily be:

```
//: C07:Mem2.h
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
```

```
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

Notice that a call to **ensureMinSize(0)** will always be quite efficient.

Although in both of these cases I based some of the decision-making process on the issue of efficiency, you must be careful not to fall into the trap of thinking only about efficiency (fascinating as it is). The most important issue in class design is the interface of the class (its **public** members, which are available to the client programmer). If these produce a class that is easy to use and reuse, then you have a success; you can always tune for efficiency if necessary but the effect of a class that is designed badly because the programmer is over-focused on efficiency issues can be dire. Your primary concern should be that the interface makes sense to those who use it and who read the resulting code. Notice that in **MemTest.cpp** the usage of **myString** does not change regardless of whether a default constructor is used or whether the efficiency is high or low.

Summary

As a guideline, you shouldn't use a default argument as a flag upon which to conditionally execute code. You should instead break the function into two or more overloaded functions if you can. A default argument should be a value you would ordinarily put in that position. It's a value that is more likely to occur than all the rest, so client programmers can generally ignore it or use it only if they want to change it from the default value.

The default argument is included to make function calls easier, especially when those functions have many arguments with typical values. Not only is it much easier to write the calls, it's easier to read them, especially if the class creator can order the arguments so the least-modified defaults appear latest in the list.

An especially important use of default arguments is when you start out with a function with a set of arguments, and after it's been used for a while you discover you need to add arguments. By defaulting all the new arguments, you ensure that all client code using the previous interface is not disturbed.

Exercises

1. Create a **Text** class that contains a **string** object to hold the text of a file. Give it two constructors: a default constructor and a constructor that takes a **string** argument which is the name of the file to open. When the second constructor is used, open the file and read the contents into the **string** member object. Add a member function **contents( )** to return the **string** so (for example) it can be printed. In **main( )**, open a file using **Text** and print the contents.
2. Create a **Message** class with a constructor that takes a single **string** with a default value. Create a private member **string**, and in the constructor simply assign the argument **string** to your internal **string**. Create two overloaded member functions called **print( )**: one that takes no arguments and simply prints the message stored in the object, and one that takes a **string** argument, which it prints in addition to the internal message. Does it make sense to use this approach rather than the one used for the constructor?
3. Determine how to generate assembly output with your compiler, and run experiments to deduce the name-decoration scheme.
4. Create a class that contains four member functions, with 0, 1, 2, and 3 **int** arguments, respectively. Create a **main( )** that makes an object of your class and calls each of the member functions. Now modify the class so it has instead a single member function with all the arguments defaulted. Does this change your **main( )**?
5. Create a function with two arguments and call it from **main( )**. Now make one of the arguments a “placeholder” (no identifier) and see if your call in **main( )** changes
6. Modify **Stash3.h** and **Stash3.cpp** to use default arguments in the constructor. Test the constructor by making two different versions of a **Stash** object.
7. Create a new version of the **Stack** class (from the previous chapter) which contains the default constructor as before, and a second constructor which takes as its arguments an array of pointers to objects and the size of that array. This constructor should move through the array and push each

pointer onto the **Stack**. Test your class with an array of **string**.

8. Modify **SuperVar** so there are **#ifdef**s around all the **vartype** code as described in the section on **enum**. Make **vartype** a regular and **public** enumeration (with no instance) and modify **print( )** so it requires a **vartype** argument to tell it what to do.
9. Implement **Mem2.h** and make sure that the modified class still works with **MemTest.cpp**.
10. Use **class Mem** to implement **Stash**. Note that, because the implementation is **private** and thus hidden from the client programmer, the test code does not need to be modified.
11. In **class Mem**, add a **bool moved( )** member function that takes the result of a call to **pointer( )** and tells you whether the pointer has moved (due to reallocation). Write a **main( )** that tests your **moved( )** member function. Does it make more sense to use something like **moved( )** or to simply call **pointer( )** every time you need to access the memory in **Mem**?

8: Constants

> The concept of *constant* (expressed by the **const** keyword) was created to allow the programmer to draw a line between what changes and what doesn't.

This provides safety and control in a C++ programming project. Since its origin, **const** has taken on a number of different purposes. In the meantime it trickled back into the C language where its meaning was changed. All this can seem a bit confusing at first, and in this chapter you'll learn when, why, and how to use the **const** keyword. At the end there's a discussion of **volatile**, which is a near cousin to **const** (because they both concern change) and has identical syntax.

The first motivation for **const** seems to have been to eliminate the use of preprocessor **#define**s for value substitution. It has since been put to use for pointers, function arguments, return types, class objects and member functions. All of these have slightly different but conceptually compatible meanings and will be looked at in separate sections in this chapter.

Value substitution

When programming in C, the preprocessor is liberally used to create macros and to substitute values. Because the preprocessor simply does text replacement and has no concept nor facility for type checking, preprocessor value substitution introduces subtle problems that can be avoided in C++ by using **const** values.

The typical use of the preprocessor to substitute values for names in C looks like this:

```
#define BUFSIZE 100
```

BUFSIZE is a name that only exists during preprocessing, therefore it doesn't occupy storage and can be placed in a header file to provide a single value for all translation units that use it. It's very important for code maintenance to use value substitution instead of so-called "magic

numbers." If you use magic numbers in your code, not only does the reader have no idea where the numbers come from or what they represent, but if you decide to change a value, you must perform hand editing, and you have no trail to follow to ensure you don't miss one of your values (or accidentally change one you shouldn't).

Most of the time, **BUFSIZE** will behave like an ordinary variable, but not all the time. In addition, there's no type information. This can hide bugs that are very difficult to find. C++ uses **const** to eliminate these problems by bringing value substitution into the domain of the compiler. Now you can say

```
const int bufsize = 100;
```

You can use **bufsize** anyplace where the compiler must know the value at compile time. The compiler can use **bufsize** to perform *constant folding*, which means the compiler will reduce a complicated constant expression to a simple one by performing the necessary calculations at compile time. This is especially important in array definitions:

```
char buf[bufsize];
```

You can use **const** for all the built-in types (**char**, **int**, **float**, and **double**) and their variants (as well as class objects, as you'll see later in this chapter). Because of subtle bugs introduced by the preprocessor, you should always use **const** instead of **#define** value substitution.

const in header files

To use **const** instead of **#define**, you must be able to place **const** definitions inside header files as you can with **#define**. This way, you can place the definition for a **const** in a single place and distribute it to translation units by including the header file. A **const** in C++ defaults to *internal linkage*; that is, it is visible only within the file where it is defined and cannot be seen at link time by other translation units. You must always assign a value to a **const** when you define it, *except* when you make an explicit declaration using **extern**:

```
extern const bufsize;
```

Normally, the C++ compiler avoids creating storage for a **const**, but instead holds the definition in its symbol table. When you use **extern** with **const**, however, you force storage to be allocated (this is also true for certain other cases, such as taking the address of a **const**). Storage must be allocated because **extern** says "use external linkage" and that means

that several translation units must be able to refer to the item, which requires it to have storage.

In the ordinary case, when **extern** is not part of the definition, no storage is allocated. When the **const** is used, it is simply folded in at compile time.

The goal of never allocating storage for a **const** also fails with complicated structures. Whenever the compiler must allocate storage, constant folding is prevented (since there's no way for the compiler to know for sure what the value of that storage is – if it could know that, it wouldn't need to allocate the storage).

Because the compiler cannot always avoid allocating storage for a **const**, **const** definitions *must* default to internal linkage, that is, linkage only *within* that particular translation unit. Otherwise, linker errors would occur with complicated **const**s because they cause storage to be allocated in multiple **cpp** files. The linker would then see the same definition in multiple object files, and complain. Because a **const** defaults to internal linkage, the linker doesn't try to link those definitions across translation units, and there are no collisions. With built-in types, which are used in the majority of cases involving constant expressions, the compiler can always perform constant folding.

Safety **const**s

The use of **const** is not limited to replacing **#define**s in constant expressions. If you initialize a variable with a value that is produced at runtime and you know it will not change for the lifetime of that variable, it is good programming practice to make it a **const** so the compiler will give you an error message if you accidentally try to change it. Here's an example:

```
//: C08:Safecons.cpp
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
```

```
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

You can see that **i** is a compile-time **const**, but **j** is calculated from **i**. However, because **i** is a **const**, the calculated value for **j** still comes from a constant expression and is itself a compile-time constant. The very next line requires the address of **j** and therefore forces the compiler to allocate storage for **j**. Yet this doesn't prevent the use of **j** in the determination of the size of **buf** because the compiler knows **j** is **const** and that the value is valid even if storage was allocated to hold that value at some point in the program.

In **main( )**, you see a different kind of **const** in the identifier **c** because the value cannot be known at compile time. This means storage is required, and the compiler doesn't attempt to keep anything in its symbol table (the same behavior as in C). The initialization must still happen at the point of definition, and once the initialization occurs, the value cannot be changed. You can see that **c2** is calculated from **c** and also that scoping works for **const**s as it does for any other type – yet another improvement over the use of **#define**.

As a matter of practice, if you think a value shouldn't change, you should make it a **const**. This not only provides insurance against inadvertent changes, it also allows the compiler to generate more efficient code by eliminating storage and memory reads.

Aggregates

It's possible to use **const** for aggregates, but you're virtually assured that the compiler will not be sophisticated enough to keep an aggregate in its symbol table, so storage will be allocated. In these situations, **const** means "a piece of storage that cannot be changed." However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time. In the following code, you can see the statements that are illegal:

```
//: C08:Constag.cpp
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
```

```
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {} ///:~
```

In an array definition, the compiler must be able to generate code that moves the stack pointer to accommodate the array. In both of the illegal definitions above, the compiler complains because it cannot find a constant expression in the array definition.

Differences with C

Constants were introduced in early versions of C++ while the Standard C specification was still being finished. Although the C committee then decided to include **const** in C, somehow it came to mean for them "an ordinary variable that cannot be changed." In C, a **const** always occupies storage and its name is global. The C compiler cannot treat a **const** as a compile-time constant. In C, if you say

```
const bufsize = 100;
char buf[bufsize];
```

you will get an error, even though it seems like a rational thing to do. Because **bufsize** occupies storage somewhere, the C compiler cannot know the value at compile time. You can optionally say

```
const bufsize;
```

in C, but not in C++, and the C compiler accepts it as a declaration indicating there is storage allocated elsewhere. Because C defaults to external linkage for **const**s, this makes sense. C++ defaults to internal linkage for **const**s so if you want to accomplish the same thing in C++, you must explicitly change the linkage to external using **extern**:

```
extern const bufsize; // Declaration only
```

This line also works in C.

In C++, a **const** doesn't necessarily create storage. In C a **const** always creates storage. Whether or not storage is reserved for a **const** in C++ depends on how it is used. In general, if a **const** is used simply to replace a name with a value (just as you would use a **#define**), then storage doesn't have to be created for the **const**. If no storage is created (this depends on the complexity of the data type and the sophistication of the compiler), the values may be folded into the code for greater efficiency after type checking, not before, as with **#define**. If, however, you take an address of a **const** (even unknowingly, by passing it to a function that

takes a reference argument) or you define it as **extern**, then storage is created for the **const**.

In C++, a **const** that is outside all functions has file scope (i.e., it is invisible outside the file). That is, it defaults to internal linkage. This is very different from all other identifiers in C++ (and from **const** in C!) that default to external linkage. Thus, if you declare a **const** of the same name in two different files and you don't take the address or define that name as **extern**, the ideal C++ compiler won't allocate storage for the **const**, but simply fold it into the code. Because **const** has implied file scope, you can put it in C++ header files with no conflicts at link time.

Since a **const** in C++ defaults to internal linkage, you can't just define a **const** in one file and reference it as an **extern** in another file. To give a **const** external linkage so it can be referenced from another file, you must explicitly define it as extern, like this:

```
extern const int x = 1;
```

Notice that by giving it an initializer and saying it is **extern**, you force storage to be created for the **const** (although the compiler still has the option of doing constant folding here). The initialization establishes this as a definition, not a declaration. The declaration:

```
extern const int x;
```

in C++ means that the definition exists elsewhere (again, this is not necessarily true in C). You can now see why C++ requires a **const** definition to have an initializer: the initializer distinguishes a declaration from a definition (in C it's always a definition, so no initializer is necessary). With an **extern const** declaration, the compiler cannot do constant folding because it doesn't know the value.

The C approach to **const** is not very useful, and if you want to use a named value inside a constant expression (one that must be evaluated at compile time), C almost *forces* you to use **#define** in the preprocessor.

Pointers

Pointers can be made **const**. The compiler will still endeavor to prevent storage allocation and do constant folding when dealing with **const** pointers, but these features seem less useful in this case. More importantly, the compiler will tell you if you attempt to change a **const** pointer, which adds a great deal of safety.

When using **const** with pointers, you have two options: **const** can be applied to what the pointer is pointing to, or the **const** can be applied to the address stored in the pointer itself. The syntax for these is a little confusing at first but becomes comfortable with practice.

Pointer to **const**

The trick with a pointer definition, as with any complicated definition, is to read it starting at the identifier and work your way out. The **const** specifier binds to the thing it is "closest to." So if you want to prevent any changes to the element you are pointing to, you write a definition like this:

```
const int* u;
```

Starting from the identifier, we read "**u** is a pointer, which points to a **const int**." Here, no initialization is required because you're saying that **u** can point to anything (that is, it is not **const**), but the thing it points to cannot be changed.

Here's the mildly confusing part. You might think that to make the pointer itself unchangeable, that is, to prevent any change to the address contained inside **u**, you would simply move the **const** to the other side of the **int** like this:

```
int const* v;
```

It's not all that crazy to think that this should read "**v** is a **const** pointer to an **int**." However, the way it *actually* reads is "**v** is an ordinary pointer to an **int** that happens to be **const**." That is, the **const** has bound itself to the **int** again, and the effect is the same as the previous definition. The fact that these two definitions are the same is the confusing point; to prevent this confusion on the part of your reader, you should probably stick to the first form.

const pointer

To make the pointer itself a **const**, you must place the **const** specifier to the right of the *****, like this:

```
int d = 1;
int* const w = &d;
```

Now it reads: "**w** is a pointer which is **const**, that points to an **int**." Because the pointer itself is now the **const**, the compiler requires that it

be given an initial value that will be unchanged for the life of that pointer. It's OK, however, to change what that value points to by saying

```
*w = 2;
```

You can also make a **const** pointer to a **const** object using either of two legal forms:

```
int d = 1;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
```

Now neither the pointer nor the object can be changed.

Some people argue that the second form is more consistent because the **const** is always placed to the right of what it modifies. You'll have to decide which is clearer for your particular coding style.

Here are the above lines in a compileable file:

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

Formatting

This book makes a point of only putting one pointer definition on a line, and initializing each pointer at the point of definition whenever possible. Because of this, the formatting style of "attaching" the '*****' to the data type is possible:

```
int* u = &i;
```

as if **int*** were a discrete type unto itself. This makes the code easier to understand, but unfortunately that's not actually the way things work. The '*****' in fact binds to the identifier, not the type. It can be placed anywhere between the type name and the identifier. So you could do this:

```
int *u = &i, v = 0;
```

which creates an **int* u**, as before, and a nonpointer **int v**. Because readers often find this confusing, it is best to follow the form shown in this book.

Assignment and type checking

C++ is very particular about type checking, and this extends to pointer assignments. You can assign the address of a non-**const** object to a **const** pointer because you're simply promising not to change something that is OK to change. However, you can't assign the address of a **const** object to a non-**const** pointer because then you're saying you might change the object via the pointer. Of course, you can always use a cast to force such an assignment, but this is bad programming practice because you are then breaking the **const**ness of the object, along with any safety promised by the **const**. For example:

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} ///:~
```

Although C++ helps prevent errors it does not protect you from yourself if you want to break the safety mechanisms.

Character array literals

The place where strict **const**ness is not enforced is with character array literals. You can say

```
char* cp = "howdy";
```

and the compiler will accept it without complaint. This is technically an error because a character array literal (**"howdy"** in this case) is created by the compiler as a constant character array, and the result of the quoted character array is its starting address in memory.

So character array literals are actually constant character arrays. Of course, the compiler lets you get away with treating them as non-**const** because there's so much existing C code that relies on this. However, if you try to change the values in a character array literal, the behavior is undefined, although it will probably work on many machines.

Function arguments & return values

The use of **const** to specify function arguments and return values is another place where the concept of constants can be confusing. If you are passing objects *by value*, specifying **const** has no meaning to the client (it means that the passed argument cannot be modified inside the function). If you are returning an object of a user-defined type by value as a **const**, it means the returned value cannot be modified. If you are passing and returning *addresses*, **const** is a promise that the destination of the address will not be changed.

Passing by **const** value

You can specify that function arguments are **const** when passing them by value, such as

```
void f1(const int i) {
  i++; // Illegal -- compile-time error
}
```

but what does this mean? You're making a promise that the original value of the variable will not be changed by the function **f1( )**. However, because the argument is passed by value, you immediately make a copy of the original variable, so the promise to the client is implicitly kept.

Inside the function, the **const** takes on meaning: the argument cannot be changed. So it's really a tool for the creator of the function, and not the caller.

To avoid confusion to the caller, you can make the argument a **const** *inside* the function, rather than in the argument list. You could do this with a pointer, but a nicer syntax is achieved with the *reference*, a subject that will be fully developed in Chapter XX. Briefly, a reference is like a constant pointer that is automatically dereferenced, so it has the effect of being an alias to an object. To create a reference, you use the **&** in the definition. So the non-confusing function definition looks like this:

```
void f2(int ic) {
  const int& i = ic;
  i++;  // Illegal -- compile-time error
}
```

Again, you'll get an error message, but this time the **const**ness of the local object is not part of the function signature; it only has meaning to the implementation of the function and therefore it's hidden from the client.

Returning by **const** value

A similar truth holds for the return value. If you say that a function's return value is **const**:

```
const int g();
```

you are promising that the original variable (inside the function frame) will not be modified. And again, because you're returning it by value, it's copied so the original value could never be modified via the return value.

At first, this can make the specification of **const** seem meaningless. You can see the apparent lack of effect of returning **const**s by value in this example:

```
//: C08:Constval.cpp
// Returning consts by value
// has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
} ///:~
```

For built-in types, it doesn't matter whether you return by value as a **const**, so you should avoid confusing the client programmer by leaving off the **const** when returning a built-in type by value.

Returning by value as a **const** becomes important when you're dealing with user-defined types. If a function returns a class object by value as a **const**, the return value of that function cannot be an lvalue (that is, it cannot be assigned to or otherwise modified). For example:

```
//: C08:ConstReturnValues.cpp
// Constant return by value
// Result cannot be used as an lvalue
```

```
class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Pass by non-const reference
  x.modify();
}

int main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
// Causes compile-time errors:
//!  f7(f5());
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

f5( ) returns a non-**const X** object, while **f6( )** returns a **const X** object. Only the non-**const** return value can be used as an lvalue. Thus, it's important to use **const** when returning an object by value if you want to prevent its use as an lvalue.

The reason **const** has no meaning when you're returning a built-in type by value is that the compiler already prevents it from being an lvalue (because it's always a value, and not a variable). Only when you're returning objects of user-defined types by value does it become an issue.

The function **f7( )** takes its argument as a non-**const** *reference* (an additional way of handling addresses in C++ which is the subject of Chapter XX). This is effectively the same as taking a non-**const** pointer; it's just that the syntax is different. The reason this won't compile in C++ is because of the creation of a temporary.

Temporaries

Sometimes, during the evaluation of an expression, the compiler must create *temporary objects*. These are objects like any other: they require storage and they must be constructed and destroyed. The difference is that you never see them – the compiler is responsible for deciding that they're needed and the details of their existence. But there is one thing about temporaries: they're automatically **const**. Because you usually won't be able to get your hands on a temporary object, telling it to do something that will change that temporary is almost certainly a mistake because you won't be able to use that information. By making all temporaries automatically **const**, the compiler informs you when you make that mistake.

In the above example, **f5( )** returns a non-**const X** object. But in the expression:

```
f7(f5());
```

the compiler must manufacture a temporary object to hold the return value of **f5( )** so it can be passed to **f7( )**. This would be fine if **f7( )** took it's argument by value; then the temporary would be copied into **f7( )** and it wouldn't matter what happened to the temporary **X**. However, **f7( )** takes its argument *by reference*, which means in this example takes the address of the temporary **X**. Since **f7( )** doesn't take it's argument by **const** reference, it has permission to modify the temporary object. But the compiler knows that the temporary will vanish as soon as the expression evaluation is complete, and thus any modifications you make to the temporary **X** will be lost. By making all temporary objects automatically **const**, this situation causes a compile-time error so you don't get caught by what would be a very difficult bug to find.

However, notice the expressions that are legal:

```
f5() = X(1);
f5().modify();
```

Although these pass muster for the compiler, they are actually problematic. **f5( )** returns an **X** object, and for the compiler to satisfy the above expressions it must create a temporary to hold that return value.

So in both expressions the temporary object is being modified, and as soon as the expression is over the temporary is cleaned up. As a result, the modifications are lost so this code is probably a bug – but the compiler doesn't tell you anything about it. Expressions like these are simple enough for you to detect the problem, but when things get more complex it's possible for a bug to slip through these cracks.

The way the **const**ness of class objects is preserved is shown later in the chapter.

Passing and returning addresses

If you pass or return an address (either a pointer or a reference), it's possible for the client programmer to take it and modify the original value. If you make the pointer or reference a **const**, you prevent this from happening, which may save you some grief. In fact, whenever you're passing an address into a function, you should make it a **const** if at all possible. If you don't, you're excluding the possibility of using that function with anything that is a **const**.

The choice of whether to return a pointer or reference to a **const** depends on what you want to allow your client programmer to do with it. Here's an example that demonstrates the use of **const** pointers as function arguments and return values:

```
//: C08:ConstPointer.cpp
// Constant pointer arg/return

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static character array:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
```

```
}

int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
} ///:~
```

The function **t( )** takes an ordinary non-**const** pointer as an argument, and **u( )** takes a **const** pointer. Inside **u( )** you can see that attempting to modify the destination of the **const** pointer is illegal, but you can of course copy the information out into a non-**const** variable. The compiler also prevents you from creating a non-**const** pointer using the address stored inside a **const** pointer.

The functions **v( )** and **w( )** test return value semantics. **v( )** returns a **const char*** that is created from a character array literal. This statement actually produces the address of the character array literal, after the compiler creates it and stores it in the static storage area. As mentioned earlier, this character array is technically a constant, which is properly expressed by the return value of **v( )**.

The return value of **w( )** requires that both the pointer and what it points to must be **const**. As with **v( )**, the value returned by **w( )** is valid after the function returns only because it is **static**. You never want to return pointers to local stack variables because they will be invalid after the function returns and the stack is cleaned up. (Another common pointer you might return is the address of storage allocated on the heap, which is still valid after the function returns.)

In **main( )**, the functions are tested with various arguments. You can see that **t( )** will accept a non-**const** pointer argument, but if you try to pass it a pointer to a **const**, there's no promise that **t( )** will leave the pointer's destination alone, so the compiler gives you an error message. **u( )** takes

a **const** pointer, so it will accept both types of arguments. Thus, a function that takes a **const** pointer is more general than one that does not.

As expected, the return value of **v( )** can be assigned only to a **const** pointer. You would also expect that the compiler refuses to assign the return value of **w( )** to a non-**const** pointer, and accepts a **const int* const**, but it might be a bit surprising to see that it also accepts a **const int***, which is not an exact match to the return type. Once again, because the value (which is the address contained in the pointer) is being copied, the promise that the original variable is untouched is automatically kept. Thus, the second **const** in **const int* const** is only meaningful when you try to use it as an lvalue, in which case the compiler prevents you.

Standard argument passing

In C it's very common to pass by value, and when you want to pass an address your only choice is to use a pointer[31]. However, neither of these approaches is preferred in C++. Instead, your first choice when passing an argument is to pass by reference, and by **const** reference at that. To the client programmer, the syntax is identical to that of passing by value, so there's no confusion about pointers – they don't even have to think about pointers. For the creator of the function, passing an address is virtually always more efficient than passing an entire class object, and if you pass by **const** reference it means your function will not change the destination of that address, so the effect from the client programmer's point of view is exactly the same as pass-by-value (only more efficient).

Because of the syntax of references (it looks like pass-by-value to the caller) it's possible to pass a temporary object to a function that takes a **const** reference, whereas you can never pass a temporary object to a function that takes a pointer – with a pointer, the address must be explicitly taken. So passing by reference produces a new situation that never occurs in C: a temporary, which is always **const**, can have its *address* passed to a function. This is why, to allow temporaries to be passed to functions by reference, the argument must be a **const** reference. The following example demonstrates this:

```
//: C08:ConstTemporary.cpp
```

[31] Some folks go as far as saying that *everything* in C is pass by value, since when you pass a pointer a copy is made (so you're passing the pointer by value). However precise this might be, I think it actually confuses things.

```
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
} ///:~
```

f( ) returns an object of **class X** *by value*. That means when you immediately take the return value of **f( )** and pass it to another function as in the calls to **g1( )** and **g2( )**, a temporary is created and that temporary is **const**. Thus, the call in **g1( )** is an error because **g1( )** doesn't take a **const** reference, but the call to **g2( )** is OK.

Classes

This section shows the ways you can use **const** with classes. You may want to create a local **const** in a class to use inside constant expressions that will be evaluated at compile time. However, the meaning of **const** is different inside classes, so you must understand the options in order to create **const** data members of a class.

You can also make an entire object **const** (and as you've just seen, the compiler always makes temporary objects **const**). But preserving the **const**ness of an object is more complex. The compiler can ensure the **const**ness of a built-in type but it cannot monitor the intricacies of a class. To guarantee the **const**ness of a class object, the **const** member function is introduced: only a **const** member function may be called for a **const** object.

const in classes

One of the places you'd like to use a **const** for constant expressions is inside classes. The typical example is when you're creating an array inside

a class and you want to use a **const** instead of a **#define** to establish the array size and to use in calculations involving the array. The array size is something you'd like to keep hidden inside the class, so if you used a name like **size**, for example, you could use that name in another class without a clash. The preprocessor treats all **#define**s as global from the point they are defined, so this will not achieve the desired effect.

You might assume that the logical choice is to place a **const** inside the class. This doesn't produce the desired result. Inside a class, **const** partially reverts to its meaning in C. It allocates storage within each object and represents a value that is initialized once and then cannot change. The use of **const** inside a class means "This is constant for the lifetime of the object." However, each different object may contain a different value for that constant.

Thus, when you create an ordinary (non-**static**) **const** inside a class, you cannot give it an initial value. This initialization must occur in the constructor, of course, but in a special place in the constructor. Because a **const** must be initialized at the point it is created, inside the main body of the constructor the **const** must *already* be initialized. Otherwise you're left with the choice of waiting until some point later in the constructor body, which means the **const** would be un-initialized for a while. Also, there would be nothing to keep you from changing the value of the **const** at various places in the constructor body.

The constructor initializer list

The special initialization point is called the *constructor initializer list*, and it was originally developed for use in inheritance (an object-oriented subject of a later chapter). The constructor initializer list – which, as the name implies, occurs only in the definition of the constructor – is a list of "constructor calls" that occur after the function argument list and a colon, but before the opening brace of the constructor body. This is to remind you that the initialization in the list occurs before any of the main constructor code is executed. This is the place to put all **const** initializations. The proper form for **const** inside a class is shown here:

```
//: C08:ConstInitialization.cpp
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
  const int size;
```

```
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

The form of the constructor initializer list shown above is confusing at first because you're not used to seeing a built-in type treated as if it has a constructor.

"Constructors" for built-in types

As the language developed and more effort was put into making user-defined types look like built-in types, it became apparent that there were times when it was helpful to make built-in types look like user-defined types. In the constructor initializer list, you can treat a built-in type as if it has a constructor, like this:

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
```

```
} ///:~
```

This is especially critical when initializing **const** data members because they must be initialized before the function body is entered.

It made sense to extend this "constructor" for built-in types (which simply means assignment) to the general case, which is why the **float pi(3.14159)** definition works in the above code.

It's often useful to encapsulate a built-in type inside a class to guarantee initialization with the constructor. For example, here's an **Integer** class:

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
    i[j].print();
} ///:~
```

The array of **Integer**s in **main( )** are all automatically initialized to zero. This initialization isn't necessarily more costly than a **for** loop or **memset( )**. Many compilers easily optimize this to a very fast process.

Compile-time constants in classes

The above use of **const** is interesting and probably useful in cases, but it does not solve the original problem which is: "how do you make a compile-time constant inside a class?" The answer requires the use of an additional keyword which will not be fully introduced until Chapter XX: **static**. The **static** keyword, in this situation, means "there's only one

instance, regardless of how many objects of the class are created," which is precisely what we need here: a member of a class which is constant, and which cannot change from one object of the class to another. Thus, a **static const** of a built-in type can be treated as a compile-time constant.

There is one feature of **static const** when used inside classes which is a bit unusual: you must provide the initializer at the point of definition of the **static const**. This is something that only occurs with the **static const**; as much as you might like to use it in other situations it won't work because all other data members must be initialized in the constructor or in other member functions.

Here's an example that shows the creation and use of a **static const** called **size** inside a class that represents a stack of string pointers:

```
//: C08:StringStack.cpp
// Using static const to create a compile-time
// constant inside a class
#include <string>
#include <iostream>
using namespace std;

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
   StringStack();
   void push(const string* s);
   const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
```

```
    stack[index] = 0;
    return rv;
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

Since **size** is used to determine the size of the array **stack**, it is indeed a compile-time constant, but one that is hidden inside the class.

Notice that **push( )** takes a **const string*** as an argument, **pop( )** returns a **const string***, and **StringStack** holds **const string***. If this were not true, you couldn't use a **StringStack** to hold the pointers in **iceCream**. However, it also prevents you from doing anything that will change the objects contained by **StringStack**. Of course, not all containers are designed with this restriction.

The "enum hack" in old code

In older versions of C++, **static const** was not supported inside classes. This meant that **const** was useless for constant expressions inside classes. However, people still wanted to do this so a common solution

(typically referred to as the “enum hack”) was to use an untagged **enum** with no instances. An enumeration must have all its values established at compile time, it’s local to the class, and its values are available for constant expressions. Thus, you will commonly see (in older code[32]):

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
  cout << "sizeof(Bunch) = " << sizeof(Bunch) <<
    ", sizeof(i[1000]) = " << sizeof(int[1000])
    << endl;
} ///:~
```

The use of **enum** here is guaranteed to occupy no storage in the object, and the enumerators are all evaluated at compile time. You can also explicitly establish the values of the enumerators:

```
enum { one = 1, two = 2, three };
```

With integral **enum** types, the compiler will continue counting from the last value, so the enumerator **three** will get the value 3.

In the **StringStack.cpp** example above, the line:

```
static const int size = 100;
```

would be instead:

```
enum { size = 100 };
```

Although you’ll often see the **enum** technique in legacy code, the **static const** feature was added to the language to solve just this problem and it produces a more flexible compile-time constant inside a class.

---

[32] If your compiler doesn’t support **static const**, you’ll have to replace all the instances in this book of **static const** inside classes with the enum hack in order to get them to compile.

const objects & member functions

Class member functions can be made **const**. What does this mean? To understand, you must first grasp the concept of **const** objects.

A **const** object is defined the same for a user-defined type as a built-in type. For example:

```
const int i = 1;
const blob b(2);
```

Here, **b** is a **const** object of type **blob**. Its constructor is called with an argument of two. For the compiler to enforce **const**ness, it must ensure that no data members of the object are changed during the object's lifetime. It can easily ensure that no public data is modified, but how is it to know which member functions will change the data and which ones are "safe" for a **const** object?

If you declare a member function **const**, you tell the compiler the function can be called for a **const** object. A member function that is not specifically declared **const** is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a **const** object.

It doesn't stop there, however. Just *claiming* a member function is **const** doesn't guarantee it will act that way, so the compiler forces you to reiterate the **const** specification when defining the function. (The **const** becomes part of the function signature, so both the compiler and linker check for **const**ness.) Then it enforces **const**ness during the function definition by issuing an error message if you try to change any members of the object *or* call a non-**const** member function. Thus, any member function you declare **const** is guaranteed to behave that way in the definition.

To understand the syntax for declaring **const** member functions, first notice that preceding the function declaration with **const** means the return value is **const**, so that doesn't produce the desired results. Instead, you must place the **const** specifier *after* the argument list. For example,

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
```

```
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Note that the **const** keyword must be repeated in the definition or the compiler sees it as a different function. Since **f( )** is a **const** member function, if it attempts to change **i** in any way *or* to call another member function that is not **const**, the compiler flags it as an error.

You can see that a **const** member function is safe to call with both **const** and non-**const** objects. Thus, you could think of it as the most general form of a member function (and because of this, it is unfortunate that member functions do not automatically default to **const**). Any function that doesn't modify member data should be declared as **const**, so it can be used with **const** objects.

Here's an example that contrasts a **const** and non-**const** member function:

```
//: C08:Quoter.cpp
// Random quote selection
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
```

```
  lastquote = -1;
  srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Neither constructors nor destructors can be **const** member functions because they virtually always perform some modification on the object during initialization and cleanup. The **quote( )** member function also cannot be **const** because it modifies the data member **lastquote** (see the **return** statement). However, **lastQuote( )** makes no modifications, and so it can be **const** and can be safely called for the **const** object **cq**.

mutable: bitwise vs. memberwise const

What if you want to create a **const** member function, but you'd still like to change some of the data in the object? This is sometimes referred to as the difference between *bitwise* **const** and *memberwise* **const**. Bitwise **const** means that every bit in the object is permanent, so a bit image of the object will never change. Memberwise **const** means that, although the entire object is conceptually constant, there may be changes on a member-by-member basis. However, if the compiler is told that an object is **const**, it will jealously guard that object to ensure bitwise **const**ness. To effect memberwise **const**ness, there are two ways to change a data member from within a **const** member function.

The first approach is the historical one and is called *casting away constness*. It is performed in a rather odd fashion. You take **this** (the keyword that produces the address of the current object) and cast it to a pointer to an object of the current type. It would seem that **this** is *already* such a pointer. However, inside a **const** member function it's actually a **const** pointer, so by casting it to an ordinary pointer, you remove the **const**ness for that operation. Here's an example:

```
//: C08:Castaway.cpp
// "Casting away" constness

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y:: Y() { i = 0; }

void Y::f() const {
//!   i++; // Error -- const member function
    ((Y*)this)->i++; // OK: cast away const-ness
}

int main() {
  const Y yy;
  yy.f(); // Actually changes it!
} ///:~
```

This approach works and you'll see it used in legacy code, but it is not the preferred technique. The problem is that this lack of **const**ness is hidden away in a member function definition, and you have no clue from the class interface that the data of the object is actually being modified unless you have access to the source code (and you must suspect that **const**ness is being cast away, and look for the cast). To put everything out in the open, you should use the **mutable** keyword in the class declaration to specify that a particular data member may be changed inside a **const** object:

```
//: C08:Mutable.cpp
// The "mutable" keyword

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() { i = j = 0; }

void Z::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
  const Z zz;
  zz.f(); // Actually changes it!
} ///:~
```

This way, the user of the class can see from the declaration which members are likely to be modified in a **const** member function.

ROMability

If an object is defined as **const**, it is a candidate to be placed in read-only memory (ROM), which is often an important consideration in embedded systems programming. Simply making an object **const**, however, is not enough – the requirements for ROMability are much more strict. Of course, the object must be bitwise-**const**, rather than memberwise-**const**. This is easy to see if memberwise **const**ness is implemented only through the **mutable** keyword, but probably not detectable by the

compiler if **const**ness is cast away inside a **const** member function. In addition,

1. The **class** or **struct** must have no user-defined constructors or destructor.
2. There can be no base classes (covered in the future chapter on inheritance) or member objects with user-defined constructors or destructors.

The effect of a write operation on any part of a **const** object of a ROMable type is undefined. Although a suitably formed object may be placed in ROM, no objects are ever *required* to be placed in ROM.

volatile

The syntax of **volatile** is identical to that for **const**, but **volatile** means "This data may change outside the knowledge of the compiler." Somehow, the environment is changing the data (possibly through multitasking, multithreading or interrupts), and **volatile** tells the compiler not to make any assumptions about that data, especially during optimization.

If the compiler says, "I read this data into a register earlier, and I haven't touched that register," normally it wouldn't need to read the data again. But if the data is **volatile**, the compiler cannot make such an assumption because the data may have been changed by another process, and it must reread that data rather than optimizing the code to remove what would normally be a redundant read.

You create **volatile** objects using the same syntax that you use to create **const** objects. You can also create **const volatile** objects, which can't be changed by the client programmer but instead change through some outside agency. Here is an example that might represent a class associated with some piece of communication hardware:

```
//: C08:Volatile.cpp
// The volatile keyword

class Comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  static const int bufsize = 100;
  unsigned char buf[bufsize];
  int index;
public:
```

```
  Comm();
  void isr() volatile;
  char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// as an interrupt service routine:
void Comm::isr() volatile {
  if(flag) flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Error, read() not volatile
} ///:~
```

As with **const**, you can use **volatile** for data members, member functions, and objects themselves. You can only call **volatile** member functions for **volatile** objects.

The reason that **isr( )** can't actually be used as an interrupt service routine is that in a member function, the address of the current object (**this**) must be secretly passed, and an ISR generally wants no arguments at all. To solve this problem, you can make **isr( )** a **static** member function, a subject covered in a future chapter.

The syntax of **volatile** is identical to **const**, so discussions of the two are often treated together. The two are referred to in combination as the *c-v qualifier*.

Summary

The **const** keyword gives you the ability to define objects, function arguments, return values and member functions as constants, and to eliminate the preprocessor for value substitution without losing any preprocessor benefits. All this provides a significant additional form of type checking and safety in your programming. The use of so-called *const correctness* (the use of **const** anywhere you possibly can) can be a lifesaver for projects.

Although you can ignore **const** and continue to use old C coding practices, it's there to help you. Chapters XX & XX begin using references heavily, and there you'll see even more about how critical it is to use **const** with function arguments.

Exercises

1. Create three **const int** values, then add them together to produce a value that determines the size of an array in an array definition. Try to compile the same code in C and see what happens (you can generally force your C++ compiler to run as a C compiler by using a command-line flag).
2. Prove to yourself that the C and C++ compilers really do treat constants differently. Create a global **const** and use it in a constant expression; then compile it under both C and C++.
3. Create example **const** definitions for all the built-in types and their variants. Use these in expressions with other **const**s to make new **const** definitions. Make sure they compile successfully.
4. Create a **const** definition in a header file, include that header file in two **.cpp** files, then compile those files and link them. You should not get any errors. Now try the same experiment with C.
5. Create a **const** whose value is determined at run time by reading the time when the program starts (you'll have to use the **<ctime>** standard header). Later in the program, try to read a second value of the time into your **const** and see what happens.

6. Create a **const** array of **char**, then try to change one of the **char**s.
7. Create an **extern const** declaration in one file, and put a **main( )** in that file that prints the value of the **extern const**. Provide an **extern const** definition in a second file, then compile and link the two files together.
8. Write two pointers to **const long** using both forms of the declaration. Point one of them to an array of **long**. Demonstrate that you can increment or decrement the pointer, but you can't change what it points to.
9. Write a **const** pointer to a **double**, and point it at an array of **double**. Show that you can change what the pointer points to, but you can't increment or decrement the pointer.
10. Write a **const** pointer to a **const** object. Show that you can only read the value that the pointer points to, but you can't change the pointer or what it points to.
11. Remove the comment on the error-generating line of code in **PointerAssignment.cpp** to see the error that your compiler generates.
12. Create a character array literal with a pointer that points to the beginning of the array. Now use the pointer to modify elements in the array. Does your compiler report this as an error? Should it? If it doesn't, why do you think that is?
13. Create a function that takes an argument by value as a **const**; then try to change that argument in the function body.
14. Create a a function that takes a **float** by value. Inside the function, bind a **const float&** to the argument, and only use the reference from then on to ensure that the argument is not changed.
15. Modify **ConstReturnValues.cpp** removing comments on the error-causing lines one at a time, to see what error messages your compiler generates.
16. Modify **ConstPointer.cpp** removing comments on the error-causing lines one at a time, to see what error messages your compiler generates.
17. Make a new version of **ConstPointer.cpp** called **ConstReference.cpp** which demonstrates references instead of pointers (you may need to look forward to Chapter XX).

18. Modify **ConstTemporary.cpp** removing the comment on the error-causing line to see what error messages your compiler generates.
19. Create a class containing both a **const** and a non-**const float**. Initialize these using the constructor initializer list.
20. Create a class called **MyString** which contains a **string** and has a constructor that initializes the **string**, and a **print( )** function. Modify **StringStack.cpp** so that the container holds **MyString** objects, and **main( )** so it prints them.
21. Create a class containing a **const** member that you initialize in the constructor initializer list and an untagged enumeration that you use to determine an array size.
22. In **ConstMember.cpp**, remove the **const** specifier on the member function definition, but leave it on the declaration, to see what kind of compiler error message you get.
23. Create a class with both **const** and non-**const** member functions. Create **const** and non-**const** objects of this class, and try calling the different types of member functions for the different types of objects.
24. Create a class with both **const** and non-**const** member functions. Try to call a non-**const** member function from a **const** member function to see what kind of compiler error message you get.
25. In **Mutable.cpp**, remove the comment on the error-causing line to see what sort of error message your compiler produces.
26. Modify **Quoter.cpp** by making **quote( )** a **const** member function and **lastquote mutable**.
27. Create a class with a **volatile** data member. Create both **volatile** and non-**volatile** member functions that modify the **volatile** data member, and see what the compiler says. Create both **volatile** and non-**volatile** objects of your class and try calling both the **volatile** and non-**volatile** member functions to see what is successful and what kind of error messages the compiler produces.
28. Create a class called **bird** that can **fly( )** and a class **rock** that can't. Create a **rock** object, take its address, and assign that to a **void***. Now take the **void***, assign it to a **bird*** (you'll have to use a cast), and call **fly( )** through that pointer. Is it clear why C's permission to openly assign

via a **void*** (without a cast) is a “hole” in the language, which couldn’t be propagated into C++?

9: Inline functions

One of the important features C++ inherits from C is efficiency. If the efficiency of C++ were dramatically less than C, there would be a significant contingent of programmers who couldn't justify its use.

In C, one of the ways to preserve efficiency is through the use of *macros*, which allow you to make what looks like a function call without the normal overhead of the function call. The macro is implemented with the preprocessor rather than the compiler proper, and the preprocessor replaces all macro calls directly with the macro code, so there's no cost involved from pushing arguments, making an assembly-language CALL, returning arguments, and performing an assembly-language RETURN. All the work is performed by the preprocessor, so you have the convenience and readability of a function call but it doesn't cost you anything.

There are two problems with the use of preprocessor macros in C++. The first is also true with C: A macro looks like a function call, but doesn't always act like one. This can bury difficult-to-find bugs. The second problem is specific to C++: The preprocessor has no permission to access **private** data. This means preprocessor macros are virtually useless as class member functions.

To retain the efficiency of the preprocessor macro, but to add the safety and class scoping of true functions, C++ has the *inline function*. In this chapter, we'll look at the problems of preprocessor macros in C++, how these problems are solved with inline functions, and guidelines and insights on the way inlines work.

Preprocessor pitfalls

The key to the problems of preprocessor macros is that you can be fooled into thinking that the behavior of the preprocessor is the same as the behavior of the compiler. Of course, it was intended that a macro look and act like a function call, so it's quite easy to fall into this fiction. The difficulties begin when the subtle differences appear.

As a simple example, consider the following:

```
#define F (x) (x + 1)
```

Now, if a call is made to **F** like this

```
F(1)
```

the preprocessor expands it, somewhat unexpectedly, to the following:

```
(x) (x + 1)(1)
```

The problem occurs because of the gap between **F** and its opening parenthesis in the macro definition. When this gap is removed, you can actually *call* the macro with the gap

```
F (1)
```

and it will still expand properly, to

```
(1 + 1)
```

The above example is fairly trivial and the problem will make itself evident right away. The real difficulties occur when using expressions as arguments in macro calls.

There are two problems. The first is that expressions may expand inside the macro so that their evaluation precedence is different from what you expect. For example,

```
#define FLOOR(x,b) x>=b?0:1
```

Now, if expressions are used for the arguments

```
if(FLOOR(a&0x0f,0x07)) // ...
```

the macro will expand to

```
if(a&0x0f>=0x07?0:1)
```

The precedence of **&** is lower than that of **>=**, so the macro evaluation will surprise you. Once you discover the problem (and as a general

practice when creating preprocessor macros) you can solve it by putting parentheses around everything in the macro definition. Thus,

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

Discovering the problem may be difficult, however, and you may not find it until after you've taken the proper macro behavior for granted. In the unparenthesized version of the preceding example, *most* expressions will work correctly, because the precedence of **>=** is lower than most of the operators like +, **/**, **– –**, and even the bitwise shift operators. So you can easily begin to think that it works with all expressions, including those using bitwise logical operators.

The preceding problem can be solved with careful programming practice: Parenthesize everything in a macro. The second difficulty is more subtle. Unlike a normal function, every time you use an argument in a macro, that argument is evaluated. As long as the macro is called only with ordinary variables, this evaluation is benign, but if the evaluation of an argument has side effects, then the results can be surprising and will definitely not mimic function behavior.

For example, this macro determines whether its argument falls within a certain range:

```
#define BAND(X) (((X)>5 && (X)<10) ? (X) : 0)
```

As long as you use an "ordinary" argument, the macro works very much like a real function. But as soon as you relax and start believing it *is* a real function, the problems start. Thus,

```
//: C09:Macro.cpp
// Side effects with macros
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(X) (((X)>5 && (X)<10) ? (X) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
```

```
  }
} ///:~
```

Here's the output produced by the program, which is not at all what you would have expected from a true function:

```
a = 4
 BAND(++a)=0
  a = 5
a = 5
 BAND(++a)=8
  a = 8
a = 6
 BAND(++a)=9
  a = 9
a = 7
 BAND(++a)=10
  a = 10
a = 8
 BAND(++a)=0
  a = 10
a = 9
 BAND(++a)=0
  a = 11
a = 10
 BAND(++a)=0
  a = 12
```

When **a** is four, only the first part of the conditional occurs, so the expression is evaluated only once, and the side effect of the macro call is that **a** becomes five, which is what you would expect from a normal function call in the same situation. However, when the number is within the band, both conditionals are tested, which results in two increments. The result is produced by evaluating the argument again, which results in a third increment. Once the number gets out of the band, both conditionals are still tested so you get two increments. The side effects are different, depending on the argument.

This is clearly not the kind of behavior you want from a macro that looks like a function call. In this case, the obvious solution is to make it a true function, which of course adds the extra overhead and may reduce efficiency if you call that function a lot. Unfortunately, the problem may not always be so obvious, and you can unknowingly get a library that contains functions and macros mixed together, so a problem like this can

hide some very difficult-to-find bugs. For example, the **putc( )** macro in **cstdio** may evaluate its second argument twice. This is specified in Standard C. Also, careless implementations of **toupper( )** as a macro may evaluate the argument more than once, which will give you unexpected results with **toupper(*p++)**.[33]

Macros and access

Of course, careful coding and use of preprocessor macros are required with C, and we could certainly get away with the same thing in C++ if it weren't for one problem: A macro has no concept of the scoping required with member functions. The preprocessor simply performs text substitution, so you cannot say something like

```
class X {
  int i;
public:
#define VAL (X::i) // Error
```

or anything even close. In addition, there would be no indication of which object you were referring to. There is simply no way to express class scope in a macro. Without some alternative to preprocessor macros, programmers will be tempted to make some data members **public** for the sake of efficiency, thus exposing the underlying implementation and preventing changes in that implementation.

Inline functions

In solving the C++ problem of a macro with access to private class members, *all* the problems associated with preprocessor macros were eliminated. This was done by bringing macros under the control of the compiler, where they belong. In C++, the concept of a macro is implemented as an *inline function*, which is a true function in every sense. Any behavior you expect from an ordinary function, you get from an inline function. The only difference is that an inline function is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated. Thus, you should (almost) never use macros, only inline functions.

---

[33] Andrew Koenig goes into more detail in his book *C Traps & Pitfalls* (Addison-Wesley, 1989).

Any function defined within a class body is automatically inline, but you can also make a nonclass function inline by preceding it with the **inline** keyword. However, for it to have any effect, you must include the function body with the declaration; otherwise the compiler will treat it as an ordinary function declaration. Thus,

```
inline int plusOne(int x);
```

has no effect at all other than declaring the function (which may or may not get an inline definition sometime later). The successful approach is

```
inline int plusOne(int x) { return ++x; }
```

Notice that the compiler will check (as it always does) for the proper use of the function argument list and return value (performing any necessary conversions), something the preprocessor is incapable of. Also, if you try to write the above as a preprocessor macro, you get an unwanted side effect.

You'll almost always want to put inline definitions in a header file. When the compiler sees such a definition, it puts the function type (signature + return value) *and* the function body in its symbol table. When you use the function, the compiler checks to ensure the call is correct and the return value is being used correctly, and then substitutes the function body for the function call, thus eliminating the overhead. The inline code does occupy space, but if the function is small, this can actually take less space than the code generated to do an ordinary function call (pushing arguments on the stack and doing the CALL).

An inline function in a header file defaults to *internal linkage* – that is, it is **static** and can only be seen in translation units where it is included. Thus, as long as they aren't declared in the same translation unit, there will be no clash at link time between an inline function and a global function with the same signature. (Remember the return value is not included in the resolution of function overloading.

Inlines inside classes

To define an inline function, you must ordinarily precede the function definition with the **inline** keyword. However, this is not necessary inside a class definition. Any function you define inside a class definition is automatically an inline. Thus,

```
//: C09:Inline.cpp
// Inlines inside classes
#include <iostream>
```

```
using namespace std;

class Point {
  int i, j, k;
public:
  Point() { i = j = k = 0; }
  Point(int ii, int jj, int kk) {
    i = ii;
    j = jj;
    k = kk;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
} ///:~
```

Of course, the temptation is to use inlines everywhere inside class declarations because they save you the extra step of making the external member function definition. Keep in mind, however, that the idea of an inline is to reduce the overhead of a function call. If the function body is large, chances are you'll spend a much larger percentage of your time inside the body versus going in and out of the function, so the gains will be small. But inlining a big function will cause that code to be duplicated everywhere the function is called, producing code bloat with little or no speed benefit.

Access functions

One of the most important uses of inlines inside classes is the *access function*. This is a small function that allows you to read or change part of the state of an object – that is, an internal variable or variables. The reason inlines are so important with access functions can be seen in the following example:

```
//: C09:Access.cpp
// Inline access functions

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
  Access A;
  A.set(100);
  int x = A.read();
} ///:~
```

Here, the class user never has direct contact with the state variables inside the class, and they can be kept **private**, under the control of the class designer. All the access to the **private** data members can be controlled through the member function interface. In addition, access is remarkably efficient. Consider the **read( )**, for example. Without inlines, the code generated for the call to **read( )** would include pushing **this** on the stack and making an assembly language CALL. With most machines, the size of this code would be larger than the code created by the inline, and the execution time would certainly be longer.

Without inline functions, an efficiency-conscious class designer will be tempted to simply make **i** a public member, eliminating the overhead by allowing the user to directly access **i**. From a design standpoint, this is disastrous because **i** then becomes part of the public interface, which means the class designer can never change it. You're stuck with an **int** called **i**. This is a problem because you may learn sometime later that it would be much more useful to represent the state information as a **float** rather than an **int**, but because **int i** is part of the public interface, you can't change it. If, on the other hand, you've always used member functions to read and change the state information of an object, you can modify the underlying representation of the object to your heart's content (and permanently remove from your mind the idea that you are going to perfect your design before you code it and try it out).

Accessors and mutators

Some people further divide the concept of access functions into *accessors* (to read state information from an object) and *mutators* (to change the

state of an object). In addition, function overloading may be used to provide the same function name for both the accessor and mutator; how you call the function determines whether you're reading or modifying state information. Thus,

```
//: C09:Rectangle.cpp
// Accessors & mutators

class Rectangle {
  int _width, _height;
public:
  Rectangle(int w = 0, int h = 0)
    : _width(w), _height(h) {}
  int width() const { return _width; } // Read
  void width(int w) { _width = w; } // Set
  int height() const { return _height; } // Read
  void height(int h) { _height = h; } // Set
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.height(2 * r.width());
  r.width(2 * r.height());
} ///:~
```

The constructor uses the constructor initializer list (briefly introduced in Chapter XX and covered fully in Chapter XX) to initialize the values of **_width** and **_height** (using the pseudoconstructor-call form for built-in types).

Since you cannot have member function names using the same identifiers as data members, the data members are distinguished with a leading underscore (this way, the coding standard described in Appendix A can be followed, whereby all variables and functions begin with lowercase letters). Because this is a bit awkward, and because overloading this way might seem confusing, you may choose instead to use "get" and "set" to indicate accessors and mutators:

```
//: C09:Rectangle2.cpp
// Accessors & mutators with "get" and "set"

class Rectangle {
  int width, height;
```

```
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

Of course, accessors and mutators don't have to be simple pipelines to an internal variable. Sometimes they can perform some sort of calculation. The following example uses the Standard C library time functions to produce a simple **Time** class:

```
//: C09:Cpptime.h
// A simple time class
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
    if(!lflag) {
      local = *std::localtime(&t);
      lflag++;
    }
  }
  void updateAscii() {
    if(!aflag) {
      updateLocal();
      std::strcpy(asciiRep,std::asctime(&local));
```

```
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
    std::time(&t);
  }
  const char* ascii() {
    updateAscii();
    return asciiRep;
  }
  // Difference in seconds:
  int delta(Time* dt) const {
    return std::difftime(t, dt->t);
  }
  int daylightSavings() {
    updateLocal();
    return local.tm_isdst;
  }
  int dayOfYear() { // Since January 1
    updateLocal();
    return local.tm_yday;
  }
  int dayOfWeek() { // Since Sunday
    updateLocal();
    return local.tm_wday;
  }
  int since1900() { // Years since 1900
    updateLocal();
    return local.tm_year;
  }
  int month() { // Since January
    updateLocal();
    return local.tm_mon;
  }
  int dayOfMonth() {
    updateLocal();
    return local.tm_mday;
  }
  int hour() { // Since midnight, 24-hour clock
```

```
    updateLocal();
    return local.tm_hour;
  }
  int minute() {
    updateLocal();
    return local.tm_min;
  }
  int second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H ///:~
```

The Standard C library functions have multiple representations for time, and these are all part of the **Time** class. However, it isn't necessary to update all of them all the time, so instead the **time_t t** is used as the base representation, and the **tm local** and ASCII character representation **asciiRep** each have flags to indicate if they've been updated to the current **time_t**. The two **private** functions **updateLocal( )** and **updateAscii( )** check the flags and conditionally perform the update.

The constructor calls the **mark( )** function (which the user can also call to force the object to represent the current time), and this clears the two flags to indicate that the local time and ASCII representation are now invalid. The **ascii( )** function calls **updateAscii( )**, which copies the result of the Standard C library function **asctime( )** into a local buffer because **asctime( )** uses a static data area that is overwritten if the function is called elsewhere. The return value is the address of this local buffer.

In the functions starting with **DaylightSavings( )**, all use the **updateLocal( )** function, which causes the composite inline to be fairly large. This doesn't seem worthwhile, especially considering you probably won't call the functions very much. However, this doesn't mean all the functions should be made out of line. If you leave **updateLocal( )** as an inline, its code will be duplicated in all the out-of-line functions, eliminating the extra overhead.

Here's a small test program:

```
//: C09:Cpptime.cpp
// Testing a simple time class
#include "Cpptime.h"
#include <iostream>
using namespace std;
```

```
int main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

A **Time** object is created, then some time-consuming activity is performed, then a second **Time** object is created to mark the ending time. These are used to show starting, ending, and elapsed times.

Stash & Stack with inlines

Inlines & the compiler

To understand when inlining is effective, it's helpful to understand what the compiler does when it encounters an inline. As with any function, the compiler holds the function *type* (that is, the function prototype including the name and argument types, in combination with the function return value) in its symbol table. In addition, when the compiler sees the inline function body *and* the function body parses without error, the code for the function body is also brought into the symbol table. Whether the code is stored in source form or as compiled assembly instructions is up to the compiler.

When you make a call to an inline function, the compiler first ensures that the call can be correctly made; that is, all the argument types must be the proper types, or the compiler must be able to make a type conversion to the proper types, and the return value must be the correct type (or convertible to the correct type) in the destination expression. This, of course, is exactly what the compiler does for any function and is markedly

different from what the preprocessor does because the preprocessor cannot check types or make conversions.

If all the function type information fits the context of the call, then the inline code is substituted directly for the function call, eliminating the call overhead. Also, if the inline is a member function, the address of the object (**this**) is put in the appropriate place(s), which of course is another thing the preprocessor is unable to perform.

Limitations

There are two situations when the compiler cannot perform inlining. In these cases, it simply reverts to the ordinary form of a function by taking the inline definition and creating storage for the function just as it does for a non-inline. If it must do this in multiple translation units (which would normally cause a multiple definition error), the linker is told to ignore the multiple definitions.

The compiler cannot perform inlining if the function is too complicated. This depends upon the particular compiler, but at the point most compilers give up, the inline probably wouldn't gain you any efficiency. Generally, any sort of looping is considered too complicated to expand as an inline, and if you think about it, looping probably entails much more time inside the function than embodied in the calling overhead. If the function is just a collection of simple statements, the compiler probably won't have any trouble inlining it, but if there are a lot of statements, the overhead of the function call will be much less than the cost of executing the body. And remember, every time you call a big inline function, the entire function body is inserted in place of each call, so you can easily get code bloat without any noticeable performance improvement. Some of the examples in this book may exceed reasonable inline sizes in favor of conserving screen real estate.

The compiler also cannot perform inlining if the address of the function is taken, implicitly or explicitly. If the compiler must produce an address, then it will allocate storage for the function code and use the resulting address. However, where an address is not required, the compiler will probably still inline the code.

It is important to understand that an inline is just a suggestion to the compiler; the compiler is not forced to inline anything at all. A good compiler will inline small, simple functions while intelligently ignoring inlines that are too complicated. This will give you the results you want – the true semantics of a function call with the efficiency of a macro.

Order of evaluation

If you're imagining what the compiler is doing to implement inlines, you can confuse yourself into thinking there are more limitations than actually exist. In particular, if an inline makes a forward reference to a function that hasn't yet been declared in the class, it can seem like the compiler won't be able to handle it:

```
//: C09:Evorder.cpp
// Inline evaluation order

class Forward {
  int i;
public:
  Forward() : i(0) {}
  // Call to undeclared function:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

int main() {
  Forward F;
  F.f();
} ///:~
```

In **f( )**, a call is made to **g( )**, although **g( )** has not yet been declared. This works because the language definition states that no inline functions in a class shall be evaluated until the closing brace of the class declaration.

Of course, if **g( )** in turn called **f( )**, you'd end up with a set of recursive calls, which are too complicated for the compiler to inline. (Also, you'd have to perform some test in **f( )** or **g( )** to force one of them to "bottom out," or the recursion would be infinite.)

Hidden activities in constructors & destructors

Constructors and destructors are two places where you can be fooled into thinking that an inline is more efficient than it actually is. Both constructors and destructors may have hidden activities, because the class can contain subobjects whose constructors and destructors must be

called. These sub-objects may be member objects, or they may exist because of inheritance (which hasn't been introduced yet). As an example of a class with member objects

```
//: C09:Hidden.cpp
// Hidden activities in inlines
#include <iostream>
using namespace std;

class Member {
  int i, j, k;
public:
  Member(int x = 0) { i = j = k = x; }
  ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
  Member q, r, s; // Have constructors
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Trivial?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
};

int main() {
  WithMembers wm(1);
} ///:~
```

In **class WithMembers**, the inline constructor and destructor look straightforward and simple enough, but there's more going on than meets the eye. The constructors and destructors for the member objects **q**, **r**, and **s** are being called automatically, and *those* constructors and destructors are also inline, so the difference is significant from normal member functions. This doesn't necessarily mean that you should always make constructor and destructor definitions out-of-line. When you're making an initial "sketch" of a program by quickly writing code, it's often more convenient to use inlines. However, if you're concerned about efficiency, it's a place to look.

Forward referencing

Although they are convenient, inline functions exacerbate a complication that already exists without them: forward referencing. The problem is this:

Reducing clutter

In a book like this, the simplicity and terseness of putting inline definitions inside classes is very useful because more fits on a page or screen (in a seminar). However, Dan Saks[34] has pointed out that in a real project this has the effect of needlessly cluttering the class interface and thereby making the class harder to use. He refers to member functions defined within classes using the Latin *in situ* (in place) and maintains that all definitions should be placed outside the class to keep the interface clean. Optimization, he argues, is a separate issue. If you want to optimize, use the **inline** keyword. Using this approach, the earlier **Rectangle.cpp** example becomes

```
//: C09:Noinsitu.cpp
// Removing in situ functions

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {
}

inline int Rectangle::getWidth() const {
```

[34] Co-author with Tom Plum of *C++ Programming Guidelines*, Plum Hall, 1991.

```
  return width;
}

inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}

inline void Rectangle::setHeight(int h) {
  height = h;
}

int main() {
  Rectangle r(19, 47);
  // Transpose width & height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

Now if you want to compare the effect of inlining with out-of-line functions, you can simply remove the **inline** keyword. (Inline functions should normally be put in header files, however, while non-inline functions must reside in their own translation unit.) If you want to put the functions into documentation, it's a simple cut-and-paste operation. *In situ* functions require more work and have greater potential for errors. Another argument for this approach is that you can always produce a consistent formatting style for function definitions, something that doesn't always occur with *in situ* functions.

More preprocessor features

Earlier, I said you *almost* always want to use **inline** functions instead of preprocessor macros. The exceptions are when you need to use three special features in the C preprocessor (which is, by inheritance, the C++ preprocessor): stringizing, string concatenation, and token pasting.

Stringizing, performed with the **#** directive, allows you to take an identifier and turn it into a string, whereas string concatenation takes place when two adjacent strings have no intervening punctuation, in which case the strings are combined. These two features are exceptionally useful when writing debug code. Thus,

```
#define DEBUG(X) cout << #X " = " << X << endl
```

This prints the value of any variable. You can also get a trace that prints out the statements as they execute:

```
#define TRACE(S) cout << #S << endl; S
```

The **#S** stringizes the statement for output, and the second **S** reiterates the statement so it is executed. Of course, this kind of thing can cause problems, especially in one-line **for** loops:

```
for(int i = 0; i < 100; i++)
  TRACE(f(i));
```

Because there are actually two statements in the **TRACE( )** macro, the one-line **for** loop executes only the first one. The solution is to replace the semicolon with a comma in the macro.

Token pasting

Token pasting is very useful when you are manufacturing code. It allows you to take two identifiers and paste them together to automatically create a new identifier. For example,

```
#define FIELD(A) char* A##_string; int A##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

Each call to the **FIELD( )** macro creates an identifier to hold a string and another to hold the length of that string. Not only is it easier to read, it can eliminate coding errors and make maintenance easier. Notice, however, the use of all upper-case characters in the name of the macro. This is a helpful practice because it tells the reader this is a macro and not a function, so if there are problems, it acts as a little reminder.

Improved error checking

The **require.h** macros have been used up to this point without defining them (although **assert( )** has also been used to help detect programmer errors, where it's appropriate). Now it's time to define this header file. Inline functions are convenient here because they allow everything to be placed in a header file, which simplifies the process of using the package. You just include the header file and you don't need to worry about linking.

You should note that exceptions (presented in detail in Chapter XX) provide a much more effective way of handling many kinds of errors – especially those that you'd like to recover from, instead of just halting the program. The conditions that **require.h** handles, however, are ones which prevent the continuation of the program, such as if the user doesn't provide enough command-line arguments or a file cannot be opened.

The following header file will be placed in the book's root directory so it's easily accessed from all chapters.

```
//: :require.h
// Test for error conditions in programs
// Local "using namespace std" for old compilers
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>

inline void require(bool requirement,
  const char* msg = "Requirement failed") {
  using namespace std;
  if (!requirement) {
    fprintf(stderr, "%s", msg);
    exit(1);
  }
}

inline void requireArgs(int argc, int args,
  const char* msg = "Must use %d arguments") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg, args);
     exit(1);
```

```
  }
}

inline void requireMinArgs(int argc, int minArgs,
  const char* msg =
    "Must use at least %d arguments") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg, minArgs);
    exit(1);
  }
}

inline void assure(std::ifstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr,
      "Could not open file %s", filename);
    exit(1);
  }
}

inline void assure(std::ofstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr,
      "Could not open file %s", filename);
    exit(1);
  }
}
#endif // REQUIRE_H ///:~
```

The default values provide reasonable messages that can be changed if necessary.

In the definitions for **requireArgs( )** and **requireMinArgs( )**, one is added to the number of arguments you need on the command line because **argc** always includes the name of the program being executed as the zeroth argument, and so always has a value that is one more than the number of actual arguments on the command line.

Note the use of local "**using namespace std**" declarations within each function. This is because some compilers at the time of this writing incorrectly did not include the C standard library functions in **namespace std**, so explicit qualification would cause a compile-time error. The local declaration allows **require.h** to work with both correct and incorrect libraries.

Here's a simple program to test **require.h**:

```
//: C09:ErrTest.cpp
// Testing require.h
#include "../require.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  int i = 1;
  require(i, "value must be nonzero");
  requireArgs(argc, 1);
  requireMinArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // Use the file name
  ifstream nofile("nofile.xxx");
  assure(nofile); // The default argument
  ofstream out("tmp.txt");
  assure(out);
} ///:~
```

You might be tempted to go one step further for opening files and add a macro to **require.h**:

```
#define IFOPEN(VAR, NAME) \
  ifstream VAR(NAME); \
  assure(VAR, NAME);
```

Which could then be used like this:

```
IFOPEN(in, argv[1])
```

At first, this might seem appealing since it means there's less to type. It's not terribly unsafe, but it's a road best avoided. Note that, once again, a macro looks like a function but behaves differently: it's actually creating an object (**in**) whose scope persists beyond the macro. You may understand this, but for new programmers and code maintainers it's just one more thing they have to puzzle out. C++ is complicated enough

without adding to the confusion, so try to talk yourself out of using macros whenever you can.

Summary

It's critical that you be able to hide the underlying implementation of a class because you may want to change that implementation sometime later. You'll do this for efficiency, or because you get a better understanding of the problem, or because some new class becomes available that you want to use in the implementation. Anything that jeopardizes the privacy of the underlying implementation reduces the flexibility of the language. Thus, the inline function is very important because it virtually eliminates the need for preprocessor macros and their attendant problems. With inlines, member functions can be as efficient as preprocessor macros.

The inline function can be overused in class definitions, of course. The programmer is tempted to do so because it's easier, so it will happen. However, it's not that big an issue because later, when looking for size reductions, you can always move the functions out of line with no effect on their functionality. The development guideline should be "First make it work, then optimize it."

Exercises

1. Take Exercise 2 from Chapter 6, and add an inline constructor, and an inline member function called **print( )** to print out all the values in the array.
2. Take the **NestFriend.cpp** example from Chapter XX and replace all the member functions with inlines. Make them non-*in situ* inline functions. Also change the **initialize( )** functions to constructors.
3. Take the **nl.cpp** example from Chapter XX and turn **nl** into an **inline** function in its own header file.
4. Create a class **A** with a default constructor that announces itself. Now make a new class **B** and put an object of **A** as a member of **B**, and give **B** an inline constructor. Create an array of **B** objects and see what happens.
5. Create a large quantity of the objects from Exercise 4, and use the **Time** class to time the difference between a non-

inline constructor and an inline constructor. (If you have a profiler, also try using that.)

10: Name control

Creating names is a fundamental activity in programming, and when a project gets large the number of names can easily be overwhelming. C++ allows you a great deal of control over both the creation and visibility of names, where storage for those names is placed, and linkage for names.

The **static** keyword was overloaded in C before people knew what the term "overload" meant, and C++ has added yet another meaning. The underlying concept with all uses of **static** seems to be "something that holds its position" (like static electricity), whether that means a physical location in memory or visibility within a file.

In this chapter, you'll learn how **static** controls storage and visibility, and an improved way to control access to names via C++'s *namespace* feature. You'll also find out how to use functions that were written and compiled in C.

Static elements from C

In both C and C++ the keyword **static** has two basic meanings, which unfortunately often step on each other's toes:

1. Allocated once at a fixed address; that is, the object is created in a special *static data area* rather than on the stack each time a function is called. This is the concept of *static storage*.

2. Local to a particular translation unit (and class scope in C++, as you will see later). Here, **static** controls the *visibility* of a name, so that name cannot be seen outside the translation unit or class. This also describes the concept of *linkage*, which determines what names the linker will see.

This section will look at the above meanings of **static** as they were inherited from C.

static variables inside functions

Normally, when you create a variable inside a function, the compiler allocates storage for that variable each time the function is called by moving the stack pointer down an appropriate amount. If there is an initializer for the variable, the initialization is performed each time that sequence point is passed.

Sometimes, however, you want to retain a value between function calls. You could accomplish this by making a global variable, but that variable would not be under the sole control of the function. C and C++ allow you to create a **static** object inside a function; the storage for this object is not on the stack but instead in the program's static storage area. This object is initialized once the first time the function is called and then retains its value between function invocations. For example, the following function returns the next character in the string each time the function is called:

```
//: C10:Statfun.cpp
// Static vars inside functions
#include "../require.h"
#include <iostream>
using namespace std;

char onechar(const char* string = 0) {
  static const char* s;
  if(string) {
    s = string;
    return *s;
  }
  else
    require(s, "un-initialized s");
  if(*s == '\0')
    return 0;
```

```
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // Onechar(); // require() fails
  onechar(a); // Initializes s to a
  char c;
  while((c = onechar()) != 0)
    cout << c << endl;
} ///:~
```

The **static char* s** holds its value between calls of **onechar( )** because its storage is not part of the stack frame of the function, but is in the static storage area of the program. When you call **onechar( )** with a **char*** argument, **s** is assigned to that argument, and the first character of the string is returned. Each subsequent call to **onechar( )** *without* an argument produces the default value of zero for **string**, which indicates to the function that you are still extracting characters from the previously initialized value of **s**. The function will continue to produce characters until it reaches the null terminator of the string, at which point it stops incrementing the pointer so it doesn't overrun the end of the string.

But what happens if you call **onechar( )** with no arguments and without previously initializing the value of **s**? In the definition for **s**, you could have provided an initializer,

static char* s = 0;

but if you do not provide an initializer for a static variable of a built-in type, the compiler guarantees that variable will be initialized to zero (converted to the proper type) at program start-up. So in **onechar( )**, the first time the function is called, **s** is zero. In this case, the **if(!s)** conditional will catch it.

The above initialization for **s** is very simple, but initialization for static objects (like all other objects) can be arbitrary expressions involving constants and previously declared variables and functions.

static class objects inside functions

The rules are the same for static objects of user-defined types, including the fact that some initialization is required for the object. However, assignment to zero has meaning only for built-in types; user-defined

types must be initialized with constructor calls. Thus, if you don't specify constructor arguments when you define the static object, the class must have a default constructor. For example,

```
//: C10:Funobj.cpp
// Static objects in functions
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Default constructor required
}

int main() {
  f();
} ///:~
```

The static objects of type **X** inside **f( )** can be initialized either with the constructor argument list or with the default constructor. This construction occurs the first time control passes through the definition, and only the first time.

Static object destructors

Destructors for static objects (all objects with static storage, not just local static objects as in the above example) are called when **main( )** exits or when the Standard C library function **exit( )** is explicitly called, **main( )** in most implementations calls **exit( )** when it terminates. This means that it can be dangerous to call **exit( )** inside a destructor because you can end up with infinite recursion. Static object destructors are *not* called if you exit the program using the Standard C library function **abort( )**.

You can specify actions to take place when leaving **main( )** (or calling **exit( )**) by using the Standard C library function **atexit( )**. In this case, the functions registered by **atexit( )** may be called before the destructors for any objects constructed before leaving **main( )** (or calling **exit( )**).

Destruction of static objects occurs in the reverse order of initialization. However, only objects that have been constructed are destroyed. Fortunately, the programming system keeps track of initialization order and the objects that have been constructed. Global objects are always constructed before **main( )** is entered, so this last statement applies only to static objects that are local to functions. If a function containing a local static object is never called, the constructor for that object is never executed, so the destructor is also not executed. For example,

```
//: C10:StaticDestructors.cpp
// Static object destructors
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
  char c; // Identifier
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() for " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() for " << c << endl;
  }
};

Obj a('a'); // Global (static storage)
// Constructor & destructor always called

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for b
  // g() not called
  out << "leaving main()" << endl;
} ///:~
```

In **Obj**, the **char c** acts as an identifier so the constructor and destructor can print out information about the object they're working on. The **Obj a** is a global object, so the constructor is always called for it before **main( )** is entered, but the constructors for the **static Obj b** inside **f( )** and the **static Obj C** inside **g( )** are called only if those functions are called.

To demonstrate which constructors and destructors are called, inside **main( )** only **f( )** is called. The output of the program is

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
leaving main()
Obj::~Obj() for b
Obj::~Obj() for a
```

The constructor for **a** is called before **main( )** is entered, and the constructor for **b** is called only because **f( )** is called. When **main( )** exits, the destructors for the objects that have been constructed are called in reverse order of their construction. This means that if **g( )** *is* called, the order in which the destructors for **b** and **c** are called depends on whether **f( )** or **g( )** is called first.

Notice that the trace file **ofstream** object **out** is also a static object. It is important that its definition (as opposed to an **extern** declaration) appear at the beginning of the file, before there is any possible use of **out**. Otherwise you'll be using an object before it is properly initialized.

In C++ the constructor for a global static object is called before **main( )** is entered, so you now have a simple and portable way to execute code before entering **main( )** and to execute code with the destructor after exiting **main( )**. In C this was always a trial that required you to root around in the compiler vendor's assembly-language startup code.

Controlling linkage

Ordinarily, any name at *file scope* (that is, not nested inside a class or function) is visible throughout all translation units in a program. This is often called *external linkage* because at link time the name is visible to the linker everywhere, external to that translation unit. Global variables and ordinary functions have external linkage.

There are times when you'd like to limit the visibility of a name. You might like to have a variable at file scope so all the functions in that file can use it, but you don't want functions outside that file to see or access that

variable, or to inadvertently cause name clashes with identifiers outside the file.

An object or function name at file scope that is explicitly declared **static** is local to its translation unit (in the terms of this book, the **cpp** file where the declaration occurs); that name has *internal linkage*. This means you can use the same name in other translation units without a name clash.

One advantage to internal linkage is that the name can be placed in a header file without worrying that there will be a clash at link time. Names that are commonly placed in header files, such as **const** definitions and **inline** functions, default to internal linkage. (However, **const** defaults to internal linkage only in C++; in C it defaults to external linkage.) Note that linkage refers only to elements that have addresses at link/load time; thus, class declarations and local variables have no linkage.

Confusion

Here's an example of how the two meanings of **static** can cross over each other. All global objects implicitly have static storage class, so if you say (at file scope),

```
int a = 0;
```

then storage for **a** will be in the program's static data area, and the initialization for **a** will occur once, before **main( )** is entered. In addition, the visibility of **a** is global, across all translation units. In terms of visibility, the opposite of **static** (visible only in this translation unit) is **extern**, which explicitly states that the visibility of the name is across all translation units. So the above definition is equivalent to saying

```
extern int a = 0;
```

But if you say instead,

```
static int a = 0;
```

all you've done is change the visibility, so **a** has internal linkage. The storage class is unchanged – the object resides in the static data area whether the visibility is **static** or **extern**.

Once you get into local variables, **static** stops altering the visibility (and **extern** has no meaning) and instead alters the storage class.

With function names, **static** and **extern** can only alter visibility, so if you say,

```
extern void f();
```

it's the same as the unadorned declaration

```
void f();
```

and if you say,

```
static void f();
```

it means **f( )** is visible only within this translation unit; this is sometimes called *file static*.

Other storage class specifiers

You will see **static** and **extern** used commonly. There are two other storage class specifiers that occur less often. The **auto** specifier is almost never used because it tells the compiler that this is a local variable. The compiler can always determine this fact from the context in which the variable is defined, so **auto** is redundant.

A **register** variable is a local (**auto**) variable, along with a hint to the compiler that this particular variable will be heavily used, so the compiler ought to keep it in a register if it can. Thus, it is an optimization aid. Various compilers respond differently to this hint; they have the option to ignore it. If you take the address of the variable, the **register** specifier will almost certainly be ignored. You should avoid using **register** because the compiler can usually do a better job at of optimization than you.

Namespaces

Although names can be nested inside classes, the names of global functions, global variables, and classes are still in a single global name space. The **static** keyword gives you some control over this by allowing you to give variables and functions internal linkage (make them file static). But in a large project, lack of control over the global name space can cause problems. To solve these problems for classes, vendors often create long complicated names that are unlikely to clash, but then you're stuck typing those names. (A **typedef** is often used to simplify this.) It's not an elegant, language-supported solution.

You can subdivide the global name space into more manageable pieces using the *namespace* feature of C++.[35] The **namespace** keyword, like

[35] Your compiler may not have implemented this feature yet; check your local documentation.

class, **struct**, **enum**, and **union**, puts the names of its members in a distinct space. While the other keywords have additional purposes, the creation of a new name space is the only purpose for **namespace**.

Creating a namespace

The creation of a namespace is notably similar to the creation of a **class**:

```
namespace MyLib {
  // Declarations
}
```

This produces a new **namespace** containing the enclosed declarations. There are significant differences with **class**, **struct**, **union** and **enum**, however:

1. A **namespace** definition can only appear at the global scope, but namespaces can be nested within each other.
2. No terminating semicolon is necessary after the closing brace of a **namespace** definition.
3. A **namespace** definition can be "continued" over multiple header files using a syntax that would appear to be a redefinition for a class:

```
//: C10:Header1.h
namespace MyLib {
  extern int x;
  void f();
  // ...
} ///:~
//: C10:Header2.h
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
  extern int y;
  void g();
  // ...
} ///:~
```

4. A namespace name can be *aliased* to another name, so you don't have to type an unwieldy name created by a library vendor:

```
namespace BobsSuperDuperLibrary {
```

```
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
```

5. You cannot create an instance of a namespace as you can with a class.

Unnamed namespaces

Each translation unit contains an unnamed namespace that you can add to by saying **namespace** without an identifier:

```
namespace {
  class Arm  { /* ... */ };
  class Leg  { /* ... */ };
  class Head { /* ... */ };
  class Robot {
    Arm arm[4];
    Leg leg[16];
    Head head[3];
    // ...
  } xanthan;
  int i, j, k;
}
```

The names in this space are automatically available in that translation unit without qualification. It is guaranteed that an unnamed space is unique for each translation unit. If you put local names in an unnamed namespace, you don't need to give them internal linkage by making them **static**.

Friends

You can *inject* a **friend** declaration into a namespace by declaring it within an enclosed class:

```
namespace me {
  class Us {
    //...
    friend you();
  };
}
```

Now the function **you( )** is a member of the namespace **me**.

Using a namespace

You can refer to a name within a namespace in two ways: one name at a time, using the scope resolution operator, and more expediently with the **using** keyword.

Scope resolution

Any name in a namespace can be explicitly specified using the scope resolution operator, just like the names within a class:

```
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void func();
}
int X::Y::i = 9;
class X::Z {
  int u, v, w;
public:
  Z(int i);
  int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::func() {
  X::Z a(1);
  a.g();
}
```

So far, namespaces look very much like classes.

The **using** directive

Because it can rapidly get tedious to type the full qualification for an identifier in a namespace, the **using** keyword allows you to import an entire namespace at once. When used in conjunction with the

namespace keyword, this is called a *using directive*. The **using** directive declares all the names of a namespace to be in the current scope, so you can conveniently use the unqualified names:

```
namespace math {
  enum sign { positive, negative };
  class Integer {
    int i;
    sign s;
  public:
    Integer(int ii = 0)
      : i(ii),
        s(i >= 0 ? positive : negative)
    {}
    sign getSign() { return s; }
    void setSign(sign sgn) { s = sgn; }
    // ...
  };
  Integer a, b, c;
  Integer divide(Integer, Integer);
  // ...
}
```

Now you can declare all the names in **math** inside a function, but leave those names nested within the function:

```
void arithmetic() {
  using namespace math;
  Integer x;
  x.setSign(positive);
}
```

Without the **using** directive, all the names in the namespace would need to be fully qualified.

One aspect of the **using** directive may seem slightly counterintuitive at first. The visibility of the names introduced with a **using** directive is the scope where the directive is made. But you can override the names from the **using** directive as if they've been declared globally to that scope!

```
void q() {
  using namespace math;
  Integer A; // Hides math::A;
  A.setSign(negative);
  math::A.setSign(positive);
```

```
}
```

If you have a second namespace:

```
namespace calculation {
  class Integer {};
  Integer divide(Integer, Integer);
  // ...
}
```

And this namespace is also introduced with a **using** directive, you have the possibility of a collision. However, the ambiguity appears at the point of *use* of the name, not at the **using** directive:

```
void s() {
  using namespace math;
  using namespace calculation;
  // Everything's ok until:
  divide(1, 2); // Ambiguity
}
```

Thus it's possible to write **using** directives to introduce a number of namespaces with conflicting names without ever producing an ambiguity.

The **using** declaration

You can introduce names one at a time into the current scope with a *using declaration*. Unlike the **using** directive, which treats names as if they were declared globally to the scope, a **using** declaration is a declaration within the current scope. This means it can override names from a **using** directive:

```
namespace U {
  void f();
  void g();
}
namespace V {
  void f();
  void g();
}
void func() {
  using namespace U; // Using directive
  using V::f; // Using declaration
  f(); // Calls V::f();
  U::f(); // Must fully qualify to call
```

```
}
```

The **using** declaration just gives the fully specified name of the identifier, but no type information. This means that if the namespace contains a set of overloaded functions with the same name, the **using** declaration declares all the functions in the overloaded set.

You can put a **using** declaration anywhere a normal declaration can occur. A **using** declaration works like a normal declaration in all ways but one: it's possible for a **using** declaration to cause the overload of a function with the same argument types (which isn't allowed with normal overloading). This ambiguity, however, doesn't show up until the point of use, rather than the point of declaration.

A using declaration can also appear within a namespace, and it has the same effect as anywhere else: that name is declared within the space:

```
namespace Q {
  using U::f;
  using V::g;
  // ...
}
void m() {
  using namespace Q;
  f(); // Calls U::f();
  g(); // Calls V::g();
}
```

A using declaration is an alias, and it allows you to declare the same function in separate namespaces. If you end up redeclaring the same function by importing different namespaces, it's OK – there won't be any ambiguities or duplications.

Static members in C++

There are times when you need a single storage space to be used by all objects of a class. In C, you would use a global variable, but this is not very safe. Global data can be modified by anyone, and its name can clash with other identical names in a large project. It would be ideal if the data could be stored as if it were global, but be hidden inside a class, and clearly associated with that class.

This is accomplished with **static** data members inside a class. There is a single piece of storage for a **static** data member, regardless of how many

objects of that class you create. All objects share the same **static** storage space for that data member, so it is a way for them to "communicate" with each other. But the **static** data belongs to the class; its name is scoped inside the class and it can be **public**, **private**, or **protected**.

Defining storage for static data members

Because **static** data has a single piece of storage regardless of how many objects are created, that storage must be defined in a single place. The compiler will not allocate storage for you, although this was once true, with some compilers. The linker will report an error if a **static** data member is declared but not defined.

The definition must occur outside the class (no inlining is allowed), and only one definition is allowed. Thus it is usual to put it in the implementation file for the class. The syntax sometimes gives people trouble, but it is actually quite logical. For example,

```
class A {
  static int i;
public:
  //...
};
```

and later, in the definition file,

```
int A::i = 1;
```

If you were to define an ordinary global variable, you would say

```
int i = 1;
```

but here, the scope resolution operator and the class name are used to specify **A::i**.

Some people have trouble with the idea that **A::i** is **private**, and yet here's something that seems to be manipulating it right out in the open. Doesn't this break the protection mechanism? It's a completely safe practice for two reasons. First, the only place this initialization is legal is in the definition. Indeed, if the **static** data were an object with a constructor, you would call the constructor instead of using the **=** operator. Secondly, once the definition has been made, the end-user cannot make a second definition – the linker will report an error. And the class creator is forced to create the definition, or the code won't link

during testing. This ensures that the definition happens only once and that it's in the hands of the class creator.

The entire initialization expression for a static member is in the scope of the class. For example,

```
//: C10:Statinit.cpp
// Scope of static initializer
#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "WithStatic::x = " << x << endl;
    cout << "WithStatic::y = " << y << endl;
  }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
  WithStatic ws;
  ws.print();
} ///:~
```

Here, the qualification **WithStatic::** extends the scope of **WithStatic** to the entire definition.

static array initialization

It's possible to create **static const** objects as well as arrays of **static** objects, both **const** and non-**const**. Here's the syntax you use to initialize such elements:

```
//: C10:StaticArray.cpp
// Initializing static arrays

class Values {
```

```
  // static consts can be initialized in-place:
  static const int scSize = 100;
  // Automatic counting works with static consts:
  static const float scTable[] = {
    1.1, 2.2, 3.3, 4.4
  };
  static const char scLetters[] = {
    'a', 'b', 'c', 'd', 'e',
    'f', 'g', 'h', 'i', 'j'
  };
  // Non-const statics must be
  // initialized externally:
  static int size;
  static float table[4];
  static char letters[10];
};

int Values::size = 100;

float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

int main() { Values v; } ///:~
```

With **static const**s you provide the definitions inline, but for ordinary **static** member data, you must provide a single external definition for the member. These definitions have internal linkage, so they can be placed in header files. The syntax for initializing static arrays is the same as any aggregate, but you cannot use automatic counting for non-**static const** arrays. The compiler must have enough knowledge about the class to create an object by the end of the class definition, including the exact sizes of all the components.

Nested and local classes

You can easily put static data members in classes that are nested inside other classes. The definition of such members is an intuitive and obvious

extension – you simply use another level of scope resolution. However, you cannot have **static** data members inside local classes (a local class is a class defined inside a function). Thus,

```
//: C10:Local.cpp
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
  class Inner {
    static int i; // OK
  };
};

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
  class Local {
  public:
//! static int i;  // Error
    // (How would you define i?)
  } x;
}

int main() { Outer x; f(); } ///:~
```

You can see the immediate problem with a **static** member in a local class: How do you describe the data member at file scope in order to define it? In practice, local classes are used very rarely.

static member functions

You can also create **static** member functions that, like **static** data members, work for the class as a whole rather than for a particular object of a class. Instead of making a global function that lives in and "pollutes" the global or local namespace, you bring the function inside the class. When you create a **static** member function, you are expressing an association with a particular class.

You can call a **static** member function in the ordinary way, with the dot or the arrow, in association with an object. However, it's more typical to call

a **static** member function by itself, without any specific object, using the scope-resolution operator, like this:

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

X::f();
```

When you see static member functions in a class, remember that the designer intended that function to be conceptually associated with the class as a whole.

A **static** member function cannot access ordinary data members, only **static** data members. It can call only other **static** member functions. Normally, the address of the current object (**this**) is quietly passed in when any member function is called, but a **static** member has no **this**, which is the reason it cannot access ordinary members. Thus, you get the tiny increase in speed afforded by a global function, which doesn't have the extra overhead of passing **this**, but the benefits of having the function inside the class.

For data members, **static** indicates that only one piece of storage for member data exists for all objects of a class. This parallels the use of **static** to define objects *inside* a function, to mean that only one copy of a local variable is used for all calls of that function.

Here's an example showing **static** data members and **static** member functions used together:

```
//: C10:StaticMemberFunctions.cpp

class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
     // Non-static member function can access
     // static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member function
```

```
    // cannot access non-static member data
    return ++j;
  }
  static int f() {
    //! val(); // Error: static member function
    // cannot access non-static member function
    return incr(); // OK -- calls static
  }
};

int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static members
} ///:~
```

Because they have no **this** pointer, **static** member functions can neither access non**static** data members nor call non**static** member functions. (Those functions require a **this** pointer.)

Notice in **main( )** that a **static** member can be selected using the usual dot or arrow syntax, associating that function with an object, but also with no object (because a **static** member is associated with a class, not a particular object), using the class name and scope resolution operator.

Here's an interesting feature: Because of the way initialization happens for **static** member objects, you can put a **static** data member of the same class *inside* that class. Here's an example that allows only a single object of type **egg** to exist by making the constructor private. You can access that object, but you can't create any new **egg** objects:

```
//: C10:Selfmem.cpp
// Static member of same type
// ensures only one object of this type exists.
// Also referred to as a "singleton" pattern.
#include <iostream>
using namespace std;

class Egg {
  static Egg e;
```

```
  int i;
  Egg(int ii) : i(ii) {}
public:
  static Egg* instance() { return &e; }
  int val() { return i; }
};

Egg Egg::e(47);

int main() {
//!  Egg x(1); // Error -- can't create an Egg
  // You can access the single instance:
  cout << Egg::instance()->val() << endl;
} ///:~
```

The initialization for **E** happens after the class declaration is complete, so the compiler has all the information it needs to allocate storage and make the constructor call.

Static initialization dependency

Within a specific translation unit, the order of initialization of static objects is guaranteed to be the order in which the object definitions appear in that translation unit. The order of destruction is guaranteed to be the reverse of the order of initialization.

However, there is no guarantee concerning the order of initialization of static objects *across* translation units, and there's no way to specify this order. This can cause significant problems. As an example of an instant disaster (which will halt primitive operating systems, and kill the process on sophisticated ones), if one file contains

```
// First file
#include <fstream>
ofstream out("out.txt");
```

and another file uses the **out** object in one of its initializers

```
// Second file
#include <fstream>
extern ofstream out;
```

```
class Oof {
public:
  Oof() { out << "ouch"; }
} oof;
```

the program may work, and it may not. If the programming environment builds the program so that the first file is initialized before the second file, then there will be no problem. However, if the second file is initialized before the first, the constructor for **oof** relies upon the existence of **out**, which hasn't been constructed yet and this causes chaos. This is only a problem with static object initializers *that depend on each other*, because by the time you get into **main( )**, all constructors for static objects have already been called.

A more subtle example can be found in the ARM.[36] In one file,

```
extern int y;
int x = y + 1;
```

and in a second file,

```
extern int x;
int y = x + 1;
```

For all static objects, the linking-loading mechanism guarantees a static initialization to zero before the dynamic initialization specified by the programmer takes place. In the previous example, zeroing of the storage occupied by the **fstream out** object has no special meaning, so it is truly undefined until the constructor is called. However, with built-in types, initialization to zero *does* have meaning, and if the files are initialized in the order they are shown above, **y** begins as statically initialized to zero, so **x** becomes one, and **y** is dynamically initialized to two. However, if the files are initialized in the opposite order, **x** is statically initialized to zero, **y** is dynamically initialized to one, and **x** then becomes two.

Programmers must be aware of this because they can create a program with static initialization dependencies and get it working on one platform, but move it to another compiling environment where it suddenly, mysteriously, doesn't work.

---

[36] Bjarne Stroustrup and Margaret Ellis, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990, pp. 20-21.

What to do

There are three approaches to dealing with this problem:

1. Don't do it. Avoiding static initializer dependencies is the best solution.
2. If you must do it, put the critical static object definitions in a single file, so you can portably control their initialization by putting them in the correct order.
3. If you're convinced it's unavoidable to scatter static objects across translation units – as in the case of a library, where you can't control the programmer who uses it – there is a technique pioneered by Jerry Schwarz while creating the iostream library (because the definitions for **cin**, **cout**, and **cerr** live in a separate file).

This technique requires an additional class in your library header file. This class is responsible for the dynamic initialization of your library's static objects. Here is a simple example:

```
//: C10:Depend.h
// Static initialization technique
#ifndef DEPEND_H
#define DEPEND_H
#include <iostream>
extern int x; // Declarations, not definitions
extern int y;

class Initializer {
  static int init_count;
public:
  Initializer() {
    std::cout << "Initializer()" << std::endl;
    // Initialize first time only
    if(init_count++ == 0) {
      std::cout << "performing initialization"
           << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Initializer() {
    std::cout << "~Initializer()" << std::endl;
    // Clean up last time only
```

```
    if(--init_count == 0) {
      std::cout << "performing cleanup"
        << std::endl;
      // Any necessary cleanup here
    }
  }
};

// The following creates one object in each
// file where DEPEND.H is included, but that
// object is only visible within that file:
static Initializer init;
#endif // DEPEND_H ///:~
```

The declarations for **x** and **y** announce only that these objects exist, but don't allocate storage for them. However, the definition for the **Initializer init** allocates storage for that object in every file where the header is included, but because the name is **static** (controlling visibility this time, not the way storage is allocated because that is at file scope by default), it is only visible within that translation unit, so the linker will not complain about multiple definition errors.

Here is the file containing the definitions for **x**, **y**, and **init_count**:

```
//: C10:Depdefs.cpp {O}
// Definitions for DEPEND.H
#include "Depend.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int Initializer::init_count;
///:~
```

(Of course, a file static instance of **init** is also placed in this file.) Suppose that two other files are created by the library user:

```
//: C10:Depend.cpp {O}
// Static initialization
#include "Depend.h"
///:~
```

and

```
//: C10:Depend2.cpp
//{L} Depdefs Depend
```

```
// Static initialization
#include "Depend.h"
using namespace std;

int main() {
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
} ///:~
```

Now it doesn't matter which translation unit is initialized first. The first time a translation unit containing **Depend.h** is initialized, **init_count** will be zero so the initialization will be performed. (This depends heavily on the fact that global objects of built-in types are set to zero before any dynamic initialization takes place.) For all the rest of the translation units, the initialization will be skipped. Cleanup happens in the reverse order, and **~Initializer( )** ensures that it will happen only once.

This example used built-in types as the global static objects. The technique also works with classes, but those objects must then be dynamically initialized by the **Initializer** class. One way to do this is to create the classes without constructors and destructors, but instead with initialization and cleanup member functions using different names. A more common approach, however, is to have pointers to objects and to create them dynamically on the heap inside **Initializer( )**. This requires the use of two C++ keywords, **new** and **delete**, which will be explored in Chapter XX.

Alternate linkage specifications

What happens if you're writing a program in C++ and you want to use a C library? If you make the C function declaration,

```
float f(int a, char b);
```

the C++ compiler will decorate this name to something like **_f_int_char** to support function overloading (and type-safe linkage). However, the C compiler that compiled your C library has most definitely *not* decorated the name, so its internal name will be **_f**. Thus, the linker will not be able to resolve your C++ calls to **f( )**.

The escape mechanism provided in C++ is the *alternate linkage specification*, which was produced in the language by overloading the **extern** keyword. The **extern** is followed by a string that specifies the linkage you want for the declaration, followed by the declaration itself:

```
extern "C" float f(int a, char b);
```

This tells the compiler to give C linkage to **f( )**; that is, don't decorate the name. The only two types of linkage specifications supported by the standard are **"C"** and **"C++,"** but compiler vendors have the option of supporting other languages in the same way.

If you have a group of declarations with alternate linkage, put them inside braces, like this:

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

Or, for a header file,

```
extern "C" {
#include "Myheader.h"
}
```

Most C++ compiler vendors handle the alternate linkage specifications inside their header files that work with both C and C++, so you don't have to worry about it.

Summary

The **static** keyword can be confusing because in some situations it controls the location of storage, and in others it controls visibility and linkage of a name.

With the introduction of C++ namespaces, you have an improved and more flexible alternative to control the proliferation of names in large projects.

The use of **static** inside classes is one more way to control names in a program. The names do not clash with global names, and the visibility and access is kept within the program, giving you greater control in the maintenance of your code.

Exercises

1. Create a class that holds an array of **int**s. Set the size of the array using an untagged enumeration inside the class. Add a **const int** variable, and initialize it in the constructor initializer list. Add a **static int** member variable and initialize it to a specific value. Add a **static** member function that prints the **static** data member. Add an **inline** constructor and an **inline** member function called **print( )** to print out all the values in the array, and to call the static member function.
2. In **StaticDestructors.cpp**, experiment with the order of constructor and destructor calls by calling **f( )** and **g( )** inside **main( )** in different orders. Does your compiler get it right?
3. In **StaticDestructors.cpp**, test the default error handling of your implementation by turning the original definition of **out** into an **extern** declaration and putting the actual definition after the definition of **A** (whose **obj** constructor sends information to **out**). Make sure there's nothing else important running on your machine when you run the program or that your machine will handle faults robustly.
4. Create a class with a destructor that prints a message and then calls **exit( )**. Create a global static object of this class and see what happens.
5. Modify **Volatile.cpp** from Chapter 8 to make **comm::isr( )** something that would actually work as an interrupt service routine.

11: References & the copy-constructor

References are a C++ feature that are like constant pointers automatically dereferenced by the compiler.

Although references also exist in Pascal, the C++ version was taken from the Algol language. They are essential in C++ to support the syntax of operator overloading (see Chapter XX), but are also a general convenience to control the way arguments are passed into and out of functions.

This chapter will first look briefly at the differences between pointers in C and C++, then introduce references. But the bulk of the chapter will delve into a rather confusing issue for the new C++ programmer: the copy-constructor, a special constructor (requiring references) that makes a new object from an existing object of the same type. The copy-constructor is used by the compiler to pass and return objects *by value* into and out of functions.

Finally, the somewhat obscure C++ *pointer-to-member* feature is illuminated.

Pointers in C++

The most important difference between pointers in C and in C++ is that C++ is a more strongly typed language. This stands out where **void*** is

concerned. C doesn't let you casually assign a pointer of one type to another, but it *does* allow you to quietly accomplish this through a **void***. Thus,

```
bird* b;
rock* r;
void* v;
v = r;
b = v;
```

C++ doesn't allow this because it leaves a big hole in the type system. The compiler gives you an error message, and if you really want to do it, you must make it explicit, both to the compiler and to the reader, using a cast. (See Chapter XX for C++'s improved casting syntax.)

References in C++

A *reference* (**&**) is like a constant pointer that is automatically dereferenced. It is usually used for function argument lists and function return values. But you can also make a free-standing reference. For example,

```
int x;
int& r = x;
```

When a reference is created, it must be initialized to a live object. However, you can also say

```
int& q = 12;
```

Here, the compiler allocates a piece of storage, initializes it with the value 12, and ties the reference to that piece of storage. The point is that any reference must be tied to someone *else's* piece of storage. When you access a reference, you're accessing that storage. Thus if you say,

```
int x = 0;
int& a = x;
a++;
```

incrementing **a** is actually incrementing **x**. Again, the easiest way to think about a reference is as a fancy pointer. One advantage of this pointer is you never have to wonder whether it's been initialized (the compiler enforces it) and how to dereference it (the compiler does it).

There are certain rules when using references:

1. A reference must be initialized when it is created. (Pointers can be initialized at any time.)
2. Once a reference is initialized to an object, it cannot be changed to refer to another object. (Pointers can be pointed to another object at any time.)
3. You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

References in functions

The most common place you'll see references is in function arguments and return values. When a reference is used as a function argument, any modification to the reference *inside* the function will cause changes to the argument *outside* the function. Of course, you could do the same thing by passing a pointer, but a reference has much cleaner syntax. (You can think of a reference as nothing more than a syntax convenience, if you want.)

If you return a reference from a function, you must take the same care as if you return a pointer from a function. Whatever the reference is connected to shouldn't go away when the function returns; otherwise you'll be referring to unknown memory.

Here's an example:

```
//: C11:Reference.cpp
// Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe; x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe; outside this scope
}

int& h() {
  int q;
//!  return q;  // Error
```

```
  static int x;
  return x; // Safe; x lives outside scope
}

int main() {
  int a = 0;
  f(&a); // Ugly (but explicit)
  g(a);  // Clean (but hidden)
} ///:~
```

The call to **f( )** doesn't have the convenience and cleanliness of using references, but it's clear that an address is being passed. In the call to **g( )**, an address is being passed (via a reference), but you don't see it.

const references

The reference argument in **Reference.cpp** works only when the argument is a non-**const** object. If it is a **const** object, the function **g( )** will not accept the argument, which is actually a good thing, because the function *does* modify the outside argument. If you know the function will respect the **const**ness of an object, making the argument a **const** reference will allow the function to be used in all situations. This means that, for built-in types, the function will not modify the argument, and for user-defined types the function will call only **const** member functions, and won't modify any **public** data members.

The use of **const** references in function arguments is especially important because your function may receive a temporary object, created as a return value of another function or explicitly by the user of your function. Temporary objects are always **const**, so if you don't use a **const** reference, that argument won't be accepted by the compiler. As a very simple example,

```
//: C11:Pasconst.cpp
// Passing references as const

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Error
  g(1);
} ///:~
```

The call to **f(1)** produces a compiler error because the compiler must first create a reference. It does so by allocating storage for an **int**, initializing it to one and producing the address to bind to the reference. The storage *must* be a **const** because changing it would make no sense – you can never get your hands on it again. With all temporary objects you must make the same assumption, that they're inaccessible. It's valuable for the compiler to tell you when you're changing such data because the result would be lost information.

Pointer references

In C, if you wanted to modify the *contents* of the pointer rather than what it points to, your function declaration would look like

```
void f(int**);
```

and you'd have to take the address of the pointer when passing it in:

```
int i = 47;
int* ip = &i;
f(&ip);
```

With references in C++, the syntax is cleaner. The function argument becomes a reference to a pointer, and you no longer have to take the address of that pointer. Thus,

```
//: C11:Refptr.cpp
// Reference to pointer
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

By running this program, you'll prove to yourself that the pointer itself is incremented, not what it points to.

Argument-passing guidelines

Your normal habit when passing an argument to a function should be to pass by **const** reference. Although this may at first seem like only an efficiency concern (and you normally don't want to concern yourself with efficiency tuning while you're designing and assembling your program), there's more at stake: as you'll see in the remainder of the chapter, a copy-constructor is required to pass an object by value, and this isn't always available.

The efficiency savings can be substantial for such a simple habit: to pass an argument by value requires a constructor and destructor call, but if you're not going to modify the argument then passing by **const** reference only needs an address pushed on the stack.

In fact, virtually the only time passing an address *isn't* preferable is when you're going to do such damage to an object that passing by value is the only safe approach (rather than modifying the outside object, something the caller doesn't usually expect). This is the subject of the next section.

The copy-constructor

Now that you understand the basics of the reference in C++, you're ready to tackle one of the more confusing concepts in the language: the copy-constructor, often called **X(X&)** ("X of X ref"). This constructor is essential to control passing and returning of user-defined types by value during function calls.

Passing & returning by value

To understand the need for the copy-constructor, consider the way C handles passing and returning variables by value during function calls. If you declare a function and make a function call,

```
int f(int x, char c);
int g = f(a, b);
```

how does the compiler know how to pass and return those variables? It just knows! The range of the types it must deal with is so small – **char**, **int**, **float**, and **double** and their variations – that this information is built into the compiler.

If you figure out how to generate assembly code with your compiler and determine the statements generated by the function call to **f( )**, you'll get the equivalent of,

```
push  b
push  a
call  f()
add  sp,4
mov  g, register a
```

This code has been cleaned up significantly to make it generic – the expressions for **b** and **a** will be different depending on whether the variables are global (in which case they will be **_b** and **_a**) or local (the compiler will index them off the stack pointer). This is also true for the expression for **g**. The appearance of the call to **f( )** will depend on your name-decoration scheme, and "register a" depends on how the CPU registers are named within your assembler. The logic behind the code, however, will remain the same.

In C and C++, arguments are pushed on the stack from right to left, the function call is made, then the calling code is responsible for cleaning the arguments off the stack (which accounts for the **add sp,4**). But notice that to pass the arguments by value, the compiler simply pushes copies on the stack – it knows how big they are and that pushing those arguments makes accurate copies of them.

The return value of **f( )** is placed in a register. Again, the compiler knows everything there is to know about the return value type because it's built into the language, so the compiler can return it by placing it in a register. The simple act of copying the bits of the value is equivalent to copying the object.

Passing & returning large objects

But now consider user-defined types. If you create a class and you want to pass an object of that class by value, how is the compiler supposed to know what to do? This is no longer a built-in type the compiler writer knows about; it's a type someone has created since then.

To investigate this, you can start with a simple structure that is clearly too large to return in registers:

```
//: C11:PassStruct.cpp
// Passing a big structure

struct Big {
```

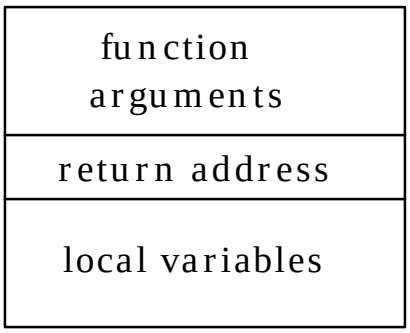
```
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

Decoding the assembly output is a little more complicated here because most compilers use "helper" functions rather than putting all functionality inline. In **main( )**, the call to **bigfun( )** starts as you might guess – the entire contents of **B** is pushed on the stack. (Here, you might see some compilers load registers with the address of **B** and its size, then call a helper function to push it onto the stack.)

In the previous example, pushing the arguments onto the stack was all that was required before making the function call. In **PassStruct.cpp**, however, you'll see an additional action: The address of **B2** is pushed before making the call, even though it's obviously not an argument. To comprehend what's going on here, you need to understand the constraints on the compiler when it's making a function call.

Function-call stack frame

When the compiler generates code for a function call, it first pushes all the arguments on the stack, then makes the call. Inside the function itself, code is generated to move the stack pointer down even further to provide storage for the function's local variables. ("Down" is relative here; your machine may increment or decrement the stack pointer during a push.) But during the assembly-language CALL, the CPU pushes the address in the program code where the function call *came from*, so the assembly-language RETURN can use that address to return to the calling point. This address is of course sacred, because without it your program will get completely lost. Here's what the stack frame looks like after the CALL and the allocation of local variable storage in the function:



The code generated for the rest of the function expects the memory to be laid out exactly this way, so it can carefully pick from the function arguments and local variables without touching the return address. I shall call this block of memory, which is everything used by a function in the process of the function call, the *function frame*.

You might think it reasonable to try to return values on the stack. The compiler could simply push it, and the function could return an offset to indicate how far down in the stack the return value begins.

Re-entrancy

The problem occurs because functions in C and C++ support interrupts; that is, the languages are *re-entrant*. They also support recursive function calls. This means that at any point in the execution of a program an interrupt can occur without disturbing the program. Of course, the person who writes the interrupt service routine (ISR) is responsible for saving and restoring all the registers he uses, but if the ISR needs to use any memory that's further down on the stack, that must be a safe thing to do. (You can think of an ISR as an ordinary function with no arguments and **void** return value that saves and restores the CPU state. An ISR function call is triggered by some hardware event rather than an explicit call from within a program.)

Now imagine what would happen if the called function tried to return values on the stack from an ordinary function. You can't touch any part of the stack that's above the return address, so the function would have to push the values below the return address. But when the assembly-language RETURN is executed, the stack pointer must be pointing to the return address (or right below it, depending on your machine), so right before the RETURN, the function must move the stack pointer up, thus clearing off all its local variables. If you're trying to return values on the stack below the return address, you become vulnerable at that moment

because an interrupt could come along. The ISR would move the stack pointer down to hold its return address and its local variables and overwrite your return value.

To solve this problem, the caller could be responsible for allocating the extra storage on the stack for the return values *before* calling the function. However, C was not designed this way, and C++ must be compatible. As you'll see shortly, the C++ compiler uses a more efficient scheme.

Your next idea might be to return the value in some global data area, but this doesn't work either. Re-entrancy means that any function can interrupt any other function, *including the same function you're currently inside*. Thus, if you put the return value in a global area, you might return into the same function, which would overwrite that return value. The same logic applies to recursion.

The only safe place to return values is in the registers, so you're back to the problem of what to do when the registers aren't large enough to hold the return value. The answer is to push the address of the return value's destination on the stack as one of the function arguments, and let the function copy the return information directly into the destination. This not only solves all the problems, it's more efficient. It's also the reason that, in **PassStruct.cpp**, the compiler pushes the address of **B2** before the call to **bigfun( )** in **main( )**. If you look at the assembly output for **bigfun( )**, you can see it expects this hidden argument and performs the copy to the destination *inside* the function.

Bitcopy versus initialization

So far, so good. There's a workable process for passing and returning large simple structures. But notice that all you have is a way to copy the bits from one place to another, which certainly works fine for the primitive way that C looks at variables. But in C++ objects can be much more sophisticated than a patch of bits; they have meaning. This meaning may not respond well to having its bits copied.

Consider a simple example: a class that knows how many objects of its type exist at any one time. From Chapter XX, you know the way to do this is by including a **static** data member:

```
//: C11:HowMany.cpp
// Class counts its objects
#include <fstream>
using namespace std;
```

```
ofstream out("HowMany.out");

class HowMany {
  static int object_count;
public:
  HowMany() {
    object_count++;
  }
  static void print(const char* msg = 0) {
    if(msg) out << msg << ": ";
    out << "object_count = "
        << object_count << endl;
  }
  ~HowMany() {
    object_count--;
    print("~HowMany()");
  }
};

int HowMany::object_count = 0;

// Pass and return BY VALUE:
HowMany f(HowMany x) {
  x.print("x argument inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
} ///:~
```

The class **HowMany** contains a **static int** and a **static** member function **print( )** to report the value of that **int**, along with an optional message argument. The constructor increments the count each time an object is created, and the destructor decrements it.

The output, however, is not what you would expect:

```
after construction of h: object_count = 1
x argument inside f(): object_count = 1
~HowMany(): object_count = 0
```

```
after call to f(): object_count = 0
~HowMany(): object_count = -1
~HowMany(): object_count = -2
```

After **h** is created, the object count is one, which is fine. But after the call to **f( )** you would expect to have an object count of two, because **h2** is now in scope as well. Instead, the count is zero, which indicates something has gone horribly wrong. This is confirmed by the fact that the two destructors at the end make the object count go negative, something that should never happen.

Look at the point inside **f( )**, which occurs after the argument is passed by value. This means the original object **h** exists outside the function frame, and there's an additional object *inside* the function frame, which is the copy that has been passed by value. However, the argument has been passed using C's primitive notion of bitcopying, whereas the C++ **HowMany** class requires true initialization to maintain its integrity, so the default bitcopy fails to produce the desired effect.

When the local object goes out of scope at the end of the call to **f( )**, the destructor is called, which decrements **object_count**, so outside the function, **object_count** is zero. The creation of **h2** is also performed using a bitcopy, so the constructor isn't called there, either, and when **h** and **h2** go out of scope, their destructors cause the negative values of **object_count**.

Copy-construction

The problem occurs because the compiler makes an assumption about how to create *a new object from an existing object*. When you pass an object by value, you create a new object, the passed object inside the function frame, from an existing object, the original object outside the function frame. This is also often true when returning an object from a function. In the expression

```
HowMany h2 = f(h);
```

h2, a previously unconstructed object, is created from the return value of **f( )**, so again a new object is created from an existing one.

The compiler's assumption is that you want to perform this creation using a bitcopy, and in many cases this may work fine but in **HowMany** it doesn't fly because the meaning of initialization goes beyond simply copying. Another common example occurs if the class contains pointers –

what do they point to, and should you copy them or should they be connected to some new piece of memory?

Fortunately, you can intervene in this process and prevent the compiler from doing a bitcopy. You do this by defining your own function to be used whenever the compiler needs to make a new object from an existing object. Logically enough, you're making a new object, so this function is a constructor, and also logically enough, the single argument to this constructor has to do with the object you're constructing from. But that object can't be passed into the constructor by value because you're trying to define the function that handles passing by value, and syntactically it doesn't make sense to pass a pointer because, after all, you're creating the new object from an existing *object*. Here, references come to the rescue, so you take the reference of the source object. This function is called the *copy-constructor* and is often referred to as **X(X&)**, which is its appearance for a class called **X**.

If you create a copy-constructor, the compiler will not perform a bitcopy when creating a new object from an existing one. It will always call your copy-constructor. So, if you don't create a copy-constructor, the compiler will do something sensible, but you have the choice of taking over complete control of the process.

Now it's possible to fix the problem in **HowMany.cpp**:

```
//: C11:HowMany2.cpp
// The copy-constructor
#include <fstream>
#include <cstring>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  static const int bufsize = 30;
  char name[bufsize]; // Object identifier
  static int object_count;
public:
  HowMany2(const char* id = 0) {
    if(id) strncpy(name, id, bufsize);
    else *name = 0;
    ++object_count;
    print("HowMany2()");
  }
  // The copy-constructor:
```

```
  HowMany2(const HowMany2& h) {
    strncpy(name, h.name, bufsize);
    strncat(name, " copy", bufsize - strlen(name));
    ++object_count;
    print("HowMany2(HowMany2&)");
  }
  // Can't be static (printing name):
  void print(const char* msg = 0) const {
    if(msg) out << msg << endl;
    out << '\t' << name << ": "
        << "object_count = "
        << object_count << endl;
  }
  ~HowMany2() {
    --object_count;
    print("~HowMany2()");
  }
};

int HowMany2::object_count = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
  x.print("x argument inside f()");
  out << "returning from f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "entering f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "call f(), no return value" << endl;
  f(h);
  out << "after call to f()" << endl;
} ///:~
```

There are a number of new twists thrown in here so you can get a better idea of what's happening. First, the character buffer **name** acts as an object identifier so you can figure out which object the information is being printed about. In the constructor, you can put an identifier string (usually the name of the object) that is copied to **name** using the

Standard C library function **strncpy( )**, which only copies a certain number of characters, preventing overrun of the buffer.

Next is the copy-constructor, **HowMany2(HowMany2&)**. The copy-constructor can create a new object only from an existing one, so the existing object's name is copied to **name**, followed by the word "copy" so you can see where it came from. Note the use of the Standard C library function **strncat( )** to copy a maximum number of characters into **name**, again to prevent overrunning the end of the buffer.

Inside the copy-constructor, the object count is incremented just as it is inside the normal constructor. This means you'll now get an accurate object count when passing and returning by value.

The **print( )** function has been modified to print out a message, the object identifier, and the object count. It must now access the **name** data of a particular object, so it can no longer be a **static** member function.

Inside **main( )**, you can see a second call to **f( )** has been added. However, this call uses the common C approach of ignoring the return value. But now that you know how the value is returned (that is, code *inside* the function handles the return process, putting the result in a destination whose address is passed as a hidden argument), you might wonder what happens when the return value is ignored. The output of the program will throw some illumination on this.

Before showing the output, here's a little program that uses iostreams to add line numbers to any file:

```
//: C11:Linenum.cpp
// Add line numbers
#include "../require.h"
#include <fstream>
#include <strstream>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Adds line numbers to file");
  strstream text;
  {
    ifstream in(argv[1]);
    assure(in, argv[1]);
    text << in.rdbuf(); // Read in whole file
```

```
  } // Close file
  ofstream out(argv[1]); // Overwrite file
  assure(out, argv[1]);
  const int bsz = 100;
  char buf[bsz];
  int line = 0;
  while(text.getline(buf, bsz)) {
    out.setf(ios::right, ios::adjustfield);
    out.width(2);
    out << ++line << ") " << buf << endl;
  }
} ///:~
```

The entire file is read into a **strstream** (which can be both written to and read from) and the **ifstream** is closed with scoping. Then an **ofstream** is created for the same file, overwriting it. **getline( )** fetches a line at a time from the **strstream** and line numbers are added as the line is written back into the file.

The line numbers are printed right-aligned in a field width of two, so the output still lines up in its original configuration. You can change the program to add an optional second command-line argument that allows the user to select a field width, *or* you can be more clever and count all the lines in the file to determine the field width automatically.

When **Linenum.cpp** is applied to **HowMany2.out**, the result is

```
 1) HowMany2()
 2)   h: object_count = 1
 3) entering f()
 4) HowMany2(HowMany2&)
 5)   h copy: object_count = 2
 6) x argument inside f()
 7)   h copy: object_count = 2
 8) returning from f()
 9) HowMany2(HowMany2&)
10)   h copy copy: object_count = 3
11) ~HowMany2()
12)   h copy: object_count = 2
13) h2 after call to f()
14)   h copy copy: object_count = 2
15) call f(), no return value
16) HowMany2(HowMany2&)
17)   h copy: object_count = 3
```

```
18) x argument inside f()
19)   h copy: object_count = 3
20) returning from f()
21) HowMany2(HowMany2&)
22)   h copy copy: object_count = 4
23) ~HowMany2()
24)   h copy: object_count = 3
25) ~HowMany2()
26)   h copy copy: object_count = 2
27) after call to f()
28) ~HowMany2()
29)   h copy copy: object_count = 1
30) ~HowMany2()
31)   h: object_count = 0
```

As you would expect, the first thing that happens is the normal constructor is called for **h**, which increments the object count to one. But then, as **f( )** is entered, the copy-constructor is quietly called by the compiler to perform the pass-by-value. A new object is created, which is the copy of **h** (thus the name "h copy") inside the function frame of **f( )**, so the object count becomes two, courtesy of the copy-constructor.

Line eight indicates the beginning of the return from **f( )**. But before the local variable "h copy" can be destroyed (it goes out of scope at the end of the function), it must be copied into the return value, which happens to be **h2**. A previously unconstructed object (**h2**) is created from an existing object (the local variable inside **f( )**), so of course the copy-constructor is used again in line nine. Now the name becomes "h copy copy" for **h2**'s identifier because it's being copied from the copy that is the local object inside **f( )**. After the object is returned, but before the function ends, the object count becomes temporarily three, but then the local object "h copy" is destroyed. After the call to **f( )** completes in line 13, there are only two objects, **h** and **h2**, and you can see that **h2** did indeed end up as "h copy copy."

Temporary objects

Line 15 begins the call to **f(h)**, this time ignoring the return value. You can see in line 16 that the copy-constructor is called just as before to pass the argument in. And also, as before, line 21 shows the copy-constructor is called for the return value. But the copy-constructor must have an address to work on as its destination (a **this** pointer). Where is the object returned to?

It turns out the compiler can create a temporary object whenever it needs one to properly evaluate an expression. In this case it creates one you don't even see to act as the destination for the ignored return value of **f( )**. The lifetime of this temporary object is as short as possible so the landscape doesn't get cluttered up with temporaries waiting to be destroyed, taking up valuable resources. In some cases, the temporary might be immediately passed to another function, but in this case it isn't needed after the function call, so as soon as the function call ends by calling the destructor for the local object (lines 23 and 24), the temporary object is destroyed (lines 25 and 26).

Now, in lines 28-31, the **h2** object is destroyed, followed by **h**, and the object count goes correctly back to zero.

Default copy-constructor

Because the copy-constructor implements pass and return by value, it's important that the compiler will create one for you in the case of simple structures – effectively, the same thing it does in C. However, all you've seen so far is the default primitive behavior: a bitcopy.

When more complex types are involved, the C++ compiler will still automatically create a copy-constructor if you don't make one. Again, however, a bitcopy doesn't make sense, because it doesn't necessarily implement the proper meaning.

Here's an example to show the more intelligent approach the compiler takes. Suppose you create a new class composed of objects of several existing classes. This is called, appropriately enough, *composition*, and it's one of the ways you can make new classes from existing classes. Now take the role of a naive user who's trying to solve a problem quickly by creating the new class this way. You don't know about copy-constructors, so you don't create one. The example demonstrates what the compiler does while creating the default copy-constructor for your new class:

```
//: C11:Autocc.cpp
// Automatic copy-constructor
#include <iostream>
#include <cstring>
using namespace std;

class WithCC { // With copy-constructor
public:
  // Explicit default constructor required:
```

```
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};

class WoCC { // Without copy-constructor
  static const int bsz = 30;
  char buf[bsz];
public:
  WoCC(const char* msg = 0) {
    memset(buf, 0, bsz);
    if(msg) strncpy(buf, msg, bsz);
  }
  void print(const char* msg = 0) const {
    if(msg) cout << msg << ": ";
    cout << buf << endl;
  }
};

class Composite {
  WithCC withcc; // Embedded objects
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const char* msg = 0) {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("contents of c");
  cout << "calling Composite copy-constructor"
       << endl;
  Composite c2 = c;  // Calls copy-constructor
  c2.print("contents of c2");
} ///:~
```

The class **WithCC** contains a copy-constructor, which simply announces it has been called, and this brings up an interesting issue. In the class **Composite**, an object of **WithCC** is created using a default constructor. If there were no constructors at all in **WithCC**, the compiler would

automatically create a default constructor, which would do nothing in this case. However, if you add a copy-constructor, you've told the compiler you're going to handle constructor creation, so it no longer creates a default constructor for you and will complain unless you explicitly create a default constructor as was done for **WithCC**.

The class **WoCC** has no copy-constructor, but its constructor will store a message in an internal buffer that can be printed out using **print( )**. This constructor is explicitly called in **Composite**'s constructor initializer list (briefly introduced in Chapter XX and covered fully in Chapter XX). The reason for this becomes apparent later.

The class **Composite** has member objects of both **WithCC** and **WoCC** (note the embedded object **wocc** is initialized in the constructor-initializer list, as it must be), and no explicitly defined copy-constructor. However, in **main( )** an object is created using the copy-constructor in the definition:

```
Composite c2 = c;
```

The copy-constructor for **Composite** is created automatically by the compiler, and the output of the program reveals how it is created.

To create a copy-constructor for a class that uses composition (and inheritance, which is introduced in Chapter XX), the compiler recursively calls the copy-constructors for all the member objects and base classes. That is, if the member object also contains another object, its copy-constructor is also called. So in this case, the compiler calls the copy-constructor for **WithCC**. The output shows this constructor being called. Because **WoCC** has no copy-constructor, the compiler creates one for it, which is the default behavior of a bitcopy, and calls that inside the **Composite** copy-constructor. The call to **Composite::print( )** in main shows that this happens because the contents of **c2.wocc** are identical to the contents of **c.wocc**. The process the compiler goes through to synthesize a copy-constructor is called *memberwise initialization*.

It's best to always create your own copy-constructor rather than letting the compiler do it for you. This guarantees it will be under your control.

Alternatives to copy-construction

At this point your head may be swimming, and you might be wondering how you could have possibly written a functional class without knowing about the copy-constructor. But remember: You need a copy-constructor

only if you're going to pass an object of your class *by value*. If that never happens, you don't need a copy-constructor.

Preventing pass-by-value

"But," you say, "if I don't make a copy-constructor, the compiler will create one for me. So how do I know that an object will never be passed by value?"

There's a simple technique for preventing pass-by-value: Declare a **private** copy-constructor. You don't even need to create a definition, unless one of your member functions or a **friend** function needs to perform a pass-by-value. If the user tries to pass or return the object by value, the compiler will produce an error message because the copy-constructor is **private**. It can no longer create a default copy-constructor because you've explicitly stated you're taking over that job.

Here's an example:

```
//: C11:Stopcc.cpp
// Preventing copy-construction

class NoCC {
  int i;
  NoCC(const NoCC&); // No definition
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
//! f(n); // Error: copy-constructor called
//! NoCC n2 = n; // Error: c-c called
//! NoCC n3(n); // Error: c-c called
} ///:~
```

Notice the use of the more general form

```
NoCC(const NoCC&);
```

using the **const**.

Functions that modify outside objects

Reference syntax is nicer to use than pointer syntax, yet it clouds the meaning for the reader. For example, in the iostreams library one overloaded version of the **get( )** function takes a **char&** as an argument, and the whole point of the function is to modify its argument by inserting the result of the **get( )**. However, when you read code using this function it's not immediately obvious to you the outside object is being modified:

```
char c;
cin.get(c);
```

Instead, the function call looks like a pass-by-value, which suggests the outside object is *not* modified.

Because of this, it's probably safer from a code maintenance standpoint to use pointers when you're passing the address of an argument to modify. If you *always* pass addresses as **const** references *except* when you intend to modify the outside object via the address, where you pass by non-**const** pointer, then your code is far easier for the reader to follow.

Pointers to members

A pointer is a variable that holds the address of some location, which can be either data or a function, so you can change what a pointer selects at runtime. The C++ *pointer-to-member* follows this same concept, except that what it selects is a location inside a class. The dilemma here is that all a pointer needs is an address, but there is no "address" inside a class; selecting a member of a class means offsetting into that class. You can't produce an actual address until you combine that offset with the starting address of a particular object. The syntax of pointers to members requires that you select an object at the same time you're dereferencing the pointer to member.

To understand this syntax, consider a simple structure:

```
struct simple { int a; };
```

If you have a pointer **sp** and an object **so** for this structure, you can select members by saying

```
sp->a;
so.a;
```

Now suppose you have an ordinary pointer to an integer, **ip**. To access what **ip** is pointing to, you dereference the pointer with a *****:

```
*ip = 4;
```

Finally, consider what happens if you have a pointer that happens to point to something inside a class object, even if it does in fact represent an offset into the object. To access what it's pointing at, you must dereference it with *****. But it's an offset into an object, so you must also refer to that particular object. Thus, the ***** is combined with the object dereferencing. As an example using the **simple** class,

```
sp->*pm = 47;
so.*pm = 47;
```

So the new syntax becomes **–>*** for a pointer to an object, and **.*** for the object or a reference. Now, what is the syntax for defining **pm**? Like any pointer, you have to say what type it's pointing at, and you use a ***** in the definition. The only difference is you must say what class of objects this pointer-to-member is used with. Of course, this is accomplished with the name of the class and the scope resolution operator. Thus,

```
int simple::*pm;
```

You can also initialize the pointer-to-member when you define it (or any other time):

```
int simple::*pm = &simple::a;
```

There is actually no "address" of **simple::a** because you're just referring to the class and not an object of that class. Thus, **&simple::a** can be used only as pointer-to-member syntax.

Functions

A similar exercise produces the pointer-to-member syntax for member functions. A pointer to a function is defined like this:

```
int (*fp)(float);
```

The parentheses around **(*fp)** are necessary to force the compiler to evaluate the definition properly. Without them this would appear to be a function that returns an **int***.

To define and use a pointer to a member function, parentheses play a similarly important role. If you have a function inside a structure:

```
struct simple2 { int f(float); };
```

you define a pointer to that member function by inserting the class name and scope resolution operator into an ordinary function pointer definition:

```
int (simple2::*fp)(float);
```

You can also initialize it when you create it, or at any other time:

```
int (simple2::*fp)(float) = &simple2::f;
```

As with normal functions, the **&** is optional; you can give the function identifier without an argument list to mean the address:

```
fp = simple2::f;
```

An example

The value of a pointer is that you can change what it points to at runtime, which provides an important flexibility in your programming because through a pointer you can select or change *behavior* at runtime. A pointer-to-member is no different; it allows you to choose a member at runtime. Typically, your classes will have only member functions publicly visible (data members are usually considered part of the underlying implementation), so the following example selects member functions at runtime.

```
//: C11:Pmem.cpp
// Pointers to members

class Widget {
public:
  void f(int);
  void g(int);
  void h(int);
  void i(int);
};

void Widget::h(int) {}

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

Of course, it isn't particularly reasonable to expect the casual user to create such complicated expressions. If the user must directly manipulate a pointer-to-member, then a **typedef** is in order. To really clean things up, you can use the pointer-to-member as part of the internal implementation mechanism. Here's the preceding example using a pointer-to-member *inside* the class. All the user needs to do is pass a number in to select a function.[37]

```
//: C11:Pmem2.cpp
// Pointers to members
#include <iostream>
using namespace std;

class Widget {
  void f(int) const {cout << "Widget::f()\n";}
  void g(int) const {cout << "Widget::g()\n";}
  void h(int) const {cout << "Widget::h()\n";}
  void i(int) const {cout << "Widget::i()\n";}
  static const int _count = 4;
  void (Widget::*fptr[_count])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= _count) return;
    (this->*fptr[i])(j);
  }
  int count() { return _count; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
} ///:~
```

[37] Thanks to Owen Mortensen for this example

In the class interface and in **main( )**, you can see that the entire implementation, including the functions themselves, has been hidden away. The code must even ask for the **count( )** of functions. This way, the class implementor can change the quantity of functions in the underlying implementation without affecting the code where the class is used.

The initialization of the pointers-to-members in the constructor may seem overspecified. Shouldn't you be able to say

```
fptr[1] = &g;
```

because the name **g** occurs in the member function, which is automatically in the scope of the class? The problem is this doesn't conform to the pointer-to-member syntax, which is required so everyone, especially the compiler, can figure out what's going on. Similarly, when the pointer-to-member is dereferenced, it seems like

```
(this->*fptr[i])(j);
```

is also over-specified; **this** looks redundant. Again, the syntax requires that a pointer-to-member always be bound to an object when it is dereferenced.

Summary

Pointers in C++ are remarkably similar to pointers in C, which is good. Otherwise a lot of C code wouldn't compile properly under C++. The only compiler errors you will produce is where dangerous assignments occur. If these are in fact what are intended, the compiler errors can be removed with a simple (and explicit!) cast.

C++ also adds the *reference* from Algol and Pascal, which is like a constant pointer that is automatically dereferenced by the compiler. A reference holds an address, but you treat it like an object. References are essential for clean syntax with operator overloading (the subject of the next chapter), but they also add syntactic convenience for passing and returning objects for ordinary functions.

The copy-constructor takes a reference to an existing object of the same type as its argument, and it is used to create a new object from an existing one. The compiler automatically calls the copy-constructor when you pass or return an object by value. Although the compiler will automatically create a copy-constructor for you, if you think one will be needed for your class you should always define it yourself to ensure that

the proper behavior occurs. If you don't want the object passed or returned by value, you should create a private copy-constructor.

Pointers-to-members have the same functionality as ordinary pointers: You can choose a particular region of storage (data or function) at runtime. Pointers-to-members just happen to work with class members rather than global data or functions. You get the programming flexibility that allows you to change behavior at runtime.

Exercises

1. Create a function that takes a **char&** argument and modifies that argument. In **main( )**, print out a **char** variable, call your function for that variable, and print it out again to prove to yourself it has been changed. How does this affect program readability?
2. Write a class with a copy-constructor that announces itself to **cout**. Now create a function that passes an object of your new class in by value and another one that creates a local object of your new class and returns it by value. Call these functions to prove to yourself that the copy-constructor is indeed quietly called when passing and returning objects by value.
3. Discover how to get your compiler to generate assembly language, and produce assembly for **PassStruct.cpp**. Trace through and demystify the way your compiler generates code to pass and return large structures.
4. (Advanced) This exercise creates an alternative to using the copy-constructor. Create a class **X** and declare (but don't define) a **private** copy-constructor. Make a public **clone( )** function as a **const** member function that returns a copy of the object created using **new** (a forward reference to Chapter XX). Now create a function that takes as an argument a **const X&** and clones a local copy that can be modified. The drawback to this approach is that you are responsible for explicitly destroying the cloned object (using **delete**) when you're done with it.

12: Operator overloading

Operator overloading is just “syntactic sugar,” which means it is simply another way for you to make a function call.

The difference is the arguments for this function don’t appear inside parentheses, but instead surrounding or next to characters you’ve always thought of as immutable operators.

There are two differences between the use of an operator and an ordinary function call. The syntax is different; an operator is often “called” by placing it between or sometimes after the arguments. The second difference is that the compiler determines what “function” to call. For instance, if you are using the operator **+** with floating-point arguments, the compiler “calls” the function to perform floating-point addition (this “call” is typically the act of inserting in-line code, or a floating-point coprocessor instruction). If you use operator **+** with a floating-point number and an integer, the compiler “calls” a special function to turn the **int** into a **float**, and then “calls” the floating-point addition code.

But in C++, it’s possible to define new operators that work with classes. This definition is just like an ordinary function definition except the name of the function begins with the keyword **operator** and ends with the operator itself. That’s the only difference, and it becomes a function like any other function, which the compiler calls when it sees the appropriate pattern.

Warning & reassurance

It’s very tempting to become overenthusiastic with operator overloading. It’s a fun toy, at first. But remember it’s *only* syntactic sugar, another way

of calling a function. Looking at it this way, you have no reason to overload an operator except that it will make the code involving your class easier to write and especially *read*. (Remember, code is read much more than it is written.) If this isn't the case, don't bother.

Another common response to operator overloading is panic: Suddenly, C operators have no familiar meaning anymore. "Everything's changed and all my C code will do different things !" This isn't true. All the operators used in expressions that contain only built-in data types cannot be changed. You can never overload operators such that

```
1 << 4;
```

behaves differently, or

```
1.414 << 2;
```

has meaning. Only an expression containing a user-defined type can have an overloaded operator.

Syntax

Defining an overloaded operator is like defining a function, but the name of that function is **operator@**, where @ represents the operator. The number of arguments in the function argument list depends on two factors:

1. Whether it's a unary (one argument) or binary (two argument) operator.
2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

Here's a small class that shows the syntax for operator overloading:

```
//: C12:Opover.cpp
// Operator overloading syntax
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) { i = ii; }
```

```
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv){
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer I(1), J(2), K(3);
  K += I + J;
} ///:~
```

The two overloaded operators are defined as inline member functions that announce when they are called. The single argument is what appears on the right-hand side of the operator for binary operators. Unary operators have no arguments when defined as member functions. The member function is called for the object on the left-hand side of the operator.

For nonconditional operators (conditionals usually return a Boolean value) you'll almost always want to return an object or reference of the same type you're operating on if the two arguments are the same type. If they're not, the interpretation of what it should produce is up to you. This way complex expressions can be built up:

```
K += I + J;
```

The **operator+** produces a new **Integer** (a temporary) that is used as the **rv** argument for the **operator+=**. This temporary is destroyed as soon as it is no longer needed.

Overloadable operators

Although you can overload almost all the operators available in C, the use is fairly restrictive. In particular, you cannot combine operators that currently have no meaning in C (such as ****** to represent exponentiation), you cannot change the evaluation precedence of operators, and you cannot change the number of arguments an operator takes. This makes sense – all these actions would produce operators that confuse meaning rather than clarify it.

The next two subsections give examples of all the "regular" operators, overloaded in the form that you'll most likely use.

Unary operators

The following example shows the syntax to overload all the unary operators, both in the form of global functions and member functions. These will expand upon the **Integer** class shown previously and add a new **byte** class. The meaning of your particular operators will depend on the way you want to use them, but consider the client programmer before doing something unexpected.

```
//: C12:Unary.cpp
// Overloading unary operators
#include <iostream>
using namespace std;

class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  // No side effects takes const& argument:
  friend const Integer&
    operator+(const Integer& a);
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
```

```
  operator!(const Integer& a);
 // Side effects don't take const& argument:
 // Prefix:
 friend const Integer&
  operator++(Integer& a);
 // Postfix:
 friend const Integer
  operator++(Integer& a, int);
 // Prefix:
 friend const Integer&
  operator--(Integer& a);
 // Postfix:
 friend const Integer
  operator--(Integer& a, int);
};

// Global operators:
const Integer& operator+(const Integer& a) {
 cout << "+Integer\n";
 return a; // Unary + has no effect
}
const Integer operator-(const Integer& a) {
 cout << "-Integer\n";
 return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
 cout << "~Integer\n";
 return Integer(~a.i);
}
Integer* operator&(Integer& a) {
 cout << "&Integer\n";
 return a.This(); // &a is recursive!
}
int operator!(const Integer& a) {
 cout << "!Integer\n";
 return !a.i;
}
// Prefix; return incremented value
const Integer& operator++(Integer& a) {
 cout << "++Integer\n";
 a.i++;
 return a;
```

```
}
// Postfix; return the value before increment:
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer r(a.i);
  a.i++;
  return r;
}
// Prefix; return decremented value
const Integer& operator--(Integer& a) {
  cout << "--Integer\n";
  a.i--;
  return a;
}
// Postfix; return the value before decrement:
const Integer operator--(Integer& a, int) {
  cout << "Integer--\n";
  Integer r(a.i);
  a.i--;
  return r;
}

void f(Integer a) {
  +a;
  -a;
  ~a;
  Integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}

// Member operators (implicit "this"):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte& operator+() const {
    cout << "+Byte\n";
```

```
    return *this;
  }
  const Byte operator-() const {
    cout << "-Byte\n";
    return Byte(-b);
  }
  const Byte operator~() const {
    cout << "~Byte\n";
    return Byte(~b);
  }
  Byte operator!() const {
    cout << "!Byte\n";
    return Byte(!b);
  }
  Byte* operator&() {
    cout << "&Byte\n";
    return this;
  }
  // Side effects: non-const member function:
  const Byte& operator++() { // Prefix
    cout << "++Byte\n";
    b++;
    return *this;
  }
  const Byte operator++(int) { // Postfix
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
  }
  const Byte& operator--() { // Prefix
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Postfix
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};
```

```
void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

The functions are grouped according to the way their arguments are passed. Guidelines for how to pass and return arguments are given later. The above forms (and the ones that follow in the next section) are typically what you'll use, so start with them as a pattern when overloading your own operators.

Increment & decrement

The overloaded **++** and **– –** operators present a dilemma because you want to be able to call different functions depending on whether they appear before (prefix) or after (postfix) the object they're acting upon. The solution is simple, but some people find it a bit confusing at first. When the compiler sees, for example, **++a** (a preincrement), it generates a call to **operator++(a)**; but when it sees **a++,** it generates a call to **operator++(a, int)**. That is, the compiler differentiates between the two forms by making different function calls. In **Unary.cpp** for the member function versions, if the compiler sees **++b**, it generates a call to **B::operator++( )**; and if it sees **b++** it calls **B::operator++(int)**.

The user never sees the result of her action except that a different function gets called for the prefix and postfix versions. Underneath, however, the two functions calls have different signatures, so they link to two different function bodies. The compiler passes a dummy constant value for the **int** argument (which is never given an identifier because the

value is never used) to generate the different signature for the postfix version.

Binary operators

The following listing repeats the example of **Unary.cpp** for binary operators. Both global versions and member function versions are shown.

```
//: C12:Binary.cpp
// Overloading binary operators
#include "../require.h"
#include <fstream>
using namespace std;

ofstream out("binary.out");

class Integer { // Combine this with Unary.cpp
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operators that create new, modified value:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
  friend const Integer
    operator-(const Integer& left,
              const Integer& right);
  friend const Integer
    operator*(const Integer& left,
              const Integer& right);
  friend const Integer
    operator/(const Integer& left,
              const Integer& right);
  friend const Integer
    operator%(const Integer& left,
              const Integer& right);
  friend const Integer
    operator^(const Integer& left,
              const Integer& right);
  friend const Integer
    operator&(const Integer& left,
              const Integer& right);
  friend const Integer
```

```
  operator|(const Integer& left,
            const Integer& right);
friend const Integer
  operator<<(const Integer& left,
             const Integer& right);
friend const Integer
  operator>>(const Integer& left,
             const Integer& right);
// Assignments modify & return lvalue:
friend Integer&
  operator+=(Integer& left,
             const Integer& right);
friend Integer&
  operator-=(Integer& left,
             const Integer& right);
friend Integer&
  operator*=(Integer& left,
             const Integer& right);
friend Integer&
  operator/=(Integer& left,
             const Integer& right);
friend Integer&
  operator%=(Integer& left,
             const Integer& right);
friend Integer&
  operator^=(Integer& left,
             const Integer& right);
friend Integer&
  operator&=(Integer& left,
             const Integer& right);
friend Integer&
  operator|=(Integer& left,
             const Integer& right);
friend Integer&
  operator>>=(Integer& left,
              const Integer& right);
friend Integer&
  operator<<=(Integer& left,
              const Integer& right);
// Conditional operators return true/false:
friend int
  operator==(const Integer& left,
```

```
            const Integer& right);
  friend int
    operator!=(const Integer& left,
            const Integer& right);
  friend int
    operator<(const Integer& left,
            const Integer& right);
  friend int
    operator>(const Integer& left,
            const Integer& right);
  friend int
    operator<=(const Integer& left,
            const Integer& right);
  friend int
    operator>=(const Integer& left,
            const Integer& right);
  friend int
    operator&&(const Integer& left,
            const Integer& right);
  friend int
    operator||(const Integer& left,
            const Integer& right);
  // Write the contents to an ostream:
  void print(ostream& os) const { os << i; }
};

const Integer
  operator+(const Integer& left,
          const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
          const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
          const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
```

```
  operator/(const Integer& left,
          const Integer& right) {
  require(right.i != 0, "divide by zero");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
          const Integer& right) {
  require(right.i != 0, "modulo by zero");
  return Integer(left.i % right.i);
}
const Integer
  operator^(const Integer& left,
          const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
          const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
          const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
           const Integer& right) {
  return Integer(left.i << right.i);
}
const Integer
  operator>>(const Integer& left,
           const Integer& right) {
  return Integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
Integer& operator+=(Integer& left,
                const Integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i += right.i;
   return left;
```

```
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i -= right.i;
  return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i *= right.i;
  return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
  require(right.i != 0, "divide by zero");
  if(&left == &right) {/* self-assignment */}
  left.i /= right.i;
  return left;
}
Integer& operator%=(Integer& left,
                    const Integer& right) {
  require(right.i != 0, "modulo by zero");
  if(&left == &right) {/* self-assignment */}
  left.i %= right.i;
  return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i ^= right.i;
  return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i &= right.i;
  return left;
}
Integer& operator|=(Integer& left,
                    const Integer& right) {
  if(&left == &right) {/* self-assignment */}
```

```
  left.i |= right.i;
  return left;
}
Integer& operator>>=(Integer& left,
                     const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i >>= right.i;
  return left;
}
Integer& operator<<=(Integer& left,
                     const Integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i <<= right.i;
  return left;
}
// Conditional operators return true/false:
int operator==(const Integer& left,
               const Integer& right) {
   return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
   return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
   return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
   return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
   return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
   return left.i >= right.i;
}
int operator&&(const Integer& left,
               const Integer& right) {
```

```
  return left.i && right.i;
}
int operator||(const Integer& left,
               const Integer& right) {
  return left.i || right.i;
}

void h(Integer& c1, Integer& c2) {
  // A complex expression:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(OP) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #OP " c2 produces "; \
  (c1 OP c2).print(out); \
  out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionals:
  #define TRYC(OP) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #OP " c2 produces "; \
  out << (c1 OP c2); \
  out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

// Member operators (implicit "this"):
class Byte { // Combine this with Unary.cpp
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // No side effects: const member function:
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
```

```
const Byte
  operator-(const Byte& right) const {
  return Byte(b - right.b);
}
const Byte
  operator*(const Byte& right) const {
  return Byte(b * right.b);
}
const Byte
  operator/(const Byte& right) const {
  require(right.b != 0, "divide by zero");
  return Byte(b / right.b);
}
const Byte
  operator%(const Byte& right) const {
  require(right.b != 0, "modulo by zero");
  return Byte(b % right.b);
}
const Byte
  operator^(const Byte& right) const {
  return Byte(b ^ right.b);
}
const Byte
  operator&(const Byte& right) const {
  return Byte(b & right.b);
}
const Byte
  operator|(const Byte& right) const {
  return Byte(b | right.b);
}
const Byte
  operator<<(const Byte& right) const {
  return Byte(b << right.b);
}
const Byte
  operator>>(const Byte& right) const {
  return Byte(b >> right.b);
}
// Assignments modify & return lvalue.
// operator= can only be a member function:
Byte& operator=(const Byte& right) {
  // Handle self-assignment:
```

```
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
Byte& operator+=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b += right.b;
  return *this;
}
Byte& operator-=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b -= right.b;
  return *this;
}
Byte& operator*=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b *= right.b;
  return *this;
}
Byte& operator/=(const Byte& right) {
  require(right.b != 0, "divide by zero");
  if(this == &right) {/* self-assignment */}
  b /= right.b;
  return *this;
}
Byte& operator%=(const Byte& right) {
  require(right.b != 0, "modulo by zero");
  if(this == &right) {/* self-assignment */}
  b %= right.b;
  return *this;
}
Byte& operator^=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b ^= right.b;
  return *this;
}
Byte& operator&=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b &= right.b;
  return *this;
}
Byte& operator|=(const Byte& right) {
```

```
  if(this == &right) {/* self-assignment */}
  b |= right.b;
  return *this;
}
Byte& operator>>=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b >>= right.b;
  return *this;
}
Byte& operator<<=(const Byte& right) {
  if(this == &right) {/* self-assignment */}
  b <<= right.b;
  return *this;
}
// Conditional operators return true/false:
int operator==(const Byte& right) const {
    return b == right.b;
}
int operator!=(const Byte& right) const {
    return b != right.b;
}
int operator<(const Byte& right) const {
    return b < right.b;
}
int operator>(const Byte& right) const {
    return b > right.b;
}
int operator<=(const Byte& right) const {
    return b <= right.b;
}
int operator>=(const Byte& right) const {
    return b >= right.b;
}
int operator&&(const Byte& right) const {
    return b && right.b;
}
int operator||(const Byte& right) const {
    return b || right.b;
}
// Write the contents to an ostream:
void print(ostream& os) const {
  os << "0x" << hex << int(b) << dec;
```

```
  }
};

void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #OP " b2 produces "; \
  (b1 OP b2).print(out); \
  out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(OP) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #OP " b2 produces "; \
  out << (b1 OP b2); \
  out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  Byte b3 = 92;
  b1 = b2 = b3;
}

int main() {
  Integer c1(47), c2(9);
  h(c1, c2);
  out << "\n member functions:" << endl;
```

```
  Byte b1(47), b2(9);
  k(b1, b2);
} ///:~
```

You can see that **operator=** is only allowed to be a member function. This is explained later.

Notice that all the assignment operators have code to check for self-assignment, as a general guideline. In some cases this is not necessary; for example, with **operator+=** you may *want* to say **A+=A** and have it add **A** to itself. The most important place to check for self-assignment is **operator=** because with complicated objects disastrous results may occur. (In some cases it's OK, but you should always keep it in mind when writing **operator=**.)

All of the operators shown in the previous two examples are overloaded to handle a single type. It's also possible to overload operators to handle mixed types, so you can add apples to oranges, for example. Before you start on an exhaustive overloading of operators, however, you should look at the section on automatic type conversion later in this chapter. Often, a type conversion in the right place can save you a lot of overloaded operators.

Arguments & return values

It may seem a little confusing at first when you look at **Unary.cpp** and **Binary.cpp** and see all the different ways that arguments are passed and returned. Although you *can* pass and return arguments any way you want to, the choices in these examples were not selected at random. They follow a very logical pattern, the same one you'll want to use in most of your choices.

1. As with any function argument, if you only need to read from the argument and not change it, default to passing it as a **const** reference. Ordinary arithmetic operations (like **+** and **–**, etc.) and Booleans will not change their arguments, so pass by **const** reference is predominantly what you'll use. When the function is a class member, this translates to making it a **const** member function. Only with the operator-assignments (like **+=**) and the **operator=**, which change the left-hand argument, is the left argument *not* a constant, but it's still passed in as an address because it will be changed.
2. The type of return value you should select depends on the expected meaning of the operator. (Again, you can do anything you want with the arguments and return values.) If the effect of the operator is to

produce a new value, you will need to generate a new object as the return value. For example, **Integer::operator+** must produce an **Integer** object that is the sum of the operands. This object is returned by value as a **const**, so the result cannot be modified as an lvalue.

3. All the assignment operators modify the lvalue. To allow the result of the assignment to be used in chained expressions, like **A=B=C**, it's expected that you will return a reference to that same lvalue that was just modified. But should this reference be a **const** or non**const**? Although you read **A=B=C** from left to right, the compiler parses it from right to left, so you're not forced to return a non**const** to support assignment chaining. However, people do sometimes expect to be able to perform an operation on the thing that was just assigned to, such as **(A=B).func( );** to call **func( )** on **A** after assigning **B** to it. Thus the return value for all the assignment operators should be a non**const** reference to the lvalue.
4. For the logical operators, everyone expects to get at worst an **int** back, and at best a **bool**. (Libraries developed before most compilers supported C++'s built-in **bool** will use **int** or an equivalent **typedef**).
5. The increment and decrement operators present a dilemma because of the pre- and postfix versions. Both versions change the object and so cannot treat the object as a **const**. The prefix version returns the value of the object after it was changed, so you expect to get back the object that was changed. Thus, with prefix you can just return ***this** as a reference. The postfix version is supposed to return the value *before* the value is changed, so you're forced to create a separate object to represent that value and return it. Thus, with postfix you must return by value if you want to preserve the expected meaning. (Note that you'll often find the increment and decrement operators returning an **int** or **bool** to indicate, for example, whether an iterator is at the end of a list). Now the question is: Should these be returned as **const** or non**const**? If you allow the object to be modified and someone writes **(++A).func( );**, **func( )** will be operating on **A** itself, but with **(A++).func( );**, **func( )** operates on the temporary object returned by the postfix **operator++**. Temporary objects are automatically **const**, so this would be flagged by the compiler, but for consistency's sake it may make more sense to make them both **const**, as was done here. Because of the variety of meanings you may want to give the increment and decrement operators, they will need to be considered on a case-by-case basis.

Return by value as **const**

Returning by value as a **const** can seem a bit subtle at first, and so deserves a bit more explanation. Consider the binary **operator+**. If you use it in an expression such as **f(A+B)**, the result of **A+B** becomes a temporary object that is used in the call to **f( )**. Because it's a temporary, it's automatically **const**, so whether you explicitly make the return value **const** or not has no effect.

However, it's also possible for you to send a message to the return value of **A+B**, rather than just passing it to a function. For example, you can say **(A+B).g( )**, where **g( )** is some member function of **Integer**, in this case. By making the return value **const**, you state that only a **const** member function can be called for that return value. This is **const**-correct, because it prevents you from storing potentially valuable information in an object that will most likely be lost.

return efficiency

When new objects are created to return by value, notice the form used. In **operator+**, for example:

```
return Integer(left.i + right.i);
```

This may look at first like a "function call to a constructor," but it's not. The syntax is that of a temporary object; the statement says "make a temporary **Integer** object and return it." Because of this, you might think that the result is the same as creating a named local object and returning that. However, it's quite different. If you were to say instead:

```
Integer tmp(left.i + right.i);
return tmp;
```

three things will happen. First, the **tmp** object is created including its constructor call. Then, the copy-constructor copies the **tmp** to the location of the outside return value. Finally, the destructor is called for **tmp** at the end of the scope.

In contrast, the "returning a temporary" approach works quite differently. When the compiler sees you do this, it knows that you have no other need for the object it's creating than to return it so it builds the object *directly* into the location of the outside return value. This requires only a single ordinary constructor call (no copy-constructor is necessary) and there's no destructor call because you never actually create a local object. Thus, while it doesn't cost anything but programmer awareness, it's significantly more efficient.

Unusual operators

Several additional operators have a slightly different syntax for overloading.

The subscript, **operator[ ]**, must be a member function and it requires a single argument. Because it implies that the object acts like an array, you will often return a reference from this operator, so it can be used conveniently on the left-hand side of an equal sign. This operator is commonly overloaded; you'll see examples in the rest of the book.

The comma operator is called when it appears next to an object of the type the comma is defined for. However, **operator,** is *not* called for function argument lists, only for objects that are out in the open, separated by commas. There doesn't seem to be a lot of practical uses for this operator; it's in the language for consistency. Here's an example showing how the comma function can be called when the comma appears *before* an object, as well as after:

```
//: C12:Comma.cpp
// Overloading the ',' operator
#include <iostream>
using namespace std;

class After {
public:
  const After& operator,(const After&) const {
    cout << "After::operator,()" << endl;
    return *this;
  }
};

class Before {};

Before& operator,(int, Before& b) {
  cout << "Before::operator,()" << endl;
  return b;
}

int main() {
  After a, b;
  a, b;  // Operator comma called

  Before c;
```

```
  1, c;  // Operator comma called
} ///:~
```

The global function allows the comma to be placed before the object in question. The usage shown is fairly obscure and questionable. Although you would probably use a comma-separated list as part of a more complex expression, it's too subtle to use in most situations.

The *function call* **operator( )** must be a member function, and it is unique in that it allows any number of arguments. It makes your object look like it's actually a function name, so it's probably best used for types that only have a single operation, or at least an especially prominent one.

The operators **new** and **delete** control dynamic storage allocation, and can be overloaded. This very important topic is covered in the next chapter.

The **operator–>*** is a binary operator that behaves like all the other binary operators. It is provided for those situations when you want to mimic the behavior provided by the built-in *pointer-to-member* syntax, described in the previous chapter.

The *smart pointer* **operator–>** is designed to be used when you want to make an object appear to be a pointer. This is especially useful if you want to "wrap" a class around a pointer to make that pointer safe, or in the common usage of an *iterator*, which is an object that moves through a *collection* or *container* of other objects and selects them one at a time, without providing direct access to the implementation of the container. (You'll often find containers and iterators in class libraries.)

A smart pointer must be a member function. It has additional, atypical constraints: It must return either an object (or reference to an object) that also has a smart pointer or a pointer that can be used to select what the smart pointer arrow is pointing at. Here's a simple example:

```
//: C12:Smartp.cpp
// Smart pointer example
#include <iostream>
#include <cstring>
using namespace std;

class Obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
```

```
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  static const int sz = 100;
  Obj* a[sz];
  int index;
public:
  ObjContainer() {
    index = 0;
    memset(a, 0, sz * sizeof(Obj*));
  }
  void add(Obj* obj) {
    if(index >= sz) return;
    a[index++] = obj;
  }
  friend class Sp;
};

// Iterator:
class Sp {
  ObjContainer* oc;
  int index;
public:
  Sp(ObjContainer* objc) {
    index = 0;
    oc = objc;
  }
  // Return value indicates end of list:
  int operator++() { // Prefix
    if(index >= oc->sz) return 0;
    if(oc->a[++index] == 0) return 0;
    return 1;
  }
  int operator++(int) { // Postfix
    return operator++(); // Use prefix version
  }
  Obj* operator->() const {
```

```
    if(oc->a[index]) return oc->a[index];
    static Obj dummy;
    return &dummy;
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Fill it up
  Sp sp(&oc); // Create an iterator
  do {
    sp->f(); // Smart pointer calls
    sp->g();
  } while(sp++);
} ///:~
```

The class **Obj** defines the objects that are manipulated in this program. The functions **f( )** and **g( )** simply print out interesting values using **static** data members. Pointers to these objects are stored inside containers of type **ObjContainer** using its **add( )** function. **ObjContainer** looks like an array of pointers, but you'll notice there's no way to get the pointers back out again. However, **Sp** is declared as a **friend** class, so it has permission to look inside the container. The **Sp** class looks very much like an intelligent pointer – you can move it forward using **operator++** (you can also define an **operator– –**), it won't go past the end of the container it's pointing to, and it returns (via the smart pointer operator) the value it's pointing to. Notice that an iterator is a custom fit for the container it's created for – unlike a pointer, there isn't a "general purpose" iterator. Containers and iterators are covered in more depth in Chapter XX.

In **main( )**, once the container **oc** is filled with **Obj** objects, an iterator **SP** is created. The smart pointer calls happen in the expressions:

```
sp->f(); // Smart pointer calls
sp->g();
```

Here, even though **sp** doesn't actually have **f( )** and **g( )** member functions, the smart pointer mechanism calls those functions for the **Obj*** that is returned by **Sp::operator–>**. The compiler performs all the checking to make sure the function call works properly.

Although the underlying mechanics of the smart pointer are more complex than the other operators, the goal is exactly the same – to provide a more convenient syntax for the users of your classes.

Operators you can't overload

There are certain operators in the available set that cannot be overloaded. The general reason for the restriction is safety: If these operators were overloadable, it would somehow jeopardize or break safety mechanisms. Often it makes things harder, or confuses existing practice.

The member selection **operator.**. Currently, the dot has a meaning for any member in a class, but if you allow it to be overloaded, then you couldn't access members in the normal way; instead you'd have to use a pointer and the arrow operator **–>**.

The pointer to member dereference **operator.*** . For the same reason as **operator.**.

There's no exponentiation operator. The most popular choice for this was **operator**** from Fortran, but this raised difficult parsing questions. Also, C has no exponentiation operator, so C++ didn't seem to need one either because you can always perform a function call. An exponentiation operator would add a convenient notation, but no new language functionality, to account for the added complexity of the compiler.

There are no user-defined operators. That is, you can't make up new operators that aren't currently in the set. Part of the problem is how to determine precedence, and part of the problem is an insufficient need to account for the necessary trouble.

You can't change the precedence rules. They're hard enough to remember as it is, without letting people play with them.

Nonmember operators

In some of the previous examples, the operators may be members or nonmembers, and it doesn't seem to make much difference. This usually raises the question, "Which should I choose?" In general, if it doesn't make any difference, they should be members, to emphasize the association between the operator and its class. When the left-hand operand is an object of the current class, it works fine.

This isn't always the case – sometimes you want the left-hand operand to be an object of some other class. A very common place to see this is when the operators **<<** and **>>** are overloaded for iostreams:

```
//: C12:Iosop.cpp
// Iostream operator overloading
// Example of non-member overloaded operators
#include "../require.h"
#include <iostream>
#include <strstream>
#include <cstring>
using namespace std;

class IntArray {
  static const int sz = 5;
  int i[sz];
public:
  IntArray() {
    memset(i, 0, sz* sizeof(*i));
  }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
          "operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os,
             const IntArray& ia);
  friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream& operator<<(ostream& os,
                const IntArray& ia){
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}
```

```
istream& operator>>(istream& is, IntArray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

int main() {
  istrstream input("47 34 56 92 103");
  IntArray I;
  input >> I;
  I[4] = -1; // Use overloaded operator[]
  cout << I;
} ///:~
```

This class also contains an overloaded **operator[ ]**, which returns a reference to a legitimate value in the array. A reference is returned, so the expression

```
I[4] = -1;
```

not only looks much more civilized than if pointers were used, it also accomplishes the desired effect.

The overloaded shift operators pass and return by reference, so the actions will affect the external objects. In the function definitions, expressions like

```
os << ia.i[j];
```

cause *existing* overloaded operator functions to be called (that is, those defined in **<iostream>**). In this case, the function called is **ostream& operator<<(ostream&, int)** because **ia.i[j]** resolves to an **int**.

Once all the actions are performed on the **istream** or **ostream**, it is returned so it can be used in a more complicated expression.

The form shown in this example for the inserter and extractor is standard. If you want to create a set for your own class, copy the function signatures and return types and follow the form of the body.

Basic guidelines

Murray[38] suggests these guidelines for choosing between members and nonmembers:

Operator	Recommended use
All unary operators	member
= ( ) [ ] ->	*must* be member
+= -= /= *= ^= &= |= %= >>= <<=	member
All other binary operators	nonmember

Overloading assignment

A common source of confusion with new C++ programmers is assignment. This is no doubt because the **=** sign is such a fundamental operation in programming, right down to copying a register at the machine level. In addition, the copy-constructor (from the previous chapter) can also be invoked when using the **=** sign:

```
MyType b;
MyType a = b;
a = b;
```

In the second line, the object **a** is being *defined*. A new object is being created where one didn't exist before. Because you know by now how defensive the C++ compiler is about object initialization, you know that a constructor must always be called at the point where an object is defined. But which constructor? **a** is being created from an existing **MyType** object, so there's only one choice: the copy-constructor. So even though an equal sign is involved, the copy-constructor is called.

In the third line, things are different. On the left side of the equal sign, there's a previously initialized object. Clearly, you don't call a constructor for an object that's already been created. In this case

[38] Rob Murray, *C++ Strategies & Tactics*, Addison-Wesley, 1993, page 47.

MyType::operator= is called for **a**, taking as an argument whatever appears on the right-hand side. (You can have multiple **operator=** functions to take different right-hand arguments.)

This behavior is not restricted to the copy-constructor. Any time you're initializing an object using an **=** instead of the ordinary function-call form of the constructor, the compiler will look for a constructor that accepts whatever is on the right-hand side:

```
//: C12:FeeFi.cpp
// Copying vs. initialization

class Fi {
public:
  Fi() {}
};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee f = 1; // Fee(int)
  Fi fi;
  Fee fum = fi; // Fee(Fi)
} ///:~
```

When dealing with the **=** sign, it's important to keep this distinction in mind: If the object hasn't been created yet, initialization is required; otherwise the assignment **operator=** is used.

It's even better to avoid writing code that uses the **=** for initialization; instead, always use the explicit constructor form; the last line becomes

```
  Fee fum(fi);
```

This way, you'll avoid confusing your readers.

Behavior of **operator=**

In **Binary.cpp**, you saw that **operator=** can be only a member function. It is intimately connected to the object on the left side of the **=**, and if you

could define **operator=** globally, you could try to redefine the built-in **=** sign:

```
int operator=(int, MyType); // Global = not allowed!
```

The compiler skirts this whole issue by forcing you to make **operator=** a member function.

When you create an **operator=**, you must copy all the necessary information from the right-hand object into yourself to perform whatever you consider "assignment" for your class. For simple objects, this is obvious:

```
//: C12:Simpcopy.cpp
// Simple operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0) {
    a = aa;
    b = bb;
    c = cc;
  }
  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
    operator<<(ostream& os, const Value& rv) {
      return os << "a = " << rv.a << ", b = "
        << rv.b << ", c = " << rv.c;
    }
};

int main() {
  Value A, B(1, 2, 3.3);
  cout << "A: " << A << endl;
  cout << "B: " << B << endl;
```

```
  A = B;
  cout << "A after assignment: " << A << endl;
} ///:~
```

Here, the object on the left side of the **=** copies all the elements of the object on the right, then returns a reference to itself, so a more complex expression can be created.

A common mistake was made in this example. When you're assigning two objects of the same type, you should always check first for self-assignment: Is the object being assigned to itself? In some cases, such as this one, it's harmless if you perform the assignment operations anyway, but if changes are made to the implementation of the class it, can make a difference, and if you don't do it as a matter of habit, you may forget and cause hard-to-find bugs.

Pointers in classes

What happens if the object is not so simple? For example, what if the object contains pointers to other objects? Simply copying a pointer means you'll end up with two objects pointing to the same storage location. In situations like these, you need to do bookkeeping of your own.

There are two common approaches to this problem. The simplest technique is to copy whatever the pointer refers to when you do an assignment or a copy-constructor. This is very straightforward:

```
//: C12:Copymem.cpp
// Duplicate during assignment
#include "../require.h"
#include <cstdlib>
#include <cstring>
using namespace std;

class WithPointer {
  char* p;
  static const int blocksz = 100;
public:
  WithPointer() {
    p = (char*)malloc(blocksz);
    require(p != 0);
    memset(p, 1, blocksz);
  }
  WithPointer(const WithPointer& wp) {
    p = (char*)malloc(blocksz);
```

```
    require(p != 0);
    memcpy(p, wp.p, blocksz);
  }
  WithPointer&
  operator=(const WithPointer& wp) {
    // Check for self-assignment:
    if(&wp != this)
      memcpy(p, wp.p, blocksz);
    return *this;
  }
  ~WithPointer() {
    free(p);
  }
};

int main() {
  WithPointer p;
  WithPointer p2 = p; // Copy construction
  p = p2; // Assignment
} ///:~
```

This shows the four functions you will always need to define when your class contains pointers: all necessary ordinary constructors, the copy-constructor, **operator=** (either define it or disallow it), and a destructor. The **operator=** checks for self-assignment as a matter of course, even though it's not strictly necessary here. This virtually eliminates the possibility that you'll forget to check for self-assignment if you *do* change the code so that it matters.

Here, the constructors allocate the memory and initialize it, the **operator=** copies it, and the destructor frees the memory. However, if you're dealing with a lot of memory or a high overhead to initialize that memory, you may want to avoid this copying. A very common approach to this problem is called *reference counting*. You make the block of memory smart, so it knows how many objects are pointing to it. Then copy-construction or assignment means attaching another pointer to an existing block of memory and incrementing the reference count. Destruction means reducing the reference count and destroying the object if the reference count goes to zero.

But what if you want to write to the block of memory? More than one object may be using this block, so you'd be modifying someone else's block as well as yours, which doesn't seem very neighborly. To solve this problem, an additional technique called *copy-on-write* is often used.

Before writing to a block of memory, you make sure no one else is using it. If the reference count is greater than one, you must make yourself a personal copy of that block before writing it, so you don't disturb someone else's turf. Here's a simple example of reference counting and copy-on-write:

```
//: C12:Refcount.cpp
// Reference count, copy-on-write
#include "../require.h"
#include <cstring>
using namespace std;

class Counted {
  class MemBlock {
    static const int size = 100;
    char c[size];
    int refcount;
  public:
    MemBlock() {
      memset(c, 1, size);
      refcount = 1;
    }
    MemBlock(const MemBlock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
    }
    void attach() { ++refcount; }
    void detach() {
      require(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this MemBlock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    MemBlock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new MemBlock(*this);
```

```
    }
  }* block;
public:
  Counted() {
    block = new MemBlock; // Sneak preview
  }
  Counted(const Counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
  }
  void unalias() { block = block->unalias(); }
  Counted& operator=(const Counted& rv) {
    // Check for self-assignment:
    if(&rv == this) return *this;
    // Clean up what you're using first:
    block->detach();
    block = rv.block; // Like copy-constructor
    block->attach();
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~Counted() { block->detach(); }
  // Copy-on-write:
  void write(char value) {
    // Do this before any write operation:
    unalias();
    // It's safe to write now.
    block->set(value);
  }
};

int main() {
  Counted A, B;
  Counted C(A);
  B = A;
  C = C;
  C.write('x');
} ///:~
```

The nested class **MemBlock** is the block of memory pointed to. (Notice the pointer **block** defined at the end of the nested class.) It contains a reference count and functions to control and read the reference count.

There's a copy-constructor so you can make a new **MemBlock** from an existing one.

The **attach( )** function increments the reference count of a **MemBlock** to indicate there's another object using it. **detach( )** decrements the reference count. If the reference count goes to zero, then no one is using it anymore, so the member function destroys its own object by saying **delete this**.

You can modify the memory with the **set( )** function, but before you make any modifications, you should ensure that you aren't walking on a **MemBlock** that some other object is using. You do this by calling **Counted::unalias( )**, which in turn calls **MemBlock::unalias( )**. The latter function will return the **block** pointer if the reference count is one (meaning no one else is pointing to that block), but will duplicate the block if the reference count is more than one.

This example includes a sneak preview of the next chapter. Instead of C's **malloc( )** and **free( )** to create and destroy the objects, the special C++ operators **new** and **delete** are used. For this example, you can think of **new** and **delete** just like **malloc( )** and **free( )**, except **new** calls the constructor after allocating memory, and **delete** calls the destructor before freeing the memory.

The copy-constructor, instead of creating its own memory, assigns **block** to the **block** of the source object. Then, because there's now an additional object using that block of memory, it increments the reference count by calling **MemBlock::attach( )**.

The **operator=** deals with an object that has already been created on the left side of the **=**, so it must first clean that up by calling **detach( )** for that **MemBlock**, which will destroy the old **MemBlock** if no one else is using it. Then **operator=** repeats the behavior of the copy-constructor. Notice that it first checks to detect whether you're assigning the same object to itself.

The destructor calls **detach( )** to conditionally destroy the **MemBlock**.

To implement copy-on-write, you must control all the actions that write to your block of memory. This means you can't ever hand a raw pointer to the outside world. Instead you say, "Tell me what you want done and I'll do it for you!" For example, the **write( )** member function allows you to change the values in the block of memory. But first, it uses **unalias( )** to prevent the modification of an aliased block (a block with more than one **Counted** object using it).

main( ) tests the various functions that must work correctly to implement reference counting: the constructor, copy-constructor, **operator=**, and destructor. It also tests the copy-on-write by calling the **write( )** function for object **C**, which is aliased to **A**'s memory block.

Tracing the output

To verify that the behavior of this scheme is correct, the best approach is to add information and functionality to the class to generate a trace output that can be analyzed. Here's **Refcount.cpp** with added trace information:

```
//: C12:RefcountTrace.cpp
// Refcount.cpp w/ trace info
#include "../require.h"
#include <cstring>
#include <fstream>
using namespace std;

ofstream out("rctrace.out");

class Counted {
  class MemBlock {
    static const int size = 100;
    char c[size];
    int refcount;
    static int blockcount;
    int blocknum;
  public:
    MemBlock() {
      memset(c, 1, size);
      refcount = 1;
      blocknum = blockcount++;
    }
    MemBlock(const MemBlock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
      blocknum = blockcount++;
      print("copied block");
      out << endl;
      rv.print("from block");
    }
    ~MemBlock() {
```

```
      out << "\tdestroying block "
          << blocknum << endl;
    }
    void print(const char* msg = "") const {
      if(*msg) out << msg << ", ";
      out << "blocknum:" << blocknum;
      out << ", refcount:" << refcount;
    }
    void attach() { ++refcount; }
    void detach() {
      require(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this MemBlock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    MemBlock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new MemBlock(*this);
    }
  }* block;
  static const int sz = 30;
  char ident[sz];
public:
  Counted(const char* id = "tmp") {
    block = new MemBlock; // Sneak preview
    strncpy(ident, id, sz);
  }
  Counted(const Counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
    strncpy(ident, rv.ident, sz);
    strncat(ident, " copy", sz - strlen(ident));
  }
  void unalias() { block = block->unalias(); }
  void addname(const char* nm) {
```

```
    strncat(ident, nm, sz - strlen(ident));
  }
  Counted& operator=(const Counted& rv) {
    print("inside operator=\n\t");
    if(&rv == this) {
      out << "self-assignment" << endl;
      return *this;
    }
    // Clean up what you're using first:
    block->detach();
    block = rv.block; // Like copy-constructor
    block->attach();
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~Counted() {
    out << "preparing to destroy: " << ident
        << "\n\tdecrementing refcount ";
    block->print();
    out << endl;
    block->detach();
  }
  // Copy-on-write:
  void write(char value) {
    unalias();
    block->set(value);
  }
  void print(const char* msg = "") {
    if(*msg) out << msg << " ";
    out << "object " << ident << ": ";
    block->print();
    out << endl;
  }
};

int Counted::MemBlock::blockcount = 0;

int main() {
  Counted A("A"), B("B");
  Counted C(A);
  C.addname(" (C) ");
  A.print();
```

```
  B.print();
  C.print();
  B = A;
  A.print("after assignment\n\t");
  B.print();
  out << "Assigning C = C" << endl;
  C = C;
  C.print("calling C.write('x')\n\t");
  C.write('x');
  out << "\n exiting main()" << endl;
} ///:~
```

Now **MemBlock** contains a **static** data member **blockcount** to keep track of the number of blocks created, and to create a unique number (stored in **blocknum**) for each block so you can tell them apart. The destructor announces which block is being destroyed, and the **print( )** function displays the block number and reference count.

The **Counted** class contains a buffer **ident** to keep track of information about the object. The **Counted** constructor creates a **new MemBlock** object and assigns the result (a pointer to the **MemBlock** object on the heap) to **block**. The identifier, copied from the argument, has the word "copy" appended to show where it's copied from. Also, the **addname( )** function lets you put additional information about the object in **ident** (the actual identifier, so you can see what it is as well as where it's copied from).

Here's the output:

```
object A: blocknum:0, refcount:2
object B: blocknum:1, refcount:1
object A copy (C) : blocknum:0, refcount:2
inside operator=
  object B: blocknum:1, refcount:1
 destroying block 1
after assignment
  object A: blocknum:0, refcount:3
object B: blocknum:0, refcount:3
Assigning C = C
inside operator=
  object A copy (C) : blocknum:0, refcount:3
self-assignment
calling C.write('x')
  object A copy (C) : blocknum:0, refcount:3
```

```
copied block, blocknum:2, refcount:1
from block, blocknum:0, refcount:2
exiting main()
preparing to destroy: A copy (C)
  decrementing refcount blocknum:2, refcount:1
  destroying block 2
preparing to destroy: B
  decrementing refcount blocknum:0, refcount:2
preparing to destroy: A
  decrementing refcount blocknum:0, refcount:1
  destroying block 0
```

By studying the output, tracing through the source code, and experimenting with the program, you'll deepen your understanding of these techniques.

Automatic **operator=** creation

Because assigning an object to another object *of the same type* is an activity most people expect to be possible, the compiler will automatically create a **type::operator=(type)** if you don't make one. The behavior of this operator mimics that of the automatically created copy-constructor: If the class contains objects (or is inherited from another class), the **operator=** for those objects is called recursively. This is called *memberwise assignment*. For example,

```
//: C12:Autoeq.cpp
// Automatic operator=()
#include <iostream>
using namespace std;

class Bar {
public:
  Bar& operator=(const Bar&) {
    cout << "inside Bar::operator=()" << endl;
    return *this;
  }
};

class MyType {
  Bar b;
};

int main() {
```

```
  MyType a, b;
  a = b; // Prints: "inside Bar::operator=()"
} ///:~
```

The automatically generated **operator=** for **MyType** calls **Bar::operator=**.

Generally you don't want to let the compiler do this for you. With classes of any sophistication (especially if they contain pointers!) you want to explicitly create an **operator=**. If you really don't want people to perform assignment, declare **operator=** as a **private** function. (You don't need to define it unless you're using it inside the class.)

Automatic type conversion

In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants. In C++, you can achieve this same effect for user-defined types by defining automatic type-conversion functions. These functions come in two flavors: a particular type of constructor and an overloaded operator.

Constructor conversion

If you define a constructor that takes as its single argument an object (or reference) of another type, that constructor allows the compiler to perform an automatic type conversion. For example,

```
//: C12:Autocnst.cpp
// Type conversion constructor

class One {
public:
  One() {}
};

class Two {
public:
  Two(const One&) {}
};
```

```
void f(Two) {}

int main() {
  One one;
  f(one); // Wants a Two, has a One
} ///:~
```

When the compiler sees **f( )** called with a **One** object, it looks at the declaration for **f( )** and notices it wants a **Two**. Then it looks to see if there's any way to get a **Two** from a **One**, and it finds the constructor **Two::Two(One)**, which it quietly calls. The resulting **Two** object is handed to **f( )**.

In this case, automatic type conversion has saved you from the trouble of defining two overloaded versions of **f( )**. However, the cost is the hidden constructor call to **Two**, which may matter if you're concerned about the efficiency of calls to **f( )**.

Preventing constructor conversion

There are times when automatic type conversion via the constructor can cause problems. To turn it off, you modify the constructor by prefacing with the keyword **explicit**[39] (which only works with constructors). Used to modify the constructor of class **Two** in the above example:

```
class One {
public:
  One() {}
};

class Two {
public:
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
```

[39] At the time of this writing, **explicit** was a new keyword in the language. Your compiler may not support it yet.

```
//!  f(one); // No auto conversion allowed
  f(Two(one)); // OK -- user performs conversion
}
```

By making **Two**'s constructor explicit, the compiler is told not to perform any automatic conversion using that particular constructor (other non-**explicit** constructors in that class can still perform automatic conversions). If the user wants to make the conversion happen, the code must be written out. In the above code, **f(Two(one))** creates a temporary object of type **Two** from **one**, just like the compiler did in the previous version.

Operator conversion

The second way to effect automatic type conversion is through operator overloading. You can create a member function that takes the current type and converts it to the desired type using the **operator** keyword followed by the type you want to convert to. This form of operator overloading is unique because you don't appear to specify a return type – the return type is the *name* of the operator you're overloading. Here's an example:

```
//: C12:Opconv.cpp
// Op overloading conversion

class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
```

```
  g(1);  // Calls Three(1,0)
} ///:~
```

With the constructor technique, the destination class is performing the conversion, but with operators, the source class performs the conversion. The value of the constructor technique is you can add a new conversion path to an existing system as you're creating a new class. However, creating a single-argument constructor *always* defines an automatic type conversion (even if it's got more than one argument, if the rest of the arguments are defaulted), which may not be what you want. In addition, there's no way to use a constructor conversion from a user-defined type to a built-in type; this is possible only with operator overloading.

Reflexivity

One of the most convenient reasons to use global overloaded operators rather than member operators is that in the global versions, automatic type conversion may be applied to either operand, whereas with member objects, the left-hand operand must already be the proper type. If you want both operands to be converted, the global versions can save a lot of coding. Here's a small example:

```
//: C12:Reflex.cpp
// Reflexivity in overloading

class Number {
  int i;
public:
  Number(int ii = 0) : i(ii) {}
  const Number
  operator+(const Number& n) const {
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
```

```
  Number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd arg converted to Number
//! 1 + a; // Wrong! 1st arg not of type Number
  a - b; // OK
  a - 1; // 2nd arg converted to Number
  1 - a; // 1st arg converted to Number
} ///:~
```

Class **Number** has a member **operator+** and a friend **operator–**. Because there's a constructor that takes a single **int** argument, an **int** can be automatically converted to a **Number**, but only under the right conditions. In **main( )**, you can see that adding a **Number** to another **Number** works fine because it's an exact match to the overloaded operator. Also, when the compiler sees a **Number** followed by a **+** and an **int**, it can match to the member function **Number::operator+** and convert the **int** argument to a **Number** using the constructor. But when it sees an **int** and a **+** and a **Number**, it doesn't know what to do because all it has is **Number::operator+**, which requires that the left operand already be a **Number** object. Thus the compiler issues an error.

With the **friend operator–**, things are different. The compiler needs to fill in both its arguments however it can; it isn't restricted to having a **Number** as the left-hand argument. Thus, if it sees **1 – a**, it can convert the first argument to a **Number** using the constructor.

Sometimes you want to be able to restrict the use of your operators by making them members. For example, when multiplying a matrix by a vector, the vector must go on the right. But if you want your operators to be able to convert either argument, make the operator a friend function.

Fortunately, the compiler will not take **1 – 1** and convert both arguments to **Number** objects and then call **operator–**. That would mean that existing C code might suddenly start to work differently. The compiler matches the "simplest" possibility first, which is the built-in operator for the expression **1 – 1**.

A perfect example: strings

An example where automatic type conversion is extremely helpful occurs with a **string** class. Without automatic type conversion, if you wanted to use all the existing string functions from the Standard C library, you'd have to create a member function for each one, like this:

```
//: C12:Strings1.cpp
```

```
// No auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
using namespace std;

class Stringc {
  char* s;
public:
  Stringc(const char* S = "") {
    s = (char*)malloc(strlen(S) + 1);
    require(s != 0);
    strcpy(s, S);
  }
  ~Stringc() { free(s); }
  int strcmp(const Stringc& S) const {
    return ::strcmp(s, S.s);
  }
  // ... etc., for every function in string.h
};

int main() {
  Stringc s1("hello"), s2("there");
  s1.strcmp(s2);
} ///:~
```

Here, only the **strcmp( )** function is created, but you'd have to create a corresponding function for every one in **<cstring>** that might be needed. Fortunately, you can provide an automatic type conversion allowing access to all the functions in **<cstring>**:

```
//: C12:Strings2.cpp
// With auto type conversion
#include "../require.h"
#include <cstring>
#include <cstdlib>
using namespace std;

class Stringc {
  char* s;
public:
  Stringc(const char* S = "") {
    s = (char*)malloc(strlen(S) + 1);
```

```
    require(s != 0);
    strcpy(s, S);
  }
  ~Stringc() { free(s); }
  operator const char*() const { return s; }
};

int main() {
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
} ///:~
```

Now any function that takes a **char*** argument can also take a **Stringc** argument because the compiler knows how to make a **char*** from a **Stringc**.

Pitfalls in automatic type conversion

Because the compiler must choose how to quietly perform a type conversion, it can get into trouble if you don't design your conversions correctly. A simple and obvious situation occurs with a class **X** that can convert itself to an object of class **Y** with an **operator Y( )**. If class **Y** has a constructor that takes a single argument of type **X**, this represents the identical type conversion. The compiler now has two ways to go from **X** to **Y**, so it will generate an ambiguity error when that conversion occurs:

```
//: C12:Ambig.cpp
// Ambiguity in type conversion

class Y; // Class declaration

class X {
public:
  operator Y() const; // Convert X to Y
};

class Y {
public:
  Y(X); // Convert X to Y
};
```

```
void f(Y);

int main() {
  X x;
//! f(x); // Error: ambiguous conversion
} ///:~
```

The obvious solution to this problem is not to do it: Just provide a single path for automatic conversion from one type to another.

A more difficult problem to spot occurs when you provide automatic conversion to more than one type. This is sometimes called *fan-out*:

```
//: C12:Fanout.cpp
// Type conversion fanout

class A {};
class B {};

class C {
public:
  operator A() const;
  operator B() const;
};

// Overloaded h():
void h(A);
void h(B);

int main() {
  C c;
//! h(c); // Error: C -> A or C -> B ???
} ///:~
```

Class **C** has automatic conversions to both **A** and **B**. The insidious thing about this is that there's no problem until someone innocently comes along and creates two overloaded versions of **h( )**. (With only one version, the code in **main( )** works fine.)

Again, the solution – and the general watchword with automatic type conversion – is to only provide a single automatic conversion from one type to another. You can have conversions to other types; they just shouldn't be *automatic*. You can create explicit function calls with names like **make_A( )** and **make_B( )**.

Hidden activities

Automatic type conversion can introduce more underlying activities than you may expect. As a little brain teaser, look at this modification of **FeeFi.cpp**:

```
//: C12:FeeFi2.cpp
// Copying vs. initialization

class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) { i = x; }
  operator Fee() const { return Fee(i); }
};

int main() {
  Fo fo;
  Fee fiddle = fo;
} ///:~
```

There is no constructor to create the **Fee fiddle** from a **Fo** object. However, **Fo** has an automatic type conversion to a **Fee**. There's no copy-constructor to create a **Fee** from a **Fee**, but this is one of the special functions the compiler can create for you. (The default constructor, copy-constructor, **operator=**, and destructor can be created automatically.) So for the relatively innocuous statement

```
Fee fiddle = fo;
```

the automatic type conversion operator is called, and a copy-constructor is created.

Automatic type conversion should be used carefully. It's excellent when it significantly reduces a coding task, but it's usually not worth using gratuitously.

Summary

The whole reason for the existence of operator overloading is for those situations when it makes life easier. There's nothing particularly magical about it; the overloaded operators are just functions with funny names, and the function calls happen to be made for you by the compiler when it spots the right pattern. But if operator overloading doesn't provide a significant benefit to you (the creator of the class) or the user of the class, don't confuse the issue by adding it.

Exercises

1. Create a simple class with an overloaded **operator++**. Try calling this operator in both pre- and postfix form and see what kind of compiler warning you get.
2. Create a class that contains a single **private char**. Overload the iostream operators **<<** and **>>** (as in **Iosop.cpp**) and test them. You can test them with **fstreams**, **strstreams**, and **stdiostreams** (**cin** and **cout**).
3. Write a **Number** class with overloaded operators for **+, –, *, /**, and assignment. Choose the return values for these functions so that expressions can be chained together, and for efficiency. Write an automatic type conversion **operator int( )**.
4. Combine the classes in **Unary.cpp** and **Binary.cpp**.
5. Fix **Fanout.cpp** by creating an explicit function to call to perform the type conversion, instead of one of the automatic conversion operators.

13: Dynamic object creation

Sometimes you know the exact quantity, type, and lifetime of the objects in your program. But not always.

How many planes will an air-traffic system have to handle? How many shapes will a CAD system need? How many nodes will there be in a network?

To solve the general programming problem, it's essential that you be able to create and destroy objects at runtime. Of course, C has always provided the *dynamic memory allocation* functions **malloc( )**and **free( )**(along with variants of **malloc( )**) that allocate storage from the *heap* (also called the *free store*) at runtime.

However, this simply won't work in C++. The constructor doesn't allow you to hand it the address of the memory to initialize, and for good reason: If you could do that, you might

1. Forget. Then guaranteed initialization of objects in C++ wouldn't be guaranteed.
2. Accidentally do something to the object before you initialize it, expecting the right thing to happen.
3. Hand it the wrong-sized object.

And of course, even if you did everything correctly, anyone who modifies your program is prone to the same errors. Improper initialization is

responsible for a large portion of programming errors, so it's especially important to guarantee constructor calls for objects created on the heap.

So how does C++ guarantee proper initialization and cleanup, but allow you to create objects dynamically, on the heap?

The answer is, "by bringing dynamic object creation into the core of the language." **malloc( )** and **free( )** are library functions, and thus outside the control of the compiler. However, if you have an *operator* to perform the combined act of dynamic storage allocation and initialization and another to perform the combined act of cleanup and releasing storage, the compiler can still guarantee that constructors and destructors will be called for all objects.

In this chapter, you'll learn how C++'s **new** and **delete** elegantly solve this problem by safely creating objects on the heap.

Object creation

When a C++ object is created, two events occur:

1. Storage is allocated for the object.
2. The constructor is called to initialize that storage.

By now you should believe that step two *always* happens. C++ enforces it because uninitialized objects are a major source of program bugs. It doesn't matter where or how the object is created – the constructor is always called.

Step one, however, can occur in several ways, or at alternate times:

1. Storage can be allocated before the program begins, in the *static storage area*. This storage exists for the life of the program.
2. Storage can be created on the stack whenever a particular execution point is reached (an opening brace). That storage is released automatically at the complementary execution point (the closing brace). These stack-allocation operations are built into the instruction set of the processor and are very efficient. However, you have to know exactly how much storage you need when you're writing the program so the compiler can generate the right code.

3. Storage can be allocated from a pool of memory called the *heap* (also known as the *free store*). This is called *dynamic memory allocation*. To allocate this memory, a function is called at runtime; this means you can decide at any time that you want some memory and how much you need. You are also responsible for determining when to release the memory, which means the lifetime of that memory can be as long as you choose – it isn't determined by scope.

Often these three regions are placed in a single contiguous piece of physical memory: the static area, the stack, and the heap (in an order determined by the compiler writer). However, there are no rules. The stack may be in a special place, and the heap may be implemented by making calls for chunks of memory from the operating system. As a programmer, these things are normally shielded from you, so all you need to think about is that the memory is there when you call for it.

C's approach to the heap

To allocate memory dynamically at runtime, C provides functions in its standard library: **malloc( )** and its variants **calloc( )** and **realloc( )** to produce memory from the heap, and **free( )** to release the memory back to the heap. These functions are pragmatic but primitive and require understanding and care on the part of the programmer. To create an instance of a class on the heap using C's dynamic memory functions, you'd have to do something like this:

```
//: C13:MallocClass.cpp
// Malloc with class objects
// What you'd have to do if not for "new"
#include "../require.h"
#include <cstdlib> // Malloc() & free()
#include <cstring> // Memset()
#include <iostream>
using namespace std;

class Obj {
  int i, j, k;
  static const int sz = 100;
  char buf[sz];
public:
  void initialize() { // Can't use constructor
    cout << "initializing Obj" << endl;
```

```
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() { // Can't use destructor
    cout << "destroying Obj" << endl;
  }
};

int main() {
  Obj* obj = (Obj*)malloc(sizeof(Obj));
  require(obj != 0);
  obj->initialize();
  // ... sometime later:
  obj->destroy();
  free(obj);
} ///:~
```

You can see the use of **malloc( )** to create storage for the object in the line:

```
Obj* obj = (Obj*)malloc(sizeof(Obj));
```

Here, the user must determine the size of the object (one place for an error). **malloc( )** returns a **void*** because it's just a patch of memory, not an object. C++ doesn't allow a **void*** to be assigned to any other pointer, so it must be cast.

Because **malloc( )** may fail to find any memory (in which case it returns zero), you must check the returned pointer to make sure it was successful.

But the worst problem is this line:

```
Obj->initialize();
```

If they make it this far correctly, users must remember to initialize the object before it is used. Notice that a constructor was not used because the constructor cannot be called explicitly – it's called for you by the compiler when an object is created. The problem here is that the user now has the option to forget to perform the initialization before the object is used, thus reintroducing a major source of bugs.

It also turns out that many programmers seem to find C's dynamic memory functions too confusing and complicated; it's not uncommon to find C programmers who use virtual memory machines allocating huge arrays of variables in the static storage area to avoid thinking about dynamic memory allocation. Because C++ is attempting to make library

use safe and effortless for the casual programmer, C's approach to dynamic memory is unacceptable.

operator new

The solution in C++ is to combine all the actions necessary to create an object into a single operator called **new**. When you create an object with **new** (using a *new-expression*), it allocates enough storage on the heap to hold the object, and calls the constructor for that storage. Thus, if you say

```
MyType *fp = new MyType(1,2);
```

at runtime, the equivalent of **malloc(sizeof(MyType))** is called (often, it is literally a call to **malloc( )**), and the constructor for **MyType** is called with the resulting address as the **this** pointer, using **(1,2)** as the argument list. By the time the pointer is assigned to **fp**, it's a live, initialized object – you can't even get your hands on it before then. It's also automatically the proper **MyType** type so no cast is necessary.

The default **new** also checks to make sure the memory allocation was successful before passing the address to the constructor, so you don't have to explicitly determine if the call was successful. Later in the chapter you'll find out what happens if there's no memory left.

You can create a new-expression using any constructor available for the class. If the constructor has no arguments, you can make the new-expression without the constructor argument list:

```
MyType *fp = new MyType;
```

Notice how simple the process of creating objects on the heap becomes – a single expression, with all the sizing, conversions, and safety checks built in. It's as easy to create an object on the heap as it is on the stack.

operator delete

The complement to the new-expression is the *delete-expression*, which first calls the destructor and then releases the memory (often with a call to **free( )**). Just as a new-expression returns a pointer to the object, a delete-expression requires the address of an object.

```
delete fp;
```

cleans up the dynamically allocated **MyType** object created earlier.

delete can be called only for an object created by **new**. If you **malloc( )** (or **calloc( )** or **realloc( )**) an object and then **delete** it, the behavior is

undefined. Because most default implementations of **new** and **delete** use **malloc( )** and **free( )**, you'll probably release the memory without calling the destructor.

If the pointer you're deleting is zero, nothing will happen. For this reason, people often recommend setting a pointer to zero immediately after you delete it, to prevent deleting it twice. Deleting an object more than once is definitely a bad thing to do, and will cause problems.

A simple example

This example shows that the initialization takes place:

```
//: C13:Newdel.cpp
// Simple demo of new & delete
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int height) {
    height = height;
  }
  ~Tree() { cout << "*"; }
  friend ostream&
  operator<<(ostream& os, const Tree* t) {
    return os << "Tree height is: "
              << t->height << endl;
  }
};

int main() {
  Tree* t = new Tree(40);
  cout << t;
  delete t;
} ///:~
```

We can prove that the constructor is called by printing out the value of the **Tree**. Here, it's done by overloading the **operator<<** to use with an **ostream**. Note, however, that even though the function is declared as a **friend**, it is defined as an inline! This is a mere convenience – defining a **friend** function as an inline to a class doesn't change the **friend** status or the fact that it's a global function and not a class member function. Also

notice that the return value is the result of the entire output expression, which is itself an **ostream&** (which it must be, to satisfy the return value type of the function).

Memory manager overhead

When you create auto objects on the stack, the size of the objects and their lifetime is built right into the generated code, because the compiler knows the exact quantity and scope. Creating objects on the heap involves additional overhead, both in time and in space. Here's a typical scenario. (You can replace **malloc( )** with **calloc( )** or **realloc( )**.)

4. You call **malloc( )**, which requests a block of memory from the pool. (This code may actually be part of **malloc( )**.)

5. The pool is searched for a block of memory large enough to satisfy the request. This is done by checking a map or directory of some sort that shows which blocks are currently in use and which blocks are available. It's a quick process, but it may take several tries so it might not be deterministic – that is, you can't necessarily count on **malloc( )** always taking exactly the same amount of time.

6. Before a pointer to that block is returned, the size and location of the block must be recorded so further calls to **malloc( )** won't use it, and so that when you call **free( )**, the system knows how much memory to release.

The way all this is implemented can vary widely. For example, there's nothing to prevent primitives for memory allocation being implemented in the processor. If you're curious, you can write test programs to try to guess the way your **malloc( )** is implemented. You can also read the library source code, if you have it.

Early examples redesigned

Now that **new** and **delete** have been introduced (as well as many other subjects), the **Stash** and **Stack** examples from the early part of this book

can be rewritten using all the features discussed in the book so far. Examining the new code will also give you a useful review of the topics.

At this point in the book, neither the **Stash** nor **Stack** classes will "own" the objects they point to; that is, when the **Stash** or **Stack** object goes out of scope, it will not call **delete** for all the objects it points to. The reason this is not possible is because, in an attempt to be generic, they hold **void** pointers. If you **delete** a **void** pointer, the only thing that happens is the memory gets released, because there's no type information and no way for the compiler to know what destructor to call. When a pointer is returned from the **Stash** or **Stack** object, you must cast it to the proper type before using it. These problems will be dealt with in the next chapter [[??]], and in Chapter XX.

Because the container doesn't own the pointer, the user must be responsible for it. This means there's a serious problem if you add pointers to objects created on the stack *and* objects created on the heap to the same container because a delete-expression is unsafe for a pointer that hasn't been allocated on the heap. (And when you fetch a pointer back from the container, how will you know where its object has been allocated?). Thus, you must be sure that objects stored in the upcoming versions of **Stash** and **Stack** are only made on the heap, either through careful programming or by creating classes that can only be built on the heap.

Stash for pointers

This version of the **Stash** class, which you last saw in Chapter XX, is changed to reflect all the new material introduced since Chapter XX. In addition, the new **PStash** holds *pointers* to objects that exist by themselves on the heap, whereas the old **Stash** in Chapter XX and earlier copied the *objects* into the **Stash** container. With the introduction of **new** and **delete**, it's easy and safe to hold pointers to objects that have been created on the heap.

Here's the header file for the "pointer **Stash**":

```
//: C13:PStash.h
// Holds pointers instead of objects
#ifndef PSTASH_H
#define PSTASH_H

class PStash {
  int quantity; // Number of storage spaces
```

```
  int next; // Next empty space
   // Pointer storage:
  void** storage;
  void inflate(int increase);
public:
  PStash() {
    quantity = 0;
    storage = 0;
    next = 0;
  }
  // No ownership:
  ~PStash() { delete []storage; }
  int add(void* element);
  void* operator[](int index) const; // Fetch
  // Number of elements in Stash:
  int count() const { return next; }
};
#endif // PSTASH_H ///:~
```

The underlying data elements are fairly similar, but now **storage** is an array of **void** pointers, and the allocation of storage for that array is performed with **new** instead of **malloc( )**. In the expression

```
void** st = new void*[quantity + increase];
```

the type of object allocated is a **void***, so the expression allocates an array of **void** pointers.

The destructor deletes the storage where the **void** pointers are held, rather than attempting to delete what they point at (which, as previously noted, will release their storage and not call the destructors because a **void** pointer has no type information).

The other change is the replacement of the **fetch( )** function with **operator[ ]**, which makes more sense syntactically. Again, however, a **void*** is returned, so the user must remember what types are stored in the container and cast the pointers when fetching them out (a problem which will be repaired in future chapters).

Here are the member function definitions:

```
//: C13:PStash.cpp {O}
// Pointer Stash definitions
#include "PStash.h"
#include <iostream>
#include <cstring> // 'mem' functions
```

```
using namespace std;

int PStash::add(void* element) {
  const int inflateSize = 10;
  if(next >= quantity)
    inflate(inflateSize);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Operator overloading replacement for fetch
void* PStash::operator[](int index) const {
  if(index >= next || index < 0)
    return 0;  // Out of bounds
  // Produce pointer to desired element:
  return storage[index];
}

void PStash::inflate(int increase) {
  const int psz = sizeof(void*);
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Old storage
  storage = st; // Point to new memory
} ///:~
```

The **add( )** function is effectively the same as before, except that the pointer is stored instead of a copy of the whole object, which, as you've seen, actually requires a copy-constructor for normal objects.

The **inflate( )** code is modified to handle the allocation of an array of **void*** instead of the previous design which was only working with raw bytes. Here, instead of using the prior approach of copying by array indexing, the Standard C library function **memset( )** is first used to set all the new memory to zero (this is not strictly necessary, since the **PStash** is presumably managing all the memory correctly – but it usually doesn't hurt to throw in a bit of extra care). Then **memcpy( )** moves the existing data from the old location to the new. Often, functions like **memset( )** and **memcpy( )** have been optimized over time and so they may be faster than the loops shown previously, but in a function like **inflate( )** that will probably not be used that often you probably won't see a

performance difference. However, the fact that the function calls are more concise than the loops may help prevent coding errors.

A test

Here's the old test program for **Stash** rewritten for the **PStash**:

```
//: C13:PStashTest.cpp
//{L} PStash
// Test of pointer Stash
#include "PStash.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  PStash intStash;
  // 'new' works with built-in types, too. Note
  // the "pseudo-constructor" syntax:
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i));
  for(int u = 0; u < intStash.count(); u++)
    cout << "intStash[" << u << "] = "
         << *(int*)intStash[u] << endl;

  ifstream infile("PStashTest.cpp");
  assure(infile, "PStashTest.cpp");
  PStash stringStash;
  string line;
  while(getline(infile, line))
    stringStash.add(new string(line));
  // Print out the strings:
  for(int v = 0; stringStash[v]; v++)
    cout << "stringStash[" << v << "] = "
         << *(string*)stringStash[v] << endl;
} ///:~
```

As before, **Stash**es are created and filled with information, but this time the information is the pointers resulting from new-expressions. In the first case, note the line:

```
intStash.add(new int(i));
```

The expression **new int(i)** uses the pseudoconstructor form, so storage for a new **int** object is created on the heap, and the **int** is initialized to the value **i**.

Note that during printing, the value returned by **PStash::operator[ ]** must be cast to the proper type; this is repeated for the rest of the **PStash** objects in the program. It's an undesirable effect of using **void** pointers as the underlying representation and will be fixed in later chapters.

The second test opens the source code file and reads it one line at a time into another **PStash**. Each line is read into a **string** using **getline( )**, then a **new string** is created from **line** to make an independent copy of that line. If we just passed in the address of **line** each time, we'd get a whole bunch of pointers pointing to **line**, which itself would only contain the last line that was read from the file.

When fetching the pointers back out, you see the expression:

```
*(string*)stringStash[v]
```

The pointer returned from **operator[ ]** must be cast to a **string*** to give it the proper type. Then the **string*** is dereferenced so the expression evaluates to an object, at which point the compiler sees a **string** object to send to **cout**.

In this example, the objects created on the heap are never destroyed. This is not harmful here because the storage is released when the program ends, but it's not something you want to do in practice. It will be fixed in later chapters.

The stack

The **Stack** benefits greatly from all the features introduced since Chapter XX. [[I think at this point only inlines have been added??]] Here's the new header file:

```
//: C13:Stack4.h
// New version of Stack
#ifndef STACK4_H
#define STACK4_H

class Stack {
  struct Link {
    void* data;
    Link* next;
```

```
    Link(void* dat, Link* nxt) {
      data = dat;
      next = nxt;
    }
  }* head;
public:
  Stack() { head = 0; }
  ~Stack();
  void push(void* dat) {
    head = new Link(dat,head);
  }
  void* peek() const { return head->data; }
  void* pop();
};
#endif // STACK4_H ///:~
```

The rest of the logic is virtually identical to what it was in Chapter XX. Here is the implementation of the two remaining (non-inline) functions:

```
//: C13:Stack4.cpp {O}
// New version of Stack
#include "Stack4.h"

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  Link* cursor = head;
  while(head) {
    cursor = cursor->next;
    delete head;
    head = cursor;
  }
} ///:~
```

The only difference is the use of **delete** instead of **free( )** in the destructor.

As with the **Stash**, the use of **void** pointers means that the objects created on the heap cannot be destroyed by the **Stack4**, so again there is the possibility of an undesirable memory leak if the user doesn't take responsibility for the pointers in the **Stack4**. You can see this in the test program:

```
//: C13:Stack4Test.cpp
//{L} Stack4
// Test new Stack
#include "Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  // Could also use command-line argument:
  ifstream file("Stack4Test.cpp");
  assure(file, " Stack4Test.cpp");
  Stack textlines;
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  // Pop lines from the Stack and print them:
  string* s;
  while((s = (string*)textlines.pop()) != 0)
    cout << *s << endl;
} ///:~
```

As with the **Stash** example, a file is opened and each line is read into a **string** object, which is duplicated via **new** as it is stored in a **Stack**. This program doesn't **delete** the pointers in the **Stack** and the **Stack** itself doesn't do it, so that memory is lost.

new & **delete** for arrays

In C++, you can create arrays of objects on the stack or on the heap with equal ease, and (of course) the constructor is called for each object in the array. There's one constraint, however: There must be a default constructor, except for aggregate initialization on the stack (see Chapter

XX), because a constructor with no arguments must be called for every object.

When creating arrays of objects on the heap using **new**, there's something else you must do. An example of such an array is

```
MyType* fp = new MyType[100];
```

This allocates enough storage on the heap for 100 **MyType** objects and calls the constructor for each one. Now, however, you simply have a **MyType***, which is exactly the same as you'd get if you said

```
MyType* fp2 = new MyType;
```

to create a single object. Because you wrote the code, you know that **fp** is actually the starting address of an array, so it makes sense to select array elements with **fp[2]**. But what happens when you destroy the array? The statements

```
delete fp2; // OK
delete fp;  // Not the desired effect
```

look exactly the same, and their effect will be the same: The destructor will be called for the **MyType** object pointed to by the given address, and then the storage will be released. For **fp2** this is fine, but for **fp** this means the other 99 destructor calls won't be made. The proper amount of storage will still be released, however, because it is allocated in one big chunk, and the size of the whole chunk is stashed somewhere by the allocation routine.

The solution requires you to give the compiler the information that this is actually the starting address of an array. This is accomplished with the following syntax:

```
delete []fp;
```

The empty brackets tell the compiler to generate code that fetches the number of objects in the array, stored somewhere when the array is created, and calls the destructor for that many array objects. This is actually an improved syntax from the earlier form, which you may still occasionally see in old code:

```
delete [100]fp;
```

which forced the programmer to include the number of objects in the array and introduced the possibility that the programmer would get it wrong. The additional overhead of letting the compiler handle it was very

low, and it was considered better to specify the number of objects in one place rather than two.

Making a pointer more like an array

As an aside, the **fp** defined above can be changed to point to anything, which doesn't make sense for the starting address of an array. It makes more sense to define it as a constant, so any attempt to modify the pointer will be flagged as an error. To get this effect, you might try

```
int const* q = new int[10];
```

or

```
const int* q = new int[10];
```

but in both cases the **const** will bind to the **int**, that is, what is being pointed *to*, rather than the quality of the pointer itself. Instead, you must say

```
int* const q = new int[10];
```

Now the array elements in **q** can be modified, but any change to **q** itself (like **q++**) is illegal, as it is with an ordinary array identifier.

Running out of storage

What happens when the **operator new** cannot find a contiguous block of storage large enough to hold the desired object? A special function called the *new-handler* is called. Or rather, a pointer to a function is checked, and if the pointer is nonzero, then the function it points to is called.

The default behavior for the new-handler is to *throw an exception*, the subject covered in Chapter XX. However, if you're using heap allocation in your program, it's wise to at least replace the new-handler with a message that says you've run out of memory and then aborts the program. That way, during debugging, you'll have a clue about what happened. For the final program you'll want to use more robust recovery.

You replace the new-handler by including **new.h** and then calling **set_new_handler( )** with the address of the function you want installed:

```
//: C13:Newhandl.cpp
```

```
// Changing the new-handler
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;

void out_of_memory() {
  cerr << "memory exhausted!" << endl;
  exit(1);
}

int main() {
  set_new_handler(out_of_memory);
  while(1)
    new int[1000]; // Exhausts memory
} ///:~
```

The new-handler function must take no arguments and have **void** return value. The **while** loop will keep allocating **int** objects (and throwing away their return addresses) until the free store is exhausted. At the very next call to **new**, no storage can be allocated, so the new-handler will be called.

Of course, you can write more sophisticated new-handlers, even one to try to reclaim memory (commonly known as a *garbage collector*). This is not a job for the novice programmer.

Overloading new & delete

When you create a new-expression, two things occur: First, storage is allocated using the **operator new**, then the constructor is called. In a delete-expression, the destructor is called, then storage is deallocated using the **operator delete**. The constructor and destructor calls are never under your control (otherwise you might accidentally subvert them), but you *can* change the storage allocation functions **operator new** and **operator delete**.

The memory allocation system used by **new** and **delete** is designed for general-purpose use. In special situations, however, it doesn't serve your needs. The most common reason to change the allocator is efficiency: You

might be creating and destroying so many objects of a particular class that it has become a speed bottleneck. C++ allows you to overload **new** and **delete** to implement your own storage allocation scheme, so you can handle problems like this.

Another issue is heap fragmentation: By allocating objects of different sizes it's possible to break up the heap so that you effectively run out of storage. That is, the storage might be available, but because of fragmentation no piece is big enough to satisfy your needs. By creating your own allocator for a particular class, you can ensure this never happens.

In embedded and real-time systems, a program may have to run for a very long time with restricted resources. Such a system may also require that memory allocation always take the same amount of time, and there's no allowance for heap exhaustion or fragmentation. A custom memory allocator is the solution; otherwise programmers will avoid using **new** and **delete** altogether in such cases and miss out on a valuable C++ asset.

When you overload **operator new** and **operator delete**, it's important to remember that you're changing only the way *raw storage is allocated*. The compiler will simply call your **new** instead of the default version to allocate storage, then call the constructor for that storage. So, although the compiler allocates storage *and* calls the constructor when it sees **new**, all you can change when you overload **new** is the storage allocation portion. (**delete** has a similar limitation.)

When you overload **operator new**, you also replace the behavior when it runs out of memory, so you must decide what to do in your **operator new**: return zero, write a loop to call the new-handler and retry allocation, or (typically) throw a **bad_alloc** exception (discussed in Chapter XX).

Overloading **new** and **delete** is like overloading any other operator. However, you have a choice of overloading the global allocator or using a different allocator for a particular class.

Overloading global new & delete

This is the drastic approach, when the global versions of **new** and **delete** are unsatisfactory for the whole system. If you overload the global versions, you make the defaults completely inaccessible – you can't even call them from inside your redefinitions.

The overloaded **new** must take an argument of **size_t** (the Standard C standard type for sizes). This argument is generated and passed to you by the compiler and is the size of the object you're responsible for allocating. You must return a pointer either to an object of that size (or bigger, if you have some reason to do so), or to zero if you can't find the memory (in which case the constructor is *not* called!). However, if you can't find the memory, you should probably do something more drastic than just returning zero, like calling the new-handler or throwing an exception, to signal that there's a problem.

The return value of **operator new** is a **void***, *not* a pointer to any particular type. All you've done is produce memory, not a finished object – that doesn't happen until the constructor is called, an act the compiler guarantees and which is out of your control.

The **operator delete** takes a **void*** to memory that was allocated by **operator new**. It's a **void*** because you get that pointer *after* the destructor is called, which removes the object-ness from the piece of storage. The return type is **void**.

Here's a very simple example showing how to overload the global **new** and **delete**:

```
//: C13:GlobalNew.cpp
// Overload global new/delete
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class S {
  int i[100];
public:
```

```
  S() { puts("S::S()"); }
  ~S() { puts("S::~S()"); }
};

int main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  S* s = new S;
  delete s;
  puts("creating & destroying S[3]");
  S* sa = new S[3];
  delete []sa;
} ///:~
```

Here you can see the general form for overloading **new** and **delete**. These use the Standard C library functions **malloc( )** and **free( )** for the allocators (which is probably what the default **new** and **delete** use, as well!). However, they also print out messages about what they are doing. Notice that **printf( )** and **puts( )** are used rather than **iostreams**. Thus, when an **iostream** object is created (like the global **cin**, **cout**, and **cerr**), they call **new** to allocate memory. With **printf( )**, you don't get into a deadlock because it doesn't call **new** to initialize itself.

In **main( )**, objects of built-in types are created to prove that the overloaded **new** and **delete** are also called in that case. Then a single object of type **s** is created, followed by an array. For the array, you'll see that extra memory is requested to put information about the number of objects in the array. In all cases, the global overloaded versions of **new** and **delete** are used.

Overloading new & delete for a class

Although you don't have to explicitly say **static**, when you overload **new** and **delete** for a class, you're creating **static** member functions. Again, the syntax is the same as overloading any other operator. When the compiler sees you use **new** to create an object of your class, it chooses the member **operator new** over the global version. However, the global versions of **new** and **delete** are used for all other types of objects (unless they have their own **new** and **delete**).

In the following example, a very primitive storage allocation system is created for the class **Framis**. A chunk of memory is set aside in the static data area at program start-up, and that memory is used to allocate space for objects of type **Framis**. To determine which blocks have been allocated, a simple array of bytes is used, one byte for each block:

```
//: C13:Framis.cpp
// Local overloaded new & delete
#include <cstddef> // Size_t
#include <fstream>
#include <new>
using namespace std;
ofstream out("Framis.out");

class Framis {
  static const int sz = 10;
  char c[sz]; // To take up space, not used
  static unsigned char pool[];
  static unsigned char alloc_map[];
public:
  static const int psize = 100;  // frami allowed
  Framis() { out << "Framis()\n"; }
  ~Framis() { out << "~Framis() ... "; }
  void* operator new(size_t) throw(bad_alloc);
  void operator delete(void*);
};
unsigned char Framis::pool[psize * sizeof(Framis)];
unsigned char Framis::alloc_map[psize] = {0};

// Size is ignored -- assume a Framis object
void* Framis::operator new(size_t)
  throw(bad_alloc) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = 1; // Mark it used
      return pool + (i * sizeof(Framis));
    }
  out << "out of memory" << endl;
  throw bad_alloc();
}

void Framis::operator delete(void* m) {
```

```
  if(!m) return; // Check for null pointer
  // Assume it was created in the pool
  // Calculate which block number it is:
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(Framis);
  out << "freeing block " << block << endl;
  // Mark it free:
  alloc_map[block] = 0;
}

int main() {
  Framis* f[Framis::psize];
  for(int i = 0; i < Framis::psize; i++)
    f[i] = new Framis;
  new Framis; // Out of memory
  delete f[10];
  f[10] = 0;
  // Use released memory:
  Framis* x = new Framis;
  delete x;
  for(int j = 0; j < Framis::psize; j++)
    delete f[j]; // Delete f[10] OK
} ///:~
```

The pool of memory for the **Framis** heap is created by allocating an array of bytes large enough to hold **psize Framis** objects. The allocation map is **psize** bytes long, so there's one byte for every block. All the bytes in the allocation map are initialized to zero using the aggregate initialization trick of setting the first element to zero so the compiler automatically initializes all the rest.

The local **operator new** has the same form as the global one. All it does is search through the allocation map looking for a zero byte, then sets that byte to one to indicate it's been allocated and returns the address of that particular block. If it can't find any memory, it issues a message and returns zero (Notice that the new-handler is not called and no exceptions are thrown because the behavior when you run out of memory is now under your control.) In this example, it's OK to use iostreams because the global **operator new** and **delete** are untouched.

The **operator delete** assumes the **Framis** address was created in the pool. This is a fair assumption, because the local **operator new** will be called whenever you create a single **Framis** object on the heap – but not

an array. Global **new** is used in that case. So the user might accidentally have called **operator delete** without using the empty bracket syntax to indicate array destruction. This would cause a problem. Also, the user might be deleting a pointer to an object created on the stack. If you think these things could occur, you might want to add a line to make sure the address is within the pool and on a correct boundary.

operator delete calculates which block in the pool this pointer represents, and then sets the allocation map's flag for that block to zero to indicate the block has been released.

In **main( )**, enough **Framis** objects are dynamically allocated to run out of memory; this checks the out-of-memory behavior. Then one of the objects is freed, and another one is created to show that the released memory is reused.

Because this allocation scheme is specific to **Framis** objects, it's probably much faster than the general-purpose memory allocation scheme used for the default **new** and **delete**.

Overloading new & delete for arrays

If you overload operator **new** and **delete** for a class, those operators are called whenever you create an object of that class. However, if you create an *array* of those class objects, the global **operator new** is called to allocate enough storage for the array all at once, and the global **operator delete** is called to release that storage. You can control the allocation of arrays of objects by overloading the special array versions of **operator new[ ]** and **operator delete[ ]** for the class. Here's an example that shows when the two different versions are called:

```
//: C13:ArrayNew.cpp
// Operator new for arrays
#include <new> // Size_t definition
#include <fstream>
using namespace std;
ofstream trace("ArrayNew.out");

class Widget {
  static const int sz = 10;
  int i[sz];
public:
```

```
  Widget() { trace << "*"; }
  ~Widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "Widget::new: "
        << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "Widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "Widget::new[]: "
        << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "Widget::delete[]" << endl;
    ::delete []p;
  }
};

int main() {
  trace << "new Widget" << endl;
  Widget* w = new Widget;
  trace << "\ndelete Widget" << endl;
  delete w;
  trace << "\nnew Widget[25]" << endl;
  Widget* wa = new Widget[25];
  trace << "\ndelete []Widget" << endl;
  delete []wa;
} ///:~
```

Here, the global versions of **new** and **delete** are called so the effect is the same as having no overloaded versions of **new** and **delete** except that trace information is added. Of course, you can use any memory allocation scheme you want in the overloaded **new** and **delete**.

You can see that the array versions of **new** and **delete** are the same as the individual-object versions with the addition of the brackets. In both cases you're handed the size of the memory you must allocate. The size handed to the array version will be the size of the entire array. It's worth keeping in mind that the *only* thing the overloaded operator **new** is

required to do is hand back a pointer to a large enough memory block. Although you may perform initialization on that memory, normally that's the job of the constructor that will automatically be called for your memory by the compiler.

The constructor and destructor simply print out characters so you can see when they've been called. Here's what the trace file looks like for one compiler:

```
new Widget
Widget::new: 20 bytes
*
delete Widget
~Widget::delete

new Widget[25]
Widget::new[]: 504 bytes
************************
delete []Widget
~~~~~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

Creating an individual object requires 20 bytes, as you might expect. (This machine uses two bytes for an **int**). The **operator new** is called, then the constructor (indicated by the *****). In a complementary fashion, calling **delete** causes the destructor to be called, then the **operator delete**.

When an array of **Widget** objects is created, the array version of **operator new** is used, as promised. But notice that the size requested is four more bytes than expected. This extra four bytes is where the system keeps information about the array, in particular, the number of objects in the array. That way, when you say

```
delete []Widget;
```

the brackets tell the compiler it's an array of objects, so the compiler generates code to look for the number of objects in the array and to call the destructor that many times.

You can see that, even though the array **operator new** and **operator delete** are only called once for the entire array chunk, the default constructor and destructor are called for each object in the array.

Constructor calls

Considering that

```
MyType* f = new MyType;
```

calls **new** to allocate a **MyType**-sized piece of storage, then invokes the **MyType** constructor on that storage, what happens if all the safeguards fail and the value returned by **operator new** is zero? The constructor is not called in that case, so although you still have an unsuccessfully created object, at least you haven't invoked the constructor and handed it a zero pointer. Here's an example to prove it:

```
//: C13:NoMemory.cpp
// Constructor isn't called
// If new returns 0
#include <iostream>
#include <new> // size_t definition
using namespace std;

void my_new_handler() {
  cout << "new handler called" << endl;
}

class NoMemory {
public:
  NoMemory() {
    cout << "NoMemory::NoMemory()" << endl;
  }
  void* operator new(size_t sz) throw(bad_alloc){
    cout << "NoMemory::operator new" << endl;
    throw bad_alloc(); // "Out of memory"
  }
};

int main() {
  set_new_handler(my_new_handler);
  NoMemory* nm = new NoMemory;
  cout << "nm = " << nm << endl;
} ///:~
```

When the program runs, it prints only the message from **operator new**. Because **new** returns zero, the constructor is never called so its message is not printed.

Object placement

There are two other, less common, uses for overloading **operator new**.

1. You may want to place an object in a specific location in memory. This is especially important with hardware-oriented embedded systems where an object may be synonymous with a particular piece of hardware.
2. You may want to be able to choose from different allocators when calling **new**.

Both of these situations are solved with the same mechanism: The overloaded **operator new** can take more than one argument. As you've seen before, the first argument is always the size of the object, which is secretly calculated and passed by the compiler. But the other arguments can be anything you want: the address you want the object placed at, a reference to a memory allocation function or object, or anything else that is convenient for you.

The way you pass the extra arguments to **operator new** during a call may seem slightly curious at first: You put the argument list (*without* the **size_t** argument, which is handled by the compiler) after the keyword **new** and before the class name of the object you're creating. For example,

```
X* xp = new(a) X;
```

will pass **a** as the second argument to **operator new**. Of course, this can work only if such an **operator new** has been declared.

Here's an example showing how you can place an object at a particular location:

```
//: C13:PlacementNew.cpp
// Placement with operator new
#include <cstddef> // Size_t
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {}
  ~X() {
    cout << "X::~X()" << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
```

```
};

int main() {
  int l[10];
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call
  // ONLY use with placement!
} ///:~
```

Notice that **operator new** only returns the pointer that's passed to it. Thus, the caller decides where the object is going to sit, and the constructor is called for that memory as part of the new-expression.

A dilemma occurs when you want to destroy the object. There's only one version of **operator delete**, so there's no way to say, "Use my special deallocator for this object." You want to call the destructor, but you don't want the memory to be released by the dynamic memory mechanism because it wasn't allocated on the heap.

The answer is a very special syntax: You can explicitly call the destructor, as in

```
xp->X::~X(); // Explicit destructor call
```

A stern warning is in order here. Some people see this as a way to destroy objects at some time before the end of the scope, rather than either adjusting the scope or (more correctly) using dynamic object creation if they want the object's lifetime to be determined at runtime. You will have serious problems if you call the destructor this way for an object created on the stack because the destructor will be called again at the end of the scope. If you call the destructor this way for an object that was created on the heap, the destructor will execute, but the memory won't be released, which probably isn't what you want. The only reason that the destructor can be called explicitly this way is to support the placement syntax for **operator new**.

Although this example shows only one additional argument, there's nothing to prevent you from adding more if you need them for other purposes.

Summary

It's convenient and optimally efficient to create automatic objects on the stack, but to solve the general programming problem you must be able to create and destroy objects at any time during a program's execution,

particularly to respond to information from outside the program. Although C's dynamic memory allocation will get storage from the heap, it doesn't provide the ease of use and guaranteed construction necessary in C++. By bringing dynamic object creation into the core of the language with **new** and **delete**, you can create objects on the heap as easily as making them on the stack. In addition, you get a great deal of flexibility. You can change the behavior of **new** and **delete** if they don't suit your needs, particularly if they aren't efficient enough. Also, you can modify what happens when the heap runs out of storage. (However, *exception handling*, described in Chapter XX, also comes into play here.)

Exercises

1. Prove to yourself that **new** and **delete** always call the constructors and destructors by creating a class with a constructor and destructor that announce themselves through **cout**. Create an object of that class with **new**, and destroy it with **delete**. Also create and destroy an array of these objects on the heap.
2. Create a **PStash** object, and fill it with **new** objects from Exercise 1. Observe what happens when this **PStash** object goes out of scope and its destructor is called.
3. Create a class with an overloaded operator **new** and **delete**, both the single-object versions and the array versions. Demonstrate that both versions work.
4. Devise a test for **Framis.cpp** to show yourself approximately how much faster the custom **new** and **delete** run than the global **new** and **delete**.

14: Inheritance & composition

One of the most compelling features about C++ is code reuse. But to be revolutionary, you need to be able to do a lot more than copy code and change it.

That's the C approach, and it hasn't worked very well. As with most everything in C++, the solution revolves around the class. You reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone else has built and debugged.

The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this. The first is quite straightforward: You simply create objects of your existing class inside the new class. This is called *composition* because the new class is composed of objects of existing classes.

The second approach is more subtle. You create a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it, without modifying the existing class. This magical act is called *inheritance*, and most of the work is done by the compiler. Inheritance is one of the cornerstones of object-oriented programming and has additional implications that will be explored in the next chapter.

It turns out that much of the syntax and behavior are similar for both composition and inheritance (which makes sense; they are both ways of making new types from existing types). In this chapter, you'll learn about these code reuse mechanisms.

Composition syntax

Actually, you've been using composition all along to create classes. You've just been composing classes using built-in types. It turns out to be almost as easy to use composition with user-defined types.

Consider an existing class that is valuable for some reason:

```
//: C14:Useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H

class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

The data members are **private** in this class, so it's completely safe to embed an object of type **X** as a **public** object in a new class, which makes the interface straightforward:

```
//: C14:Compose.cpp
// Reuse code with composition
#include "Useful.h"

class Y {
  int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
```

```
  y.x.set(37); // Access the embedded object
} ///:~
```

Accessing the member functions of the embedded object (referred to as a *subobject*) simply requires another member selection.

It's probably more common to make the embedded objects **private**, so they become part of the underlying implementation (which means you can change the implementation if you want). The **public** interface functions for your new class then involve the use of the embedded object, but they don't necessarily mimic the object's interface:

```
//: C14:Compose2.cpp
// Private embedded objects
#include "Useful.h"

class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
  y.permute();
} ///:~
```

Here, the **permute( )** function is carried through to the new class interface, but the other member functions of **X** are used within the members of **Y**.

Inheritance syntax

The syntax for composition is obvious, but to perform inheritance there's a new and different form.

When you inherit, you are saying, "This new class is like that old class." You state this in code by giving the name of the class, as usual, but before the opening brace of the class body, you put a colon and the name of the

base class (or classes, for multiple inheritance). When you do this, you automatically get all the data members and member functions in the base class. Here's an example:

```
//: C14:Inherit.cpp
// Simple inheritance
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // Same-name function call
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
} ///:~
```

In **Y** you can see inheritance going on, which means that **Y** will contain all the data elements in **X** and all the member functions in **X**. In fact, **Y** contains a subobject of **X** just as if you had created a member object of **X** inside **Y** instead of inheriting from **X**. Both member objects and base class storage are referred to as subobjects.

In **main( )** you can see that the data elements are added because the **sizeof(Y)** is twice as big as **sizeof(X)**.

You'll notice that the base class is preceded by **public**. During inheritance, everything defaults to **private**, which means all the **public** members of the base class are **private** in the derived class. This is almost never what you want; the desired result is to keep all the **public** members of the base class **public** in the derived class. You do this by using the **public** keyword during inheritance.

In **change( )**, the base-class **permute( )** function is called. The derived class has direct access to all the **public** base-class functions.

The **set( )** function in the derived class *redefines* the **set( )** function in the base class. That is, if you call the functions **read( )** and **permute( )** for an object of type **Y**, you'll get the base-class versions of those functions (you can see this happen inside **main( )**), but if you call **set( )** for a **Y** object, you get the redefined version. This means that if you don't like the version of a function you get during inheritance, you can change what it does. (You can also add completely new functions like **change( )**.)

However, when you're redefining a function, you may still want to call the base-class version. If, inside **set( )**, you simply call **set( )** you'll get the local version of the function – a recursive function call. To call the base-class version, you must explicitly name it, using the base-class name and the scope resolution operator.

The constructor initializer list

You've seen how important it is in C++ to guarantee proper initialization, and it's no different during composition and inheritance. When an object is created, the compiler guarantees that constructors for all its subobjects are called. In the examples so far, all the subobjects have default constructors, and that's what the compiler automatically calls. But what happens if your subobjects don't have default constructors, or if you want to change a default argument in a constructor? This is a problem because the new class constructor doesn't have permission to access the **private** data elements of the subobject, so it can't initialize them directly.

The solution is simple: Call the constructor for the subobject. C++ provides a special syntax for this, the *constructor initializer list*. The form

of the constructor initializer list echoes the act of inheritance. With inheritance, you put the base classes after a colon and before the opening brace of the class body. In the constructor initializer list, you put the calls to subobject constructors after the constructor argument list and a colon, but before the opening brace of the function body. For a class **MyType**, inherited from **Bar**, this might look like

```
MyType::MyType(int i) : Bar(i) { // ...
```

if **Bar** has a constructor that takes a single **int** argument.

Member object initialization

It turns out that you use this very same syntax for member object initialization when using composition. For composition, you give the names of the objects rather than the class names. If you have more than one constructor call in the initializer list, you separate the calls with commas:

```
MyType2::MyType2(int i) : Bar(i), memb(i+1) { // ...
```

This is the beginning of a constructor for class **MyType2**, which is inherited from **Bar** and contains a member object called **memb**. Note that while you can see the type of the base class in the constructor initializer list, you only see the member object identifier.

Built-in types in the initializer list

The constructor initializer list allows you to explicitly call the constructors for member objects. In fact, there's no other way to call those constructors. The idea is that the constructors are all called before you get into the body of the new class's constructor. That way, any calls you make to member functions of subobjects will always go to initialized objects. There's no way to get to the opening brace of the constructor without *some* constructor being called for all the member objects and base-class objects, even if the compiler must make a hidden call to a default constructor. This is a further enforcement of the C++ guarantee that no object (or part of an object) can get out of the starting gate without its constructor being called.

This idea that all the member objects are initialized by the opening brace of the constructor is a convenient programming aid, as well. Once you hit the opening brace, you can assume all subobjects are properly initialized

and focus on specific tasks you want to accomplish in the constructor. However, there's a hitch: What about embedded objects of built-in types, which don't *have* constructors?

To make the syntax consistent, you're allowed to treat built-in types as if they have a single constructor, which takes a single argument: a variable of the same type as the variable you're initializing. Thus, you can say

```
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
  // ...
```

The action of these "pseudoconstructor calls" is to perform a simple assignment. It's a convenient technique and a good coding style, so you'll often see it used.

It's even possible to use the pseudoconstructor syntax when creating a variable of this type outside of a class:

```
int i(100);
```

This makes built-in types act a little bit more like objects. Remember, though, that these are not real constructors. In particular, if you don't explicitly make a pseudo-constructor call, no initialization is performed.

Combining composition & inheritance

Of course, you can use the two together. The following example shows the creation of a more complex class, using both inheritance and composition.

```
//: C14:Combined.cpp
// Inheritance & composition

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
```

```
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};

int main() {
  C c(47);
} ///:~
```

C inherits from **B** and has a member object ("is composed of") **A**. You can see the constructor initializer list contains calls to both the base-class constructor and the member-object constructor.

The function **C::f( )** redefines **B::f( )** that it inherits, and also calls the base-class version. In addition, it calls **a.f( )**. Notice that the only time you can talk about redefinition of functions is during inheritance; with a member object you can only manipulate the public interface of the object, not redefine it. In addition, calling **f( )** for an object of class **C** would not call **a.f( )** if **C::f( )** had not been defined, whereas it *would* call **B::f( )**.

Automatic destructor calls

Although you are often required to make explicit constructor calls in the initializer list, you never need to make explicit destructor calls because there's only one destructor for any class, and it doesn't take any arguments. However, the compiler still ensures that all destructors are

called, and that means all the destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root.

It's worth emphasizing that constructors and destructors are quite unusual in that every one in the hierarchy is called, whereas with a normal member function only that function is called, but not any of the base-class versions. If you also want to call the base-class version of a normal member function that you're overriding, you must do it explicitly.

Order of constructor & destructor calls

It's interesting to know the order of constructor and destructor calls when an object has many subobjects. The following example shows exactly how it works:

```
//: C14:Order.cpp
// Constructor/destructor order
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 constructor\n";
  }
  ~Derived1() {
```

```
    out << "Derived1 destructor\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 constructor\n";
  }
  ~Derived2() {
    out << "Derived2 destructor\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

First, an **ofstream** object is created to send all the output to a file. Then, to save some typing and demonstrate a macro technique that will be replaced by a much improved technique in Chapter XX, a macro is created to build some of the classes, which are then used in inheritance and composition. Each of the constructors and destructors report themselves to the trace file. Note that the constructors are not default constructors; they each have an **int** argument. The argument itself has no identifier; its only job is to force you to explicitly call the constructors in the initializer list. (Eliminating the identifier prevents compiler warning messages.)

The output of this program is

```
Base1 constructor
Member1 constructor
Member2 constructor
Derived1 constructor
Member3 constructor
Member4 constructor
Derived2 constructor
Derived2 destructor
Member4 destructor
Member3 destructor
Derived1 destructor
Member2 destructor
```

```
Member1 destructor
Base1 destructor
```

You can see that construction starts at the very root of the class hierarchy, and that at each level the base class constructor is called first, followed by the member object constructors. The destructors are called in exactly the reverse order of the constructors – this is important because of potential dependencies.

It's also interesting that the order of constructor calls for member objects is completely unaffected by the order of the calls in the constructor initializer list. The order is determined by the order that the member objects are declared in the class. If you could change the order of constructor calls via the constructor initializer list, you could have two different call sequences in two different constructors, but the poor destructor wouldn't know how to properly reverse the order of the calls for destruction, and you could end up with a dependency problem.

Name hiding

If a base class has a function name that's overloaded several times, redefining that function name in the derived class will hide *all* the base-class versions. That is, they become unavailable in the derived class:

```
//: C14:Hide.cpp
// Name hiding during inheritance

class Homer {
public:
  int doh(int) const { return 1; }
  char doh(char) const { return 'd';}
  float doh(float) const { return 1.0; }
};

class Bart : public Homer {
public:
  class Milhouse {};
  void doh(Milhouse) const {}
};

int main() {
  Bart b;
//! b.doh(1); // Error
//! b.doh('x'); // Error
```

```
//! b.doh(1.0); // Error
} ///:~
```

Because **Bart** redefines **doh( )**, none of the base-class versions can be called for a **Bart** object. In each case, the compiler attempts to convert the argument into a **Milhouse** object and complains because it can't find a conversion.

As you'll see in the next chapter, it's far more common to redefine functions using exactly the same signature and return type as in the base class.

Functions that don't automatically inherit

Not all functions are automatically inherited from the base class into the derived class. Constructors and destructors deal with the creation and destruction of an object, and they can know what to do with the aspects of the object only for their particular level, so all the constructors and destructors in the entire hierarchy must be called. Thus, constructors and destructors don't inherit.

In addition, the **operator=** doesn't inherit because it performs a constructor-like activity. That is, just because you know how to initialize all the members of an object on the left-hand side of the **=** from an object on the right-hand side doesn't mean that initialization will still have meaning after inheritance.

In lieu of inheritance, these functions are synthesized by the compiler if you don't create them yourself. (With constructors, you can't create *any* constructors for the default constructor and the copy-constructor to be automatically created.) This was briefly described in Chapter XX. The synthesized constructors use memberwise initialization and the synthesized **operator=** uses memberwise assignment. Here's an example of the functions that are created by the compiler rather than inherited:

```
//: C14:Ninherit.cpp
// Non-inherited functions
#include <fstream>
using namespace std;
ofstream out("ninherit.out");

class Root {
public:
```

```
  Root() { out << "Root()\n"; }
  Root(Root&) { out << "Root(Root&)\n"; }
  Root(int) { out << "Root(int)\n"; }
  Root& operator=(const Root&) {
    out << "Root::operator=()\n";
    return *this;
  }
  class Other {};
  operator Other() const {
    out << "Root::operator Other()\n";
    return Other();
  }
  ~Root() { out << "~Root()\n"; }
};

class Derived : public Root {};

void f(Root::Other) {}

int main() {
  Derived d1;  // Default constructor
  Derived d2 = d1; // Copy-constructor
//! Derived d3(1); // Error: no int constructor
  d1 = d2; // Operator= not inherited
  f(d1); // Type-conversion IS inherited
} ///:~
```

All the constructors and the **operator=** announce themselves so you can see when they're used by the compiler. In addition, the **operator Other( )** performs automatic type conversion from a **Root** object to an object of the nested class **Other**. The class **Derived** simply inherits from **Root** and creates no functions (to see how the compiler responds). The function **f( )** takes an **Other** object to test the automatic type conversion function.

In **main( )**, the default constructor and copy-constructor are created and the **Root** versions are called as part of the constructor-call hierarchy. Even though it looks like inheritance, new constructors are actually created. As you might expect, no constructors with arguments are automatically created because that's too much for the compiler to intuit.

The **operator=** is also synthesized as a new function in **Derived** using memberwise assignment because that function was not explicitly written in the new class.

Because of all these rules about rewriting functions that handle object creation, it may seem a little strange at first that the automatic type conversion operator *is* inherited. But it's not too unreasonable – if there are enough pieces in **Root** to make an **Other** object, those pieces are still there in anything derived from **Root** and the type conversion operator is still valid (even though you may in fact want to redefine it).

Choosing composition vs. inheritance

Both composition and inheritance place subobjects inside your new class. Both use the constructor initializer list to construct these subobjects. You may now be wondering what the difference is between the two, and when to choose one over the other.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object that you're planning on using to implement features of your new class, but the user of your new class sees the interface you've defined rather than the interface from the original class. For this effect, you embed **private** objects of existing classes inside your new class.

Sometimes it makes sense to allow the class user to directly access the composition of your new class, that is, to make the member objects **public**. The member objects use implementation hiding themselves, so this is a safe thing to do and when the user knows you're assembling a bunch of parts, it makes the interface easier to understand. A **car** object is a good example:

```
//: C14:Car.cpp
// Public composition

class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
```

```
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
  Door left, right; // 2-door
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

Because the composition of a car is part of the analysis of the problem (and not simply part of the underlying design), making the members public assists the client programmer's understanding of how to use the class and requires less code complexity for the creator of the class.

With a little thought, you'll also see that it would make no sense to compose a car using a vehicle object – a car doesn't contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

Subtyping

Now suppose you want to create a type of **ifstream** object that not only opens a file but also keeps track of the name of the file. You can use

composition and embed both an **ifstream** and a **strstream** into the new class:

```
//: C14:FName1.cpp
// An fstream with a file name
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;

class FName1 {
  ifstream file;
  static const int bsize = 100;
  char buf[bsize];
  ostrstream fname;
  int nameset;
public:
  FName1() : fname(buf, bsize), nameset(0) {}
  FName1(const char* filename)
    : file(filename), fname(buf, bsize) {
      assure(file, filename);
      fname << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    fname << newname << ends;
    nameset = 1;
  }
  operator ifstream&() { return file; }
};

int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Error: rdbuf() not a member:
//!  cout << file.rdbuf() << endl;
} ///:~
```

There's a problem here, however. An attempt is made to allow the use of the **FName1** object anywhere an **ifstream** object is used, by including an

automatic type conversion operator from **FName1** to an **ifstream&**. But in main, the line

```
cout << file.rdbuf() << endl;
```

will not compile because automatic type conversion happens only in function calls, not during member selection. So this approach won't work.

A second approach is to add the definition of **rdbuf( )** to **FName1**:

```
filebuf* rdbuf() { return file.rdbuf(); }
```

This will work if there are only a few functions you want to bring through from the **ifstream** class. In that case you're only using part of the class, and composition is appropriate.

But what if you want everything in the class to come through? This is called *subtyping* because you're making a new type from an existing type, and you want your new type to have exactly the same interface as the existing type (plus any other member functions you want to add), so you can use it everywhere you'd use the existing type. This is where inheritance is essential. You can see that subtyping solves the problem in the preceding example perfectly:

```
//: C14:FName2.cpp
// Subtyping solves the problem
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;

class FName2 : public ifstream {
  static const int bsize = 100;
  char buf[bsize];
  ostrstream fname;
  int nameset;
public:
  FName2() : fname(buf, bsize), nameset(0) {}
  FName2(const char* filename)
    : ifstream(filename), fname(buf, bsize) {
      assure(*this, filename);
      fname << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
```

```
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    fname << newname << ends;
    nameset = 1;
  }
};

int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "name: " << file.name() << endl;
  const int bsize = 100;
  char buf[bsize];
  file.getline(buf, bsize); // This works too!
  file.seekg(-200, ios::end);
  cout << file.rdbuf() << endl;
} ///:~
```

Now any member function that works with an **ifstream** object also works with an **FName2** object. That's because an **FName2** *is* a type of **ifstream**; it doesn't simply contain one. This is a very important issue that will be explored at the end of this chapter and in the next one.

Specialization

When you inherit, you take an existing class and make a special version of it. Generally, this means you're taking a general-purpose class and specializing it for a particular need.

For example, consider the **Stack** class from the previous chapter. One of the problems with that class is that you had to perform a cast every time you fetched a pointer from the container. This is not only tedious, it's unsafe – you could cast the pointer to anything you want.

An approach that seems better at first glance is to specialize the general **Stack** class using inheritance. Here's an example that uses the class from the previous chapter:

```
//: C14:InheritStack.cpp
//{L} ../C13/Stack4
// Specializing the Stack class
#include "../C13/Stack4.h"
#include "../require.h"
#include <iostream>
```

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

class StringList : public Stack {
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
};

int main() {
  ifstream file("InheritStack.cpp");
  assure(file, "InheritStack.cpp");
  string line;
  StringList textlines;
  while(getline(file,line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
} ///:~
```

The **Stack4.h** header file is brought in from Chapter XX. (The Stack4 object file must be linked in as well.)

Stringlist specializes **Stack** so that **push( )** will accept only **String** pointers. Before, **Stack** would accept **void** pointers, so the user had no type checking to make sure the proper pointers were inserted. In addition, **peek( )** and **pop( )** now return **String** pointers rather than **void** pointers, so no cast is necessary to use the pointer.

Amazingly enough, this extra type-checking safety is free! The compiler is being given extra type information, that it uses at compile-time, but the functions are inline and no extra code is generated.

Unfortunately, inheritance doesn't solve all the problems with this container class. The destructor still causes trouble. You'll remember from

Chapter XX that the **Stack::~Stack( )** destructor moves through the list and calls **delete** for all the pointers. The problem is, **delete** is called for **void** pointers, which only releases the memory and doesn't call the destructors (because **void*** has no type information). If a **Stringlist::~Stringlist( )** destructor is created to move through the list and call **delete** for all the **String** pointers in the list, the problem is solved *if*

1. The **Stack** data members are made **protected** so the **Stringlist** destructor can access them. (**protected** is described a bit later in the chapter.)
2. The **Stack** base class destructor is removed so the memory isn't released twice.
3. No more inheritance is performed, because you'd end up with the same dilemma again: multiple destructor calls versus an incorrect destructor call (to a **String** object rather than what the class derived from **Stringlist** might contain).

This issue will be revisited in the next chapter, but will not be fully solved until templates are introduced in Chapter XX.

A more important observation to make about this example is that it *changes the interface* of the **Stack** in the process of inheritance. If the interface is different, then a **Stringlist** really isn't a **Stack**, and you will never be able to correctly use a **Stringlist** as a **Stack**. This questions the use of inheritance here: if you're not creating a **Stringlist** that *is-a* type of **Stack**, then why are you inheriting? A more appropriate version of **Stringlist** will be shown later in the chapter.

private inheritance

You can inherit a base class privately by leaving off the **public** in the base-class list, or by explicitly saying **private** (probably a better policy because it is clear to the user that you mean it). When you inherit privately, you're "implementing in terms of"; that is, you're creating a new class that has all the data and functionality of the base class, but that functionality is hidden, so it's only part of the underlying implementation. The class user has no access to the underlying functionality, and an object cannot be treated as a member of the base class (as it was in **FName2.cpp**).

You may wonder what the purpose of **private** inheritance is, because the alternative of creating a **private** object in the new class seems more

appropriate. **private** inheritance is included in the language for completeness, but if for no other reason than to reduce confusion, you'll usually want to use a **private** member rather than **private** inheritance. However, there may occasionally be situations where you want to produce part of the same interface as the base class *and* disallow the treatment of the object as if it were a base-class object. **private** inheritance provides this ability.

Publicizing privately inherited members

When you inherit privately, all the **public** members of the base class become **private**. If you want any of them to be visible, just say their names (no arguments or return values) in the **public** section of the derived class:

```
//: C14:Privinh.cpp
// Private inheritance

class Base1 {
public:
  char f() const { return 'a'; }
  int g() const { return 2; }
  float h() const { return 3.0; }
};

class Derived : Base1 { // Private inheritance
public:
  Base1::f; // Name publicizes member
  Base1::h;
};

int main() {
  Derived d;
  d.f();
  d.h();
//! d.g(); // Error -- private function
} ///:~
```

Thus, **private** inheritance is useful if you want to hide part of the functionality of the base class.

You should think carefully before using **private** inheritance instead of member objects; **private** inheritance has particular complications when combined with runtime type identification (the subject of Chapter XX).

protected

Now that you've been introduced to inheritance, the keyword **protected** finally has meaning. In an ideal world, **private** members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is a nod to pragmatism; it says, "This is **private** as far as the class user is concerned, but available to anyone who inherits from this class."

The best approach is to leave the data members **private** – you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** member functions:

```
//: C14:Protect.cpp
// The protected keyword
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
} ///:~
```

You will find examples of the need for **protected** in examples later in this book.

protected inheritance

When you're inheriting, the base class defaults to **private**, which means that all the public member functions are **private** to the user of the new class. Normally, you'll make the inheritance **public** so the interface of the base class is also the interface of the derived class. However, you can also use the **protected** keyword during inheritance.

Protected derivation means "implemented-in-terms-of" to other classes but "is-a" for derived classes and friends. It's something you don't use very often, but it's in the language for completeness.

Multiple inheritance

You can inherit from one class, so it would seem to make sense to inherit from more than one class at a time. Indeed you can, but whether it makes sense as part of a design is a subject of continuing debate. One thing is generally agreed upon: You shouldn't try this until you've been programming quite a while and understand the language thoroughly. By that time, you'll probably realize that no matter how much you think you absolutely must use multiple inheritance, you can almost always get away with single inheritance.

Initially, multiple inheritance seems simple enough: You add more classes in the base-class list during inheritance, separated by commas. However, multiple inheritance introduces a number of possibilities for ambiguity, which is why Chapter XX is devoted to the subject.

Incremental development

One of the advantages of inheritance is that it supports *incremental development* by allowing you to introduce new code without causing bugs in existing code and isolating new bugs to the new code. By inheriting from an existing, functional class and adding data members and member functions (and redefining existing member functions) you leave the

existing code – that someone else may still be using – untouched and unbugged. If a bug happens, you know it's in your new code, which is much shorter and easier to read than if you had modified the body of existing code.

It's rather amazing how cleanly the classes are separated. You don't even need the source code for the member functions to reuse the code, just the header file describing the class and the object file or library file with the compiled member functions. (This is true for both inheritance and composition.)

It's important to realize that program development is an incremental process, just like human learning. You can do as much analysis as you want, but you still won't know all the answers when you set out on a project. You'll have much more success – and more immediate feedback – if you start out to "grow" your project as an organic, evolutionary creature, rather than constructing it all at once like a glass-box skyscraper.

Although inheritance for experimentation is a useful technique, at some point after things stabilize you need to take a new look at your class hierarchy with an eye to collapsing it into a sensible structure. Remember that underneath it all, inheritance is meant to express a relationship that says, "This new class is a *type of* that old class." Your program should not be concerned with pushing bits around, but instead with creating and manipulating objects of various types to express a model in the terms given you by the problem's space.

Upcasting

Earlier in the chapter, you saw how an object of a class derived from **ofstream** has all the characteristics and behaviors of an **ofstream** object. In **FName2.cpp**, any **ofstream** member function could be called for an **FName2** object.

The most important aspect of inheritance is not that it provides member functions for the new class, however. It's the relationship expressed between the new class and the base class. This relationship can be summarized by saying, "The new class *is a type of* the existing class."

This description is not just a fanciful way of explaining inheritance – it's supported directly by the compiler. As an example, consider a base class called **Instrument** that represents musical instruments and a derived class called **Wind**. Because inheritance means that all the functions in the

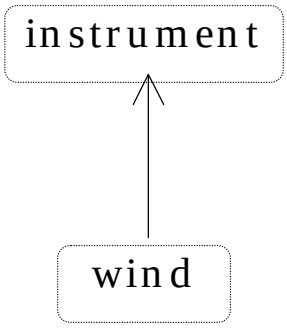
base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. So if the **Instrument** class has a **play( )** member function, so will **Wind** instruments. This means we can accurately say that a **Wind** object is also a type of **Instrument**. The following example shows how the compiler supports this notion:

```
//: C14:Wind.cpp
// Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

What's interesting in this example is the **tune( )** function, which accepts an **Instrument** reference. However, in **main( )** the **tune( )** function is called by giving it a **Wind** object. Given that C++ is very particular about type checking, it seems strange that a function that accepts one type will readily accept another type, until you realize that a **Wind** object is also an **Instrument** object, and there's no function that **tune( )** could call for an **Instrument** that isn't also in **Wind**. Inside **tune( )**, the code works for **Instrument** and anything derived from **Instrument**, and the act of converting a **Wind** object, reference, or pointer into an **Instrument** object, reference, or pointer is called *upcasting*.

Why “upcasting”?

The reason for the term is historical and is based on the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for **Wind.cpp** is then:



Casting from derived to base moves *up* on the inheritance diagram, so it’s commonly referred to as upcasting. Upcasting is always safe because you’re going from a more specific type to a more general type – the only thing that can occur to the class interface is that it can lose member functions, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

Downcasting

You can also perform the reverse of upcasting, called *downcasting*, but this involves a dilemma that is the subject of Chapter XX.

Upcasting and the copy-constructor (not indexed)

If you allow the compiler to synthesize a copy-constructor for a derived class, it will automatically call the base-class copy-constructor, and then the copy-constructors for all the member objects (or perform a bitcopy on built-in types) so you’ll get the right behavior:

```
//: C14:Ccright.cpp
// Correctly synthesizing the CC
#include <iostream>
using namespace std;
```

```
class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(Parent&)\n";
  }
  Parent() :i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(Member&)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& d){
    return os << (Parent&)d << d.m
```

```
            << "Child: " << d.i << endl;
  }
};

int main() {
  Child d(2);
  cout << "calling copy-constructor: " << endl;
  Child d2 = d; // Calls copy-constructor
  cout << "values in d2:\n" << d2;
} ///:~
```

The **operator<<** for **Child** is interesting because of the way that it calls the **operator<<** for the **Parent** part within it: by casting the **Child** object to a **Parent&** (if you cast to a **Parent** object instead of a reference you'll end up creating a temporary):

```
return os << (Parent&)d << d.m
```

Since the compiler then sees it as a **Parent**, it calls the **Parent** version of **operator<<**.

You can see that **Child** has no explicitly-defined copy-constructor. The compiler then synthesizes the copy-constructor (since that is one of the four functions it will synthesize, along with the default constructor – if you don't create any constructors – the **operator=** and the destructor) by calling the **Parent** copy-constructor and the **Member** copy-constructor. This is shown in the output

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent(Parent&)
Member(Member&)
values in d2:
Parent: 2
Member: 2
Child: 2
```

However, if you try to write your own copy-constructor for **Child** and you make an innocent mistake and do it badly:

```
Child(const Child& d) : i(d.i), m(d.m) {}
```

The *default* constructor will be automatically called, since that's what the compiler falls back on when it has no other choice of constructor to call (remember that *some* constructor must always be called for every object,

regardless of whether it's a subobject of another class). The output will then be:

```
Parent(int ii)
Member(int ii)
Child(int ii)
calling copy-constructor:
Parent()
Member(Member&)
values in d2:
Parent: 0
Member: 2
Child: 2
```

This is probably not what you expect, since generally you'll want the base-class portion to be copied from the existing object to the new object as part of copy-construction.

To repair the problem you must remember to properly call the base-class copy-constructor (as the compiler does) whenever you write your own copy-constructor. This can seem a little strange-looking at first but it's another example of upcasting:

```
Child(const Child& d)
  : Parent(d), i(d.i), m(d.m) {
  cout << "Child(Child&)\n";
}
```

The strange part is where the **Parent** copy-constructor is called: **Parent(d)**. What does it mean to pass a **Child** object to a **Parent** constructor? Here's the trick: **Child** is inherited from **Parent**, so a **Child** reference *is* a **Parent** reference. So the base-class copy-constructor upcasts a reference to **Child** to a reference to **Parent** and uses it to perform the copy-construction. When you write your own copy constructors you'll generally want to do this.

Composition vs. inheritance (revisited)

One of the clearest ways to determine whether you should be using composition or inheritance is by asking whether you'll ever need to upcast from your new class. Earlier in this chapter, the **Stack** class was specialized using inheritance. However, chances are the **Stringlist** objects

will be used only as **String** containers, and never upcast, so a more appropriate alternative is composition:

```
//: C14:InheritStack2.cpp
//{L} ../C13/Stack4
// Composition vs. inheritance
#include "../C13/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringList {
  Stack stack; // Embed instead of inherit
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream file("InheritStack2.cpp");
  assure(file, "InheritStack2.cpp");
  string line;
  StringList textlines;
  while(getline(file,line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
} ///:~
```

The file is identical to **Inhstack.cpp**, except that a **Stack** object is embedded in **Stringlist**, and member functions are called for the embedded object. There's still no time or space overhead because the

subobject takes up the same amount of space, and all the additional type checking happens at compile time.

You can also use **private** inheritance to express "implemented in terms of." The method you use to create the **Stringlist** class is not critical in this situation – they all solve the problem adequately. One place it becomes important, however, is when multiple inheritance might be warranted. In that case, if you can detect a class where composition can be used instead of inheritance, you may be able to eliminate the need for multiple inheritance.

Pointer & reference upcasting

In **Wind.cpp**, the upcasting occurs during the function call – a **Wind** object outside the function has its reference taken and becomes an **Instrument** reference inside the function. Upcasting can also occur during a simple assignment to a pointer or reference:

```
Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast
```

Like the function call, neither of these cases require an explicit cast.

A crisis

Of course, any upcast loses type information about an object. If you say

```
Wind w;
Instrument* ip = &w;
```

the compiler can deal with **ip** only as an **Instrument** pointer and nothing else. That is, it cannot know that **ip** *actually* happens to point to a **Wind** object. So when you call the **play( )** member function by saying

```
ip->play(middleC);
```

the compiler can know only that it's calling **play( )** for an **Instrument** pointer, and call the base-class version of **Instrument::play( )** instead of what it should do, which is call **Wind::play( )**. Thus you won't get the correct behavior.

This is a significant problem; it is solved in the next chapter by introducing the third cornerstone of object-oriented programming: polymorphism (implemented in C++ with **virtual** functions).

Summary

Both inheritance and composition allow you to create a new type from existing types, and both embed subobjects of the existing types inside the new type. Typically, however, you use composition to reuse existing types as part of the underlying implementation of the new type and inheritance when you want to reuse the interface as well as the implementation. If the derived class has the base-class interface, it can be *upcast* to the base, which is critical for polymorphism as you'll see in the next chapter.

Although code reuse through composition and inheritance is very helpful for rapid project development, you'll generally want to redesign your class hierarchy before allowing other programmers to become dependent on it. Your goal is a hierarchy where each class has a specific use and is neither too big (encompassing so much functionality that it's unwieldy to reuse) nor annoyingly small (you can't use it by itself or without adding functionality). Your finished classes should themselves be easily reused.

Exercises

1. Modify **Car.cpp** so it also inherits from a class called **vehicle**, placing appropriate member functions in **vehicle** (that is, make up some member functions). Add a nondefault constructor to **vehicle**, which you must call, inside **car**'s constructor.
2. Create two classes, **A** and **B,** with default constructors that announce themselves. Inherit a new class called **C** from **A**, and create a member object of **B** in **C**, but do not create a constructor for **C**. Create an object of class **C** and observe the results.
3. Use inheritance to specialize the **PStash** class in Chapter XX (**Pstash.h** & **Pstash.cpp**) so it accepts and returns **String** pointers. Also modify **PStashTest.cpp** and test it. Change the class so **PStash** is a member object.
4. Use **private** and **protected** inheritance to create two new classes from a base class. Then attempt to upcast objects of the derived class to the base class. Explain what happens.
5. Take the example **ccright.cpp** in this chapter and modify it by adding your own copy-constructor *without* calling the base-class copy-constructor and see what happens. Fix the

problem by making a proper explicit call to the base-class copy constructor in the constructor-initializer list of the **Child** copy-constructor.

15: Polymorphism & virtual functions

Polymorphism (implemented in C++ with **virtual** functions) is the third essential feature of an object-oriented programming language, after data abstraction and inheritance.

It provides another dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs that can be "grown" not only during the original creation of the project, but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details **private**. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But virtual functions deal with decoupling in terms of *types*. In the last chapter, you saw how inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The virtual function allows one type to

express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the functions you can call through the base class.

In this chapter, you'll learn about virtual functions starting from the very basics, with simple examples that strip away everything but the "virtualness" of the program.

Evolution of C++ programmers

C programmers seem to acquire C++ in three steps. First, as simply a "better C," because C++ forces you to declare all functions before using them and is much pickier about how variables are used. You can often find the errors in a C program simply by compiling it with a C++ compiler.

The second step is "object-based" C++. This means that you easily see the code organization benefits of grouping a data structure together with the functions that act upon it, the value of constructors and destructors, and perhaps some simple inheritance. Most programmers who have been working with C for a while quickly see the usefulness of this because, whenever they create a library, this is exactly what they try to do. With C++, you have the aid of the compiler.

You can get stuck at the object-based level because it's very easy to get to and you get a lot of benefit without much mental effort. It's also easy to feel like you're creating data types – you make classes, and objects, and you send messages to those objects, and everything is nice and neat.

But don't be fooled. If you stop here, you're missing out on the greatest part of the language, which is the jump to true object-oriented programming. You can do this only with virtual functions.

Virtual functions enhance the concept of type rather than just encapsulating code inside structures and behind walls, so they are without a doubt the most difficult concept for the new C++ programmer to fathom. However, they're also the turning point in the understanding of object-oriented programming. If you don't use virtual functions, you don't understand OOP yet.

Because the virtual function is intimately bound with the concept of type, and type is at the core of object-oriented programming, there is no analog to the virtual function in a traditional procedural language. As a procedural programmer, you have no referent with which to think about virtual functions, as you do with almost every other feature in the language.

Features in a procedural language can be understood on an algorithmic level, but virtual functions can be understood only from a design viewpoint.

Upcasting

In the last chapter you saw how an object can be used as its own type or as an object of its base type. In addition, it can be manipulated through an address of the base type. Taking the address of an object (either a pointer or a reference) and treating it as the address of the base type is called *upcasting* because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, which is embodied in the following code:

```
//: C15:Wind2.cpp
// Inheritance & upcasting
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}
```

```
int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

The function **tune( )** accepts (by reference) an **Instrument**, but also without complaint anything derived from **Instrument**. In **main( )**, you can see this happening as a **Wind** object is passed to **tune( )**, with no cast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is publicly inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may "narrow" that interface, but it cannot make it any less than the full interface to **Instrument**.

The same arguments are true when dealing with pointers; the only difference is that the user must explicitly take the addresses of objects as they are passed into the function.

The problem

The problem with **Wind2.cpp** can be seen by running the program. The output is **Instrument::play**. This is clearly not the desired output, because you happen to know that the object is actually a **Wind** and not just an **Instrument**. The call should resolve to **Wind::play**. For that matter, any object of a class derived from **Instrument** should have its version of **play** used, regardless of the situation.

However, the behavior of **Wind2.cpp** is not surprising, given C's approach to functions. To understand the issues, you need to be aware of the concept of *binding*.

Function call binding

Connecting a function call to a function body is called *binding*. When binding is performed before the program is run (by the compiler and linker), it's called *early binding*. You may not have heard the term before because it's never been an option with procedural languages: C compilers have only one kind of function call, and that's early binding.

The problem in the above program is caused by early binding because the compiler cannot know the correct function to call when it has only an **Instrument** address.

The solution is called *late binding*, which means the binding occurs at runtime, based on the type of the object. Late binding is also called *dynamic binding* or *runtime binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at runtime and call the appropriate member function. That is, the compiler still doesn't know the actual object type, but it inserts code that finds out and calls the correct function body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects themselves. You'll see how this works later.

virtual functions

To cause late binding to occur for a particular function, C++ requires that you use the **virtual** keyword when declaring the function in the base class. Late binding occurs only with **virtual** functions, and only when you're using an address of the base class where those **virtual** functions exist, although they may also be defined in an earlier base class.

To create a member function as **virtual**, you simply precede the declaration of the function with the keyword **virtual**. You don't repeat it for the function definition, and you don't *need* to repeat it in any of the derived-class function redefinitions (though it does no harm to do so). If a function is declared as **virtual** in the base class, it is **virtual** in all the derived classes. The redefinition of a **virtual** function in a derived class is often called *overriding.*

To get the desired behavior from **Wind2.cpp**, simply add the **virtual** keyword in the base class before **play( )**:

```
//: C15:Wind3.cpp
// Late binding with virtual
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

This file is identical to **Wind2.cpp** except for the addition of the **virtual** keyword, and yet the behavior is significantly different: Now the output is **Wind::play**.

Extensibility

With **play( )** defined as **virtual** in the base class, you can add as many new types as you want to the system without changing the **tune( )** function. In a well-designed OOP program, most or all of your functions will follow the model of **tune( )** and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The functions that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Here's the instrument example with more virtual functions and a number of new classes, all of which work correctly with the old, unchanged **tune( )** function:

```
//: C15:Wind4.cpp
// Extensibility in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.
```

```
class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
```

```
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
```

```
  f(flugelhorn);
} ///:~
```

You can see that another inheritance level has been added beneath **Wind**, but the **virtual** mechanism works correctly no matter how many levels there are. The **adjust( )** function is *not* redefined for **Brass** and **Woodwind**. When this happens, the previous definition is automatically used – the compiler guarantees there's always *some* definition for a virtual function, so you'll never end up with a call that doesn't bind to a function body. (This would spell disaster.)

The array **A[ ]** contains pointers to the base class **Instrument**, so upcasting occurs during the process of array initialization. This array and the function **f( )** will be used in later discussions.

In the call to **tune( )**, upcasting is performed on each different type of object, yet the desired behavior always takes place. This can be described as "sending a message to an object and letting the object worry about what to do with it." The **virtual** function is the lens to use when you're trying to analyze a project: Where should the base classes occur, and how might you want to extend the program? However, even if you don't discover the proper base class interfaces and virtual functions at the initial creation of the program, you'll often discover them later, even much later, when you set out to extend or otherwise maintain the program. This is not an analysis or design error; it simply means you didn't have all the information the first time. Because of the tight class modularization in C++, it isn't a large problem when this occurs because changes you make in one part of a system tend not to propagate to other parts of the system as they do in C.

How C++ implements late binding

How can late binding happen? All the work goes on behind the scenes by the compiler, which installs the necessary late-binding mechanism when you ask it to (you ask by creating virtual functions). Because programmers often benefit from understanding the mechanism of virtual functions in C++, this section will elaborate on the way the compiler implements this mechanism.

The keyword **virtual** tells the compiler it should not perform early binding. Instead, it should automatically install all the mechanisms necessary to

perform late binding. This means that if you call **play( )** for a **Brass** object *through an address for the base-class* **Instrument**, you'll get the proper function.

To accomplish this, the compiler creates a single table (called the VTABLE) for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. In each class with virtual functions, it secretly places a pointer, called the *vpointer* (abbreviated as VPTR), which points to the VTABLE for that object. When you make a virtual function call through a base-class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the right function and causing late binding to take place.

All of this – setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call – happens automatically, so you don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.

The following sections go into this process in more detail.

Storing type information

You can see that there is no explicit type information stored in any of the classes. But the previous examples, and simple logic, tell you that there must be some sort of type information stored in the objects; otherwise the type could not be established at runtime. This is true, but the type information is hidden. To see it, here's an example to examine the sizes of classes that use virtual functions compared with those that don't:

```
//: C15:Sizes.cpp
// Object sizes vs. virtual funcs
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
```

```
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

With no virtual functions, the size of the object is exactly what you'd expect: the size of a single **int**. With a single virtual function in **OneVirtual**, the size of the object is the size of **NoVirtual** plus the size of a **void** pointer. It turns out that the compiler inserts a single pointer (the VPTR) into the structure if you have one *or more* virtual functions. There is no size difference between **OneVirtual** and **TwoVirtuals**. That's because the VPTR points to a table of function addresses. You need only one because all the virtual function addresses are contained in that single table.

This example required at least one data member. If there had been no data members, the C++ compiler would have forced the objects to be a nonzero size because each object must have a distinct address. If you imagine indexing into an array of zero-sized objects, you'll understand. A "dummy" member is inserted into objects that would otherwise be zero-sized. When the type information is inserted because of the **virtual** keyword, this takes the place of the "dummy" member. Try commenting out the **int a** in all the classes in the above example to see this.

Picturing virtual functions

To understand exactly what's going on when you use a virtual function, it's helpful to visualize the activities going on behind the curtain. Here's a drawing of the array of pointers **A[ ]** in **Wind4.cpp**:

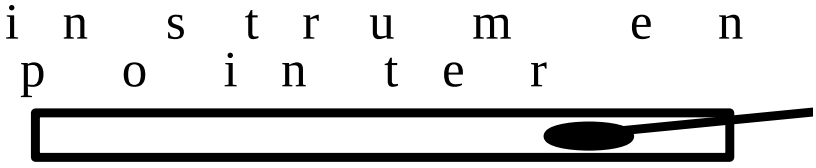
The array of **Instrument** pointers has no specific type information; they each point to an object of type **Instrument**. **Wind**, **Percussion**, **Stringed**, and **Brass** all fit into this category because they are derived from **Instrument** (and thus have the same interface as **Instrument**, and can respond to the same messages), so their addresses can also be placed into the array. However, the compiler doesn't know they are anything more than **Instrument** objects, so left to its own devices, it would normally call the base-class versions of all the functions. But in this case, all those functions have been declared with the **virtual** keyword, so something different happens.

Each time you create a class that contains virtual functions, or you derive from a class that contains virtual functions, the compiler creates a VTABLE for that class, seen on the right of the diagram. In that table it places the addresses of all the functions that are declared virtual in this class or in the base class. If you don't redefine a function that was declared virtual in the base class, the compiler uses the address of the base-class version in the derived class. (You can see this in the **adjust** entry in the **Brass** VTABLE.) Then it places the VPTR (discovered in **Sizes.cpp**) into the class. There is only one VPTR for each object when using simple inheritance like this. The VPTR must be initialized to point to the starting

address of the appropriate VTABLE. (This happens in the constructor, which you'll see later in more detail.)

Once the VPTR is initialized to the proper VTABLE, the object in effect "knows" what type it is. But this self-knowledge is worthless unless it is used at the point a virtual function is called.

When you call a virtual function through a base class address (the situation when the compiler doesn't have all the information necessary to perform early binding), something special happens. Instead of performing a typical function call, which is simply an assembly-language **CALL** to a particular address, the compiler generates different code to perform the function call. Here's what a call to **adjust( )** for a **Brass** object it looks like, if made through an **Instrument** pointer. An **Instrument** reference produces the same result:



The compiler starts with the **Instrument** pointer, which points to the starting address of the object. All **Instrument** objects or objects derived from **Instrument** have their VPTR in the same place (often at the beginning of the object), so the compiler can pick the VPTR out of the object. The VPTR points to the starting address of the VTABLE. All the VTABLEs are laid out in the same order, regardless of the specific type of the object. **play( )** is first, **what( )** is second, and **adjust( )** is third. The compiler knows that regardless of the specific object type, the **adjust( )** function is at the location VPTR+2. Thus instead of saying, "Call the function at the absolute location **Instrument::adjust**" (early binding; the wrong action), it generates code that says, in effect, "Call the function at VPTR+2." Because the fetching of the VPTR and the determination of the actual function address occur at runtime, you get the desired late binding. You send a message to the object, and the object figures out what to do with it.

Under the hood

It can be helpful to see the assembly-language code generated by a virtual function call, so you can see that late-binding is indeed taking place. Here's the output from one compiler for the call

```
i.adjust(1);
```

inside the function **f(Instrument& i)**:

```
push  1
push  si
mov  bx,word ptr [si]
call  word ptr [bx+4]
add  sp,4
```

The arguments of a C++ function call, like a C function call, are pushed on the stack from right to left (this order is required to support C's variable argument lists), so the argument **1** is pushed on the stack first. At this point in the function, the register **si** (part of the Intel X86 processor architecture) contains the address of **i**. This is also pushed on the stack because it is the starting address of the object of interest. Remember that the starting address corresponds to the value of **this**, and **this** is quietly pushed on the stack as an argument before every member function call, so the member function knows which particular object it is working on. Thus you'll always see the number of arguments plus one pushed on the stack before a member function call (except for **static** member functions, which have no **this**).

Now the actual virtual function call must be performed. First, the VPTR must be produced, so the VTABLE can be found. For this compiler the VPTR is inserted at the beginning of the object, so the contents of **this** correspond to the VPTR. The line

```
mov bx,word ptr [si]
```

fetches the word that **si** (that is, **this**) points to, which is the VPTR. It places the VPTR into the register **bx**.

The VPTR contained in **bx** points to the starting address of the VTABLE, but the function pointer to call isn't at the zeroth location of the VTABLE, but instead the second location (because it's the third function in the list). For this memory model each function pointer is two bytes long, so the compiler adds four to the VPTR to calculate where the address of the proper function is. Note that this is a constant value, established at compile time, so the only thing that matters is that the function pointer at

location number two is the one for **adjust( )**. Fortunately, the compiler takes care of all the bookkeeping for you and ensures that all the function pointers in all the VTABLEs occur in the same order.

Once the address of the proper function pointer in the VTABLE is calculated, that function is called. So the address is fetched and called all at once in the statement

```
call word ptr [bx+4]
```

Finally, the stack pointer is moved back up to clean off the arguments that were pushed before the call. In C and C++ assembly code you'll often see the caller clean off the arguments but this may vary depending on processors and compiler implementations.

Installing the vpointer

Because the VPTR determines the virtual function behavior of the object, you can see how it's critical that the VPTR always be pointing to the proper VTABLE. You don't ever want to be able to make a call to a virtual function before the VPTR is properly initialized. Of course, the place where initialization can be guaranteed is in the constructor, but none of the WIND examples has a constructor.

This is where creation of the default constructor is essential. In the WIND examples, the compiler creates a default constructor that does nothing except initialize the VPTR. This constructor, of course, is automatically called for all **Instrument** objects before you can do anything with them, so you know that it's always safe to call virtual functions.

The implications of the automatic initialization of the VPTR inside the constructor are discussed in a later section.

Objects are different

It's important to realize that upcasting deals only with addresses. If the compiler has an object, it knows the exact type and therefore (in C++) will not use late binding for any function calls – or at least, the compiler doesn't *need* to use late binding. For efficiency's sake, most compilers will perform early binding when they are making a call to a virtual function for an object because they know the exact type. Here's an example:

```
//: C15:Early.cpp
// Early binding & virtuals
#include <iostream>
```

```
using namespace std;

class Base {
public:
  virtual int f() const { return 1; }
};

class Derived : public Base {
public:
  int f() const { return 2; }
};

int main() {
  Derived d;
  Base* b1 = &d;
  Base& b2 = d;
  Base b3;
  // Late binding for both:
  cout << "b1->f() = " << b1->f() << endl;
  cout << "b2.f() = " << b2.f() << endl;
  // Early binding (probably):
  cout << "b3.f() = " << b3.f() << endl;
} ///:~
```

In **b1->f( )** and **b2.f( )** addresses are used, which means the information is incomplete: **b1** and **b2** can represent the address of a **Base** *or* something derived from **Base**, so the virtual mechanism must be used. When calling **b3.f( )** there's no ambiguity. The compiler knows the exact type and that it's an object, so it can't possibly be an object derived from **Base** – it's *exactly* a **Base**. Thus early binding is probably used. However, if the compiler doesn't want to work so hard, it can still use late binding and the same behavior will occur.

Why **virtual** functions?

At this point you may have a question: "If this technique is so important, and if it makes the 'right' function call all the time, why is it an option? Why do I even need to know about it?"

This is a good question, and the answer is part of the fundamental philosophy of C++: "Because it's not quite as efficient." You can see from the previous assembly-language output that instead of one simple CALL to

an absolute address, there are two more sophisticated assembly instructions required to set up the virtual function call. This requires both code space and execution time.

Some object-oriented languages have taken the approach that late binding is so intrinsic to object-oriented programming that it should always take place, that it should not be an option, and the user shouldn't have to know about it. This is a design decision when creating a language, and that particular path is appropriate for many languages.[40] However, C++ comes from the C heritage, where efficiency is critical. After all, C was created to replace assembly language for the implementation of an operating system (thereby rendering that operating system – Unix – far more portable than its predecessors). One of the main reasons for the invention of C++ was to make C programmers more efficient.[41] And the first question asked when C programmers encounter C++ is "What kind of size and speed impact will I get?" If the answer were, "Everything's great except for function calls when you'll always have a little extra overhead," many people would stick with C rather than make the change to C++. In addition, inline functions would not be possible, because virtual functions must have an address to put into the VTABLE. So the virtual function is an option, *and* the language defaults to nonvirtual, which is the fastest configuration. Stroustrup stated that his guideline was "If you don't use it, you don't pay for it."

Thus the **virtual** keyword is provided for efficiency tuning. When designing your classes, however, you shouldn't be worrying about efficiency tuning. If you're going to use polymorphism, use virtual functions everywhere. You only need to look for functions to make non-virtual when looking for ways to speed up your code (and there are usually much bigger gains to be had in other areas).

Anecdotal evidence suggests that the size and speed impacts of going to C++ are within 10% of the size and speed of C, and often much closer to the same. The reason you might get better size and speed efficiency is because you may design a C++ program in a smaller, faster way than you would using C.

---

[40] Smalltalk, for instance, uses this approach with great success.

[41] At Bell labs, where C++ was invented, there are a *lot* of C programmers. Making them all more efficient, even just a bit, saves the company many millions.

Abstract base classes and pure **virtual** functions

Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class *abstract* by giving it at least one *pure virtual function*. You can recognize a pure virtual function because it uses the **virtual** keyword and is followed by **= 0**. If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

When an abstract class is inherited, all pure virtual functions must be implemented, or the inherited class becomes abstract as well. Creating a pure virtual function allows you to put a member function in an interface without being forced to provide a possibly meaningless body of code for that member function, and at the same time forcing inherited classes to provide a definition for it.

In all the instrument examples, the functions in the base class **Instrument** were always "dummy" functions. If these functions are ever called, they indicate you've done something wrong. That's because the intent of **Instrument** is to create a common interface for all the classes derived from it.

[[corrected diagram here]]

The only reason to establish the common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what's in common with all the derived classes. Nothing else. Another way of saying this is to call **Instrument** an *abstract base class* (or simply an *abstract class*). You create an abstract class when you want to manipulate a set of classes through this common interface.

Notice you are only required to declare a function as **virtual** in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. You *can* use the **virtual** keyword in the derived-class declarations (and some people do, for clarity), but it is redundant.

If you have a genuine abstract class (like **Instrument**), objects of that class almost always have no meaning. That is, **Instrument** is meant to express only the interface, and not a particular implementation, so creating an **Instrument** object makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the virtual functions in **Instrument** print error messages, but this delays the information until runtime and requires reliable exhaustive testing on the part of the user. It is much better to catch the problem at compile time.

C++ provides a mechanism for doing this called the *pure virtual function*. Here is the syntax used for a declaration:

```
virtual void X() = 0;
```

By doing this, you tell the compiler to reserve a slot for a function in the VTABLE, but not to put an address in that particular slot. If only one

function in a class is declared as pure virtual, the VTABLE is incomplete. A class containing pure virtual functions is called a *pure abstract base class*.

If the VTABLE for a class is incomplete, what is the compiler supposed to do when someone tries to make an object of that class? It cannot safely create an object of a pure abstract class, so you get an error message from the compiler if you try to make an object of a pure abstract class. Thus, the compiler ensures the purity of the abstract class, and you don't have to worry about misusing it.

Here's **Wind4.cpp** modified to use pure virtual functions:

```
//: C15:Wind5.cpp
// Pure abstract base classes
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
```

```
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
```

```
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Pure virtual functions are very helpful because they make explicit the abstractness of a class and tell both the user and the compiler how it was intended to be used.

Note that pure virtual functions prevent a function call with the pure abstract class being passed in by value. Thus it is also a way to prevent object slicing from accidentally upcasting by value. This way you can ensure that a pointer or reference is always used during upcasting.

Because one pure virtual function prevents the VTABLE from being generated doesn't mean you don't want function bodies for some of the others. Often you will want to call a base-class version of a function, even if it is virtual. It's always a good idea to put common code as close as possible to the root of your hierarchy. Not only does this save code space, it allows easy propagation of changes.

Pure **virtual** definitions

It's possible to provide a definition for a pure virtual function in the base class. You're still telling the compiler not to allow objects of that pure abstract base class, and the pure virtual functions must be defined in derived classes in order to create objects. However, there may be a piece of code you want some or all the derived class definitions to use in common, and you don't want to duplicate that code in every function. Here's what it looks like:

```
//: C15:Pvdef.cpp
// Pure virtual base definition
#include <iostream>
using namespace std;

class Base {
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // Inline pure virtual definitions illegal:
```

```
  //!  virtual void g() const = 0 {}
};

// OK, not defined inline
void Base::f() const {
  cout << "Base::f()\n";
}

void Base::v() const { cout << "Base::v()\n";}

class D : public Base {
public:
  // Use the common Base code:
  void v() const { Base::v(); }
  void f() const { Base::f(); }
};

int main() {
  D d;
  d.v();
  d.f();
} ///:~
```

The slot in the **Base** VTABLE is still empty, but there happens to be a function by that name you can call in the derived class.

The other benefit to this feature is that it allows you to change to a pure virtual without disturbing the existing code. (This is a way for you to locate classes that don't redefine that virtual function).

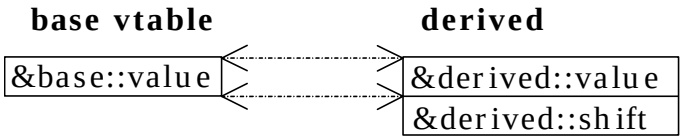
Inheritance and the **VTABLE**

You can imagine what happens when you perform inheritance and redefine some of the virtual functions. The compiler creates a new VTABLE for your new class, and it inserts your new function addresses, using the base-class function addresses for any virtual functions you don't redefine. One way or another, there's always a full set of function addresses in the VTABLE, so you'll never be able to make a call to an address that isn't there (which would be disastrous).

But what happens when you inherit and add new virtual functions in the *derived* class? Here's a simple example:

```
//: C15:Addv.cpp
// Adding virtuals in derivation
#include <iostream>
using namespace std;

class Base {
  int i;
public:
  Base(int ii) : i(ii) {}
  virtual int value() const { return i; }
};

class Derived : public Base {
public:
  Derived(int ii) : Base(ii) {}
  int value() const {
    return Base::value() * 2;
  }
  // New virtual function in the Derived class:
  virtual int shift(int x) const {
    return Base::value() << x;
  }
};

int main() {
  Base* B[] = { new Base(7), new Derived(7) };
  cout << "B[0]->value() = "
       << B[0]->value() << endl;
  cout << "B[1]->value() = "
       << B[1]->value() << endl;
//! cout << "B[1]->shift(3) = "
//!      << B[1]->shift(3) << endl; // Illegal
} ///:~
```

The class **Base** contains a single virtual function **value( )**, and **Derived** adds a second one called **shift( )**, as well as redefining the meaning of **value( )**. A diagram will help visualize what's happening. Here are the VTABLEs created by the compiler for **Base** and **Derived**:



Notice the compiler maps the location of the **value** address into exactly the same spot in the **Derived** VTABLE as it is in the **Base** VTABLE. Similarly, if a class is inherited from **Derived**, its version of **shift** would be placed in its VTABLE in exactly the same spot as it is in **Derived**. This is because (as you saw with the assembly-language example) the compiler generates code that uses a simple numerical offset into the VTABLE to select the virtual function. Regardless of what specific subtype the object belongs to, its VTABLE is laid out the same way, so calls to the virtual functions will always be made the same way.

In this case, however, the compiler is working only with a pointer to a base-class object. The base class has only the **value( )** function, so that is the only function the compiler will allow you to call. How could it possibly know that you are working with a **Derived** object, if it has only a pointer to a base-class object? That pointer might point to some other type, which doesn't have a **shift** function. It may or may not have some other function address at that point in the VTABLE, but in either case, making a virtual call to that VTABLE address is not what you want to do. So it's fortunate and logical that the compiler protects you from making virtual calls to functions that exist only in derived classes.

There are some less-common cases where you may know that the pointer actually points to an object of a specific subclass. If you want to call a function that only exists in that subclass, then you must cast the pointer. You can remove the error message produced by the previous program like this:

```
((Derived*)B[1])->shift(3)
```

Here, you happen to know that **B[1]** points to a **Derived** object, but generally you don't know that. If your problem is set up so that you must know the exact types of all objects, you should rethink it, because you're probably not using virtual functions properly. However, there are some situations where the design works best (or you have no choice) if you know the exact type of all objects kept in a generic container. This is the problem of *run-time type identification* (RTTI)**.**

Run-time type identification is all about casting base-class pointers *down* to derived-class pointers ("up" and "down" are relative to a typical class diagram, with the base class at the top). Casting *up* happens

automatically, with no coercion, because it's completely safe. Casting *down* is unsafe because there's no compile time information about the actual types, so you must know exactly what type the object really is. If you cast it into the wrong type, you'll be in trouble.

Chapter XX describes the way C++ provides run-time type information.

Object slicing

There is a distinct difference between passing addresses and passing values when treating objects polymorphically. All the examples you've seen here, and virtually all the examples you should see, pass addresses and not values. This is because addresses all have the same size,[42] so passing the address of an object of a derived type (which is usually bigger) is the same as passing the address of an object of the base type (which is usually smaller). As explained before, this is the goal when using polymorphism – code that manipulates objects of a base type can transparently manipulate derived-type objects as well.

If you use an object instead of a pointer or reference as the recipient of your upcast, something will happen that may surprise you: the object is "sliced" until all that remains is the subobject that corresponds to your destination. In the following example you can see what's left after slicing by examining the size of the objects:

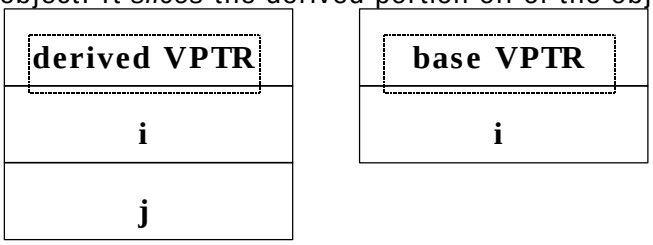
```
//: C15:Slice.cpp
// Object slicing
#include <iostream>
using namespace std;

class Base {
  int i;
public:
  Base(int ii = 0) : i(ii) {}
  virtual int sum() const { return i; }
};

class Derived : public Base {
  int j;
public:
```

[42] Actually, not all pointers are the same size on all machines. In the context of this discussion, however, they can be considered to be the same.

```
  Derived(int ii = 0, int jj = 0)
    : Base(ii), j(jj) {}
  int sum() const { return Base::sum() + j; }
};

void call(Base b) {
  cout << "sum = " << b.sum() << endl;
}

int main() {
  Base b(10);
  Derived d(10, 47);
  call(b);
  call(d);
} ///:~
```

The function **call( )** is passed an object of type **Base** *by value*. It then calls the virtual function **sum( )** for the **Base** object. In **main( )**, you might expect the first call to produce 10, and the second to produce 57. In fact, both calls produce 10.

Two things are happening in this program. First, **call( )** accepts only a **Base** object, so all the code inside the function body will manipulate only members associated with **Base**. Any calls to **call( )** will cause an object the size of **Base** to be pushed on the stack and cleaned up after the call. This means that if an object of a class inherited from **Base** is passed to **call( )**, the compiler accepts it, but it copies only the **Base** portion of the object. It *slices* the derived portion off of the object, like this:



before slice **after slice**

Now you may wonder about the virtual function call. Here, the virtual function makes use of portions of both **Base** (which still exists) and **Derived**, which no longer exists because it was sliced off! So what happens when the virtual function is called?

You're saved from disaster precisely because the object is being passed by value. Because of this, the compiler thinks it knows the precise type of the object (and it does, here, because any information that contributed extra features to the objects has been lost). In addition, when passing by value, it uses the copy-constructor for a **Base** object, which initializes the VPTR to the **Base** VTABLE and copies only the **Base** parts of the object. There's no explicit copy-constructor here, so the compiler synthesizes one. Under all interpretations, the object truly becomes a **Base** during slicing.

Object slicing actually removes part of the object rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not often done; in fact, it's usually something to watch out for and prevent. You can explicitly prevent object slicing by putting pure virtual functions in the base class; this will cause a compile-time error message for an object slice.

virtual functions & constructors

When an object containing virtual functions is created, its VPTR must be initialized to point to the proper VTABLE. This must be done before there's any possibility of calling a virtual function. As you might guess, because the constructor has the job of bringing an object into existence, it is also the constructor's job to set up the VPTR. The compiler secretly inserts code into the beginning of the constructor that initializes the VPTR. In fact, even if you don't explicitly create a constructor for a class, the compiler will create one for you with the proper VPTR initialization code (if you have virtual functions). This has several implications.

The first concerns efficiency. The reason for **inline** functions is to reduce the calling overhead for small functions. If C++ didn't provide **inline** functions, the preprocessor might be used to create these "macros." However, the preprocessor has no concept of access or classes, and therefore couldn't be used to create member function macros. In addition, with constructors that must have hidden code inserted by the compiler, a preprocessor macro wouldn't work at all.

You must be aware when hunting for efficiency holes that the compiler is inserting hidden code into your constructor function. Not only must it initialize the VPTR, it must also check the value of **this** (in case the **operator new** returns zero) and call base-class constructors. Taken

together, this code can impact what you thought was a tiny inline function call. In particular, the size of the constructor can overwhelm the savings you get from reduced function-call overhead. If you make a lot of inline constructor calls, your code size can grow without any benefits in speed.

Of course, you probably won't make all tiny constructors non-inline right away, because they're much easier to write as inlines. But when you're tuning your code, remember to remove inline constructors.

Order of constructor calls

The second interesting facet of constructors and virtual functions concerns the order of constructor calls and the way virtual calls are made within constructors.

All base-class constructors are always called in the constructor for an inherited class. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access only to its own members, and not those of the base class; only the base-class constructor can properly initialize its own elements. Therefore it's essential that all constructors get called; otherwise the entire object wouldn't be constructed properly. That's why the compiler enforces a constructor call for every portion of a derived class. It will call the default constructor if you don't explicitly call a base-class constructor in the constructor initializer list. If there is no default constructor, the compiler will complain. (In this example, **class X** has no constructors so the compiler can automatically make a default constructor.)

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class. This means you must be able to assume that all the members of the base class are valid when you're in the derived class. In a normal member function, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you're in the derived-class constructor, all the members you can access in the base class have been initialized. "Knowing all members are valid" inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) in the constructor initializer list. If you follow this practice, you can assume that all base class members *and* member objects of the current object have been initialized.

Behavior of virtual functions inside constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a virtual function? Inside an ordinary member function you can imagine what will happen – the virtual call is resolved at runtime because the object cannot know whether it belongs to the class the member function is in, or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not the case. If you call a virtual function inside a constructor, only the local version of the function is used. That is, the virtual mechanism doesn't work within the constructor.

This behavior makes sense for two reasons. Conceptually, the constructor's job is to bring the object into existence (which is hardly an ordinary feat). Inside any constructor, the object may only be partially formed – you can only know that the base-class objects have been initialized, but you cannot know which classes are inherited from you. A virtual function call, however, reaches "forward" or "outward" into the inheritance hierarchy. It calls a function in a derived class. If you could do this inside a constructor, you'd be calling a function that might manipulate members that hadn't been initialized yet, a sure recipe for disaster.

The second reason is a mechanical one. When a constructor is called, one of the first things it does is initialize its VPTR. However, it can only know that it is of the "current" type. The constructor code is completely ignorant of whether or not the object is in the base of another class. When the compiler generates code for that constructor, it generates code for a constructor of that class, not a base class and not a class derived from it (because a class can't know who inherits it). So the VPTR it uses must be for the VTABLE of that class. The VPTR remains initialized to that VTABLE for the rest of the object's lifetime *unless* this isn't the last constructor call. If a more-derived constructor is called afterwards, that constructor sets the VPTR to *its* VTABLE, and so on, until the last constructor finishes. The state of the VPTR is determined by the constructor that is called last. This is another reason why the constructors are called in order from base to most-derived.

But while all this series of constructor calls is taking place, each constructor has set the VPTR to its own VTABLE. If it uses the virtual mechanism for function calls, it will produce only a call through its own

VTABLE, not the most-derived VTABLE (as would be the case after *all* the constructors were called). In addition, many compilers recognize that a virtual function call is being made inside a constructor, and perform early binding because they know that late-binding will produce a call only to the local function. In either event, you won't get the results you might expect from a virtual function call inside a constructor.

Destructors and **virtual** destructors

Constructors cannot be made explicitly virtual (and the technique in Appendix B only simulates virtual constructors), but destructors can and often must be virtual.

The constructor has the special job of putting an object together piece-by-piece, first by calling the base constructor, then the more derived constructors in order of inheritance. Similarly, the destructor also has a special job – it must disassemble an object that may belong to a hierarchy of classes. To do this, the compiler generates code that calls all the destructors, but in the *reverse* order that they are called by the constructor. That is, the destructor starts at the most-derived class and works its way down to the base class. This is the safe and desirable thing to do: The current destructor always knows that the base-class members are alive and active because it knows what it is derived from. Thus, the destructor can perform its own cleanup, then call the next-down destructor, which will perform *its* own cleanup, knowing what it is derived from, but not what is derived from it.

You should keep in mind that constructors and destructors are the only places where this hierarchy of calls must happen (and thus the proper hierarchy is automatically generated by the compiler). In all other functions, only *that* function will be called, whether it's virtual or not. The only way for base-class versions of the same function to be called in ordinary functions (virtual or not) is if you *explicitly* call that function.

Normally, the action of the destructor is quite adequate. But what happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? This is certainly a major objective in object-oriented programming. The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base

class, the compiler can only know to call the base-class version of the destructor during **delete**. Sound familiar? This is the same problem that virtual functions were created to solve for the general case. Fortunately virtual functions work for destructors as they do for all other functions except constructors.

Even though the destructor, like the constructor, is an "exceptional" function, it is possible for the destructor to be virtual because the object already knows what type it is (whereas it doesn't during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

For a time, pure virtual destructors were legal and worked if you combined them with a function body, but in the final C++ standard function bodies combined with pure virtual functions were outlawed. This means that a virtual destructor cannot be pure, and must have a function body because (unlike ordinary functions) all destructors in a class hierarchy are always called. Here's an example:

```
//: C15:Pvdest.cpp
// Pure virtual destructors
// require a function body.
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "~Base()" << endl;
  }
};

class Derived : public Base {
public:
  ~Derived() {
    cout << "~Derived()" << endl;
  }
};

int main() {
  Base* bp = new Derived; // Upcast
  delete bp; // Virtual destructor call
} ///:~
```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Virtuals in destructors

There's something that happens during destruction that you might not immediately expect. If you're inside an ordinary member function and you call a virtual function, that function is called using the late-binding mechanism. This is not true with destructors, virtual or not. Inside a destructor, only the "local" version of the member function is called; the virtual mechanism is ignored.

Why is this? Suppose the virtual mechanism *were* used inside the destructor. Then it would be possible for the virtual call to resolve to a function that was "further out" (more derived) on the inheritance hierarchy than the current destructor. But destructors are called from the "outside in" (from the most-derived destructor down to the base destructor), so the actual function called would rely on portions of an object that has *already been destroyed*! Thus, the compiler resolves the calls at compile-time and calls only the "local" version of the function. Notice that the same is true for the constructor (as described earlier), but in the constructor's case the information wasn't available, whereas in the destructor the information (that is, the VPTR) is there, but is isn't reliable.

Summary

Polymorphism – implemented in C++ with virtual functions – means "different forms." In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the virtual functions.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like **const** or a **switch** statement, for example), but instead works only in concert, as part of a "big picture" of class relationships. People are often confused by other, non-object-oriented features of C++, like overloading and default arguments, which are sometimes presented as object-oriented. Don't be fooled: If it isn't late binding, it isn't polymorphism.

To use polymorphism, and thus object-oriented techniques, effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.

Polymorphism completes the object-oriented features of the language, but there are two more major features in C++: templates (Chapter XX), and exception handling (Chapter XX). These features provide you as much increase in programming power as each of the object-oriented features: abstract data typing, inheritance, and polymorphism.

Exercises

1. Create a very simple "shape" hierarchy: a base class called **Shape** and derived classes called **Circle**, **Square**, and **Triangle**. In the base class, make a virtual function called **draw( ),** and redefine this in the derived classes. Create an array of pointers to **Shape** objects you create on the heap (and thus perform upcasting of the pointers), and call **draw( )** through the base-class pointers, to verify the behavior of the virtual function. If your debugger supports it, single-step through the example.
2. Modify Exercise 1 so **draw( )** is a pure virtual function. Try creating an object of type **Shape**. Try to call the pure virtual function inside the constructor and see what happens. Give **draw( )** a definition.
3. Write a small program to show the difference between calling a virtual function inside a normal member function and calling a virtual function inside a constructor. The program should prove that the two calls produce different results.
4. In **Early.cpp**, how can you tell whether the compiler makes the call using early or late binding? Determine the case for your own compiler.
5. (Intermediate) Create a base **class X** with no members and no constructor, but with a **virtual** function. Create a **class Y** that inherits from **X**, but without an explicit constructor. Generate assembly code and examine it to determine if a

constructor is created and called for **X**, and if so, what the code does. Explain what you discover. **X** has no default constructor, so why doesn't the compiler complain?

6. (Intermediate) Modify exercise 5 so each constructor calls a virtual function. Generate assembly code. Determine where the VPTR is being assigned inside each constructor. Is the virtual mechanism being used by your compiler inside the constructor? Establish why the local version of the function is still being called.
7. (Advanced) If function calls to an object passed by value *weren't* early-bound, a virtual call might access parts that didn't exist. Is this possible? Write some code to force a virtual call, and see if this causes a crash. To explain the behavior, examine what happens when you pass an object by value.
8. (Advanced) Find out exactly how much more time is required for a virtual function call by going to your processor's assembly-language information or other technical manual and finding out the number of clock states required for a simple call versus the number required for the virtual function instructions.

16: Introduction to templates

Inheritance and composition provide a way to reuse object code. The *template* feature in C++ provides a way to reuse *source* code.

Although C++ templates are a general-purpose programming tool, when they were introduced in the language, they seemed to discourage the use of object-based container-class hierarchies. Later versions of container-class libraries are built exclusively with templates and are much easier for the programmer to use.

This chapter begins with an introduction to containers and the way they are implemented with templates, followed by examples of container classes and how to use them.

Containers & iterators

Suppose you want to create a stack. In C, you would make a data structure and associated functions, but of course in C++ you package the two together into an abstract data type. This **stack** class will hold **int**s, to keep it simple:

```
//: C16:IStack.cpp
// Simple integer stack
#include "../require.h"
#include <iostream>
```

```
using namespace std;

class IStack {
  static const int ssize = 100;
  int stack[ssize];
  int top;
public:
  IStack() : top(0) { stack[top] = 0; }
  void push(int i) {
    if(top < ssize) stack[top++] = i;
  }
  int pop() {
    return stack[top > 0 ? --top : top];
  }
  friend class IStackIter;
};

// An iterator is a "super-pointer":
class IStackIter {
  IStack& S;
  int index;
public:
  IStackIter(IStack& is)
    : S(is), index(0) {}
  int operator++() { // Prefix form
    if (index < S.top - 1) index++;
    return S.stack[index];
  }
  int operator++(int) { // Postfix form
    int returnval = S.stack[index];
    if (index < S.top - 1) index++;
    return returnval;
  }
};
// For interest, generate Fibonacci numbers:
int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initialized to zero
  f[0] = f[1] = 1;
  // Scan for unfilled array elements:
  int i;
```

```
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
}

int main() {
  IStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  IStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

The class **IStack** is an almost trivial example of a push-down stack. For simplicity it has been created here with a fixed size, but you can also modify it to automatically expand by allocating memory off the heap. (This will be demonstrated in later examples.)

The second class, **IStackIter**, is an example of an *iterator*, which you can think of as a superpointer that has been customized to work only with an **IStack**. Notice that **IStackIter** is a **friend** of **IStack**, which gives it access to all the **private** elements of **IStack**.

Like a pointer, **IStackIter**'s job is to move through an **IStack** and retrieve values. In this simple example, the **IStackIter** can move only forward (using both the pre- and postfix forms of the **operator++**) and it can fetch only values. However, there is no boundary to the way an iterator can be defined. It is perfectly acceptable for an iterator to move around any way within its associated container and to cause the contained values to be modified. However, it is customary that an iterator is created with a constructor that attaches it to a single container object and that it is not reattached during its lifetime. (Most iterators are small, so you can easily make another one.)

To make the example more interesting, the **fibonacci( )** function generates the traditional rabbit-reproduction numbers. This is a fairly

efficient implementation, because it never generates the numbers more than once. (Although if you've been out of school awhile, you've probably figured out that you don't spend your days researching more efficient implementations of algorithms, as textbooks might lead you to believe.)

In **main( )** you can see the creation and use of the stack and its associated iterator. Once you have the classes built, they're quite simple to use.

The need for containers

Obviously an integer stack isn't a crucial tool. The real need for containers comes when you start making objects on the heap using **new** and destroying them with **delete**. In the general programming problem, you don't know how many objects you're going to need while you're writing the program. For example, in an air-traffic control system you don't want to limit the number of planes your system can handle. You don't want the program to abort just because you exceed some number. In a computer-aided design system, you're dealing with lots of shapes, but only the user determines (at runtime) exactly how many shapes you're going to need. Once you notice this tendency, you'll discover lots of examples in your own programming situations.

C programmers who rely on virtual memory to handle their "memory management" often find the idea of **new**, **delete,** and container classes disturbing. Apparently, one practice in C is to create a huge global array, larger than anything the program would appear to need. This may not require much thought (or awareness of **malloc( )** and **free( )**), but it produces programs that don't port well and can hide subtle bugs.

In addition, if you create a huge global array of objects in C++, the constructor and destructor overhead can slow things down significantly. The C++ approach works much better: When you need an object, create it with **new,** and put its pointer in a container. Later on, fish it out and do something to it. This way, you create only the objects you absolutely need. And generally you don't have all the initialization conditions at the start-up of the program; you have to wait until something happens in the environment before you can actually create the object.

So in the most common situation, you'll create a container that holds pointers to some objects of interest. You will create those objects using **new** and put the resulting pointer in the container (potentially upcasting it in the process), fishing it out later when you want to do something with

the object. This technique produces the most flexible, general sort of program.

Overview of templates

Now a problem arises. You have an **IStack**, which holds integers. But you want a stack that holds shapes or airliners or plants or something else. Reinventing your source-code every time doesn't seem like a very intelligent approach with a language that touts reusability. There must be a better way.

There are three techniques for source-code reuse: the C way, presented here for contrast; the Smalltalk approach, which significantly affected C++; and the C++ approach: templates.

The C approach

Of course you're trying to get away from the C approach because it's messy and error prone and completely inelegant. You copy the source code for a **Stack** and make modifications by hand, introducing new errors in the process. This is certainly not a very productive technique.

The Smalltalk approach

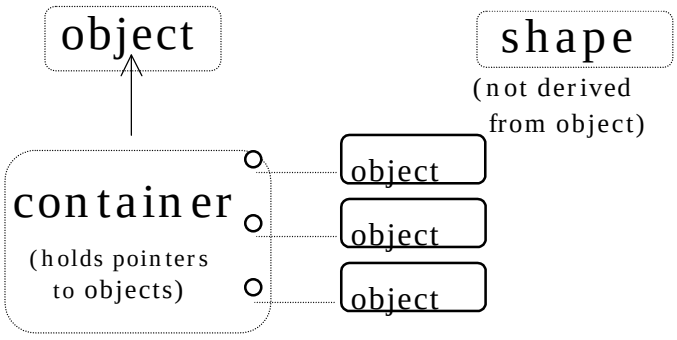
Smalltalk took a simple and straightforward approach: You want to reuse code, so use inheritance. To implement this, each container class holds items of the generic base class **object**. But, as mentioned before, the library in Smalltalk is of fundamental importance, so fundamental, in fact, that you don't ever create a class from scratch. Instead, you must always inherit it from an existing class. You find a class as close as possible to the one you want, inherit from it, and make a few changes. Obviously this is a benefit because it minimizes your effort (and explains why you spend a lot of time learning the class library before becoming an effective Smalltalk programmer).

But it also means that all classes in Smalltalk end up being part of a single inheritance tree. You must inherit from a branch of this tree when creating a new class. Most of the tree is already there (it's the Smalltalk class library), and at the root of the tree is a class called **object** – the same class that each Smalltalk container holds.

This is a neat trick because it means that every class in the Smalltalk class hierarchy is derived from **object**, so every class can be held in every container, including that container itself. This type of single-tree hierarchy based on a fundamental generic type (often named **object**) is referred to as an "object-based hierarchy." You may have heard this term before and assumed it was some new fundamental concept in OOP, like polymorphism. It just means a class tree with **object** (or some similar name) at its root and container classes that hold **object**.

Because the Smalltalk class library had a much longer history and experience behind it than C++, and the original C++ compilers had *no* container class libraries, it seemed like a good idea to duplicate the Smalltalk library in C++. This was done as an experiment with a very early C++ implementation,[43] and because it represented a significant body of code, many people began using it. In the process of trying to use the container classes, they discovered a problem.

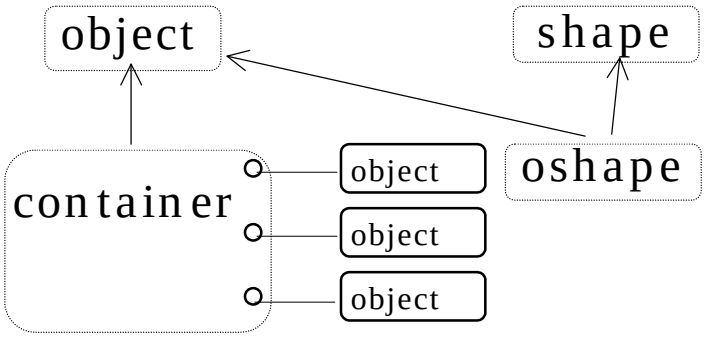
The problem was that in Smalltalk, you could force people to derive everything from a single hierarchy, but in C++ you can't. You might have your nice object-based hierarchy with its container classes, but then you might buy a set of shape classes or airline classes from another vendor who didn't use that hierarchy. (For one thing, the hierarchy imposes overhead, which C programmers eschew.) How do you shoehorn a separate class tree into the container class in your object-based hierarchy? Here's what the problem looks like:



Because C++ supports multiple independent hierarchies, Smalltalk's object-based hierarchy does not work so well.

[43] The OOPS library, by Keith Gorlen while he was at NIH. Generally available from public sources.

The solution seemed obvious. If you can have many inheritance hierarchies, then you should be able to inherit from more than one class: Multiple inheritance will solve the problem. So you do the following:



Now **oshape** has **shape**'s characteristics and behaviors, but because it is also derived from **object** it can be placed in **container**.

But multiple inheritance wasn't originally part of C++. Once the container problem was seen, there came a great deal of pressure to add the feature. Other programmers felt (and still feel) multiple inheritance wasn't a good idea and that it adds unneeded complexity to the language. An oft-repeated statement at that time was, "C++ is not Smalltalk," which, if you knew enough to translate it, meant "Don't use object based hierarchies for container classes." But in the end[44] the pressure persisted, and multiple inheritance was added to the language. Compiler vendors followed suit by including object-based container-class hierarchies, most of which have since been replaced by template versions. You can argue that multiple inheritance is needed for solving general programming problems, but you'll see in the next chapter that its complexity is best avoided except in certain cases.

The template approach

Although an object-based hierarchy with multiple inheritance is conceptually straightforward, it turns out to be painful to use. In his original book[45] Stroustrup demonstrated what he considered a preferable

[44] We'll probably never know the full story because control of the language was still within AT&T at the time.

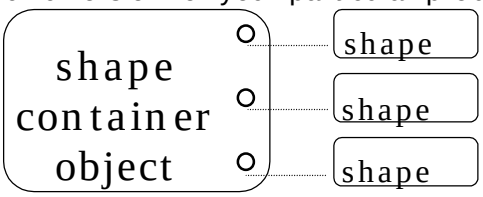
[45] *The C++ Programming Language* by Bjarne Stroustrup (1st edition, Addison-Wesley, 1986).

alternative to the object-based hierarchy. Container classes were created as large preprocessor macros with arguments that could be substituted for your desired type. When you wanted to create a container to hold a particular type, you made a couple of macro calls.

Unfortunately, this approach was confused by all the existing Smalltalk literature, and it was a bit unwieldy. Basically, nobody got it.

In the meantime, Stroustrup and the C++ team at Bell Labs had modified his original macro approach, simplifying it and moving it from the domain of the preprocessor into the compiler itself. This new code-substitution device is called a **template**[46], and it represents a completely different way to reuse code: Instead of reusing object code, as with inheritance and composition, a template reuses *source code*. The container no longer holds a generic base class called **object**, but instead an unspecified parameter. When you use a template, the parameter is substituted *by the compiler*, much like the old macro approach, but cleaner and easier to use.

Now, instead of worrying about inheritance or composition when you want to use a container class, you take the template version of the container and stamp out a specific version for your particular problem, like this:



The compiler does the work for you, and you end up with exactly the container you need to do your job, rather than an unwieldy inheritance hierarchy. In C++, the template implements the concept of a *parameterized type*. Another benefit of the template approach is that the novice programmer who may be unfamiliar or uncomfortable with inheritance can still use canned container classes right away.

Template syntax

The **template** keyword tells the compiler that the following class definition will manipulate one or more unspecified types. At the time the

[46] The inspiration for templates appears to be ADA generics.

object is defined, those types must be specified so the compiler can substitute them.

Here's a small example to demonstrate the syntax:

```
//: C16:Stemp.cpp
// Simple template example
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
class Array {
  static const int size = 100;
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size);
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

You can see that it looks like a normal class except for the line

```
template<class T>
```

which says that **T** is the substitution parameter, and it represents a type name. Also, you see **T** used everywhere in the class where you would normally see the specific type the container holds.

In **Array**, elements are inserted *and* extracted with the same function, the overloaded **operator[ ]**. It returns a reference, so it can be used on both sides of an equal sign. Notice that if the index is out of bounds, the

require( ) function is used to print a message. This is actually a case where throwing an exception is more appropriate, because then the class user can recover from the error, but that topic is not covered until Chapter XX.

In **main( )**, you can see how easy it is to create **Array**s that hold different types of objects. When you say

```
Array<int> ia;
Array<float> fa;
```

the compiler expands the **Array** template (this is called *instantiation*) twice, to create two new *generated classes*, which you can think of as **Array_int** and **Array_float.** (Different compilers may decorate the names in different ways.) These are classes just like the ones you would have produced if you had performed the substitution by hand, except that the compiler creates them for you as you define the objects **ia** and **fa**. Also note that duplicate class definitions are either avoided by the compiler or merged by the linker.

Non-inline function definitions

Of course, there are times when you'll want to have non-inline member function definitions. In this case, the compiler needs to see the **template** declaration before the member function definition. Here's the above example, modified to show the non-inline member definition:

```
//: C16:Stemp2.cpp
// Non-inline template example
#include "../require.h"

template<class T>
class Array {
  static const int size = 100;
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Index out of range");
  return A[index];
```

```
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

Notice that in the member function definition the class name is now qualified with the template argument type: **Array<T>**. You can imagine that the compiler does indeed carry both the name and the argument type(s) in some decorated form.

Header files

Even if you create non-inline function definitions, you'll generally want to put all declarations *and* definitions for a template in a header file. This may seem to violate the normal header file rule of "Don't put in anything that allocates storage" to prevent multiple definition errors at link time, but template definitions are special. Anything preceded by **template<...>** means the compiler won't allocate storage for it at that point, but will instead wait until it's told to (by a template instantiation), and that somewhere in the compiler and linker there's a mechanism for removing multiple definitions of an identical template. So you'll almost always put the entire template declaration *and* definition in the header file, for ease of use.

There are times when you may need to place the template definitions in a separate **cpp** file to satisfy special needs (for example, forcing template instantiations to exist in only a single Windows **dll** file). Most compilers have some mechanism to allow this; you'll have to investigate your particular compiler's documentation to use it.

The stack as a template

Here is the container and iterator from **Istack.cpp**, implemented as a generic container class using templates:

```
//: C16:Stackt.h
// Simple stack template
#ifndef STACKT_H
#define STACKT_H
template<class T> class StacktIter; // Declare

template<class T>
```

```
class Stackt {
  static const int ssize = 100;
  T stack[ssize];
  int top;
public:
  Stackt() : top(0) { stack[top] = 0; }
  void push(const T& i) {
    if(top < ssize) stack[top++] = i;
  }
  T pop() {
    return stack[top > 0 ? --top : top];
  }
  friend class StacktIter<T>;
};

template<class T>
class StacktIter {
  Stackt<T>& s;
  int index;
public:
  StacktIter(Stackt<T>& is)
    : s(is), index(0) {}
  T& operator++() { // Prefix form
    if (index < s.top - 1) index++;
    return s.stack[index];
  }
  T& operator++(int) { // Postfix form
    int returnIndex = index;
    if (index < s.top - 1) index++;
    return s.stack[returnIndex];
  }
};
#endif // STACKT_H ///:~
```

Notice that anywhere a template's class name is referred to, it must be accompanied by its template argument list, as in **Stackt<T>& s**. You can imagine that internally, the arguments in the template argument list are also being decorated to produce a unique class name for each template instantiation.

Also notice that a template makes certain assumptions about the objects it is holding. For example, **Stackt** assumes there is some sort of

assignment operation for **T** inside the **push( )** function. You could say that a template "implies an interface" for the types it is capable of holding.

Here's the revised example to test the template:

```
//: C16:Stackt.cpp
// Test simple stack template
#include "Stackt.h"
#include "../require.h"
#include <iostream>
using namespace std;

// For interest, generate Fibonacci numbers:
int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initialized to zero
  f[0] = f[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
}

int main() {
  Stackt<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  StacktIter<int> it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

The only difference is in the creation of **is** and **it**: You specify the type of object the stack and iterator should hold inside the template argument list.

Constants in templates

Template arguments are not restricted to class types; you can also use built-in types. The values of these arguments then become compile-time constants for that particular instantiation of the template. You can even use default values for these arguments:

```
//: C16:Mblock.cpp
// Built-in types in templates
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Mblock {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size);
    return array[index];
  }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};

template<class T, int sz = 20>
class Holder {
  Mblock<T, sz>* np;
public:
  Holder() : np(0) {}
```

```
  T& operator[](int i) {
    require(i >= 0 && i < sz);
    if(!np) np = new Mblock<T, sz>;
    return np->operator[](i);
  }
};

int main() {
  Holder<Number, 20> h;
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Class **Mblock** is a checked array of objects; you cannot index out of bounds. (Again, the exception approach in Chapter XX may be more appropriate than **assert( )** in this situation.)

The class **Holder** is much like **Mblock** except that it has a pointer to an **Mblock** instead of an embedded object of type **Mblock**. This pointer is not initialized in the constructor; the initialization is delayed until the first access. You might use a technique like this if you are creating a lot of objects, but not accessing them all, and want to save storage.

Stash and **stack** as templates

It turns out that the **Stash** and **Stack** classes that have been updated periodically throughout this book are actually container classes, so it makes sense to convert them to templates. But first, one other important issue arises with container classes: When a container releases a pointer to an object, does it destroy that object? For example, when a container object goes out of scope, does it destroy all the objects it points to?

The ownership problem

This issue is commonly referred to as *ownership*. Containers that hold entire objects don't usually worry about ownership because they clearly own the objects they contain. But if your container holds pointers (which

is more common with C++, especially with polymorphism), then it's very likely those pointers may also be used somewhere else in the program, and you don't necessarily want to delete the object because then the other pointers in the program would be referencing a destroyed object. To prevent this from happening, you must consider ownership when designing and using a container.

Many programs are very simple, and one container holds pointers to objects that are used only by that container. In this case ownership is very straightforward: The container owns its objects. Generally, you'll want this to be the default case for a container because it's the most common situation.

The best approach to handling the ownership problem is to give the client programmer the choice. This is often accomplished by a constructor argument that defaults to indicating ownership (typically desired for simple programs). In addition there may be read and set functions to view and modify the ownership of the container. If the container has functions to remove an object, the ownership state usually affects that removal, so you may also find options to control destruction in the removal function. You could conceivably also add ownership data for every element in the container, so each position would know whether it needed to be destroyed; this is a variant of reference counting where the container and not the object knows the number of references pointing to an object.

Stash as a template

The "stash" class that has been evolving throughout the book (last seen in Chapter XX) is an ideal candidate for a template. Now an iterator has been added along with ownership operations:

```
//: C16:TStash.h
// PSTASH using templates
#ifndef TSTASH_H
#define TSTASH_H
#include "../require.h"
#include <cstdlib>

// More convenient than nesting in TStash:
enum Owns { no = 0, yes = 1, Default };
// Declaration required:
template<class Type, int sz> class TStashIter;

template<class Type, int chunksize = 20>
```

```
class TStash {
  int quantity;
  int next;
  Owns _owns; // Flag
  void inflate(int increase = chunksize);
protected:
  Type** storage;
public:
  TStash(Owns owns = yes);
  ~TStash();
  Owns owns() const { return _owns; }
  void owns(Owns newOwns) { _owns = newOwns; }
  int add(Type* element);
  int remove(int index, Owns d = Default);
  Type* operator[](int index);
  int count() const { return next; }
  friend class TStashIter<Type, chunksize>;
};

template<class Type, int sz = 20>
class TStashIter {
  TStash<Type, sz>& ts;
  int index;
public:
  TStashIter(TStash<Type, sz>& TS)
    : ts(TS), index(0) {}
  TStashIter(const TStashIter& rv)
    : ts(rv.ts), index(rv.index) {}
  // Jump interator forward or backward:
  void forward(int amount) {
    index += amount;
    if(index >= ts.next) index = ts.next -1;
  }
  void backward(int amount) {
    index -= amount;
    if(index < 0) index = 0;
  }
  // Return value of ++ and -- to be
  // used inside conditionals:
  int operator++() {
    if(++index >= ts.next) return 0;
    return 1;
```

```
  }
  int operator++(int) { return operator++(); }
  int operator--() {
    if(--index < 0) return 0;
    return 1;
  }
  int operator--(int) { return operator--(); }
  operator int() {
    return index >= 0 && index < ts.next;
  }
  Type* operator->() {
    Type* t = ts.storage[index];
    if(t) return t;
    require(0,"TStashIter::operator->return 0");
    return 0; // To allow inlining
  }
  // Remove the current element:
  int remove(Owns d = Default){
    return ts.remove(index, d);
  }
};

template<class Type, int sz>
TStash<Type, sz>::TStash(Owns owns) : _owns(owns) {
  quantity = 0;
  storage = 0;
  next = 0;
}

// Destruction of contained objects:
template<class Type, int sz>
TStash<Type, sz>::~TStash() {
  if(!storage) return;
  if(_owns == yes)
    for(int i = 0; i < count(); i++)
      delete storage[i];
  free(storage);
}

template<class Type, int sz>
int TStash<Type, sz>::add(Type* element) {
  if(next >= quantity)
```

```
    inflate();
  storage[next++] = element;
  return(next - 1); // Index number
}

template<class Type, int sz>
int TStash<Type, sz>::remove(int index, Owns d){
  if(index >= next || index < 0)
    return 0;
  switch(d) {
    case Default:
      if(_owns != yes) break;
    case yes:
      delete storage[index];
    case no:
      storage[index] = 0; // Position is empty
  }
  return 1;
}

template<class Type, int sz> inline
Type* TStash<Type, sz>::operator[](int index) {
  // Remove check for shipping application:
  require(index >= 0 && index < next);
  return storage[index];
}

template<class Type, int sz>
void TStash<Type, sz>::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*sizeof(Type*));
  require(v != 0);  // Was it successful?
  storage = (Type**)v;
  quantity += increase;
}
#endif // TSTASH_H ///:~
```

The **enum owns** is global, although you'd normally want to nest it inside the class. Here it's more convenient to use, but you can try moving it if you want to see the effect.

The **storage** pointer is made **protected** so inherited classes can directly access it. This means that the inherited classes may become dependent

on the specific implementation of **TStash**, but as you'll see in the **Sorted.cpp** example, it's worth it.

The **own** flag indicates whether the container defaults to owning its objects. If so, in the destructor each object whose pointer is in the container is destroyed. This is straightforward; the container knows the type it contains. You can also change the default ownership in the constructor or read and modify it with the overloaded **owns( )** function.

You should be aware that if the container holds pointers to a base-class type, that type should have a **virtual** destructor to ensure proper cleanup of derived objects whose addresses have been upcast when placing them in the container.

The **TStashIter** follows the iterator model of bonding to a single container object for its lifetime. In addition, the copy-constructor allows you to make a new iterator pointing at the same location as the existing iterator you create it from, effectively making a bookmark into the container. The **forward( )** and **backward( )** member functions allow you to jump an iterator by a number of spots, respecting the boundaries of the container. The overloaded increment and decrement operators move the iterator by one place. The smart pointer is used to operate on the element the iterator is referring to, and **remove( )** destroys the current object by calling the container's **remove( )**.

The following example creates and tests two different kinds of **Stash** objects, one for a new class called **Int** that announces its construction and destruction and one that holds objects of the class **String** from Chapter XX.

```
//: C16:TStashTest.cpp
// Test TStash
#include "TStash.h"
#include "../require.h"
#include <fstream>
#include <vector>
#include <string>
using namespace std;
ofstream out("tstest.out");

class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
    out << ">" << i << endl;
```

```
  }
  ~Int() { out << "~" << i << endl; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << x.i;
  }
};

int main() {
  TStash<Int> intStash; // Instantiate for Int
  for(int i = 0; i < 30; i++)
    intStash.add(new Int(i));
  TStashIter<Int> intIter(intStash);
  intIter.forward(5);
  for(int j = 0; j < 20; j++, intIter++)
    intIter.remove(); // Default removal
  for(int k = 0; k < intStash.count(); k++)
    if(intStash[k]) // Remove() causes "holes"
      out << *intStash[k] << endl;

  ifstream file("TStashTest.cpp");
  assure(file, "TStashTest.cpp");
  // Instantiate for String:
  TStash<string> stringStash;
  string line;
  while(getline(file, line))
    stringStash.add(new string(line));
  for(int u = 0; u < stringStash.count(); u++)
    if(stringStash[u])
      out << *stringStash[u] << endl;
  TStashIter<string> it(stringStash);
  int n = 25;
  it.forward(n);
  while(it) {
    out << n++ << ": " << it->c_str() << endl;
    it++;
  }
} ///:~
```

In both cases an iterator is created and used to move through the container. Notice the elegance produced by using these constructs: You aren't assailed with the implementation details of using an array. You tell

the container and iterator objects *what* to do, not how. This makes the solution easier to conceptualize, to build, and to modify.

stack as a template

The **Stack** class, last seen in Chapter XX, is also a container and is also best expressed as a template with an associated iterator. Here's the new header file:

```
//: C16:TStack.h
// Stack using templates
#ifndef TSTACK_H
#define TSTACK_H

// Declaration required:
template<class T> class TStackIterator;

template<class T> class TStack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt) {
      data = dat;
      next = nxt;
    }
  }* head;
  int _owns;
public:
  TStack(int own = 1) : head(0), _owns(own) {}
  ~TStack();
  void push(T* dat) {
    head = new Link(dat,head);
  }
  T* peek() const { return head->data; }
  T* pop();
  int owns() const { return _owns; }
  void owns(int newownership) {
    _owns = newownership;
  }
  friend class TStackIterator<T>;
};

template<class T> T* TStack<T>::pop() {
```

```
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> TStack<T>::~TStack() {
  Link* cursor = head;
  while(head) {
    cursor = cursor->next;
    // Conditional cleanup of data:
    if(_owns) delete head->data;
    delete head;
    head = cursor;
  }
}

template<class T> class TStackIterator {
  TStack<T>::Link* p;
public:
  TStackIterator(const TStack<T>& tl)
    : p(tl.head) {}
  TStackIterator(const TStackIterator& tl)
    : p(tl.p) {}
  // operator++ returns boolean indicating end:
  int operator++() {
    if(p->next)
      p = p->next;
    else p = 0; // Indicates end of list
    return int(p);
  }
  int operator++(int) { return operator++(); }
  // Smart pointer:
  T* operator->() const {
    if(!p) return 0;
    return p->data;
  }
  T* current() const {
    if(!p) return 0;
    return p->data;
```

```
  }
  // int conversion for conditional test:
  operator int() const { return p ? 1 : 0; }
};
#endif // TSTACK_H ///:~
```

You'll also notice the class has been changed to support ownership, which works now because the class knows the exact type (or at least the base type, which will work assuming virtual destructors are used). As with **TStash**, the default is for the container to destroy its objects but you can change this by either modifying the constructor argument or using the **owns( )** read/write member functions.

The iterator is very simple and very small – the size of a single pointer. When you create a **TStackIterator**, it's initialized to the head of the linked list, and you can only increment it forward through the list. If you want to start over at the beginning, you create a new iterator, and if you want to remember a spot in the list, you create a new iterator from the existing iterator pointing at that spot (using the copy-constructor).

To call functions for the object referred to by the iterator, you can use the smart pointer (a very common sight in iterators) or a function called **current( )** that *looks* identical to the smart pointer because it returns a pointer to the current object, but is different because the smart pointer performs the extra levels of dereferencing (see Chapter XX). Finally, the **operator int** indicates whether or not you are at the end of the list and allows the iterator to be used in conditional statements.

The entire implementation is contained in the header file, so there's no separate **cpp** file. Here's a small test that also exercises the iterator:

```
//: C16:TStackTest.cpp
// Use template list & iterator
#include "TStack.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStackTest.cpp");
  assure(file, "TStackTest.cpp");
  TStack<string> textlines;
  // Read file and store lines in the list:
```

```
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Use iterator to print lines from the list:
  TStackIterator<string> it(textlines);
  TStackIterator<string>* it2 = 0;
  while(it) {
    cout << *it.current() << endl;
    it++;
    if(++i == 10) // Remember 10th line
      it2 = new TStackIterator<string>(it);
  }
  cout << *(it2->current()) << endl;
  delete it2;
} ///:~
```

A **TStack** is instantiated to hold **String** objects and filled with lines from a file. Then an iterator is created and used to move through the linked list. The tenth line is remembered by copy-constructing a second iterator from the first; later this line is printed and the iterator – created dynamically – is destroyed. Here, dynamic object creation is used to control the lifetime of the object.

This is very similar to earlier test examples for the **Stack** class, but now the contained objects are properly destroyed when the **TStack** is destroyed.

Polymorphism & containers

It's common to see polymorphism, dynamic object creation and containers used together in a true object-oriented program. Containers and dynamic object creation solve the problem of not knowing how many or what type of objects you'll need, and because the container is configured to hold pointers to base-class objects, an upcast occurs every time you put a derived-class pointer into the container (with the associated code organization and extensibility benefits). The following example is a little simulation of trash recycling. All the trash is put into a single bin, then later it's sorted out into separate bins. There's a function that goes through any trash bin and figures out what the resource value is. Notice

this is not the most elegant way to implement this simulation; the example will be revisited in Chapter XX when Run-Time Type Identification (RTTI) is explained:

```
//: C16:Recycle.cpp
// Containers & polymorphism
#include "TStack.h"
#include <fstream>
#include <cstdlib>
#include <ctime>
using namespace std;
ofstream out("recycle.out");

enum TrashType { AluminumT, PaperT, GlassT };

class Trash {
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual TrashType trashType() const = 0;
  virtual const char* name() const = 0;
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() {}
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  TrashType trashType() const { return AluminumT; }
  virtual const char* name() const {
    return "Aluminum";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float Aluminum::val = 1.67;

class Paper : public Trash {
```

```
  static float val;
public:
  Paper(float wt) : Trash(wt) {}
  TrashType trashType() const { return PaperT; }
  virtual const char* name() const {
    return "Paper";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float Paper::val = 0.10;

class Glass : public Trash {
  static float val;
public:
  Glass(float wt) : Trash(wt) {}
  TrashType trashType() const { return GlassT; }
  virtual const char* name() const {
    return "Glass";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float Glass::val = 0.23;

// Sums up the value of the Trash in a bin:
void sumValue(const TStack<Trash>& bin,ostream& os){
  TStackIterator<Trash> tally(bin);
  float val = 0;
  while(tally) {
    val += tally->weight() * tally->value();
    os << "weight of " << tally->name()
       << " = " << tally->weight() << endl;
    tally++;
  }
  os << "Total value = " << val << endl;
```

```
}

int main() {
  srand(time(0)); // Seed random number generator
  TStack<Trash> bin; // Default to ownership
  // Fill up the Trash bin:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push(new Aluminum(rand() % 100));
        break;
      case 1 :
        bin.push(new Paper(rand() % 100));
        break;
      case 2 :
        bin.push(new Glass(rand() % 100));
        break;
    }
  // Bins to sort into:
  TStack<Trash> glassBin(0); // No ownership
  TStack<Trash> paperBin(0);
  TStack<Trash> alBin(0);
  TStackIterator<Trash> sorter(bin);
  // Sort the Trash:
  // (RTTI offers a nicer solution)
  while(sorter) {
    // Smart pointer call:
    switch(sorter->trashType()) {
      case AluminumT:
        alBin.push(sorter.current());
        break;
      case PaperT:
        paperBin.push(sorter.current());
        break;
      case GlassT:
        glassBin.push(sorter.current());
        break;
    }
    sorter++;
  }
  sumValue(alBin, out);
  sumValue(paperBin, out);
```

```
  sumValue(glassBin, out);
  sumValue(bin, out);
} ///:~
```

This uses the classic structure of virtual functions in the base class that are redefined in the derived class. The container **TStack** is instantiated for **Trash**, so it holds **Trash** pointers, which are pointers to the base class. However, it will also hold pointers to objects of classes derived from **Trash**, as you can see in the call to **push( )**. When these pointers are added, they lose their specific identities and become simply **Trash** pointers (they are *upcast*). However, because of polymorphism the proper behavior still occurs when the virtual function is called through the **tally** and **sorter** iterators. (Notice the use of the iterator's smart pointer, which causes the virtual function call.)

The **Trash** class also includes a **virtual** destructor, something you should automatically add to any class with **virtual** functions. When the **bin** container goes out of scope, the container's destructor calls all the **virtual** destructors for the objects it contains, and thus properly cleans everything up.

Because container class templates are rarely subject to the inheritance and upcasting you see with "ordinary" classes, you'll almost never see **virtual** functions in these types of classes. Their reuse is implemented with templates, not with inheritance.

Summary

Container classes are an essential part of object-oriented programming; they are another way to simplify and hide the details of a program and to speed the process of program development. In addition, they provide a great deal of safety and flexibility by replacing the primitive arrays and relatively crude data structure techniques found in C.

Because the client programmer needs containers, it's essential that they be easy to use. This is where the **template** comes in. With templates the syntax for source-code reuse (as opposed to object-code reuse provided by inheritance and composition) becomes trivial enough for the novice user. In fact, reusing code with templates is notably easier than inheritance and composition.

Although you've learned about creating container and iterator classes in this book, in practice it's much more expedient to learn the containers and iterators that come with your compiler or, failing that, to buy a library

from a third-party vendor.[47] The standard C++ library includes a very complete but nonexhaustive set of containers and iterators.

The issues involved with container-class design have been touched upon in this chapter, but you may have gathered that they can go much further. A complicated container-class library may cover all sorts of additional issues, including persistence (introduced in Chapter XX) and garbage collection (introduced in Chapter XX), as well as additional ways to handle the ownership problem.

Exercises

1. Modify the result of Exercise 1 from Chapter XX to use a **TStack** and **TStackIterator** instead of an array of **Shape** pointers. Add destructors to the class hierarchy so you can see that the **Shape** objects are destroyed when the **TStack** goes out of scope.
2. Modify the **Sshape2.cpp** example from Chapter XX to use **TStack** instead of an array.
3. Modify **Recycle.cpp** to use a **TStash** instead of a **TStack**.
4. Change **SetTest.cpp** to use a **SortedSet** instead of a **set**.
5. Duplicate the functionality of **Applist.cpp** for the **TStash** class.
6. You can do this exercise only if your compiler supports member function templates. Copy **Tstack.h** to a new header file and add the function templates in **Applist.cpp** as *member* function templates of **TStack**.
7. (Advanced) Modify the **TStack** class to further increase the granularity of ownership: add a flag to each link indicating whether that link owns the object it points to, and support this information in the **add( )** function and destructor. Add member functions to read and change the ownership for each link, and decide what the **_owns** flag means in this new context.
8. (Advanced) Modify the **TStack** class so each entry contains reference-counting information (*not* the objects they

[47] See, for example, Rogue Wave, which has a well-designed set of C++ tools for all platforms.

contain), and add member functions to support the reference counting behavior.

9. (Advanced) Change the underlying implementation of **Urand** in **Sorted.cpp** so it is space-efficient (as described in the paragraph following **Sorted.cpp**) rather than time-efficient.
10. (Advanced) Change the **typedef cntr** from an **int** to a **long** in **Getmem.h** and modify the code to eliminate the resulting warning messages about the loss of precision. This is a pointer arithmetic problem.
11. (Advanced) Devise a test to compare the execution speed of an **SString** created on the stack versus one created on the heap.

Part 2: The Standard C++ Library

Standard C++ not only incorporates all the Standard C libraries, with small additions and changes to support type safety, it also adds libraries of its own. These libraries are far more powerful than those in Standard C; the leverage you get from them is analogous to the leverage you get from changing from C to C++.

This section of the book gives you an in-depth introduction to the most important portions of the Standard C++ library.

The most complete and also the most obscure reference to the full libraries is the Standard itself. Somewhat more readable (and yet still a self-described "expert's guide") is Bjarne Stroustrup's 3rd Edition of *The C++ Programming Language* (Addison-Wesley, 1997). Another valuable reference is the 3rd edition of *C++ Primer*, by Lippman & Lajoie. The goal of the chapters in this book that cover the libraries is to provide you with an encyclopedia of descriptions and examples so you'll have a good starting point for solving any problem that requires the use of the Standard libraries. However, there are some techniques and topics that are used rarely enough that they are not covered here, so if you can't find it in these chapters you should reach for the other two books; this book is not intended to replace those but rather to complement them. In

particular, I hope that after going through the material in the following chapters you'll have a much easier time understanding those books.

You will notice that this section does not contain exhaustive documentation describing every function and class in the Standard C++ library. I've left the full descriptions to others; in particular there a particularly good on-line sources of standard library documentation in HTML format that you can keep resident on your computer and view with a Web browser whenever you need to look something up. This is PJ Plauger's Dinkumware C/C++ Library reference at http://www.dinkumware.com. You can view this on-line, and purchase it for local viewing. It contains complete reference pages for the both the C and C++ libraries (so it's good to use for all your Standard C/C++ programming questions). I am particularly fond of electronic documentation not only because you can always have it with you, but also because you can do an electronic search for what you're seeking.

When you're actively programming, these resources should adequately satisfy your reference needs (and you can use them to look up anything in this chapter that isn't clear to you). Appendix XX lists additional references.

Library overview

[[Still needs work]]

The first chapter in this section introduces the Standard C++ **string** class, which is a powerful tool that simplifies most of the text processing chores you might have to do. The **string** class may be the most thorough string manipulation tool you've ever seen. Chances are, anything you've done to character strings with lines of code in C can be done with a member function call in the string class, including **append( )**, **assign( )**, **insert( )**, **remove( )**, **replace( )**, **resize( )**, **copy( )**, **find( )**, **rfind( )**, **find_first_of( )**, **find_last_of( )**, **find_first_not_of( )**, **find_last_not_of( )**, **substr( )**, and **compare( )**. The operators **=**, **+=**, and **[ ]** are also overloaded to perform the intuitive operations. In addition, there's a "wide" **wstring** class designed to support international character sets. Both **string** and **wstring** (declared in **<string>**, not to be confused with C's **<string.h>**, which is, in strict C++, **<cstring>**) are created from a common template class called **basic_string**. Note that the string classes are seamlessly integrated with iostreams, virtually eliminating the need for you to ever use **strstream**.

The next chapter covers the **iostream** library.

Language Support. Elements inherent to the language itself, like implementation limits in **<climits>** and **<cfloat>**; dynamic memory declarations in **<new>** like **bad_alloc** (the exception thrown when you're out of memory) and **set_new_handler**; the **<typeinfo>** header for RTTI and the **<exception>** header that declares the **terminate( )** and **unexpected( )** functions.

Diagnostics Library. Components C++ programs can use to detect and report errors. The **<exception>** header declares the standard exception classes and **<cassert>** declares the same thing as C's **assert.h**.

General Utilities Library. These components are used by other parts of the Standard C++ library, but you can also use them in your own programs. Included are templatized versions of operators **!=**, **>**, **<=**, and **>=** (to prevent redundant definitions), a **pair** template class with a **tuple**-making template function, a set of *function objects* for support of the STL, and storage allocation functions for use with the STL so you can easily modify the storage allocation mechanism.

Localization Library. This allows you to localize strings in your program to adapt to usage in different countries, including money, numbers, date, time, and so on.

Containers Library. This includes the Standard Template Library (described in the next section of this appendix) and also the **bits** and **bit_string** classes in **<bits>** and **<bitstring>**, respectively. Both **bits** and **bit_string** are more complete implementations of the bitvector concept introduced in Chapter XX. The **bits** template creates a fixed-sized array of bits that can be manipulated with all the bitwise operators, as well as member functions like **set( )**, **reset( )**, **count( )**, **length( )**, **test( )**, **any( )**, and **none( )**. There are also conversion operators **to_ushort( )**, **to_ulong( )**, and **to_string( )**.

The **bit_string** class is, by contrast, a dynamically sized array of bits, with similar operations to **bits**, but also with additional operations that make it act somewhat like a **string**. There's a fundamental difference in bit weighting: With **bits**, the right-most bit (bit zero) is the least significant bit, but with **bit_string**, the right-most bit is the *most* significant bit. There are no conversions between **bits** and **bit_string**. You'll use **bits** for a space-efficient set of on-off flags and **bit_string** for manipulating arrays of binary values (like pixels).

Iterators Library. Includes iterators that are tools for the STL (described in the next section of this appendix), streams, and stream buffers.

Algorithms Library. These are the template functions that perform operations on the STL containers using iterators. The algorithms include: **adjacent_find**, **prev_permutation**, **binary_search**, **push_heap**, **copy**, **random_shuffle**, **copy_backward**, **remove**, **count**, **remove_copy**, **count_if**, **remove_copy_if**, **equal**, **remove_if**, **equal_range**, **replace**, **fill**, **replace_copy**, **fill_n**, **replace_copy_if**, **find**, **replace_if**, **find_if**, **reverse**, **for_each**, **reverse_copy**, **generate**, **rotate**, **generate_n**, **rotate_copy**, **includes**, **search**, **inplace_merge**, **set_difference**, **lexicographical_compare**, **set_intersection**, **lower_bound**, **set_symmetric_difference**, **make_heap**, **set_union**, **max**, **sort**, **max_element**, **sort_heap**, **merge**, **stable_partition**, **min**, **stable_sort**, **min_element**, **swap**, **mismatch**, **swap_ranges**, **next_permutation**, **transform**, **nth_element**, **unique**, **partial_sort**, **unique_copy**, **partial_sort_copy**, **upper_bound**, and **partition**.

Numerics Library. The goal of this library is to allow the compiler implementor to take advantage of the architecture of the underlying machine when used for numerical operations. This way, creators of higher level numerical libraries can write to the numerics library and produce efficient algorithms without having to customize to every possible machine. The numerics library also includes the complex number class (which appeared in the first version of C++ as an example, and has become an expected part of the library) in **float**, **double**, and **long double** forms.

17: Strings

> [48]One of the biggest time-wasters in C is character arrays: keeping track of the difference between static quoted strings and arrays created on the stack and the heap, and the fact that sometimes you're passing around a **char*** and sometimes you must copy the whole array.

(This is the general problem of *shallow copy* vs. *deep copy*.) Especially because string manipulation is so common, character arrays are a great source of misunderstandings and bugs.

Despite this, creating string classes remained a common exercise for beginning C++ programmers for many years. The Standard C++ library **string** class solves the problem of character array manipulation once and for all**,** keeping track of memory even during assignments and copy-constructions. You simply don't need to think about it.

This chapter examines the Standard C++ **string** class, beginning with a look at what constitutes a C++ string and how the C++ version differs from a traditional C character array. You'll learn about operations and manipulations using **string** objects, and see how C++ **string**s accommodate variation in character sets and string data conversion.

Handling text is perhaps one of the oldest of all programming applications, so it's not surprising that the C++ **string** draws heavily on the ideas and terminology that have long been used for this purpose in C and other languages. As you begin to acquaint yourself with C++ **string**s this fact should be reassuring, in the respect that no matter what programming idiom you choose, there are really only about three things you can do with a **string**: create or modify the sequence of characters stored in the

---

[48] Much of the material in this chapter was originally created by Nancy Nicolaisen

string, detect the presence or absence of elements within the **string**, and translate between various schemes for representing **string** characters.

You'll see how each of these jobs is accomplished using C++ **string** objects.

What's in a string

In C, a string is simply an array of characters that always includes a binary zero (often called the *null terminator*) as its final array element. There are two significant differences between C++ **string**s and their C progenitors. First, C++ **string** objects associate the array of characters which constitute the **string** with methods useful for managing and operating on it. A **string** also contains certain "housekeeping" information about the size and storage location of its data. Specifically, a C++ **string** object knows its starting location in memory, its content, its length in characters, and the length in characters to which it can grow before the **string** object must resize its internal data buffer. This gives rise to the second big difference between C **char** arrays and C++ **string**s. C++ **string**s do not include a null terminator, nor do the C++ **string** handling member functions rely on the existence of a null terminator to perform their jobs. C++ **string**s greatly reduce the likelihood of making three of the most common and destructive C programming errors: overwriting array bounds, trying to access arrays through uninitialized or incorrectly valued pointers, and leaving pointers "dangling" after an array ceases to occupy the storage that was once allocated to it.

The exact implementation of memory layout for the string class is not defined by the C++ Standard. This architecture is intended to be flexible enough to allow differing implementations by compiler vendors, yet guarantee predictable behavior for users. In particular, the exact conditions under which storage is allocated to hold data for a string object are not defined. String allocation rules were formulated to allow but not require a reference-counted implementation, but whether or not the implementation uses reference counting, the semantics must be the same. To put this a bit differently, in C, every **char** array occupies a unique physical region of memory. In C++, individual **string** objects may or may not occupy unique physical regions of memory, but if reference counting is used to avoid storing duplicate copies of data, the individual objects must look and act as though they do exclusively own unique regions of storage. For example:

```
//: C17:StringStorage.cpp
```

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

int main() {
  string s1("12345");
  // Set the iterator indicate the first element
  string::iterator it = s1.begin();
  // This may copy the first to the second or
  // use reference counting to simulate a copy
  string s2 = s1;
  // Either way, this statement may ONLY modify first
  *it = '0';
  cout << "s1 = " << s1 << endl;
  cout << "s2 = " << s2 << endl;
} ///:~
```

Reference counting may serve to make an implementation more memory efficient, but it is transparent to users of the **string** class.

Creating and initializing C++ strings

Creating and initializing **string**s is a straightforward proposition, and fairly flexible as well. In the example shown below, the first **string**, **imBlank**, is declared but contains no initial value. Unlike a C **char** array, which would contain a random and meaningless bit pattern until initialization, **imBlank** does contain meaningful information. This **string** object has been initialized to hold “no characters,” and can properly report its 0 length and absence of data elements through the use of class member functions.

The next **string**, **heyMom**, is initialized by the literal argument "Where are my socks?". This form of initialization uses a quoted character array as a parameter to the **string** constructor. By contrast, **standardReply** is simply initialized with an assignment. The last **string** of the group, **useThisOneAgain**, is initialized using an existing C++ **string** object. Put another way, this example illustrates that **string** objects let you:

- Create an empty **string** and defer initializing it with character data
- Initialize a **string** by passing a literal, quoted character array as an argument to the constructor
- Initialize a **string** using ‘**=**‘

- Use one **string** to initialize another

```
//: C17:SmallString.cpp
#include <string>
using namespace std;

int main() {
  string imBlank;
  string heyMom("Where are my socks?");
  string standardReply = "Beamed into deep "
    "space on wide angle dispersion?";
  string useThisOneAgain(standardReply);
} ///:~
```

These are the simplest forms of **string** initialization, but there are other variations which offer more flexibility and control. You can :

- Use a portion of either a C **char** array or a C++ **string**
- Combine different sources of initialization data using **operator+**
- Use the **string** object's **substr( )** member function to create a substring

```
//: C17:SmallString2.cpp
#include <string>
using namespace std;

int main() {
  string s1
    ("What is the sound of one clam napping?");
  string s2
    ("Anything worth doing is worth overdoing.");
  string s3("I saw Elvis in a UFO.");
  // Copy the first 8 chars
  string s4(s1, 0, 8);
  // Copy 6 chars from the middle of the source
  string s5(s2, 15, 6);
  // Copy from middle to end
  string s6(s3, 6, 15);
  // Copy all sorts of stuff
  string quoteMe = s4 + "that" +
  // substr() copies 10 chars at element 20
  s1.substr(20, 10) + s5 +
  // substr() copies up to either 100 char
```

```
  // or eos starting at element 5
  "with" + s3.substr(5, 100) +
  // OK to copy a single char this way
  s1.substr(37, 1);
} ///:~
```

The **string** member function **substr( )** takes a starting position as its first argument and the number of characters to select as the second argument. Both of these arguments have default values and if you say **substr( )** with an empty argument list you produce a copy of the entire **string**, so this is a convenient way to duplicate a **string**.

Here's what the **string quoteMe** contains after the initialization shown above :

```
"What is that one clam doing with Elvis in a UFO?"
```

Notice the final line of example above. C++ allows **string** initialization techniques to be mixed in a single statement, a flexible and convenient feature. Also note that the last initializer copies *just one character* from the source **string**.

Another slightly more subtle initialization technique involves the use of the **string** iterators **string.begin( )** and **string.end( ).** This treats a **string** like a *container* object (which you've seen primarily in the form of **vector** so far in this book – you'll see many more containers soon) which has *iterators* indicating the start and end of the "container." This way you can hand a **string** constructor two iterators and it will copy from one to the other into the new **string**:

```
//: C17:StringIterators.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string source("xxx");
  string s(source.begin(), source.end());
  cout << s << endl;
} ///:~
```

The iterators are not restricted to **begin( )** and **end( )**, so you can choose a subset of characters from the source **string**.

Initialization limitations

C++ **string**s may *not* be initialized with single characters or with ASCII or other integer values.

```
//: C17:UhOh.cpp
#include <string>
using namespace std;

int main() {
  // Error: no single char inits
  //! string nothingDoing1('a');
  // Error: no integer inits
  //! string nothingDoing2(0x37);
} ///:~
```

This is true both for initialization by assignment and by copy constructor.

Operating on strings

If you've programmed in C, you are accustomed to the convenience of a large family of functions for writing, searching, rearranging, and copying **char** arrays. However, there are two unfortunate aspects of the Standard C library functions for handling **char** arrays. First, there are three loosely organized families of them: the "plain" group, the group that manipulates the characters *without* respect to case, and the ones which require you to supply a count of the number of characters to be considered in the operation at hand. The roster of function names in the C **char** array handling library literally runs to several pages, and though the kind and number of arguments to the functions are somewhat consistent within each of the three groups, to use them properly you must be very attentive to details of function naming and parameter passing.

The second inherent trap of the standard C **char** array tools is that they all rely explicitly on the assumption that the character array includes a null terminator. If by oversight or error the null is omitted or overwritten, there's very little to keep the C **char** array handling functions from manipulating the memory beyond the limits of the allocated space, sometimes with disastrous results.

C++ provides a vast improvement in the convenience and safety of **string** objects. For purposes of actual string handling operations, there are a modest two or three dozen member function names. It's worth your while

to become acquainted with these. Each function is overloaded, so you don't have to learn a new **string** member function name simply because of small differences in their parameters.

Appending, inserting and concatenating strings

One of the most valuable and convenient aspects of C++ strings is that they grow as needed, without intervention on the part of the programmer. Not only does this make string handling code inherently more trustworthy, it also almost entirely eliminates a tedious "housekeeping" chore – keeping track of the bounds of the storage in which your strings live. For example, if you create a string object and initialize it with a string of 50 copies of 'X', and later store in it 50 copies of "Zowie", the object itself will reallocate sufficient storage to accommodate the growth of the data. Perhaps nowhere is this property more appreciated than when the strings manipulated in your code will change in size, but when you don't know big the change is. Appending, concatenating, and inserting strings often give rise to this circumstance, but the string member functions **append( )** and **insert( )** transparently reallocate storage when a string grows.

```
//: C17:StrSize.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string bigNews("I saw Elvis in a UFO. ");
  cout << bigNews << endl;
  // How much data have we actually got?
  cout << "Size = " << bigNews.size() << endl;
  // How much can we store without reallocating
  cout << "Capacity = "
    << bigNews.capacity() << endl;
  // Insert this string in bigNews immediately
  // following bigNews[1]
  bigNews.insert(1, " thought I ");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = "
    << bigNews.capacity() << endl;
  // Make sure that there will be this much space
```

```
  bigNews.reserve(500);
  // Add this to the end of the string
  bigNews.append("I've been working too hard.");
  cout << bigNews << endl;
  cout << "Size = " << bigNews.size() << endl;
  cout << "Capacity = "
    << bigNews.capacity() << endl;
} ///:~
```

Here is the output:

```
I saw Elvis in a UFO.
Size = 21
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 63
I thought I saw Elvis in a UFO. I've been
working too hard.
Size = 66
Capacity = 511
```

This example demonstrates that even though you can safely relinquish much of the responsibility for allocating and managing the memory your **string**s occupy, C++ **string**s provide you with several tools to monitor and manage their size. The **size( )**, **resize( )**, **capacity( )**, and **reserve( )** member functions can be very useful when its necessary to work back and forth between data contained in C++ style strings and traditional null terminated C **char** arrays. Note the ease with which we changed the size of the storage allocated to the string.

The exact fashion in which the **string** member functions will allocate space for your data is dependent on the implementation of the library. When one implementation was tested with the example above, it appeared that reallocations occurred on even word boundaries, with one byte held back. The architects of the **string** class have endeavored to make it possible to mix the use of C **char** arrays and C++ string objects, so it is likely that figures reported by **StrSize.cpp** for capacity reflect that in this particular implementation, a byte is set aside to easily accommodate the insertion of a null terminator.

Replacing string characters

insert( ) is particularly nice because it absolves you of making sure the insertion of characters in a string won't overrun the storage space or overwrite the characters immediately following the insertion point. Space grows and existing characters politely move over to accommodate the new elements. Sometimes, however, this might not be what you want to happen. If the data in string needs to retain the ordering of the original characters relative to one another or must be a specific constant size, use the **replace( )** function to overwrite a particular sequence of characters with another group of characters. There are quite a number of overloaded versions of **replace( )**, but the simplest one takes three arguments: an integer telling where to start in the string, an integer telling how many characters to eliminate from the original string, and the replacement string (which can be a different number of characters than the eliminated quantity). Here's a very simple example:

```
//: C17:StringReplace.cpp
// Simple find-and-replace in strings
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  cout << s << endl;
  int start = s.find(tag);
  cout << "start = " << start << endl;
  cout << "size = " << tag.size() << endl;
  s.replace(start, tag.size(), "hello there");
  cout << s << endl;
} ///:~
```

The **tag** is first inserted into **s** (notice that the insert happens *before* the value indicating the insert point, and that an extra space was added after **tag**), then it is found and replaced.

You should actually check to see if you've found anything before you perform a **replace( )**. The above example replaces with a **char***, but there's an overloaded version that replaces with a **string**. Here's a more complete demonstration **replace( )**

```
//: C17:Replace.cpp
#include <string>
#include <iostream>
using namespace std;

void replaceChars(string& modifyMe,
  string findMe, string newChars){
  // Look in modifyMe for the "find string"
  // starting at position 0
  int i = modifyMe.find(findMe, 0);
  // Did we find the string to replace?
  if(i != string::npos)
    // Replace the find string with newChars
    modifyMe.replace(i,newChars.size(),newChars);
}

int main() {
  string bigNews =
   "I thought I saw Elvis in a UFO. "
   "I have been working too hard.";
  string replacement("wig");
  string findMe("UFO");
  // Find "UFO" in bigNews and overwrite it:
  replaceChars(bigNews, findMe,  replacement);
  cout << bigNews << endl;
} ///:~
```

Now the last line of output from **replace.cpp** looks like this:

```
I thought I saw Elvis in a wig. I have been
working too hard.
```

If replace doesn't find the search string, it returns **npos**. **npos** is a static constant member of the **basic_string** class.

Unlike **insert( )**, **replace( )** won't grow the **string**'s storage space if you copy new characters into the middle of an existing series of array elements. However, it *will* grow the storage space if you make a "replacement" that writes beyond the end of an existing array. Here's an example:

```
//: C17:ReplaceAndGrow.cpp
#include <string>
#include <iostream>
using namespace std;
```

```
int main() {
  string bigNews("I saw Elvis in a UFO. "
    "I have been working too hard.");
  string replacement("wig");
  // The first arg says "replace chars
  // beyond the end of the existing string":
  bigNews.replace(bigNews.size(),
    replacement.size(), replacement);
  cout << bigNews << endl;
} ///:~
```

The call to **replace( )** begins “replacing” beyond the end of the existing array. The output looks like this:

```
I saw Elvis in a UFO. I have
been working too hard.wig
```

Notice that **replace( )** expands the array to accommodate the growth of the string due to “replacement” beyond the bounds of the existing array.

Simple character replacement using the STL replace( ) algorithm

You may have been hunting through this chapter trying to do something relatively simple like replace all the instances of one character with a different character. Upon finding the above section on replacing, you thought you found the answer but then you started seeing groups of characters and counts and other things that looked a bit too complex. Doesn’t **string** have a way to just replace one character with another everywhere?

The answer comes in observing that the **string** class doesn’t solve all these problems alone. It contains a significant number of functions, but there other problems it doesn’t solve. These are relegated to the STL algorithms, and enabled because the **string** class can look just like an STL container, which is what the STL algorithms want to work with. All the STL algorithms specify a “range” of elements within a container that the algorithm will work upon. Usually that range is just “from the beginning of the container to the end.” A **string** object looks like a container of characters, and to get the beginning of the range you use **string::begin( )** and to get the end of the range you use **string::end( )**. The following example shows the use of the STL **replace( )** algorithm to replace all the instances of ‘X’ with ‘Y’:

```
//: C17:StringCharReplace.cpp
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
  string s("aaaXaaaXXaaXXXaXXXXaaa");
  cout << s << endl;
  replace(s.begin(), s.end(), 'X', 'Y');
  cout << s << endl;
} ///:~
```

Notice that this **replace( )** is *not* called as a member function of **string**. Also, unlike the **string::replace( )** functions which only perform one replacement, the STL replace is replacing all instances of one character with another.

The STL **replace( )** algorithm only works with single objects (in this case, **char** objects), and will not perform replacements of quoted **char** arrays or of **string** objects.

Since a **string** looks like an STL container, there are a number of other STL algorithms that can be applied to it, which may solve other problems you have that are not directly addressed by the **string** member functions. See Chapter XX for more information on the STL algorithms.

Concatenation using non-member overloaded operators

One of the most delightful discoveries awaiting a C programmer learning about C++ **string** handling is how simply **string**s can be combined and appended using **operator+** and **operator+=**. These operators make combining **string**s syntactically equivalent to adding numeric data.

```
//: C17:AddStrings.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1("This ");
  string s2("That ");
```

```
  string s3("The other ");
  // operator+ concatenates strings
  s1 = s1 + s2;
  cout << s1 << endl;
  // Another way to concatenates strings
  s1 += s3;
  cout << s1 << endl;
  // You can index the string on the right
  s1 += s3 + s3[4] + "oh lala";
  cout << s1 << endl;
} ///:~
```

The output looks like this:

```
This
This That
This That The other
This That The other ooh lala
```

operator+ and **operator+=** are a very flexible and convenient means of combining **string** data. On the right hand side of the statement, you can use almost any type that evaluates to a group of one or more characters.

Searching in strings

The **find** family of **string** member functions allows you to locate a character or group of characters within a given string. Here are the members of the **find** family and their general usage:

string find member function	What/how it finds
find( )	Searches a string for a specified character or group of characters and returns the starting position of the first occurrence found or **npos** (this member datum holds the current actual length of the string which is being searched) if no match is found.
find_first_of( )	Searches a target string and returns the position of the first match of *any* character in a specified group. If no match is found, it returns

	npos.
find_last_of( )	Searches a target string and returns the position of the last match of *any* character in a specified group. If no match is found, it returns **npos**.
find_first_not_of( )	Searches a target string and returns the position of the first element that *doesn't* match of *any* character in a specified group. If no such element is found, it returns **npos**.
find_last_not_of( )	Searches a target string and returns the position of the element with the largest subscript that *doesn't* match of *any* character in a specified group. If no such element is found, it returns **npos**.
rfind( )	Searches a string from end to beginning for a specified character or group of characters and returns the starting position of the match if one is found. If no match is found, it returns **npos**.

String searching member functions and their general uses

The simplest use of **find( )** searches for one or more characters in a **string**. This overloaded version of **find( )** takes a parameter that specifies the character(s) for which to search, and optionally one that tells it where in the string to begin searching for the occurrence of a substring. (The default position at which to begin searching is 0.) By setting the call to **find** inside a loop, you can easily move through a string, repeating a search in order to find all of the occurrences of a given character or group of characters within the string.

Notice that we define the string object **sieveChars** using a constructor idiom which sets the initial size of the character array and writes the value 'P' to each of its member.

```
//: C17:Sieve.cpp
#include <string>
#include <iostream>
using namespace std;
```

```
int main() {
  // Create a 50 char string and set each
  // element to 'P' for Prime
  string sieveChars(50, 'P');
  // By definition neither 0 nor 1 is prime.
  // Change these elements to "N" for Not Prime
  sieveChars.replace(0, 2, "NN");
  // Walk through the array:
  for(int i = 2;
    i <= (sieveChars.size() / 2) - 1; i++)
    // Find all the factors:
    for(int factor = 2;
      factor * i < sieveChars.size();factor++)
      sieveChars[factor * i] = 'N';

  cout << "Prime:" << endl;
  // Return the index of the first 'P' element:
  int j = sieveChars.find('P');
  // While not at the end of the string:
  while(j != sieveChars.npos) {
    // If the element is P, the index is a prime
    cout << j << " ";
    // Move past the last prime
    j++;
    // Find the next prime
    j = sieveChars.find('P', j);
  }
  cout << "\n Not prime:" << endl;
  // Find the first element value not equal P:
  j = sieveChars.find_first_not_of('P');
  while(j != sieveChars.npos) {
    cout << j << " ";
    j++;
    j = sieveChars.find_first_not_of('P', j);
  }
} ///:~
```

The output from **Sieve.cpp** looks like this:

```
Prime:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
Not prime:
```

```
0 1 4 6 8 9 10 12 14 15 16 18 20 21 22
24 25 26 27 28 30 32 33 34 35 36 38 39
40 42 44 45 46 48 49
```

find( ) allows you to walk forward through a **string**, detecting multiple occurrences of a character or group of characters, while **find_first_not_of( )** allows you to test for the absence of a character or group.

The **find** member is also useful for detecting the occurrence of a sequence of characters in a **string**:

```
//: C17:Find.cpp
// Find a group of characters in a string
#include <string>
#include <iostream>
using namespace std;

int main() {
  string chooseOne("Eenie, meenie, miney, mo");
  int i = chooseOne.find("een");
  while(i != string::npos) {
    cout << i << endl;
    i++;
    i = chooseOne.find("een", i);
  }
} ///:~
```

Find.cpp produces a single line of output :

```
 8
```

This tells us that the first 'e' of the search group "een" was found in the word "meenie," and is the eighth element in the string. Notice that **find** passed over the "Een" group of characters in the word "Eenie". The **find** member function performs a *case sensitive* search.

There are no functions in the **string** class to change the case of a string, but these functions can be easily created using the Standard C library functions **toupper( )** and **tolower( )**, which change the case of one character at a time. A few small changes will make **Find.cpp** perform a case insensitive search:

```
//: C17:NewFind.cpp
#include <string>
#include <iostream>
using namespace std;
```

```
// Make an uppercase copy of s:
string upperCase(string& s) {
  char* buf = new char[s.length()];
  s.copy(buf, s.length());
  for(int i = 0; i < s.length(); i++)
    buf[i] = toupper(buf[i]);
  string r(buf, s.length());
  delete buf;
  return r;
}

// Make a lowercase copy of s:
string lowerCase(string& s) {
  char* buf = new char[s.length()];
  s.copy(buf, s.length());
  for(int i = 0; i < s.length(); i++)
    buf[i] = tolower(buf[i]);
  string r(buf, s.length());
  delete buf;
  return r;
}

int main() {
  string chooseOne("Eenie, meenie, miney, mo");
  cout << chooseOne << endl;
  cout << upperCase(chooseOne) << endl;
  cout << lowerCase(chooseOne) << endl;
  // Case sensitive search
  int i = chooseOne.find("een");
  while(i != string::npos) {
    cout << i << endl;
    i++;
    i = chooseOne.find("een", i);
  }
  // Search lowercase:
  string lcase = lowerCase(chooseOne);
  cout << lcase << endl;
  i = lcase.find("een");
  while(i != lcase.npos) {
    cout << i << endl;
    i++;
```

```
    i = lcase.find("een", i);
  }
  // Search uppercase:
  string ucase = upperCase(chooseOne);
  cout << ucase << endl;
  i = ucase.find("EEN");
  while(i != ucase.npos) {
    cout << i << endl;
    i++;
    i = ucase.find("EEN", i);
  }
} ///:~
```

Both the **upperCase( )** and **lowerCase( )** functions follow the same form: they allocate storage to hold the data in the argument **string**, copy the data and change the case. Then they create a new **string** with the new data, release the buffer and return the result **string**. The **c_str( )** function cannot be used to produce a pointer to directly manipulate the data in the **string** because **c_str( )** returns a pointer to **const**. That is, you're not allowed to manipulate **string** data with a pointer, only with member functions. If you need to use the more primitive **char** array manipulation, you should use the technique shown above.

The output looks like this:

```
Eenie, meenie, miney, mo
eenie, meenie, miney, mo
EENIE, MEENIE, MINEY, MO
8
eenie, meenie, miney, mo
0
8
EENIE, MEENIE, MINEY, MO
0
8
```

The case insensitive searches found both occurrences on the "een" group.

NewFind.cpp isn't the best solution to the case sensitivity problem, so we'll revisit it when we examine **string** comparisons.

Finding in reverse

Sometimes it's necessary to search through a **string** from end to beginning, if you need to find the data in "last in / first out " order. The string member function **rfind( )** handles this job.

```
//: C17:Rparse.cpp
// Reverse the order of words in a string
#include <string>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  // The ';' characters will be delimiters
  string s("now.;sense;make;to;going;is;This");
  cout << s << endl;
  // To store the words:
  vector<string> strings;
  // The last element of the string:
  int last = s.size();
  // The beginning of the current word:
  int current = s.rfind(';');
  // Walk backward through the string:
  while(current != string::npos){
    // Push each word into the vector.
    // Current is incremented before copying to
    // avoid copying the delimiter.
    strings.push_back(
      s.substr(++current,last - current));
    // Back over the delimiter we just found,
    // and set last to the end of the next word
    current -= 2;
    last = current;
    // Find the next delimiter
    current = s.rfind(';', current);
  }
  // Pick up the first word - it's not
  // preceded by a delimiter
  strings.push_back(s.substr(0, last - current));
  // Print them in the new order:
  for(int j = 0; j < strings.size(); j++)
    cout << strings[j] << " ";
```

```
} ///:~
```

Here's how the output from **Rparse.cpp** looks:

```
now.;sense;make;to;going;is;This
This is going to make sense now.
```

rfind( ) backs through the string looking for tokens, reporting the array index of matching characters or **string::npos** if it is unsuccessful.

Finding first/last of a set

The **find_first_of( )** and **find_last_of( )** member functions can be conveniently put to work to create a little utility that will strip whitespace characters off of both ends of a string. Notice it doesn't touch the original string, but instead returns a new string:

```
//: C17:trim.h
#ifndef TRIM_H
#define TRIM_H
#include <string>
// General tool to strip spaces from both ends:
inline std::string trim(const std::string& s) {
  if(s.length() == 0)
    return s;
  int b = s.find_first_not_of(" \t");
  int e = s.find_last_not_of(" \t");
  if(b == -1) // No non-spaces
    return "";
  return std::string(s, b, e - b + 1);
}
#endif // TRIM_H ///:~
```

The first test checks for an empty **string**; in that case no tests are made and a copy is returned. Notice that once the end points are found, the **string** constructor is used to build a new **string** from the old one, giving the starting count and the length. This form also utilizes the "return value optimization" (see the index for more details).

Testing such a general-purpose tool needs to be thorough:

```
//: C17:TrimTest.cpp
#include "trim.h"
#include <iostream>
using namespace std;
```

```
string s[] = {
  " \t abcdefghijklmnop \t ",
  "abcdefghijklmnop \t ",
  " \t abcdefghijklmnop",
  "a", "ab", "abc", "a b c",
  " \t a b c \t ", " \t a \t b \t c \t ",
  "", // Must also test the empty string
};

void test(string s) {
  cout << "[" << trim(s) << "]" << endl;
}

int main() {
  for(int i = 0; i < sizeof s / sizeof *s; i++)
    test(s[i]);
} ///:~
```

In the array of **string s**, you can see that the character arrays are automatically converted to **string** objects. This array provides cases to check the removal of spaces and tabs from both ends, as well as ensuring that spaces and tabs do not get removed from the middle of a **string**.

Removing characters from strings

My word processor/page layout program (Microsoft Word) will save a document in HTML, but it doesn't recognize that the code listings in this book should be tagged with the HTML "preformatted" tag (<PRE>), and it puts paragraph marks (<P> and </P>) around every listing line. This means that all the indentation in the code listings is lost. In addition, Word saves HTML with reduced font sizes for body text, which makes it hard to read.

To convert the book to HTML form[49], then, the original output must be reprocessed, watching for the tags that mark the start and end of code listings, inserting the <PRE> and </PRE> tags at the appropriate places, removing all the <P> and </P> tags within the listings, and adjusting the font sizes. Removal is accomplished with the **erase( )** member function,

[49] I subsequently found better tools to accomplish this task, but the program is still interesting.

but you must correctly determine the starting and ending points of the substring you wish to erase. Here's the program that reprocesses the generated HTML file:

```
//: C17:ReprocessHTML.cpp
// Take Word's html output and fix up
// the code listings and html tags
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Produce a new string which is the original
// string with the html paragraph break marks
// stripped off:
string stripPBreaks(string s) {
  int br;
  while((br = s.find("<P>")) != string::npos)
    s.erase(br, strlen("<P>"));
  while((br = s.find("</P>")) != string::npos)
    s.erase(br, strlen("</P>"));
  return s;
}

// After the beginning of a code listing is
// detected, this function cleans up the listing
// until the end marker is found. The first line
// of the listing is passed in by the caller,
// which detects the start marker in the line.
void fixupCodeListing(istream& in,
  ostream& out, string& line, int tag) {
  out << line.substr(0, tag)
    << "<PRE>" // Means "preformatted" in html
    << stripPBreaks(line.substr(tag)) << endl;
  string s;
  while(getline(in, s)) {
    int endtag = s.find("/""/""/"":~");
    if(endtag != string::npos) {
      endtag += strlen("/""/""/"":~");
      string before = s.substr(0, endtag);
      string after = s.substr(endtag);
      out << stripPBreaks(before) << "</PRE>"
```

```
      << after << endl;
    return;
   }
   out << stripPBreaks(s) << endl;
  }
}

string removals[] = {
  "<FONT SIZE=2>",
  "<FONT SIZE=1>",
  "<FONT FACE=\"Times\" SIZE=1>",
  "<FONT FACE=\"Times\" SIZE=2>",
  "<FONT FACE=\"Courier\" SIZE=1>",
  "SIZE=1", // Eliminate all other '1' & '2' size
  "SIZE=2",
};
const int rmsz =
  sizeof(removals)/sizeof(*removals);

int main(int argc, char* argv[]) {
  requireArgs(argc, 2);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  ofstream out(argv[2]);
  string line;
  while(getline(in, line)) {
    // The "Body" tag only appears once:
    if(line.find("<BODY") != string::npos) {
      out << "<BODY BGCOLOR=\"#FFFFFF\" "
      "TEXT=\"#000000\">" << endl;
      continue; // Get next line
    }
    // Eliminate each of the removals strings:
    for(int i = 0; i < rmsz; i++) {
      int find = line.find(removals[i]);
      if(find != string::npos)
        line.erase(find, removals[i].size());
    }
    int tag1 = line.find("/""/"":");
    int tag2 = line.find("/""*"":");
    if(tag1 != string::npos)
      fixupCodeListing(in, out, line, tag1);
```

```
    else if(tag2 != string::npos)
      fixupCodeListing(in, out, line, tag2);
    else
      out << line << endl;
  }
} ///:~
```

Notice the lines that detect the start and end listing tags by indicating them with each character in quotes. These tags are treated in a special way by the logic in the **Extractcode.cpp** tool for extracting code listings. To present the code for the tool in the text of the book, the tag sequence itself must not occur in the listing. This was accomplished by taking advantage of a C++ preprocessor feature that causes text strings delimited by adjacent pairs of double quotes to be merged into a single string during the preprocessor pass of the build.

```
int tag1 = line.find("/""/"":");
```

The effect of the sequence of **char** arrays is to produce the starting tag for code listings.

Stripping HTML tags

Sometimes it's useful to take an HTML file and strip its tags so you have something approximating the text that would be displayed in the Web browser, only as an ASCII text file. The **string** class once again comes in handy. The following has some variation on the theme of the previous example:

```
//: C17:HTMLStripper.cpp
// Filter to remove html tags and markers
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

string replaceAll(string s, string f, string r) {
  unsigned int found = s.find(f);
  while(found != string::npos) {
    s.replace(found, f.length(), r);
    found = s.find(f);
  }
  return s;
}
```

```
string stripHTMLTags(string s) {
  while(true) {
    unsigned int left = s.find('<');
    unsigned int right = s.find('>');
    if(left==string::npos || right==string::npos)
      break;
    s = s.erase(left, right - left + 1);
  }
  s = replaceAll(s, "&lt;", "<");
  s = replaceAll(s, "&gt;", ">");
  s = replaceAll(s, "&amp;", "&");
  s = replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  const int sz = 4096;
  char buf[sz];
  while(in.getline(buf, sz)) {
    string s(buf);
    cout << stripHTMLTags(s) << endl;
  }
} ///:~
```

The **string** class can replace one string with another but there's no facility for replacing all the strings of one type with another, so the **replaceAll( )** function does this for you, inside a **while** loop that keeps finding the next instance of the find string **f**. That function is used inside **stripHTMLTags** after it uses **erase( )** to remove everything that appears inside angle braces ('**<**' and '**>**'). Note that I probably haven't gotten all the necessary replacement values, but you can see what to do (you might even put all the find-replace pairs in a table…). In **main( )** the arguments are checked, and the file is read and converted. It is sent to standard output so you must redirect it with '**>**' if you want to write it to a file.

Comparing strings

Comparing strings is inherently different than comparing numbers. Numbers have constant, universally meaningful values. To evaluate the relationship between the magnitude of two strings, you must make a *lexical comparison*. Lexical comparison means that when you test a character to see if it is “greater than” or “less than” another character, you are actually comparing the numeric representation of those characters as specified in the collating sequence of the character set being used. Most often, this will be the ASCII collating sequence, which assigns the printable characters for the English language numbers in the range from 32 to 127 decimal. In the ASCII collating sequence, the first “character” in the list is the space, followed by several common punctuation marks, and then uppercase and lowercase letters. With respect to the alphabet, this means that the letters nearer the front have lower ASCII values than those nearer the end. With these details in mind, it becomes easier to remember that when a lexical comparison that reports s1 is “greater than” s2, it simply means that when the two were compared, the first differing character in s1 came later in the alphabet than the character in that same position in s2.

C++ provides several ways to compare strings, and each has their advantages. The simplest to use are the non member overloaded operator functions **operator ==, operator != operator >, operator <, operator >=,** and **operator <=.**

```
//: C17:CompStr.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {
  // Strings to compare
  string s1("This ");
  string s2("That ");
  for(int i = 0; i< s1.size() &&
    i < s2.size(); i++)
    // See if the string elements are the same:
    if(s1[i] == s2[i])
      cout << s1[i] << "  " << i << endl;
  // Use the string inequality operators
  if(s1 != s2) {
    cout << "Strings aren't the same:" << " ";
```

```
    if(s1 > s2)
      cout << "s1 is > s2" << endl;
    else
      cout << "s2 is > s1" << endl;
  }
} ///:~
```

Here's the output from **CompStr.cpp**:

```
T 0
h 1
  4
Strings aren't the same: s1 is > s2
```

The overloaded comparison operators are useful for comparing both full strings and individual string elements.

Notice in the code fragment below the flexibility of argument types on both the left and right hand side of the comparison operators. The overloaded operator set allows the direct comparison of string objects, quoted literals, and pointers to C style strings.

```
// The lvalue is a quoted literal and
// the rvalue is a string
if("That " == s2)
  cout << "A match" << endl;
// The lvalue is a string and the rvalue is a
// pointer to a c style null terminated string
if(s1 != s2.c_str())
  cout << "No match" << endl;
```

You won't find the logical not (!) or the logical comparison operators (&& and ||) among operators for string. (Neither will you find overloaded versions of the bitwise C operators &, |, ^, or ~.) The overloaded non member comparison operators for the string class are limited to the subset which has clear, unambiguous application to single characters or groups of characters.

The **compare( )** member function offers you a great deal more sophisticated and precise comparison than the non member operator set, because it returns a lexical comparison value, and provides for comparisons that consider subsets of the string data. It provides overloaded versions that allow you to compare two complete strings, part of either string to a complete string, and subsets of two strings. This example compares complete strings:

```
//: C17:Compare.cpp
// Demonstrates compare(), swap()
#include <string>
#include <iostream>
using namespace std;

int main() {
  string first("This");
  string second("That");
  // Which is lexically greater?
  switch(first.compare(second)) {
    case 0: // The same
      cout << first << " and " << second <<
        " are lexically equal" << endl;
      break;
    case -1: // Less than
      first.swap(second);
      // Fall through this case...
    case 1: // Greater than
      cout << first <<
        " is lexically greater than " <<
        second << endl;
  }
} ///:~
```

The output from **Compare.cpp** looks like this:

```
This is lexically greater than That
```

To compare a subset of the characters in one or both strings, you add arguments that define where to start the comparison and how many characters to consider. For example, we can use the overloaded version of **compare( )**:

s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars);

If we substitute the above version of **compare( )** in the previous program so that it only looks at the first two characters of each string, the program becomes:

```
//: C17:Compare2.cpp
// Overloaded compare()
#include <string>
#include <iostream>
```

```
using namespace std;

int main() {
  string first("This");
  string second("That");
  // Compare first two characters of each string:
  switch(first.compare(0, 2, second, 0, 2)) {
    case 0: // The same
      cout << first << " and " << second <<
        " are lexically equal" << endl;
      break;
    case -1: // Less than
      first.swap(second);
      // Fall through this case...
    case 1: // Greater than
      cout << first <<
        " is lexically greater than " <<
        second << endl;
  }
} ///:~
```

The output is:

```
This and That are lexically equal
```

which is true, for the first two characters of “This” and “That.”

Indexing with **[ ]** vs. **at( )**

In the examples so far, we have used C style array indexing syntax to refer to an individual character in a string. C++ strings provide an alternative to the **s[n]** notation: the **at( )** member. These two idioms produce the same result in C++ if all goes well:

```
//: C17:StringIndexing.cpp
#include <string>
#include <iostream>
using namespace std;
int main(){
  string s("1234");
  cout << s[1] << " ";
  cout << s.at(1) << endl;
} ///:~
```

The output from this code looks like this:

```
2 2
```

However, there is one important difference between **[ ]** and **at( )**. When you try to reference an array element that is out of bounds, **at( )** will do you the kindness of throwing an exception, while ordinary **[ ]** subscripting syntax will leave you to your own devices:

```
//: C17:BadStringIndexing.cpp
#include <string>
#include <iostream>
using namespace std;

int main(){
  string s("1234");
  // Runtime problem: goes beyond array bounds:
  cout << s[5] << endl;
  // Saves you by throwing an exception:
  cout << s.at(5) << endl;
} ///:~
```

Using **at( )** in place of **[ ]** will give you a chance to gracefully recover from references to array elements that don't exist. **at( )** throws an object of class **out_of_range.** By catching this object in an exception handler, you can take appropriate remedial actions such as recalculating the offending subscript or growing the array. (You can read more about Exception Handling in Chapter XX)

Using iterators

In the example program **NewFind.cpp**, we used a lot of messy and rather tedious C **char** array handling code to change the case of the characters in a string and then search for the occurrence of matches to a substring. Sometimes the "quick and dirty" method is justifiable, but in general, you won't want to sacrifice the advantages of having your string data safely and securely encapsulated in the C++ object where it lives.

Here is a better, safer way to handle case insensitive comparison of two C++ string objects. Because no data is copied out of the objects and into C style strings, you don't have to use pointers and you don't have to risk overwriting the bounds of an ordinary character array. In this example, we use the string **iterator**. Iterators are themselves objects which move through a collection or container of other objects, selecting them one at a time, but never providing direct access to the implementation of the

container. Iterators are *not* pointers, but they are useful for many of the same jobs.

```
//: C17:CmpIter.cpp
// Find a group of characters in a string
#include <string>
#include <iostream>
using namespace std;

// Case insensitive compare function:
int
stringCmpi(const string& s1, const string& s2) {
  // Select the first element of each string:
  string::const_iterator
    p1 = s1.begin(), p2 = s2.begin();
  // Don't run past the end:
  while(p1 != s1.end() && p2 != s2.end()) {
    // Compare upper-cased chars:
    if(toupper(*p1) != toupper(*p2))
      // Report which was lexically  greater:
      return (toupper(*p1)<toupper(*p2))? -1 : 1;
    p1++;
    p2++;
  }
  // If they match up to the detected eos, say
  // which was longer. Return 0 if the same.
  return(s2.size() - s1.size());
}

int main() {
  string s1("Mozart");
  string s2("Modigliani");
  cout << stringCmpi(s1, s2) << endl;
} ///:~
```

Notice that the iterators **p1** and **p2** use the same syntax as C pointers – the '*****' operator makes the *value of* element at the location given by the iterators available to the **toupper( )** function. **toupper( )** doesn't actually change the content of the element in the string. In fact, it can't. This definition of **p1** tells us that we can only use the elements **p1** points to as constants.

```
string::const_iterator p1 = s1.begin();
```

The way **toupper( )** and the iterators are used in this example is called a *case preserving* case insensitive comparison. This means that the string didn't have to be copied or rewritten to accommodate case insensitive comparison. Both of the strings retain their original data, unmodified.

Iterating in reverse

Just as the standard C pointer gives us the increment (++) and decrement (--) operators to make pointer arithmetic a bit more convenient, C++ string iterators come in two basic varieties. You've seen **end( )** and **begin( )**, which are the tools for moving forward through a string one element at a time. The reverse iterators **rend( )** and **rbegin( )** allow you to step backwards through a string. Here's how they work:

```
//: C17:RevStr.cpp
// Print a string in reverse
#include <string>
#include <iostream>
using namespace std;
int main() {
  string s("987654321");
  // Use this iterator to walk backwards:
  string::reverse_iterator rev;
  // "Incrementing" the reverse iterator moves
  // it to successively lower string elements:
  for(rev = s.rbegin(); rev != s.rend(); rev++)
    cout << *rev << " ";
} ///:~
```

The output from **RevStr.cpp** looks like this:

```
1 2 3 4 5 6 7 8 9
```

Reverse iterators act like pointers to elements of the string's character array, *except that when you apply the increment operator to them, they move backward rather than forward*. **rbegin( )** and **rend( )** supply string locations that are consistent with this behavior, to wit, **rbegin( )** locates the position just beyond the end of the string, and **rend( )** locates the beginning. Aside from this, the main thing to remember about reverse iterators is that they *aren't* type equivalent to ordinary iterators. For example, if a member function parameter list includes an iterator as an argument, you can't substitute a reverse iterator to get the function to perform it's job walking backward through the string. Here's an illustration:

```
// The compiler won't accept this
string sBackwards(s.rbegin(), s.rend());
```

The string constructor won't accept reverse iterators in place of forward iterators in its parameter list. This is also true of string members such as **copy( )**, **insert( ),** and **assign( ).**

Strings and character traits

We seem to have worked our way around the margins of case insensitive string comparisons using C++ string objects, so maybe it's time to ask the obvious question: "Why isn't case-insensitive comparison part of the standard **string** class ?" The answer provides interesting background on the true nature of C++ string objects.

Consider what it means for a character to have "case." Written Hebrew, Farsi, and Kanji don't use the concept of upper and lower case, so for those languages this idea has no meaning at all. This the first impediment to built-in C++ support for case-insensitive character search and comparison: the idea of case sensitivity is not universal, and therefore not portable.

It would seem that if there were a way of designating that some languages were "all uppercase" or "all lowercase" we could design a generalized solution. However, some languages which employ the concept of "case" *also* change the meaning of particular characters with diacritical marks: the cedilla in Spanish, the circumflex in French, and the umlaut in German. For this reason, any case-sensitive collating scheme that attempts to be comprehensive will be nightmarishly complex to use.

Although we usually treat the C++ **string** as a class, this is really not the case. **string** is a **typedef** of a more general constituent, the **basic_string< >** template. Observe how **string** is declared in the standard C++ header file:

```
typedef basic_string<char> string;
```

To really understand the nature of strings, it's helpful to delve a bit deeper and look at the template on which it is based. Here's the declaration of the **basic_string< >** template:

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

Earlier in this book, templates were examined in a great deal of detail. The main thing to notice about the two declarations above are that the **string** type is created when the **basic_string** template is instantiated with **char.** Inside the **basic_string< >** template declaration, the line

```
class traits = char_traits<charT>,
```

tells us that the behavior of the class made from the **basic_string< >** template is specified by a class based on the template **char_traits< >**. Thus, the **basic_string< >** template provides for cases where you need string oriented classes that manipulate types other than **char** (wide characters or unicode, for example). To do this, the **char_traits< >** template controls the content and collating behaviors of a variety of character sets using the character comparison functions **eq( )** (equal), **ne( )** (not equal), and **lt( )** (less than) upon which the **basic_string< >** string comparison functions rely.

This is why the string class doesn't include case insensitive member functions: That's not in its job description. To change the way the string class treats character comparison, you must supply a different of **char_traits< >** template, because that defines the behavior of the individual character comparison member functions.

This information can be used to make a new type of **string** class that ignores case. First, we'll define a new case insensitive **char_traits< >** template that inherits the existing one. Next, we'll override only the members we need to change in order to make character-by-character comparison case insensitive. (In addition to the three lexical character comparison members mentioned above, we'll also have to supply new implementation of **find( )** and **compare( )**.) Finally, we'll **typedef** a new class based on **basic_string**, but using the case insensitive **ichar_traits** template for its second argument.

```
//: C17:ichar_traits.h
// Creating your own character traits
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <string>
#include <cctype>

struct ichar_traits : std::char_traits<char> {
  // We'll only change character by
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return
```

```
    std::toupper(c1st) == std::toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return
      std::toupper(c1st) != std::toupper(c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return
      std::toupper(c1st) < std::toupper(c2nd);
  }
  static int compare(const char* str1,
    const char* str2, size_t n) {
    for(int i = 0; i < n; i++) {
      if(std::tolower(*str1)>std::tolower(*str2))
        return 1;
      if(std::tolower(*str1)<std::tolower(*str2))
        return -1;
      if(*str1 == 0 || *str2 == 0)
        return 0;
      str1++; str2++; // Compare the other chars
    }
    return 0;
  }
  static const char* find(const char* s1,
    int  n, char c) {
    while(n-- > 0 &&
      std::toupper(*s1) != std::toupper(c))
      s1++;
    return s1;
  }
};
#endif // ICHAR_TRAITS_H  ///:~
```

If we **typedef** an **istring** class like this:

```
typedef basic_string<char, ichar_traits,
  allocator<char> > istring;
```

Then this **istring** will act like an ordinary **string** in every way, except that it will make all comparisons without respect to case. Here's an example:

```
//: C17:ICompare.cpp
#include "ichar_traits.h"
#include <string>
```

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

typedef basic_string<char, ichar_traits,
  allocator<char> > istring;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first.compare(second) << endl;
} ///:~
```

The output from the program is “0”, indicating that the strings compare as equal. This is just a simple example – in order to make **istring** fully equivalent to **string**, we’d have to create the other functions necessary to support the new **istring** type.

A string application

My friend Daniel (who designed the cover and page layout for this book) does a lot of work with Web pages. One tool he uses creates a “site map” consisting of a Java applet to display the map and an HTML tag that invoked the applet and provided it with the necessary data to create the map. Daniel wanted to use this data to create an ordinary HTML page (sans applet) that would contain regular links as the site map. The resulting program turns out to be a nice practical application of the **string** class, so it is presented here.

The input is an HTML file that contains the usual stuff along with an applet tag with a parameter that begins like this:

```
<param name="source_file" value="
```

The rest of the line contains encoded information about the site map, all combined into a single line (it’s rather long, but fortunately **string** objects don’t care). Each entry may or may not begin with a number of ‘**#**’ signs; each of these indicates one level of depth. If no ‘**#**’ sign is present the entry will be considered to be at level one. After the ‘**#**’ is the text to be displayed on the page, followed by a ‘**%**’ and the URL to use as the link. Each entry is terminated by a ‘*****’. Thus, a single entry in the line might look like this:

```
###|Useful Art%./Build/useful_art.html*
```

The '**|**' at the beginning is an artifact that needs to be removed.

My solution was to create an **Item** class whose constructor would take input text and create an object that contains the text to be displayed, the URL and the level. The objects essentially parse themselves, and at that point you can read any value you want. In **main( )**, the input file is opened and read until the line contains the parameter that we're interested in. Everything but the site map codes are stripped away from this **string**, and then it is parsed into **Item** objects:

```
//: C17:SiteMapConvert.cpp
// Using strings to create a custom conversion
// program that generates HTML output
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;

class Item {
  string id, url;
  int depth;
  string removeBar(string s) {
    if(s[0] == '|')
      return s.substr(1);
    else return s;
  }
public:
  Item(string in, int& index) : depth(0) {
    while(in[index] == '#' && index < in.size()){
      depth++;
      index++;
    }
    // 0 means no '#' marks were found:
    if(depth == 0) depth = 1;
    while(in[index] != '%' && index < in.size())
      id += in[index++];
    id = removeBar(id);
    index++; // Move past '%'
    while(in[index] != '*' && index < in.size())
      url += in[index++];
    url = removeBar(url);
```

```
    index++; // To move past '*'
  }
  string identifier() { return id; }
  string path() { return url; }
  int level() { return depth; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: SiteMapConvert inputfilename");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  ofstream out("plainmap.html");
  string line;
  while(getline(in, line)) {
    if(line.find("<param name=\"source_file\"")
       != string::npos) {
      // Extract data of from start of sequence
      // until the terminating quote mark:
      line = line.substr(line.find("value=\"")
            + string("value=\"").size());
      line = line.substr(0,
             line.find_last_of("\""));
      int index = 0;
      while(index < line.size()) {
        Item item(line, index);
        string startLevel, endLevel;
        if(item.level() == 1) {
          startLevel = "<h1>";
          endLevel = "</h1>";
        } else
          for(int i = 0; i < item.level(); i++)
            for(int j = 0; j < 5; j++)
              out << " ";
        string htmlLine = "<a href=\""
          + item.path() + "\">"
          + item.identifier() + "</a><br>";
        out << startLevel << htmlLine
            << endLevel << endl;
      }
      break; // Out of while loop
    }
```

```
  }
} ///:~
```

Item contains a private member function **removeBar( )** that is used internally to strip off the leading bars, if they appear.

The constructor for **Item** initializes **depth** to **0** to indicate that no '#' signs were found yet; if none are found then it is assumed the **Item** should be displayed at level one. Each character in the **string** is examined using **operator[ ]** to find the **depth**, **id** and **url** values. The other member functions simply return these values.

After opening the files, **main( )** uses **string::find( )** to locate the line containing the site map data. At this point, **substr( )** is used in order to strip off the information before and after the site map data. The subsequent **while** loop performs the parsing, but notice that the value **index** is passed *by reference* into the **Item** constructor, and that constructor increments **index** as it parses each new **Item**, thus moving forward in the sequence.

If an **Item** is at level one, then an HTML **h1** tag is used, otherwise the elements are indented using HTML non-breaking spaces. Note in the initialization of **htmlLine** how easy it is to construct a **string** – you can just combine quoted character arrays and other **string** objects using **operator+**.

When the output is written to the destination file, **startLevel** and **endLevel** will only produce results if they have been given any value other than their default initialization values.

Summary

C++ string objects provide developers with a number of great advantages over their C counterparts. For the most part, the **string** class makes referring to strings through the use of character pointers unnecessary. This eliminates an entire class of software defects that arise from the use of uninitialized and incorrectly valued pointers. C++ strings dynamically and transparently grow their internal data storage space to accommodate increases in the size of the string data. This means that when the data in a string grows beyond the limits of the memory initially allocated to it, the string object will make the memory management calls that take space from and return space to the heap. Consistent allocation schemes prevent memory leaks and have the potential to be much more efficient than "roll your own" memory management.

The **string** class member functions provide a fairly comprehensive set of tools for creating, modifying, and searching in strings. **string** comparisons are always case sensitive, but you can work around this by copying string data to C style null terminated strings and using case insensitive string comparison functions, temporarily converting the data held in sting objects to a single case, or by creating a case insensitive string class which overrides the character traits used to create the **basic_string** object.

Exercises

1. A palindrome is a word or group of words that read the same forward and backward. For example “madam” or “wow”. Write a program that takes a string argument from the command line and returns TRUE if the string was a palindrome.
2. Sometimes the input from a file stream contains a two character sequence to represent a newline. These two characters (0x0a 0x0d) produce extra blank lines when the stream is printed to standard out. Write a program that finds the character 0x0d (ASCII carriage return) and deletes it from the string.
3. Write a program that reverses the order of the characters in a string.

18: Iostreams

So far in this book we've used the old reliable C standard I/O library, a perfect example of a library that begs to be turned into a class.

In fact, there's much more you can do with the general I/O problem than just take standard I/O and turn it into a class. Wouldn't it be nice if you could make all the usual "receptacles" – standard I/O, files and even blocks of memory – look the same, so you need to remember only one interface? That's the idea behind iostreams. They're much easier, safer, and often more efficient than the assorted functions from the Standard C stdio library.

Iostream is usually the first class library that new C++ programmers learn to use. This chapter explores the *use* of iostreams, so they can replace the C I/O functions through the rest of the book. In future chapters, you'll see how to set up your own classes so they're compatible with iostreams.

Why iostreams?

You may wonder what's wrong with the good old C library. And why not "wrap" the C library in a class and be done with it? Indeed, there are situations when this is the perfect thing to do, when you want to make a C library a bit safer and easier to use. For example, suppose you want to make sure a stdio file is always safely opened and properly closed, without relying on the user to remember to call the **close( )** function:

```
//: C18:FileClass.h
// Stdio files wrapped
#ifndef FILECLAS_H
#define FILECLAS_H
#include <cstdio>

class FileClass {
  std::FILE* f;
public:
```

```
  FileClass(const char* fname, const char* mode="r");
  ~FileClass();
  std::FILE* fp();
};
#endif // FILECLAS_H ///:~
```

In C when you perform file I/O, you work with a naked pointer to a FILE **struct**, but this class wraps around the pointer and guarantees it is properly initialized and cleaned up using the constructor and destructor. The second constructor argument is the file mode, which defaults to "r" for "read."

To fetch the value of the pointer to use in the file I/O functions, you use the **fp( )** access function. Here are the member function definitions:

```
//: C18:FileClass.cpp {O}
// Implementation
#include "FileClass.h"
#include <cstdlib>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode){
  f = fopen(fname, mode);
  if(f == NULL) {
    printf("%s: file not found\n", fname);
    exit(1);
  }
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

The constructor calls **fopen( )**,as you would normally do, but it also checks to ensure the result isn't zero, which indicates a failure upon opening the file. If there's a failure, the name of the file is printed and **exit( )** is called.

The destructor closes the file, and the access function **fp( )**returns **f**. Here's a simple example using **class FileClass**:

```
//: C18:FileClassTest.cpp
//{L} FileClass
// Testing class File
#include "FileClass.h"
```

```
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  FileClass f(argv[1]); // Opens and tests
  const int bsize = 100;
  char buf[bsize];
  while(fgets(buf, bsize, f.fp()))
    puts(buf);
} // File automatically closed by destructor
///:~
```

You create the **FileClass** object and use it in normal C file I/O function calls by calling **fp( )**. When you're done with it, just forget about it, and the file is closed by the destructor at the end of the scope.

True wrapping

Even though the FILE pointer is private, it isn't particularly safe because **fp( )** retrieves it. The only effect seems to be guaranteed initialization and cleanup, so why not make it public, or use a **struct** instead? Notice that while you can get a copy of **f** using **fp( )**, you cannot assign to **f** – that's completely under the control of the class. Of course, after capturing the pointer returned by **fp( )**, the client programmer can still assign to the structure elements, so the safety is in guaranteeing a valid FILE pointer rather than proper contents of the structure.

If you want complete safety, you have to prevent the user from direct access to the FILE pointer. This means some version of all the normal file I/O functions will have to show up as class members, so everything you can do with the C approach is available in the C++ class:

```
//: C18:Fullwrap.h
// Completely hidden file IO
#ifndef FULLWRAP_H
#define FULLWRAP_H

class File {
  std::FILE* f;
  std::FILE* F(); // Produces checked pointer to f
public:
  File(); // Create object but don't open file
  File(const char* path,
```

```
    const char* mode = "r");
  ~File();
  int open(const char* path,
      const char* mode = "r");
  int reopen(const char* path,
       const char* mode);
  int getc();
  int ungetc(int c);
  int putc(int c);
  int puts(const char* s);
  char* gets(char* s, int n);
  int printf(const char* format, ...);
  size_t read(void* ptr, size_t size,
         size_t n);
  size_t write(const void* ptr,
          size_t size, size_t n);
  int eof();
  int close();
  int flush();
  int seek(long offset, int whence);
  int getpos(fpos_t* pos);
  int setpos(const fpos_t* pos);
  long tell();
  void rewind();
  void setbuf(char* buf);
  int setvbuf(char* buf, int type, size_t sz);
  int error();
  void clearErr();
};
#endif // FULLWRAP_H ///:~
```

This class contains almost all the file I/O functions from **cstdio**. **vfprintf( )** is missing; it is used to implement the **printf( )** member function.

File has the same constructor as in the previous example, and it also has a default constructor. The default constructor is important if you want to create an array of **File** objects or use a **File** object as a member of another class where the initialization doesn't happen in the constructor (but sometime after the enclosing object is created).

The default constructor sets the private **FILE** pointer **f** to zero. But now, before any reference to **f**, its value must be checked to ensure it isn't zero. This is accomplished with the last member function in the class,

F( ), which is **private** because it is intended to be used only by other member functions. (We don't want to give the user direct access to the **FILE** structure in this class.)[50]

This is not a terrible solution by any means. It's quite functional, and you could imagine making similar classes for standard (console) I/O and for in-core formatting (reading/writing a piece of memory rather than a file or the console).

The big stumbling block is the runtime interpreter used for the variable-argument list functions. This is the code that parses through your format string at runtime and grabs and interprets arguments from the variable argument list. It's a problem for four reasons.

1. Even if you use only a fraction of the functionality of the interpreter, the whole thing gets loaded. So if you say:
   printf("%c", 'x');
   you'll get the whole package, including the parts that print out floating-point numbers and strings. There's no option for reducing the amount of space used by the program.
2. Because the interpretation happens at runtime there's a performance overhead you can't get rid of. It's frustrating because all the information is *there* in the format string at compile time, but it's not evaluated until runtime. However, if you could parse the arguments in the format string at compile time you could make hard function calls that have the potential to be much faster than a runtime interpreter (although the **printf( )** family of functions is usually quite well optimized).
3. A worse problem occurs because the evaluation of the format string doesn't happen until runtime: there can be no compile-time error checking. You're probably very familiar with this problem if you've tried to find bugs that came from using the wrong number or type of arguments in a **printf( )** statement. C++ makes a big deal out of compile-time error checking to find errors early and make your life easier. It seems a shame to throw it away for an I/O library, especially because I/O is used a lot.

[50] The implementation and test files for FULLWRAP are available in the freely distributed source code for this book. See preface for details.

4. For C++, the most important problem is that the **printf( )** family of functions is not particularly extensible. They're really designed to handle the four basic data types in C (**char**, **int**, **float**, **double** and their variations). You might think that every time you add a new class, you could add an overloaded **printf( )** and **scanf( )** function (and their variants for files and strings) but remember, overloaded functions must have different types in their argument lists and the **printf( )** family hides its type information in the format string and in the variable argument list. For a language like C++, whose goal is to be able to easily add new data types, this is an ungainly restriction.

Iostreams to the rescue

All these issues make it clear that one of the first standard class libraries for C++ should handle I/O. Because "hello, world" is the first program just about everyone writes in a new language, and because I/O is part of virtually every program, the I/O library in C++ must be particularly easy to use. It also has the much greater challenge that it can never know all the classes it must accommodate, but it must nevertheless be adaptable to use any new class. Thus its constraints required that this first class be a truly inspired design.

This chapter won't look at the details of the design and how to add iostream functionality to your own classes (you'll learn that in a later chapter). First, you need to learn to use iostreams. In addition to gaining a great deal of leverage and clarity in your dealings with I/O and formatting, you'll also see how a really powerful C++ library can work.

Sneak preview of operator overloading

Before you can use the iostreams library, you must understand one new feature of the language that won't be covered in detail until a later chapter. To use iostreams, you need to know that in C++ all the operators can take on different meanings. In this chapter, we're particularly interested in **<<** and **>>**. The statement "operators can take on different meanings" deserves some extra insight.

In Chapter XX, you learned how function overloading allows you to use the same function name with different argument lists. Now imagine that when the compiler sees an expression consisting of an argument followed by an operator followed by an argument, it simply calls a function. That is, an operator is simply a function call with a different syntax.

Of course, this is C++, which is very particular about data types. So there must be a previously declared function to match that operator and those particular argument types, or the compiler will not accept the expression.

What most people find immediately disturbing about operator overloading is the thought that maybe everything they know about operators in C is suddenly wrong. This is absolutely false. Here are two of the sacred design goals of C++:

1. A program that compiles in C will compile in C++. The only compilation errors and warnings from the C++ compiler will result from the "holes" in the C language, and fixing these will require only local editing. (Indeed, the complaints by the C++ compiler usually lead you directly to undiscovered bugs in the C program.)
2. The C++ compiler will not secretly change the behavior of a C program by recompiling it under C++.

Keeping these goals in mind will help answer a lot of questions; knowing there are no capricious changes to C when moving to C++ helps make the transition easy. In particular, operators for built-in types won't suddenly start working differently – you cannot change their meaning. Overloaded operators can be created only where new data types are involved. So you can create a new overloaded operator for a new class, but the expression

```
1 << 4;
```

won't suddenly change its meaning, and the illegal code

```
1.414 << 1;
```

won't suddenly start working.

Inserters and extractors

In the iostreams library, two operators have been overloaded to make the use of iostreams easy. The operator **<<** is often referred to as an *inserter* for iostreams, and the operator **>>** is often referred to as an *extractor*.

A *stream* is an object that formats and holds bytes. You can have an input stream (*istream*) or an output stream (*ostream*). There are different types of istreams and ostreams: *ifstreams* and *ofstreams* for files, *istrstreams* , and *ostrstreams* for **char*** memory (in-core formatting), and *istringstreams* & *ostringstreams* for interfacing with the Standard C++ **string** class. All these stream objects have the same interface, regardless of whether you're working with a file, standard I/O, a piece of memory or a **string** object. The single interface you learn also works for extensions added to support new classes.

If a stream is capable of producing bytes (an istream), you can get information from the stream using an extractor. The extractor produces and formats the type of information that's expected by the destination object. To see an example of this, you can use the **cin** object, which is the iostream equivalent of **stdin** in C, that is, redirectable standard input. This object is pre-defined whenever you include the **iostream.h** header file. (Thus, the iostream library is automatically linked with most compilers.)

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

There's an overloaded **operator >>** for every data type you can use as the right-hand argument of **>>** in an iostream statement. (You can also overload your own, which you'll see in a later chapter.)

To find out what you have in the various variables, you can use the **cout** object (corresponding to standard output; there's also a **cerr** object corresponding to standard error) with the inserter **<<**:

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
```

```
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

This is notably tedious, and doesn't seem like much of an improvement over **printf( )**, type checking or no. Fortunately, the overloaded inserters and extractors in iostreams are designed to be chained together into a complex expression that is much easier to write:

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

You'll understand how this can happen in a later chapter, but for now it's sufficient to take the attitude of a class user and just know it works that way.

Manipulators

One new element has been added here: a *manipulator* called **endl**. A manipulator acts on the stream itself; in this case it inserts a newline and *flushes* the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output). You can also just flush the stream:

```
cout << flush;
```

There are additional basic manipulators that will change the number base to **oct** (octal), **dec** (decimal) or **hex** (hexadecimal):

```
cout << hex << "0x" << i << endl;
```

There's a manipulator for extraction that "eats" white space:

```
cin >> ws;
```

and a manipulator called **ends**, which is like **endl**, only for strstreams (covered in a while). These are all the manipulators in **<iostream>**, but there are more in **<iomanip>** you'll see later in the chapter.

Common usage

Although **cin** and the extractor **>>** provide a nice balance to **cout** and the inserter **<<**, in practice using formatted input routines, especially with

standard input, has the same problems you run into with **scanf( )**. If the input produces an unexpected value, the process is skewed, and it's very difficult to recover. In addition, formatted input defaults to whitespace delimiters. So if you collect the above code fragments into a program

```
//: C18:Iosexamp.cpp
// Iostream examples
#include <iostream>
using namespace std;

int main() {
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
} ///:~
```

and give it the following input,

```
12 1.4 c this is a test
```

you'll get the same output as if you give it

```
12
1.4
c
this is a test
```

and the output is, somewhat unexpectedly,

```
i = 12
```

```
f = 1.4
c = c
buf = this
0xc
```

Notice that **buf** got only the first word because the input routine looked for a space to delimit the input, which it saw after "this." In addition, if the continuous input string is longer than the storage allocated for **buf**, you'll overrun the buffer.

It seems **cin** and the extractor are provided only for completeness, and this is probably a good way to look at it. In practice, you'll usually want to get your input a line at a time as a sequence of characters and then scan them and perform conversions once they're safely in a buffer. This way you don't have to worry about the input routine choking on unexpected data.

Another thing to consider is the whole concept of a command-line interface. This has made sense in the past when the console was little more than a glass typewriter, but the world is rapidly changing to one where the graphical user interface (GUI) dominates. What is the meaning of console I/O in such a world? It makes much more sense to ignore **cin** altogether other than for very simple examples or tests, and take the following approaches:

1. If your program requires input, read that input from a file – you'll soon see it's remarkably easy to use files with iostreams. Iostreams for files still works fine with a GUI.
2. Read the input without attempting to convert it. Once the input is someplace where it can't foul things up during conversion, then you can safely scan it.
3. Output is different. If you're using a GUI, **cout** doesn't work and you must send it to a file (which is identical to sending it to **cout**) or use the GUI facilities for data display. Otherwise it often makes sense to send it to **cout**. In both cases, the output formatting functions of iostreams are highly useful.

Line-oriented input

To grab input a line at a time, you have two choices: the member functions **get( )** and **getline( )**. Both functions take three arguments: a pointer to a character buffer in which to store the result, the size of that

buffer (so they don't overrun it), and the terminating character, to know when to stop reading input. The terminating character has a default value of **'\n'**, which is what you'll usually use. Both functions store a zero in the result buffer when they encounter the terminating character in the input.

So what's the difference? Subtle, but important: **get( )** stops when it *sees* the delimiter in the input stream, but it doesn't extract it from the input stream. Thus, if you did another **get( )** using the same delimiter it would immediately return with no fetched input. (Presumably, you either use a different delimiter in the next **get( )** statement or a different input function.) **getline( )**, on the other hand, extracts the delimiter from the input stream, but still doesn't store it in the result buffer.

Generally, when you're processing a text file that you read a line at a time, you'll want to use **getline( )**.

Overloaded versions of **get( )**

get( ) also comes in three other overloaded versions: one with no arguments that returns the next character, using an **int** return value; one that stuffs a character into its **char** argument, using a *reference* (You'll have to jump forward to Chapter XX if you want to understand it right this minute . . . .); and one that stores directly into the underlying buffer structure of another iostream object. That is explored later in the chapter.

Reading raw bytes

If you know exactly what you're dealing with and want to move the bytes directly into a variable, array, or structure in memory, you can use **read( )**. The first argument is a pointer to the destination memory, and the second is the number of bytes to read. This is especially useful if you've previously stored the information to a file, for example, in binary form using the complementary **write( )** member function for an output stream. You'll see examples of all these functions later.

Error handling

All the versions of **get( )** and **getline( )** return the input stream from which the characters came *except* for **get( )** with no arguments, which returns the next character or EOF. If you get the input stream object back, you can ask it if it's still OK. In fact, you can ask *any* iostream object if it's OK using the member functions **good( )**, **eof( )**, **fail( )**, and **bad( )**. These return state information based on the **eofbit** (indicates the buffer is at the end of sequence), the **failbit** (indicates some operation has failed because of formatting issues or some other problem that does

not affect the buffer) and the **badbit** (indicates something has gone wrong with the buffer).

However, as mentioned earlier, the state of an input stream generally gets corrupted in weird ways only when you're trying to do input to specific types and the type read from the input is inconsistent with what is expected. Then of course you have the problem of what to do with the input stream to correct the problem. If you follow my advice and read input a line at a time or as a big glob of characters (with **read( )**) and don't attempt to use the input formatting functions except in simple cases, then all you're concerned with is whether you're at the end of the input (EOF). Fortunately, testing for this turns out to be simple and can be done inside of conditionals, such as **while(cin)** or **if(cin)**. For now you'll have to accept that when you use an input stream object in this context, the right value is safely, correctly and magically produced to indicate whether the object has reached the end of the input. You can also use the Boolean NOT operator **!**, as in **if(!cin)**, to indicate the stream is *not* OK; that is, you've probably reached the end of input and should quit trying to read the stream.

There are times when the stream becomes not-OK, but you understand this condition and want to go on using it. For example, if you reach the end of an input file, the **eofbit** and **failbit** are set, so a conditional on that stream object will indicate the stream is no longer good. However, you may want to continue using the file, by seeking to an earlier position and reading more data. To correct the condition, simply call the **clear( )** member function.[51]

File iostreams

Manipulating files with iostreams is much easier and safer than using **cstdio** in C. All you do to open a file is create an object; the constructor does the work. You don't have to explicitly close a file (although you can, using the **close( )** member function) because the destructor will close it when the object goes out of scope.

To create a file that defaults to input, make an **ifstream** object. To create one that defaults to output, make an **ofstream** object.

---

[51] Newer implementations of iostreams will still support this style of handling errors, but in some cases will also throw exceptions.

Here's an example that shows many of the features discussed so far. Note the inclusion of **<fstream>** to declare the file I/O classes; this also includes **<iostream>**.

```
//: C18:Strfile.cpp
// Stream I/O with files
// The difference between get() & getline()
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main() {
  const int sz = 100; // Buffer size;
  char buf[sz];
  {
    ifstream in("Strfile.cpp"); // Read
    assure(in, "Strfile.cpp"); // Verify open
    ofstream out("Strfile.out"); // Write
    assure(out, "Strfile.out");
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, sz)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("Strfile.out");
  assure(in, "Strfile.out");
  // More convenient line input:
  while(in.getline(buf, sz)) { // Removes \n
    char* cp = buf;
    while(*cp != ':')
      cp++;
    cp += 2; // Past ": "
    cout << cp << endl; // Must still add \n
  }
} ///:~
```

The creation of both the **ifstream** and **ofstream** are followed by an **assure( )** to guarantee the file has been successfully opened. Here again the object, used in a situation where the compiler expects an integral result, produces a value that indicates success or failure. (To do this, an automatic type conversion member function is called. These are discussed in Chapter XX.)

The first **while** loop demonstrates the use of two forms of the **get( )** function. The first gets characters into a buffer and puts a zero terminator in the buffer when either **sz – 1** characters have been read or the third argument (defaulted to **'\n'**) is encountered. **get( )** leaves the terminator character in the input stream, so this terminator must be thrown away via **in.get( )** using the form of **get( )** with no argument, which fetches a single byte and returns it as an **int**. You can also use the **ignore( )** member function, which has two defaulted arguments. The first is the number of characters to throw away, and defaults to one. The second is the character at which the **ignore( )** function quits (after extracting it) and defaults to EOF.

Next you see two output statements that look very similar: one to **cout** and one to the file **out**. Notice the convenience here; you don't need to worry about what kind of object you're dealing with because the formatting statements work the same with all **ostream** objects. The first one echoes the line to standard output, and the second writes the line out to the new file and includes a line number.

To demonstrate **getline( )**, it's interesting to open the file we just created and strip off the line numbers. To ensure the file is properly closed before opening it to read, you have two choices. You can surround the first part of the program in braces to force the **out** object out of scope, thus calling the destructor and closing the file, which is done here. You can also call **close( )** for both files; if you want, you can even reuse the **in** object by calling the **open( )** member function (you can also create and destroy the object dynamically on the heap as is in Chapter XX).

The second **while** loop shows how **getline( )** removes the terminator character (its third argument, which defaults to **'\n'**) from the input stream when it's encountered. Although **getline( )**, like **get( )**, puts a zero in the buffer, it still doesn't insert the terminating character.

Open modes

You can control the way a file is opened by changing a default argument. The following table shows the flags that control the mode of the file:

Flag	Function
ios::in	Opens an input file. Use this as an open mode for an **ofstream** to prevent truncating an existing file.
ios::out	Opens an output file. When used for an **ofstream** without **ios::app**, **ios::ate** or **ios::in**, **ios::trunc** is implied.
ios::app	Opens an output file for appending.
ios::ate	Opens an existing file (either input or output) and seeks the end.
ios::nocreate	Opens a file only if it already exists. (Otherwise it fails.)
ios::noreplace	Opens a file only if it does not exist. (Otherwise it fails.)
ios::trunc	Opens a file and deletes the old file, if it already exists.
ios::binary	Opens a file in binary mode. Default is text mode.

These flags can be combined using a bitwise *or*.

Iostream buffering

Whenever you create a new class, you should endeavor to hide the details of the underlying implementation as possible from the user of the class. Try to show them only what they need to know and make the rest **private** to avoid confusion. Normally when using iostreams you don't know or care where the bytes are being produced or consumed; indeed, this is different depending on whether you're dealing with standard I/O, files, memory, or some newly created class or device.

There comes a time, however, when it becomes important to be able to send messages to the part of the iostream that produces and consumes bytes. To provide this part with a common interface and still hide its underlying implementation, it is abstracted into its own class, called **streambuf**. Each iostream object contains a pointer to some kind of **streambuf**. (The kind depends on whether it deals with standard I/O, files, memory, etc.) You can access the **streambuf** directly; for example, you can move raw bytes into and out of the **streambuf**, without formatting them through the enclosing iostream. This is accomplished, of course, by calling member functions for the **streambuf** object.

Currently, the most important thing for you to know is that every iostream object contains a pointer to a **streambuf** object, and the **streambuf** has some member functions you can call if you need to.

To allow you to access the **streambuf**, every iostream object has a member function called **rdbuf( )** that returns the pointer to the object's **streambuf**. This way you can call any member function for the underlying **streambuf**. However, one of the most interesting things you can do with the **streambuf** pointer is to connect it to another iostream object using the **<<** operator. This drains all the bytes from your object into the one on the left-hand side of the **<<**. This means if you want to move all the bytes from one iostream to another, you don't have to go through the tedium (and potential coding errors) of reading them one byte or one line at a time. It's a much more elegant approach.

For example, here's a very simple program that opens a file and sends the contents out to standard output (similar to the previous example):

```
//: C18:Stype.cpp
// Type a file to standard output
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Must have a command line
  ifstream in(argv[1]);
  assure(in, argv[1]); // Ensure file exists
  cout << in.rdbuf(); // Outputs entire file
} ///:~
```

After making sure there is an argument on the command line, an **ifstream** is created using this argument. The open will fail if the file doesn't exist, and this failure is caught by the **assert(in)**.

All the work really happens in the statement

```
cout << in.rdbuf();
```

which causes the entire contents of the file to be sent to **cout**. This is not only more succinct to code, it is often more efficient than moving the bytes one at a time.

Using **get( )** with a streambuf

There is a form of **get( )** that allows you to write directly into the **streambuf** of another object. The first argument is the destination **streambuf** (whose address is mysteriously taken using a *reference*, discussed in Chapter XX), and the second is the terminating character, which stops the **get( )** function. So yet another way to print a file to standard output is

```
//: C18:Sbufget.cpp
// Get directly into a streambuf
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main() {
  ifstream in("Sbufget.cpp");
  assure(in, "Sbufget.cpp");
  while(in.get(*cout.rdbuf()))
    in.ignore();
} ///:~
```

rdbuf( ) returns a pointer, so it must be dereferenced to satisfy the function's need to see an object. The **get( )** function, remember, doesn't pull the terminating character from the input stream, so it must be removed using **ignore( )** so **get( )** doesn't just bonk up against the newline forever (which it will, otherwise).

You probably won't need to use a technique like this very often, but it may be useful to know it exists.

Seeking in iostreams

Each type of iostream has a concept of where its "next" character will come from (if it's an **istream**) or go (if it's an **ostream**). In some situations you may want to move this stream position. You can do it using two models: One uses an absolute location in the stream called the **streampos**; the second works like the Standard C library functions **fseek( )** for a file and moves a given number of bytes from the beginning, end, or current position in the file.

The **streampos** approach requires that you first call a "tell" function: **tellp( )** for an **ostream** or **tellg( )** for an **istream**. (The "p" refers to the "put pointer" and the "g" refers to the "get pointer.") This function returns a **streampos** you can later use in the single-argument version of **seekp( )** for an **ostream** or **seekg( )** for an **istream**, when you want to return to that position in the stream.

The second approach is a relative seek and uses overloaded versions of **seekp( )** and **seekg( )**. The first argument is the number of bytes to move: it may be positive or negative. The second argument is the seek direction:

ios::beg	From beginning of stream
ios::cur	Current position in stream
ios::end	From end of stream

Here's an example that shows the movement through a file, but remember, you're not limited to seeking within files, as you are with C and **cstdio**. With C++, you can seek in any type of iostream (although the behavior of **cin** & **cout** when seeking is undefined):

```
//: C18:Seeking.cpp
// Seeking in iostreams
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // File must already exist
```

```
  in.seekg(0, ios::end); // End of file
  streampos sp = in.tellg(); // Size of file
  cout << "file size = " << sp << endl;
  in.seekg(-sp/10, ios::end);
  streampos sp2 = in.tellg();
  in.seekg(0, ios::beg); // Start of file
  cout << in.rdbuf(); // Print whole file
  in.seekg(sp2); // Move to streampos
  // Prints the last 1/10th of the file:
  cout << endl << endl << in.rdbuf() << endl;
} ///:~
```

This program picks a file name off the command line and opens it as an **ifstream**. **assert( )** detects an open failure. Because this is a type of **istream**, **seekg( )** is used to position the "get pointer." The first call seeks zero bytes off the end of the file, that is, to the end. Because a **streampos** is a **typedef** for a **long**, calling **tellg( )** at that point also returns the size of the file, which is printed out. Then a seek is performed moving the get pointer 1/10 the size of the file – notice it's a negative seek from the end of the file, so it backs up from the end. If you try to seek positively from the end of the file, the get pointer will just stay at the end. The **streampos** at that point is captured into **sp2**, then a **seekg( )** is performed back to the beginning of the file so the whole thing can be printed out using the **streambuf** pointer produced with **rdbuf( )**. Finally, the overloaded version of **seekg( )** is used with the **streampos sp2** to move to the previous position, and the last portion of the file is printed out.

Creating read/write files

Now that you know about the **streambuf** and how to seek, you can understand how to create a stream object that will both read and write a file. The following code first creates an **ifstream** with flags that say it's both an input and an output file. The compiler won't let you write to an **ifstream**, however, so you need to create an **ostream** with the underlying stream buffer:

```
ifstream in("filename", ios::in|ios::out);
ostream out(in.rdbuf());
```

You may wonder what happens when you write to one of these objects. Here's an example:

```
//: C18:Iofile.cpp
```

```
// Reading & writing one file
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("Iofile.out", ios::in | ios::out);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
} ///:~
```

The first five lines copy the source code for this program into a file called **iofile.out**, and then close the files. This gives us a safe text file to play around with. Then the aforementioned technique is used to create two objects that read and write to the same file. In **cout << in2.rdbuf( )**, you can see the “get” pointer is initialized to the beginning of the file. The “put” pointer, however, is set to the end of the file because “Where does this end up?” appears appended to the file. However, if the put pointer is moved to the beginning with a **seekp( )**, all the inserted text *overwrites* the existing text. Both writes are seen when the get pointer is moved back to the beginning with a **seekg( )**, and the file is printed out. Of course, the file is automatically saved and closed when **out2** goes out of scope and its destructor is called.

stringstreams

strstreams

Before there were **stringstream**s, there were the more primitive **strstream**s. Although these are not an official part of Standard C++, they have been around a long time so compilers will no doubt leave in the **strstream** support in perpetuity, to compile legacy code. You should always use **stringstream**s, but it's certainly likely that you'll come across code that uses **strstream**s and at that point this section should come in handy. In addition, this section should make it fairly clear why **stringstream**s have replace **strstream**s.

A **strstream** works directly with memory instead of a file or standard output. It allows you to use the same reading and formatting functions to manipulate bytes in memory. On old computers the memory was referred to as *core* so this type of functionality is often called *in-core formatting*.

The class names for strstreams echo those for file streams. If you want to create a strstream to extract characters from, you create an **istrstream**. If you want to put characters into a strstream, you create an **ostrstream**.

String streams work with memory, so you must deal with the issue of where the memory comes from and where it goes. This isn't terribly complicated, but you must understand it and pay attention (it turned out is was too easy to lose track of this particular issue, thus the birth of **stringstream**s).

User-allocated storage

The easiest approach to understand is when the user is responsible for allocating the storage. With **istrstream**s this is the only allowed approach. There are two constructors:

```
istrstream::istrstream(char* buf);
istrstream::istrstream(char* buf, int size);
```

The first constructor takes a pointer to a zero-terminated character array; you can extract bytes until the zero. The second constructor additionally requires the size of the array, which doesn't have to be zero-terminated. You can extract bytes all the way to **buf[size]**, whether or not you encounter a zero along the way.

When you hand an **istrstream** constructor the address of an array, that array must already be filled with the characters you want to extract and presumably format into some other data type. Here's a simple example:[52]

```
//: C18:Istring.cpp
// Input strstreams
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  istrstream s("47 1.414 This is a test");
  int i;
  float f;
  s >> i >> f; // Whitespace-delimited input
  char buf2[100];
  s >> buf2;
  cout << "i = " << i << ", f = " << f;
  cout << " buf2 = " << buf2 << endl;
  cout << s.rdbuf(); // Get the rest...
} ///:~
```

You can see that this is a more flexible and general approach to transforming character strings to typed values than the Standard C Library functions like **atof( )**, **atoi( )**, and so on.

The compiler handles the static storage allocation of the string in

```
  istrstream s("47 1.414 This is a test");
```

You can also hand it a pointer to a zero-terminated string allocated on the stack or the heap.

In **s >> i >> f**, the first number is extracted into **i** and the second into **f**. This isn't "the first whitespace-delimited set of characters" because it depends on the data type it's being extracted into. For example, if the string were instead, "**1.414 47 This is a test**," then **i** would get the value one because the input routine would stop at the decimal point. Then **f** would get **0.414**. This could be useful if you want to break a floating-point

[52] Note the name has been truncated to handle the DOS limitation on file names. You may need to adjust the header file name if your system supports longer file names (or simply copy the header file).

number into a whole number and a fraction part. Otherwise it would seem to be an error.

As you may already have guessed, **buf2** doesn't get the rest of the string, just the next whitespace-delimited word. In general, it seems the best place to use the extractor in iostreams is when you know the exact sequence of data in the input stream and you're converting to some type other than a character string. However, if you want to extract the rest of the string all at once and send it to another iostream, you can use **rdbuf( )** as shown.

Output strstreams

Output strstreams also allow you to provide your own storage; in this case it's the place in memory the bytes are formatted *into*. The appropriate constructor is

```
ostrstream::ostrstream(char*, int, int = ios::out);
```

The first argument is the preallocated buffer where the characters will end up, the second is the size of the buffer, and the third is the mode. If the mode is left as the default, characters are formatted into the starting address of the buffer. If the mode is either **ios::ate** or **ios::app** (same effect), the character buffer is assumed to already contain a zero-terminated string, and any new characters are added starting at the zero terminator.

The second constructor argument is the size of the array and is used by the object to ensure it doesn't overwrite the end of the array. If you fill the array up and try to add more bytes, they won't go in.

An important thing to remember about **ostrstream**s is that the zero terminator you normally need at the end of a character array *is not* inserted for you. When you're ready to zero-terminate the string, use the special manipulator **ends**.

Once you've created an **ostrstream** you can insert anything you want, and it will magically end up formatted in the memory buffer. Here's an example:

```
//: C18:Ostring.cpp
// Output strstreams
#include <iostream>
#include <strstream>
using namespace std;
```

```
int main() {
  const int sz = 100;
  cout << "type an int, a float and a string:";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Throw away white space
  char buf[sz];
  cin.getline(buf, sz); // Get rest of the line
  // (cin.rdbuf() would be awkward)
  ostrstream os(buf, sz, ios::app);
  os << endl;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << ends;
  cout << buf;
  cout << os.rdbuf(); // Same effect
  cout << os.rdbuf(); // NOT the same effect
} ///:~
```

This is similar to the previous example in fetching the **int** and **float**. You might think the logical way to get the rest of the line is to use **rdbuf( )**; this works, but it's awkward because all the input including newlines is collected until the user presses control-Z (control-D on Unix) to indicate the end of the input. The approach shown, using **getline( )**, gets the input until the user presses the carriage return. This input is fetched into **buf**, which is subsequently used to construct the **ostrstream os**. If the third argument **ios::app** weren't supplied, the constructor would default to writing at the beginning of **buf**, overwriting the line that was just collected. However, the "append" flag causes it to put the rest of the formatted information at the end of the string.

You can see that, like the other output streams, you can use the ordinary formatting tools for sending bytes to the **ostrstream**. The only difference is that you're responsible for inserting the zero at the end with **ends**. Note that **endl** inserts a newline in the strstream, but no zero.

Now the information is formatted in **buf**, and you can send it out directly with **cout << buf**. However, it's also possible to send the information out with **os.rdbuf( )**. When you do this, the get pointer inside the **streambuf** is moved forward as the characters are output. For this reason, if you say **cout << os.rdbuf( )** a second time, nothing happens – the get pointer is already at the end.

Automatic storage allocation

Output strstreams (but *not* **istrstream**s) give you a second option for memory allocation: they can do it themselves. All you do is create an **ostrstream** with no constructor arguments:

```
ostrstream a;
```

Now **a** takes care of all its own storage allocation on the heap. You can put as many bytes into **a** as you want, and if it runs out of storage, it will allocate more, moving the block of memory, if necessary.

This is a very nice solution if you don't know how much space you'll need, because it's completely flexible. And if you simply format data into the strstream and then hand its **streambuf** off to another iostream, things work perfectly:

```
a << "hello, world. i = " << i << endl << ends;
cout << a.rdbuf();
```

This is the best of all possible solutions. But what happens if you want the physical address of the memory that **a**'s characters have been formatted into? It's readily available – you simply call the **str( )** member function:

```
char* cp = a.str();
```

There's a problem now. What if you want to put more characters into **a**? It would be OK if you knew **a** had already allocated enough storage for all the characters you want to give it, but that's not true. Generally, **a** will run out of storage when you give it more characters, and ordinarily it would try to allocate more storage on the heap. This would usually require moving the block of memory. But the stream objects has just handed you the address of its memory block, so it can't very well move that block, because you're expecting it to be at a particular location.

The way an **ostrstream** handles this problem is by "freezing" itself. As long as you don't use **str( )** to ask for the internal **char***, you can add as many characters as you want to the **ostrstream**. It will allocate all the necessary storage from the heap, and when the object goes out of scope, that heap storage is automatically released.

However, if you call **str( )**, the **ostrstream** becomes "frozen." You can't add any more characters to it. Rather, you aren't *supposed* to – implementations are not required to detect the error. Adding characters to a frozen **ostrstream** results in undefined behavior. In addition, the **ostrstream** is no longer responsible for cleaning up the storage. You took over that responsibility when you asked for the **char*** with **str( )**.

To prevent a memory leak, the storage must be cleaned up somehow. There are two approaches. The more common one is to directly release the memory when you're done. To understand this, you need a sneak preview of two new keywords in C++: **new** and **delete**. As you'll see in Chapter XX, these do quite a bit, but for now you can think of them as replacements for **malloc( )** and **free( )** in C. The operator **new** returns a chunk of memory, and **delete** frees it. It's important to know about them here because virtually all memory allocation in C++ is performed with **new**, and this is also true with **ostrstream**. If it's allocated with **new**, it must be released with **delete**, so if you have an **ostrstream a** and you get the **char*** using **str( )**, the typical way to clean up the storage is

```
delete []a.str();
```

This satisfies most needs, but there's a second, much less common way to release the storage: You can unfreeze the **ostrstream**. You do this by calling **freeze( )**, which is a member function of the **ostrstream**'s **streambuf**. **freeze( )** has a default argument of one, which freezes the stream, but an argument of zero will unfreeze it:

```
a.rdbuf()->freeze(0);
```

Now the storage is deallocated when **a** goes out of scope and its destructor is called. In addition, you can add more bytes to **a**. However, this may cause the storage to move, so you better not use any pointer you previously got by calling **str( )** – it won't be reliable after adding more characters.

The following example tests the ability to add more characters after a stream has been unfrozen:

```
//: C18:Walrus.cpp
// Freezing a strstream
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  ostrstream s;
  s << "'The time has come', the walrus said,";
  s << ends;
  cout << s.str() << endl; // String is frozen
  // s is frozen; destructor won't delete
  // the streambuf storage on the heap
  s.seekp(-1, ios::cur); // Back up before NULL
```

```
  s.rdbuf()->freeze(0); // Unfreeze it
  // Now destructor releases memory, and
  // you can add more characters (but you
  // better not use the previous str() value)
  s << " 'To speak of many things'" << ends;
  cout << s.rdbuf();
} ///:~
```

After putting the first string into **s**, an **ends** is added so the string can be printed using the **char*** produced by **str( )**. At that point, **s** is frozen. We want to add more characters to **s**, but for it to have any effect, the put pointer must be backed up one so the next character is placed on top of the zero inserted by **ends**. (Otherwise the string would be printed only up to the original zero.) This is accomplished with **seekp( )**. Then **s** is unfrozen by fetching the underlying **streambuf** pointer using **rdbuf( )** and calling **freeze(0)**. At this point **s** is like it was before calling **str( )**: We can add more characters, and cleanup will occur automatically, with the destructor.

It is *possible* to unfreeze an **ostrstream** and continue adding characters, but it is not common practice. Normally, if you want to add more characters once you've gotten the **char*** of a **ostrstream**, you create a new one, pour the old stream into the new one using **rdbuf( )** and continue adding new characters to the new **ostrstream**.

Proving movement

If you're still not convinced you should be responsible for the storage of a **ostrstream** if you call **str( )**, here's an example that demonstrates the storage location is moved, therefore the old pointer returned by **str( )** is invalid:

```
//: C18:Strmove.cpp
// ostrstream memory movement
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  ostrstream s;
  s << "hi";
  char* old = s.str(); // Freezes s
  s.rdbuf()->freeze(0); // Unfreeze
  for(int i = 0; i < 100; i++)
```

```
    s << "howdy"; // Should force reallocation
  cout << "old = " << (void*)old << endl;
  cout << "new = " << (void*)s.str(); // Freezes
  delete s.str(); // Release storage
} ///:~
```

After inserting a string to **s** and capturing the **char*** with **str( )**, the string is unfrozen and enough new bytes are inserted to virtually assure the memory is reallocated and most likely moved. After printing out the old and new **char*** values, the storage is explicitly released with **delete** because the second call to **str( )** froze the string again.

To print out addresses instead of the strings they point to, you must cast the **char*** to a **void***. The operator **<<** for **char*** prints out the string it is pointing to, while the operator **<<** for **void*** prints out the hex representation of the pointer.

It's interesting to note that if you don't insert a string to **s** before calling **str( )**, the result is zero. This means no storage is allocated until the first time you try to insert bytes to the **ostrstream**.

A better way

Again, remember that this section was only left in to support legacy code. You should always use **string** and **stringstream** rather than character arrays and **strstream**. The former is much safer and easier to use and will help ensure your projects get finished faster.

Output stream formatting

The whole goal of this effort, and all these different types of iostreams, is to allow you to easily move and translate bytes from one place to another. It certainly wouldn't be very useful if you couldn't do all the formatting with the **printf( )** family of functions. In this section, you'll learn all the output formatting functions that are available for iostreams, so you can get your bytes the way you want them.

The formatting functions in iostreams can be somewhat confusing at first because there's often more than one way to control the formatting: through both member functions and manipulators. To further confuse things, there is a generic member function to set state flags to control

formatting, such as left- or right-justification, whether to use uppercase letters for hex notation, whether to always use a decimal point for floating-point values, and so on. On the other hand, there are specific member functions to set and read values for the fill character, the field width, and the precision.

In an attempt to clarify all this, the internal formatting data of an iostream is examined first, along with the member functions that can modify that data. (Everything can be controlled through the member functions.) The manipulators are covered separately.

Internal formatting data

The class **ios** (which you can see in the header file **<iostream>**) contains data members to store all the formatting data pertaining to that stream. Some of this data has a range of values and is stored in variables: the floating-point precision, the output field width, and the character used to pad the output (normally a space). The rest of the formatting is determined by flags, which are usually combined to save space and are referred to collectively as the *format flags*. You can find out the value of the format flags with the **ios::flags( )** member function, which takes no arguments and returns a **long** (**typedef**ed to **fmtflags**) that contains the current format flags. All the rest of the functions make changes to the format flags and return the previous value of the format flags.

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

The first function forces *all* the flags to change, which you do sometimes. More often, you change one flag at a time using the remaining three functions.

The use of **setf( )** can seem more confusing: To know which overloaded version to use, you must know what type of flag you're changing. There are two types of flags: ones that are simply on or off, and ones that work in a group with other flags. The on/off flags are the simplest to understand because you turn them on with **setf(fmtflags)** and off with **unsetf(fmtflags)**. These flags are

on/off flag	effect

on/off flag	effect
ios::skipws	Skip white space. (For input; this is the default.)
ios::showbase	Indicate the numeric base (dec, oct, or hex) when printing an integral value. The format used can be read by the C++ compiler.
ios::showpoint	Show decimal point and trailing zeros for floating-point values.
ios::uppercase	Display uppercase A-F for hexadecimal values and E for scientific values.
ios::showpos	Show plus sign (+) for positive values.
ios::unitbuf	"Unit buffering." The stream is flushed after each insertion.
ios::stdio	Synchronizes the stream with the C standard I/O system.

For example, to show the plus sign for **cout**, you say **cout.setf(ios::showpos)**. To stop showing the plus sign, you say **cout.unsetf(ios::showpos)**.

The last two flags deserve some explanation. You turn on unit buffering when you want to make sure each character is output as soon as it is inserted into an output stream. You could also use unbuffered output, but unit buffering provides better performance.

The **ios::stdio** flag is used when you have a program that uses both iostreams and the C standard I/O library (not unlikely if you're using C libraries). If you discover your iostream output and **printf( )** output are occurring in the wrong order, try setting this flag.

Format fields

The second type of formatting flags work in a group. You can have only one of these flags on at a time, like the buttons on old car radios – you push one in, the rest pop out. Unfortunately this doesn't happen automatically, and you have to pay attention to what flags you're setting so you don't accidentally call the wrong **setf( )** function. For example, there's a flag for each of the number bases: hexadecimal, decimal, and octal. Collectively, these flags are referred to as the **ios::basefield**. If the **ios::dec** flag is set and you call **setf(ios::hex)**, you'll set the **ios::hex** flag, but you *won't* clear the **ios::dec** bit, resulting in undefined behavior. The proper thing to do is call the second form of **setf( )** like this: **setf(ios::hex, ios::basefield)**. This function first clears all the bits in the **ios::basefield**, *then* sets **ios::hex**. Thus, this form of **setf( )** ensures that the other flags in the group "pop out" whenever you set one. Of course, the **hex( )** manipulator does all this for you, automatically, so you don't have to concern yourself with the internal details of the implementation of this class or to even *care* that it's a set of binary flags. Later you'll see there are manipulators to provide equivalent functionality in all the places you would use **setf( )**.

Here are the flag groups and their effects:

ios::basefield	**effect**
ios::dec	Format integral values in base 10 (decimal) (default radix).
ios::hex	Format integral values in base 16 (hexadecimal).
ios::oct	Format integral values in base 8 (octal).

ios::floatfield	**effect**
ios::scientific	Display floating-point numbers in scientific format. Precision field indicates number of digits after the decimal point.

ios::floatfield	effect
ios::fixed	Display floating-point numbers in fixed format. Precision field indicates number of digits after the decimal point.
“automatic” (Neither bit is set.)	Precision field indicates the total number of significant digits.

ios::adjustfield	effect
ios::left	Left-align values; pad on the right with the fill character.
ios::right	Right-align values. Pad on the left with the fill character. This is the default alignment.
ios::internal	Add fill characters after any leading sign or base indicator, but before the value.

Width, fill and precision

The internal variables that control the width of the output field, the fill character used when the data doesn’t fill the output field, and the precision for printing floating-point numbers are read and written by member functions of the same name.

function	effect
int ios::width( )	Reads the current width. (Default is 0.) Used for both insertion and extraction.

function	effect
int ios::width(int n)	Sets the width, returns the previous width.
int ios::fill( )	Reads the current fill character. (Default is space.)
int ios::fill(int n)	Sets the fill character, returns the previous fill character.
int ios::precision( )	Reads current floating-point precision. (Default is 6.)
int ios::precision(int n)	Sets floating-point precision, returns previous precision. See **ios::floatfield** table for the meaning of "precision."

The fill and precision values are fairly straightforward, but width requires some explanation. When the width is zero, inserting a value will produce the minimum number of characters necessary to represent that value. A positive width means that inserting a value will produce at least as many characters as the width; if the value has less than width characters, the fill character is used to pad the field. However, the value will never be truncated, so if you try to print 123 with a width of two, you'll still get 123. The field width specifies a *minimum* number of characters; there's no way to specify a maximum number.

The width is also distinctly different because it's reset to zero by each inserter or extractor that could be influenced by its value. It's really not a state variable, but an implicit argument to the inserters and extractors. If you want to have a constant width, you have to call **width( )** after each insertion or extraction.

An exhaustive example

To make sure you know how to call all the functions previously discussed, here's an example that calls them all:

```
//: C18:Format.cpp
// Formatting functions
#include <fstream>
using namespace std;
#define D(A) T << #A << endl; A
ofstream T("format.out");

int main() {
  D(int i = 47;)
  D(float f = 2300114.414159;)
  char* s = "Is there any more?";

  D(T.setf(ios::unitbuf);)
//  D(T.setf(ios::stdio);)  // SOMETHING MAY HAVE CHANGED

  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase);)
  D(T.setf(ios::showpos);)
  D(T << i << endl;) // Default to dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "fill char: " << T.fill() << endl;)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T << i << endl;) // Without width(10)

  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "prec = " << T.precision() << endl;)
```

```
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "prec = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;

  D(T.unsetf(ios::showpoint);)
  D(T.unsetf(ios::unitbuf);)
//  D(T.unsetf(ios::stdio);) // SOMETHING MAY HAVE CHANGED
} ///:~
```

This example uses a trick to create a trace file so you can monitor what's happening. The macro **D(a)** uses the preprocessor "stringizing" to turn **a** into a string to print out. Then it reiterates **a** so the statement takes effect. The macro sends all the information out to a file called **T**, which is the trace file. The output is

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::stdio);
T.setf(ios::showbase);
T.setf(ios::uppercase);
T.setf(ios::showpos);
T << i << endl;
```

```
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
+0X2F
T.unsetf(ios::uppercase);
T.setf(ios::oct, ios::basefield);
T << i << endl;
+057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300115e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.setf(0, ios::floatfield);
T << f << endl;
2.300115e+06
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;
```

```
2300114.50000000020000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114500000000020000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.setf(0, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.width(10);
Is there any more?
T.width(40);
000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?00000000000000000000
T.unsetf(ios::showpoint);
T.unsetf(ios::unitbuf);
T.unsetf(ios::stdio);
```

Studying this output should clarify your understanding of the iostream formatting member functions.

Formatting manipulators

As you can see from the previous example, calling the member functions can get a bit tedious. To make things easier to read and write, a set of manipulators is supplied to duplicate the actions provided by the member functions.

Manipulators with no arguments are provided in **<iostream>**. These include **dec**, **oct**, and **hex** , which perform the same action as, respectively, **setf(ios::dec, ios::basefield)**, **setf(ios::oct, ios::basefield)**, and **setf(ios::hex, ios::basefield)**, albeit more

succinctly. **<iostream>**[53] also includes **ws**, **endl**, **ends**, and **flush** and the additional set shown here:

[53] These only appear in the revised library; you won't find them in older implementations of iostreams.

manipulator	effect
showbase noshowbase	Indicate the numeric base (dec, oct, or hex) when printing an integral value. The format used can be read by the C++ compiler.
showpos noshowpos	Show plus sign (+) for positive values
uppercase nouppercase	Display uppercase A-F for hexadecimal values, and E for scientific values
showpoint noshowpoint	Show decimal point and trailing zeros for floating-point values.
skipws noskipws	Skip white space on input.
left right internal	Left-align, pad on right. Right-align, pad on left. Fill between leading sign or base indicator and value.
scientific fixed	Use scientific notation **setprecision( )** or **ios::precision( )** sets number of places after the decimal point.

Manipulators with arguments

If you are using manipulators with arguments, you must also include the header file <**iomanip>**. This contains code to solve the general problem

of creating manipulators with arguments. In addition, it has six predefined manipulators:

manipulator	**effect**
setiosflags (fmtflags n)	Sets only the format flags specified by n. Setting remains in effect until the next change, like **ios::setf( )**.
resetiosflags(fmtflags n)	Clears only the format flags specified by n. Setting remains in effect until the next change, like **ios::unsetf( )**.
setbase(base n)	Changes base to n, where n is 10, 8, or 16. (Anything else results in 0.) If n is zero, output is base 10, but input uses the C conventions: 10 is 10, 010 is 8, and 0xf is 15. You might as well use **dec**, **oct**, and **hex** for output.
setfill(char n)	Changes the fill character to n, like **ios::fill( )**.
setprecision(int n)	Changes the precision to n, like **ios::precision( )**.
setw(int n)	Changes the field width to n, like **ios::width( )**.

If you're using a lot of inserters, you can see how this can clean things up. As an example, here's the previous program rewritten to use the manipulators. (The macro has been removed to make it easier to read.)

```
//: C18:Manips.cpp
// Format.cpp using manipulators
```

```
#include <fstream>
#include <iomanip>
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  trc << setiosflags(
        ios::unitbuf /*| ios::stdio */ /// ?????
        | ios::showbase | ios::uppercase
        | ios::showpos);
  trc << i << endl; // Default to dec
  trc << hex << i << endl;
  trc << resetiosflags(ios::uppercase)
    << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
    << dec << setfill('0');
  trc << "fill char: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc.setf(ios::internal, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc << i << endl; // Without setw(10)

  trc << resetiosflags(ios::showpos)
    << setiosflags(ios::showpoint)
    << "prec = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc.setf(0, ios::floatfield); // Automatic
  trc << f << endl;
  trc << setprecision(20);
  trc << "prec = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
```

```
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc.setf(0, ios::floatfield); // Automatic
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;

  trc << resetiosflags(
        ios::showpoint | ios::unitbuf
        // | ios::stdio // ?????????
 );
} ///:~
```

You can see that a lot of the multiple statements have been condensed into a single chained insertion. Note the calls to **setiosflags( )** and **resetiosflags( )**, where the flags have been bitwise-ORed together. This could also have been done with **setf( )** and **unsetf( )** in the previous example.

Creating manipulators

(Note: This section contains some material that will not be introduced until later chapters.) Sometimes you'd like to create your own manipulators, and it turns out to be remarkably simple. A zero-argument manipulator like **endl** is simply a function that takes as its argument an **ostream** reference (references are a different way to pass arguments, discussed in Chapter XX). The declaration for **endl** is

```
ostream& endl(ostream&);
```

Now, when you say:

```
cout << "howdy" << endl;
```

the **endl** produces the *address* of that function. So the compiler says "is there a function I can call that takes the address of a function as its argument?" There is a pre-defined function in **Iostream.h** to do this; it's called an *applicator*. The applicator calls the function, passing it the **ostream** object as an argument.

You don't need to know how the applicator works to create your own manipulator; you only need to know the applicator exists. Here's an example that creates a manipulator called **nl** that emits a newline *without* flushing the stream:

```
//: C18:nl.cpp
// Creating a manipulator
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
} ///:~
```

The expression

```
os << '\n';
```

calls a function that returns **os**, which is what is returned from **nl**.[54]

People often argue that the **nl** approach shown above is preferable to using **endl** because the latter always flushes the output stream, which may incur a performance penalty.

Effectors

As you've seen, zero-argument manipulators are quite easy to create. But what if you want to create a manipulator that takes arguments? The iostream library has a rather convoluted and confusing way to do this, but Jerry Schwarz, the creator of the iostream library, suggests[55] a scheme he calls *effectors*. An effector is a simple class whose constructor performs the desired operation, along with an overloaded **operator<<** that works with the class. Here's an example with two effectors. The first outputs a truncated character string, and the second prints a number in binary (the

[54] Before putting **nl** into a header file, you should make it an **inline** function (see Chapter 7).

[55] In a private conversation.

process of defining an overloaded **operator<<** will not be discussed until Chapter XX):

```
//: C18:Effector.txt
// (Should be "cpp" but I can't get it to compile with
// My windows compilers, so making it a txt file will
// keep it out of the makefile for the time being)
// Jerry Schwarz's "effectors"
#include<iostream>
#include <cstdlib>
#include <string>
#include <climits> // ULONG_MAX
using namespace std;

// Put out a portion of a string:
class Fixw {
  string str;
public:
  Fixw(const string& s, int width)
    : str(s, 0, width) {}
  friend ostream&
  operator<<(ostream& os, Fixw& fw) {
    return os << fw.str;
  }
};

typedef unsigned long ulong;

// Print a number in binary:
class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
  friend ostream& operator<<(ostream&, Bin&);
};

ostream& operator<<(ostream& os, Bin& b) {
  ulong bit = ~(ULONG_MAX >> 1); // Top bit set
  while(bit) {
    os << (b.n & bit ? '1' : '0');
    bit >>= 1;
  }
  return os;
```

```
}

int main() {
  char* string =
    "Things that make us happy, make us wise";
  for(int i = 1; i <= strlen(string); i++)
    cout << Fixw(string, i) << endl;
  ulong x = 0xCAFEBABEUL;
  ulong y = 0x76543210UL;
  cout << "x in binary: " << Bin(x) << endl;
  cout << "y in binary: " << Bin(y) << endl;
} ///:~
```

The constructor for **Fixw** creates a shortened copy of its **char*** argument, and the destructor releases the memory created for this copy. The overloaded **operator<<** takes the contents of its second argument, the **Fixw** object, and inserts it into the first argument, the **ostream**, then returns the **ostream** so it can be used in a chained expression. When you use **Fixw** in an expression like this:

```
cout << Fixw(string, i) << endl;
```

a *temporary object* is created by the call to the **Fixw** constructor, and that temporary is passed to **operator<<**. The effect is that of a manipulator with arguments.

The **Bin** effector relies on the fact that shifting an unsigned number to the right shifts zeros into the high bits. ULONG_MAX (the largest **unsigned long** value, from the standard include file **<climits>**) is used to produce a value with the high bit set, and this value is moved across the number in question (by shifting it), masking each bit.

Initially the problem with this technique was that once you created a class called **Fixw** for **char*** or **Bin** for **unsigned long**, no one else could create a different **Fixw** or **Bin** class for their type. However, with *namespaces* (covered in Chapter XX), this problem is eliminated.

Iostream examples

In this section you'll see some examples of what you can do with all the information you've learned in this chapter. Although many tools exist to manipulate bytes (stream editors like **sed** and **awk** from Unix are perhaps the most well known, but a text editor also fits this category), they generally have some limitations. **sed** and **awk** can be slow and can only

handle lines in a forward sequence, and text editors usually require human interaction, or at least learning a proprietary macro language. The programs you write with iostreams have none of these limitations: They're fast, portable, and flexible. It's a very useful tool to have in your kit.

Code generation

The first examples concern the generation of programs that, coincidentally, fit the format used in this book. This provides a little extra speed and consistency when developing code. The first program creates a file to hold **main( )** (assuming it takes no command-line arguments and uses the iostream library):

```
//: C18:Makemain.cpp
// Create a shell main() file
#include "../require.h"
#include <fstream>
#include <strstream>
#include <cstring>
#include <cctype>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ofstream mainfile(argv[1]);
  assure(mainfile, argv[1]);
  istrstream name(argv[1]);
  ostrstream CAPname;
  char c;
  while(name.get(c))
    CAPname << char(toupper(c));
  CAPname << ends;
  mainfile << "//" << ": " << CAPname.rdbuf()
    << " -- " << endl
    << "#include <iostream>" << endl
    << endl
    << "main() {" << endl << endl
    << "}" << endl;
} ///:~
```

The argument on the command line is used to create an **istrstream**, so the characters can be extracted one at a time and converted to upper case with the Standard C library macro **toupper( )**. This returns an **int** so

it must be explicitly cast to a **char**. This name is used in the headline, followed by the remainder of the generated file.

Maintaining class library source

The second example performs a more complex and useful task. Generally, when you create a class you think in library terms, and make a header file **Name.h** for the class declaration and a file where the member functions are implemented, called **Name.cpp**. These files have certain requirements: a particular coding standard (the program shown here will use the coding format for this book), and in the header file the declarations are generally surrounded by some preprocessor statements to prevent multiple declarations of classes. (Multiple declarations confuse the compiler – it doesn't know which one you want to use. They could be different, so it throws up its hands and gives an error message.)

This example allows you to create a new header-implementation pair of files, or to modify an existing pair. If the files already exist, it checks and potentially modifies the files, but if they don't exist, it creates them using the proper format.

```
//: C18:Cppcheck.cpp
// Configures .h & .cpp files
// To conform to style standard.
// Tests existing files for conformance
#include "../require.h"
#include <fstream>
#include <strstream>
#include <cstring>
#include <cctype>
using namespace std;

int main(int argc, char* argv[]) {
  const int sz = 40;  // Buffer sizes
  const int bsz = 100;
  requireArgs(argc, 1); // File set name
  enum bufs { base, header, implement,
    Hline1, guard1, guard2, guard3,
    CPPline1, include, bufnum };
  char b[bufnum][sz];
  ostrstream osarray[] = {
    ostrstream(b[base], sz),
    ostrstream(b[header], sz),
    ostrstream(b[implement], sz),
```

```
  ostrstream(b[Hline1], sz),
  ostrstream(b[guard1], sz),
  ostrstream(b[guard2], sz),
  ostrstream(b[guard3], sz),
  ostrstream(b[CPPline1], sz),
  ostrstream(b[include], sz),
};
osarray[base] << argv[1] << ends;
// Find any '.' in the string using the
// Standard C library function strchr():
char* period = strchr(b[base], '.');
if(period) *period = 0; // Strip extension
// Force to upper case:
for(int i = 0; b[base][i]; i++)
  b[base][i] = toupper(b[base][i]);
// Create file names and internal lines:
osarray[header] << b[base] << ".h" << ends;
osarray[implement] << b[base] << ".cpp" << ends;
osarray[Hline1] << "//" << ": " << b[header]
  << " -- " << ends;
osarray[guard1] << "#ifndef " << b[base]
            << "_H" << ends;
osarray[guard2] << "#define " << b[base]
            << "_H" << ends;
osarray[guard3] << "#endif // " << b[base]
            << "_H" << ends;
osarray[CPPline1] << "//" << ": "
              << b[implement]
              << " -- " << ends;
osarray[include] << "#include \""
             << b[header] << "\"" <<ends;
// First, try to open existing files:
ifstream existh(b[header]),
       existcpp(b[implement]);
if(!existh) { // Doesn't exist; create it
  ofstream newheader(b[header]);
  assure(newheader, b[header]);
  newheader << b[Hline1] << endl
    << b[guard1] << endl
    << b[guard2] << endl << endl
    << b[guard3] << endl;
}
```

```
  if(!existcpp) { // Create cpp file
    ofstream newcpp(b[implement]);
    assure(newcpp, b[implement]);
    newcpp << b[CPPline1] << endl
      << b[include] << endl;
  }
  if(existh) { // Already exists; verify it
    strstream hfile; // Write & read
    ostrstream newheader; // Write
    hfile << existh.rdbuf() << ends;
    // Check that first line conforms:
    char buf[bsz];
    if(hfile.getline(buf, bsz)) {
      if(!strstr(buf, "//" ":") ||
         !strstr(buf, b[header]))
        newheader << b[Hline1] << endl;
    }
    // Ensure guard lines are in header:
    if(!strstr(hfile.str(), b[guard1]) ||
       !strstr(hfile.str(), b[guard2]) ||
       !strstr(hfile.str(), b[guard3])) {
       newheader << b[guard1] << endl
         << b[guard2] << endl
         << buf
         << hfile.rdbuf() << endl
         << b[guard3] << endl << ends;
    } else
      newheader << buf
        << hfile.rdbuf() << ends;
    // If there were changes, overwrite file:
    if(strcmp(hfile.str(),newheader.str())!=0){
      existh.close();
      ofstream newH(b[header]);
      assure(newH, b[header]);
      newH << "//@//" << endl // Change marker
        << newheader.rdbuf();
    }
    delete hfile.str();
    delete newheader.str();
  }
  if(existcpp) { // Already exists; verify it
    strstream cppfile;
```

```
    ostrstream newcpp;
    cppfile << existcpp.rdbuf() << ends;
    char buf[bsz];
    // Check that first line conforms:
    if(cppfile.getline(buf, bsz))
      if(!strstr(buf, "//" ":") ||
         !strstr(buf, b[implement]))
        newcpp << b[CPPline1] << endl;
    // Ensure header is included:
    if(!strstr(cppfile.str(), b[include]))
      newcpp << b[include] << endl;
    // Put in the rest of the file:
    newcpp << buf << endl; // First line read
    newcpp << cppfile.rdbuf() << ends;
    // If there were changes, overwrite file:
    if(strcmp(cppfile.str(),newcpp.str())!=0){
      existcpp.close();
      ofstream newCPP(b[implement]);
      assure(newCPP, b[implement]);
      newCPP << "//@//" << endl // Change marker
        << newcpp.rdbuf();
    }
    delete cppfile.str();
    delete newcpp.str();
  }
} ///:~
```

This example requires a lot of string formatting in many different buffers. Rather than creating a lot of individually named buffers and **ostrstream** objects, a single set of names is created in the **enum bufs**. Then two arrays are created: an array of character buffers and an array of **ostrstream** objects built from those character buffers. Note that in the definition for the two-dimensional array of **char** buffers **b**, the number of **char** arrays is determined by **bufnum**, the last enumerator in **bufs**. When you create an enumeration, the compiler assigns integral values to all the **enum** labels starting at zero, so the sole purpose of **bufnum** is to be a counter for the number of enumerators in **buf**. The length of each string in **b** is **sz**.

The names in the enumeration are **base**, the capitalized base file name without extension; **header**, the header file name; **implement**, the implementation file (**cpp**) name; **Hline1**, the skeleton first line of the header file; **guard1**, **guard2**, and **guard3**, the “guard” lines in the

header file (to prevent multiple inclusion); **CPPline1**, the skeleton first line of the **cpp** file; and **include**, the line in the **cpp** file that includes the header file.

osarray is an array of **ostrstream** objects created using aggregate initialization and automatic counting. Of course, this is the form of the **ostrstream** constructor that takes two arguments (the buffer address and buffer size), so the constructor calls must be formed accordingly inside the aggregate initializer list. Using the **bufs** enumerators, the appropriate array element of **b** is tied to the corresponding **osarray** object. Once the array is created, the objects in the array can be selected using the enumerators, and the effect is to fill the corresponding **b** element. You can see how each string is built in the lines following the **ostrstream** array definition.

Once the strings have been created, the program attempts to open existing versions of both the header and **cpp** file as **ifstream**s. If you test the object using the operator '**!**' and the file doesn't exist, the test will fail. If the header or implementation file doesn't exist, it is created using the appropriate lines of text built earlier.

If the files *do* exist, then they are verified to ensure the proper format is followed. In both cases, a **strstream** is created and the whole file is read in; then the first line is read and checked to make sure it follows the format by seeing if it contains both a "**//:**" and the name of the file. This is accomplished with the Standard C library function **strstr( )**. If the first line doesn't conform, the one created earlier is inserted into an **ostrstream** that has been created to hold the edited file.

In the header file, the whole file is searched (again using **strstr( )**) to ensure it contains the three "guard" lines; if not, they are inserted. The implementation file is checked for the existence of the line that includes the header file (although the compiler effectively guarantees its existence).

In both cases, the original file (in its **strstream**) and the edited file (in the **ostrstream**) are compared to see if there are any changes. If there are, the existing file is closed, and a new **ofstream** object is created to overwrite it. The **ostrstream** is output to the file after a special change marker is added at the beginning, so you can use a text search program to rapidly find any files that need reviewing to make additional changes.

Detecting compiler errors

All the code in this book is designed to compile as shown without errors. Any line of code that should generate a compile-time error is commented out with the special comment sequence “//!”. The following program will remove these special comments and append a numbered comment to the line, so that when you run your compiler it should generate error messages and you should see all the numbers appear when you compile all the files. It also appends the modified line to a special file so you can easily locate any lines that don’t generate errors:

```
//: C18:Showerr.cpp
// Un-comment error generators
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
#include <cctype>
#include <cstring>
using namespace std;
char* marker = "//!";

char* usage =
"usage: showerr filename chapnum\n"
"where filename is a C++ source file\n"
"and chapnum is the chapter name it's in.\n"
"Finds lines commented with //! and removes\n"
"comment, appending //(#) where # is unique\n"
"across all files, so you can determine\n"
"if your compiler finds the error.\n"
"showerr /r\n"
"resets the unique counter.";

// File containing error number counter:
char* errnum = "../errnum.txt";
// File containing error lines:
char* errfile = "../errlines.txt";
ofstream errlines(errfile,ios::app);

int main(int argc, char* argv[]) {
  requireArgs(argc, 2, usage);
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
```

```
  switch(argv[1][1]) {
    case 'r': case 'R':
      cout << "reset counter" << endl;
      remove(errnum); // Delete files
      remove(errfile);
      return 0;
    default:
      cerr << usage << endl;
      return 1;
  }
}
char* chapter = argv[2];
strstream edited; // Edited file
int counter = 0;
{
  ifstream infile(argv[1]);
  assure(infile, argv[1]);
  ifstream count(errnum);
  assure(count, errnum);
  if(count) count >> counter;
  int linecount = 0;
  const int sz = 255;
  char buf[sz];
  while(infile.getline(buf, sz)) {
    linecount++;
    // Eat white space:
    int i = 0;
    while(isspace(buf[i]))
      i++;
    // Find marker at start of line:
    if(strstr(&buf[i], marker) == &buf[i]) {
      // Erase marker:
      memset(&buf[i], ' ', strlen(marker));
      // Append counter & error info:
      ostrstream out(buf, sz, ios::ate);
      out << "//(" << ++counter << ") "
          << "Chapter " << chapter
          << " File: " << argv[1]
          << " Line " << linecount << endl
          << ends;
        edited << buf;
      errlines << buf; // Append error file
```

```
      } else
        edited << buf << "\n"; // Just copy
      }
    } // Closes files
    ofstream outfile(argv[1]); // Overwrites
    assure(outfile, argv[1]);
    outfile << edited.rdbuf();
    ofstream count(errnum); // Overwrites
    assure(count, errnum);
    count << counter; // Save new counter
  } ///:~
```

The marker can be replaced with one of your choice.

Each file is read a line at a time, and each line is searched for the marker appearing at the head of the line; the line is modified and put into the error line list and into the **strstream edited**. When the whole file is processed, it is closed (by reaching the end of a scope), reopened as an output file and **edited** is poured into the file. Also notice the counter is saved in an external file, so the next time this program is invoked it continues to sequence the counter.

A simple datalogger

This example shows an approach you might take to log data to disk and later retrieve it for processing. The example is meant to produce a temperature-depth profile of the ocean at various points. To hold the data, a class is used:

```
//: C18:DataLogger.h
// Datalogger record layout
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iostream>

class DataPoint {
  std::tm time; // Time & day
  static const int bsz = 10;
  // Ascii degrees (*) minutes (') seconds ("):
  char latitude[bsz], longitude[bsz];
  double depth, temperature;
public:
  std::tm getTime();
```

```
  void setTime(std::tm t);
  const char* getLatitude();
  void setLatitude(const char* l);
  const char* getLongitude();
  void setLongitude(const char* l);
  double getDepth();
  void setDepth(double d);
  double getTemperature();
  void setTemperature(double t);
  void print(std::ostream& os);
};
#endif // DATALOG_H ///:~
```

The access functions provide controlled reading and writing to each of the data members. The **print( )** function formats the **DataPoint** in a readable form onto an **ostream** object (the argument to **print( )**). Here's the definition file:

```
//: C18:Datalog.cpp {O}
// Datapoint member functions
#include "DataLogger.h"
#include <iomanip>
#include <cstring>
using namespace std;

tm DataPoint::getTime() { return time; }

void DataPoint::setTime(tm t) { time = t; }

const char* DataPoint::getLatitude() {
  return latitude;
}

void DataPoint::setLatitude(const char* l) {
  latitude[bsz - 1] = 0;
  strncpy(latitude, l, bsz - 1);
}

const char* DataPoint::getLongitude() {
  return longitude;
}

void DataPoint::setLongitude(const char* l) {
```

```
  longitude[bsz - 1] = 0;
  strncpy(longitude, l, bsz - 1);
}

double DataPoint::getDepth() { return depth; }

void DataPoint::setDepth(double d) { depth = d; }

double DataPoint::getTemperature() {
  return temperature;
}

void DataPoint::setTemperature(double t) {
  temperature = t;
}

void DataPoint::print(ostream& os) {
  os.setf(ios::fixed, ios::floatfield);
  os.precision(4);
  os.fill('0'); // Pad on left with '0'
  os << setw(2) << getTime().tm_mon << '\\'
     << setw(2) << getTime().tm_mday << '\\'
     << setw(2) << getTime().tm_year << ' '
     << setw(2) << getTime().tm_hour << ':'
     << setw(2) << getTime().tm_min << ':'
     << setw(2) << getTime().tm_sec;
  os.fill(' '); // Pad on left with ' '
  os << " Lat:" << setw(9) << getLatitude()
     << ", Long:" << setw(9) << getLongitude()
     << ", depth:" << setw(9) << getDepth()
     << ", temp:" << setw(9) << getTemperature()
     << endl;
} ///:~
```

In **print( )**, the call to **setf( )** causes the floating-point output to be fixed-precision, and **precision( )** sets the number of decimal places to four.

The default is to right-justify the data within the field. The time information consists of two digits each for the hours, minutes and seconds, so the width is set to two with **setw( )** in each case. (Remember that any changes to the field width affect only the next output operation, so **setw( )** must be given for each output.) But first, to put a zero in the

left position if the value is less than 10, the fill character is set to '0'. Afterwards, it is set back to a space.

The latitude and longitude are zero-terminated character fields, which hold the information as degrees (here, '*' denotes degrees), minutes ('), and seconds("). You can certainly devise a more efficient storage layout for latitude and longitude if you desire.

Generating test data

Here's a program that creates a file of test data in binary form (using **write( )**) and a second file in ASCII form using **DataPoint::print( )**. You can also print it out to the screen but it's easier to inspect in file form.

```
//: C18:Datagen.cpp
//{L} Datalog
// Test data generator
#include "DataLogger.h"
#include "../require.h"
#include <fstream>
#include <cstdlib>
#include <cstring>
using namespace std;

int main() {
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  time_t timer;
  // Seed random number generator:
  srand(time(&timer));
  for(int i = 0; i < 100; i++) {
    DataPoint d;
    // Convert date/time to a structure:
    d.setTime(*localtime(&timer));
    timer += 55; // Reading each 55 seconds
    d.setLatitude("45*20'31\"");
    d.setLongitude("22*34'18\"");
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += double(1) / fraction;
    d.setDepth(newdepth);
```

```
    double newtemp = 150 + rand()%200; // Kelvin
    fraction = rand() % 100 + 1;
    newtemp += (double)1 / fraction;
    d.setTemperature(newtemp);
    d.print(data);
    bindata.write((unsigned char*)&d,
              sizeof(d));
  }
} ///:~
```

The file DATA.TXT is created in the ordinary way as an ASCII file, but DATA.BIN has the flag **ios::binary** to tell the constructor to set it up as a binary file.

The Standard C library function **time( )**, when called with a zero argument, returns the current time as a **time_t** value, which is the number of seconds elapsed since 00:00:00 GMT, January 1 1970 (the dawning of the age of Aquarius?). The current time is the most convenient way to seed the random number generator with the Standard C library function **srand( )**, as is done here.

Sometimes a more convenient way to store the time is as a **tm** structure, which has all the elements of the time and date broken up into their constituent parts as follows:

```
struct tm {
  int tm_sec; // 0-59 seconds
  int tm_min; // 0-59 minutes
  int tm_hour; // 0-23 hours
  int tm_mday; // Day of month
  int tm_mon; // 1-12 months
  int tm_year; // Calendar year
  int tm_wday; // Sunday == 0, etc.
  int tm_yday; // 0-365 day of year
  int tm_isdst; // Daylight savings?
};
```

To convert from the time in seconds to the local time in the **tm** format, you use the Standard C library **localtime( )** function, which takes the number of seconds and returns a pointer to the resulting **tm**. This **tm**, however, is a **static** structure inside the **localtime( )** function, which is rewritten every time **localtime( )** is called. To copy the contents into the **tm struct** inside **DataPoint**, you might think you must copy each element individually. However, all you must do is a structure assignment, and the compiler will take care of the rest. This means the right-hand side

must be a structure, not a pointer, so the result of **localtime( )** is dereferenced. The desired result is achieved with

```
d.setTime(*localtime(&timer));
```

After this, the **timer** is incremented by 55 seconds to give an interesting interval between readings.

The latitude and longitude used are fixed values to indicate a set of readings at a single location. Both the depth and the temperature are generated with the Standard C library **rand( )** function, which returns a pseudorandom number between zero and the constant RAND_MAX. To put this in a desired range, use the modulus operator **%** and the upper end of the range. These numbers are integral; to add a fractional part, a second call to **rand( )** is made, and the value is inverted after adding one (to prevent divide-by-zero errors).

In effect, the DATA.BIN file is being used as a container for the data in the program, even though the container exists on disk and not in RAM. To send the data out to the disk in binary form, **write( )** is used. The first argument is the starting address of the source block – notice it must be cast to an **unsigned char*** because that's what the function expects. The second argument is the number of bytes to write, which is the size of the **DataPoint** object. Because no pointers are contained in **DataPoint**, there is no problem in writing the object to disk. If the object is more sophisticated, you must implement a scheme for *serialization* . (Most vendor class libraries have some sort of serialization structure built into them.)

Verifying & viewing the data

To check the validity of the data stored in binary format, it is read from the disk and put in text form in DATA2.TXT, so that file can be compared to DATA.TXT for verification. In the following program, you can see how simple this data recovery is. After the test file is created, the records are read at the command of the user.

```
//: C18:Datascan.cpp
//{L} Datalog
// Verify and view logged data
#include "DataLogger.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
```

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

int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  // Create comparison file to verify data.txt:
  ofstream verify("data2.txt");
  assure(verify, "data2.txt");
  DataPoint d;
  while(bindata.read(
    (unsigned char*)&d, sizeof d))
    d.print(verify);
  bindata.clear(); // Reset state to "good"
  // Display user-selected records:
  int recnum = 0;
  // Left-align everything:
  cout.setf(ios::left, ios::adjustfield);
  // Fixed precision of 4 decimal places:
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(4);
  for(;;) {
    bindata.seekg(recnum* sizeof d, ios::beg);
    cout << "record " << recnum << endl;
    if(bindata.read(
      (unsigned char*)&d, sizeof d)) {
      cout << asctime(&(d.getTime()));
      cout << setw(11) << "Latitude"
           << setw(11) << "Longitude"
           << setw(10) << "Depth"
           << setw(12) << "Temperature"
           << endl;
      // Put a line after the description:
      cout << setfill('-') << setw(43) << '-'
           << setfill(' ') << endl;
      cout << setw(11) << d.getLatitude()
           << setw(11) << d.getLongitude()
           << setw(10) << d.getDepth()
           << setw(12) << d.getTemperature()
           << endl;
    } else {
      cout << "invalid record number" << endl;
```

```
      bindata.clear(); // Reset state to "good"
    }
    cout << endl
      << "enter record number, x to quit:";
    char buf[10];
    cin.getline(buf, 10);
    if(buf[0] == 'x') break;
    istrstream input(buf, 10);
    input >> recnum;
  }
} ///:~
```

The **ifstream bindata** is created from DATA.BIN as a binary file, with the **ios::nocreate** flag on to cause the **assert( )** to fail if the file doesn't exist. The **read( )** statement reads a single record and places it directly into the **DataPoint d**. (Again, if **DataPoint** contained pointers this would result in meaningless pointer values.) This **read( )** action will set **bindata**'s **failbit** when the end of the file is reached, which will cause the **while** statement to fail. At this point, however, you can't move the get pointer back and read more records because the state of the stream won't allow further reads. So the **clear( )** function is called to reset the **failbit**.

Once the record is read in from disk, you can do anything you want with it, such as perform calculations or make graphs. Here, it is displayed to further exercise your knowledge of iostream formatting.

The rest of the program displays a record number (represented by **recnum**) selected by the user. As before, the precision is fixed at four decimal places, but this time everything is left justified.

The formatting of this output looks different from before:

```
record 0
Tue Nov 16 18:15:49 1993
Latitude   Longitude  Depth    Temperature
----------------------------------------
45*20'31"  22*34'18"  186.0172  269.0167
```

To make sure the labels and the data columns line up, the labels are put in the same width fields as the columns, using **setw( )**. The line in between is generated by setting the fill character to '-', the width to the desired line width, and outputting a single '-'.

If the **read( )** fails, you'll end up in the **else** part, which tells the user the record number was invalid. Then, because the **failbit** was set, it must be

reset with a call to **clear( )** so the next **read( )** is successful (assuming it's in the right range).

Of course, you can also open the binary data file for writing as well as reading. This way you can retrieve the records, modify them, and write them back to the same location, thus creating a flat-file database management system. In my very first programming job, I also had to create a flat-file DBMS – but using BASIC on an Apple II. It took months, while this took minutes. Of course, it might make more sense to use a packaged DBMS now, but with C++ and iostreams you can still do all the low-level operations that are necessary in a lab.

Counting editor

Often you have some editing task where you must go through and sequentially number something, but all the other text is duplicated. I encountered this problem when pasting digital photos into a Web page – I got the formatting just right, then duplicated it, then had the problem of incrementing the photo number for each one. So I replaced the photo number with XXX, duplicated that, and wrote the following program to find and replace the "XXX" with an incremented count. Notice the formatting, so the value will be "001," "002," etc.:

```
//: C18:NumberPhotos.cpp
// Find the marker "XXX" and replace it with an
// incrementing number whereever it appears. Used
// to help format a web page with photos in it
#include "../require.h"
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 2);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  ofstream out(argv[2]);
  assure(out, argv[2]);
  string line;
  int counter = 1;
  while(getline(in, line)) {
```

```
    int xxx = line.find("XXX");
    if(xxx != string::npos) {
      ostringstream cntr;
      cntr << setfill('0') << setw(3) << counter++;
      line.replace(xxx, 3, cntr.str());
    }
    out << line << endl;
  }
} ///:~
```

Breaking up big files

This program was created to break up big files into smaller ones, in particular so they could be more easily downloaded from an Internet server (since hangups sometimes occur, this allows someone to download a file a piece at a time and then re-assemble it at the client end). You'll note that the program also creates a reassembly batch file for DOS (where it is messier), whereas under Linux/Unix you simply say something like "**cat *piece* > destination.file**".

This program reads the entire file into memory, which of course relies on having a 32-bit operating system with virtual memory for big files. It then pieces it out in chunks to the smaller files, generating the names as it goes. Of course, you can come up with a possibly more reasonable strategy that reads a chunk, creates a file, reads another chunk, etc.

Note that this program can be run on the server, so you only have to download the big file once and then break it up once it's on the server.

```
//: C18:Breakup.cpp
// Breaks a file up into smaller files for
// easier downloads
#include "../require.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <strstream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
```

```
  ifstream in(argv[1], ios::binary);
  assure(in, argv[1]);
  in.seekg(0, ios::end); // End of file
  long fileSize = in.tellg(); // Size of file
  cout << "file size = " << fileSize << endl;
  in.seekg(0, ios::beg); // Start of file
  char* fbuf = new char[fileSize];
  require(fbuf != 0);
  in.read(fbuf, fileSize);
  in.close();
  string infile(argv[1]);
  int dot = infile.find('.');
  while(dot != string::npos) {
    infile.replace(dot, 1, "-");
    dot = infile.find('.');
  }
  string batchName(
    "DOSAssemble" + infile + ".bat");
  ofstream batchFile(batchName.c_str());
  batchFile << "copy /b ";
  int filecount = 0;
  const int sbufsz = 128;
  char sbuf[sbufsz];
  const long pieceSize = 1000L * 100L;
  long byteCounter = 0;
  while(byteCounter < fileSize) {
    ostrstream name(sbuf, sbufsz);
    name << argv[1] << "-part" << setfill('0')
      << setw(2) << filecount++ << ends;
    cout << "creating " << sbuf << endl;
    if(filecount > 1)
      batchFile << "+";
    batchFile << sbuf;
    ofstream out(sbuf, ios::out | ios::binary);
    assure(out, sbuf);
    long byteq;
    if(byteCounter + pieceSize < fileSize)
      byteq = pieceSize;
    else
      byteq = fileSize - byteCounter;
    out.write(fbuf + byteCounter, byteq);
    cout << "wrote " << byteq << " bytes, ";
```

```
    byteCounter += byteq;
    out.close();
    cout << "ByteCounter = " << byteCounter
      << ", fileSize = " << fileSize << endl;
  }
  batchFile << " " << argv[1] << endl;
} ///:~
```

Summary

This chapter has given you a fairly thorough introduction to the iostream class library. In all likelihood, it is all you need to create programs using iostreams. (In later chapters you'll see simple examples of adding iostream functionality to your own classes.) However, you should be aware that there are some additional features in iostreams that are not used often, but which you can discover by looking at the iostream header files and by reading your compiler's documentation on iostreams.

Exercises

1. Open a file by creating an **ifstream** object called **in**. Make an **ostrstream** object called **os**, and read the entire contents into the **ostrstream** using the **rdbuf( )** member function. Get the address of **os**'s **char*** with the **str( )** function, and capitalize every character in the file using the Standard C **toupper( )** macro. Write the result out to a new file, and **delete** the memory allocated by **os**.
2. Create a program that opens a file (the first argument on the command line) and searches it for any one of a set of words (the remaining arguments on the command line). Read the input a line at a time, and print out the lines (with line numbers) that match.
3. Write a program that adds a copyright notice to the beginning of all source-code files. This is a small modification to exercise 1.
4. Use your favorite text-searching program (**grep**, for example) to output the names (only) of all the files that contain a particular pattern. Redirect the output into a file.

Write a program that uses the contents of that file to generate a batch file that invokes your editor on each of the files found by the search program.

19: Templates in depth

Nontype template arguments

Here is a random number generator class that always produces a unique number and overloads **operator( )** to produce a familiar function-call syntax:

```
//: C19:Urand.h
// Unique random number generator
#ifndef URAND_H
#define URAND_H
#include <cstdlib>
#include <ctime>

template<int upperBound>
class Urand {
  int used[upperBound];
  bool recycle;
public:
  Urand(bool recycle = false);
  int operator()(); // The "generator" function
};

template<int upperBound>
Urand<upperBound>::Urand(bool recyc)
  : recycle(recyc) {
```

```
  memset(used, 0, upperBound * sizeof(int));
  srand(time(0)); // Seed random number generator
}

template<int upperBound>
int Urand<upperBound>::operator()() {
  if(!memchr(used, 0, upperBound)) {
    if(recycle)
      memset(used,0,sizeof(used) * sizeof(int));
    else
      return -1; // No more spaces left
  }
  int newval;
  while(used[newval = rand() % upperBound])
    ; // Until unique value is found
  used[newval]++; // Set flag
  return newval;
}
#endif // URAND_H ///:~
```

The uniqueness of **Urand** is produced by keeping a map of all the numbers possible in the random space (the upper bound is set with the template argument) and marking each one off as it's used. The optional second constructor argument allows you to reuse the numbers once they're all used up. Notice that this implementation is optimized for speed by allocating the entire map, regardless of how many numbers you're going to need. If you want to optimize for size, you can change the underlying implementation so it allocates storage for the map dynamically and puts the random numbers themselves in the map rather than flags. Notice that this change in implementation will not affect any client code.

Default template arguments

The typename keyword

Consider the following:

```
//: C19:TypenamedID.cpp
```

```
// Using 'typename' to say it's a type,
// and not something other than a type

template<class T> class X {
  // Without typename, you should get an error:
  typename T::id i;
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  Y y;
  X<Y> xy;
  xy.f();
} ///:~
```

The template definition assumes that the class **T** that you hand it must have a nested identifier of some kind called **id**. But **id** could be a member object of **T**, in which case you can perform operations on **id** directly, but you couldn't "create an object" of "the type **id**." However, that's exactly what is happening here: the identifier **id** is being treated as if it were actually a nested type inside **T**. In the case of class **Y**, **id** is in fact a nested type, but (without the **typename** keyword) the compiler can't know that when it's compiling **X**.

If, when it sees an identifier in a template, the compiler has the option of treating that identifier as a type or as something other than a type, then it will assume that the identifier refers to something other than a type. That is, it will assume that the identifier refers to an object (including variables of primitive types), an enumeration or soemthing similar. However, it will not – cannot – just assume that it is a type. Thus, the compiler gets confused when we pretend it's a type.

The **typename** keyword tells the compiler to interpret a particular name as a type. It must be used for a name that:

1. Is a qualified name, one that is nested within another type.
2. Depends on a template argument. That is, a template argument is somehow involved in the name. The template argument causes the ambiguity when the compiler makes the simplest assumption: that the name refers to something other than a type.

Because the default behavior of the compiler is to assume that a name that fits the above two points is not a type, you must use **typename** even in places where you think that the compiler ought to be able to figure out the right way to interpret the name on its own. In the above example, when the compiler sees **T::id**, it knows (because of the **typename** keyword) that **id** refers to a nested type and thus it can create an object of that type.

The short version of the rule is: if your type is a qualified name that involves a template argument, you must use **typename**.

Typedefing a typename

The **typename** keyword does not automatically create a **typedef**. A line which reads:

```
typename Seq::iterator It;
```

causes a variable to be declared of type **Seq::iterator**. If you mean to make a **typedef**, you must say:

```
typedef typename Seq::iterator It;
```

Using **typename** instead of **class**

With the introduction of the **typename** keyword, you now have the option of using **typename** instead of **class** in the template argument list of a template definition. This may produce code which is clearer:

```
//: C19:UsingTypename.cpp
// Using 'typename' in the template argument list

template<typename T> class X { };

int main() {
  X<int> x;
```

```
} ///:~
```

You'll probably see a great deal of code which does not use **typename** in this fashion, since the keyword was added to the language a relativly long time after templates were introduced.

Function templates

A class template describes an infinite set of classes, and the most common place you'll see templates is with classes. However, C++ also supports the concept of an infinite set of functions, which is sometimes useful. The syntax is virtually identical, except that you create a function instead of a class.

The clue that you should create a function template is, as you might suspect, if you find you're creating a number of functions that look identical except that they are dealing with different types. The classic example of a function template is a sorting function.[56] However, a function template is useful in all sorts of places, as demonstrated in the first example that follows. The second example shows a function template used with containers and iterators.

A string conversion system

```
//: C19:stringConv.h
// Chuck Allison's string converter
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>

template<typename T>
T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
```

[56] See *C++ Inside & Out* (Osborne/McGraw-Hill, 1993) by the author, Chapter 10.

```
}

template<typename T>
std::string toString(const T& t) {
  std::ostringstream s;
  s << t;
  return s.str();
}
#endif // STRINGCONV_H ///:~
```

Here's a test program, that includes the use of the Standard Library **complex** number type:

```
//: C19:stringConvTest.cpp
#include "stringConv.h"
#include <iostream>
#include <complex>
using namespace std;

int main() {
  int i = 1234;
  cout << "i == \"" << toString(i) << "\"\n";
  float x = 567.89;
  cout << "x == \"" << toString(x) << "\"\n";
  complex<float> c(1.0, 2.0);
  cout << "c == \"" << toString(c) << "\"\n";
  cout << endl;

  i = fromString<int>(string("1234"));
  cout << "i == " << i << endl;
  x = fromString<float>(string("567.89"));
  cout << "x == " << x << endl;
  c = fromString< complex<float> >(string("(1.0,2.0)"));
  cout << "c == " << c << endl;
} ///:~
```

The output is what you'd expect:

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
```

```
c == (1,2)
```

A memory allocation system

There's a few things you can do to make the raw memory allocation routines **malloc( )**, **calloc( )** and **realloc( )** safer. The following function template produces one function **getmem( )** that either allocates a new piece of memory or resizes an existing piece (like **realloc( )**). In addition, it zeroes *only the new memory*, and it checks to see that the memory is successfully allocated. Also, you only tell it the number of elements of the type you want, not the number of bytes, so the possibility of a programmer error is reduced. Here's the header file:

```
//: C19:Getmem.h
// Function template for memory
#ifndef GETMEM_H
#define GETMEM_H
#include "../require.h"
#include <cstdlib>
#include <cstring>

template<class T>
void getmem(T*& oldmem, int elems) {
  typedef int cntr; // Type of element counter
  const int csz = sizeof(cntr); // And size
  const int tsz = sizeof(T);
  if(elems == 0) {
    free(&(((cntr*)oldmem)[-1]));
    return;
  }
  T* p = oldmem;
  cntr oldcount = 0;
  if(p) { // Previously allocated memory
    // Old style:
    // ((cntr*)p)--; // Back up by one cntr
    // New style:
    cntr* tmp = reinterpret_cast<cntr*>(p);
    p = reinterpret_cast<T*>(--tmp);
    oldcount = *(cntr*)p; // Previous # elems
  }
  T* m = (T*)realloc(p, elems * tsz + csz);
```

```
  require(m != 0);
  *((cntr*)m) = elems; // Keep track of count
  const cntr increment = elems - oldcount;
  if(increment > 0) {
    // Starting address of data:
    long startadr = (long)&(m[oldcount]);
    startadr += csz;
    // Zero the additional new memory:
    memset((void*)startadr, 0, increment * tsz);
  }
  // Return the address beyond the count:
  oldmem = (T*)&(((cntr*)m)[1]);
}

template<class T>
inline void freemem(T * m) { getmem(m, 0); }

#endif // GETMEM_H ///:~
```

To be able to zero only the new memory, a counter indicating the number of elements allocated is attached to the beginning of each block of memory. The **typedef cntr** is the type of this counter; it allows you to change from **int** to **long** if you need to handle larger chunks (other issues come up when using **long**, however – these are seen in compiler warnings).

A pointer reference is used for the argument **oldmem** because the outside variable (a pointer) must be changed to point to the new block of memory. **oldmem** must point to zero (to allocate new memory) or to an existing block of memory *that was created with* **getmem( )**. This function assumes you're using it properly, but for debugging you could add an additional tag next to the counter containing an identifier, and check that identifier in **getmem( )** to help discover incorrect calls.

If the number of elements requested is zero, the storage is freed. There's an additional function template **freemem( )** that aliases this behavior.

You'll notice that **getmem( )** is very low-level – there are lots of casts and byte manipulations. For example, the **oldmem** pointer doesn't point to the true beginning of the memory block, but just *past* the beginning to allow for the counter. So to **free( )** the memory block, **getmem( )** must back up the pointer by the amount of space occupied by **cntr**. Because **oldmem** is a **T***, it must first be cast to a **cntr***, then indexed backwards

one place. Finally the address of that location is produced for **free( )** in the expression:

```
free(&(((cntr*)oldmem)[-1]));
```

Similarly, if this is previously allocated memory, **getmem( )** must back up by one **cntr** size to get the true starting address of the memory, and then extract the previous number of elements. The true starting address is required inside **realloc( )**. If the storage size is being increased, then the difference between the new number of elements and the old number is used to calculate the starting address and the amount of memory to zero in **memset( )**. Finally, the address beyond the count is produced to assign to **oldmem** in the statement:

```
oldmem = (T*)&(((cntr*)m)[1]);
```

Again, because **oldmem** is a reference to a pointer, this has the effect of changing the outside argument passed to **getmem( )**.

Here's a program to test **getmem( )**. It allocates storage and fills it up with values, then increases that amount of storage:

```
//: C19:Getmem.cpp
// Test memory function template
#include "Getmem.h"
#include <iostream>
using namespace std;

int main() {
  int* p = 0;
  getmem(p, 10);
  for(int i = 0; i < 10; i++) {
    cout << p[i] << ' ';
    p[i] = i;
  }
  cout << '\n';
  getmem(p, 20);
  for(int j = 0; j < 20; j++) {
    cout << p[j] << ' ';
    p[j] = j;
  }
  cout << '\n';
  getmem(p, 25);
  for(int k = 0; k < 25; k++)
    cout << p[k] << ' ';
```

```
  freemem(p);
  cout << '\n';

  float* f = 0;
  getmem(f, 3);
  for(int u = 0; u < 3; u++) {
    cout << f[u] << ' ';
    f[u] = u + 3.14159;
  }
  cout << '\n';
  getmem(f, 6);
  for(int v = 0; v < 6; v++)
    cout << f[v] << ' ';
  freemem(f);
} ///:~
```

After each **getmem( )**, the values in memory are printed out to show that the new ones have been zeroed.

Notice that a different version of **getmem( )** is instantiated for the **int** and **float** pointers. You might think that because all the manipulations are so low-level you could get away with a single non-template function and pass a **void*&** as **oldmem**. This doesn't work because then the compiler must do a conversion from your type to a **void***. To take the reference, it makes a temporary. This produces an error because then you're modifying the temporary pointer, not the pointer you want to change. So the function template is necessary to produce the exact type for the argument.

Type induction in function templates

As a simple but very useful example, consider the following:

```
//: :arraySize.h
// Uses template type induction to
// discover the size of an array
#ifndef ARRAYSIZE_H
#define ARRAYSIZE_H

template<typename T, int size>
```

```
int asz(T (&)[size]) { return size; }

#endif // ARRAYSIZE_H ///:~
```

This actually figures out the size of an array as a compile-time constant value, without using any **sizeof( )** operations! Thus you can have a much more succinct way to calculate the size of an array at compile time:

```
//: C19:ArraySize.cpp
// The return value of the template function
// asz() is a compile-time constant
#include "../arraySize.h"

int main() {
  int a[12], b[20];
  const int sz1 = asz(a);
  const int sz2 = asz(b);
  int c[sz1], d[sz2];
} ///:~
```

Of course, just making a variable of a built-in type a **const** does not guarantee it's actually a compile-time constant, but if it's used to define the size of an array (as it is in the last line of **main( )**), then it *must* be a compile-time constant.

Taking the address of a generated function template

There are a number of situations where you need to take the address of a function. For example, you may have a function that takes an argument of a pointer to another function. Of course it's possible that this other function might be generated from a template function so you need some way to take that kind of address[57]:

```
//: C19:TemplateFunctionAddress.cpp
// Taking the address of a function generated
```

[57] I am indebted to Nathan Myers for this example.

```
// from a template.

template <typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template <class T>
  void g(void (*pf)(T*)) {}

int main() {
  // Full type exposition:
  h(&f<int>);
  // Type induction:
  h(&f);
  // Full type exposition:
  g<int>(&f<int>);
  // Type inductions:
  g(&f<int>);
  g<int>(&f);
} ///:~
```

This example demonstrates a number of different issues. First, even though you're using templates, the signatures must match – the function **h( )** takes a pointer to a function that takes an **int*** and returns **void**, and that's what the template **f** produces. Second, the function that wants the function pointer as an argument can itself be a template, as in the case of the template **g**.

In **main( )** you can see that type induction works here, too. The first call to **h( )** explicitly gives the template argument for **f**, but since **h( )** says that it will only take the address of a function that takes an **int***, that part can be induced by the compiler. With **g( )** the situation is even more interesting because there are two templates involved. The compiler cannot induce the type with nothing to go on, but if either **f** or **g** is given **int**, then the rest can be induced.

Local classes in templates

Applying a function to an STL sequence

Suppose you want to take an STL sequence container (which you'll learn more about in subsequent chapters; for now we can just use the familiar **vector**) and apply a function to all the objects it contains. Because a **vector** can contain any type of object, you need a function that works with any type of **vector** and any type of object it contains:

```
//: C19:applySequence.h
// Apply a function to an STL sequence container

// 0 arguments, any type of return value:
template<class Seq, class T, class R>
void apply(Seq& sq, R (T::*f)()) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it)->*f)();
    it++;
  }
}

// 1 argument, any type of return value:
template<class Seq, class T, class R, class A>
void apply(Seq& sq, R(T::*f)(A), A a) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it)->*f)(a);
    it++;
  }
}

// 2 arguments, any type of return value:
template<class Seq, class T, class R,
```

```
        class A1, class A2>
void apply(Seq& sq, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  typename Seq::iterator it = sq.begin();
  while(it != sq.end()) {
    ((*it)->*f)(a1, a2);
    it++;
  }
}
// Etc., to handle maximum likely arguments ///:~
```

The **apply( )** function template takes a reference to the container class and a pointer-to-member for a member function of the objects contained in the class. It uses an iterator to move through the **Stack** and apply the function to every object. If you've (understandably) forgotten the pointer-to-member syntax, you can refresh your memory at the end of Chapter XX.

Notice that there are no STL header files (or any header files, for that matter) included in **applySequence.h**, so it is actually not limited to use with an STL sequence. However, it does make assumptions (primarily, the name and behavior of the **iterator**) that apply to STL sequences.

You can see there is more than one version of **apply( )**, so it's possible to overload function templates. Although they all take any type of return value (which is ignored, but the type information is required to match the pointer-to-member), each version takes a different number of arguments, and because it's a template, those arguments can be of any type. The only limitation here is that there's no "super template" to create templates for you; thus you must decide how many arguments will ever be required.

To test the various overloaded versions of **apply( )**, the class **Gromit**[58] is created containing functions with different numbers of arguments:

```
//: C19:Gromit.h
// The techno-dog. Has member functions
// with various numbers of arguments.
#include <iostream>

class Gromit {
  int arf;
public:
```

[58] A reference to the British animated short *The Wrong Trousers* by Nick Park.

```
  Gromit(int arf = 1) : arf(arf + 1) {}
  void speak(int) {
    for(int i = 0; i < arf; i++)
      std::cout << "arf! ";
    std::cout << std::endl;
  }
  char eat(float) {
    std::cout << "chomp!" << std::endl;
    return 'z';
  }
  int sleep(char, double) {
    std::cout << "zzz..." << std::endl;
    return 0;
  }
  void sit(void) {}
}; ///:~
```

Now the **apply( )** template functions can be combined with a **vector<Gromit*>** to make a container that will call member functions of the contained objects, like this:

```
//: C19:applyGromit.cpp
// Test applySequence.h
#include "Gromit.h"
#include "applySequence.h"
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<Gromit*> dogs;
  for(int i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  apply(dogs, &Gromit::speak, 1);
  apply(dogs, &Gromit::eat, 2.0f);
  apply(dogs, &Gromit::sleep, 'z', 3.0);
  apply(dogs, &Gromit::sit);
} ///:~
```

Although the definition of **apply( )** is somewhat complex and not something you'd ever expect a novice to understand, its use is remarkably clean and simple, and a novice could easily use it knowing only *what* it is intended to accomplish, not *how*. This is the type of division you should strive for in all of your program components: The tough details are all

isolated on the designer's side of the wall, and users are concerned only with accomplishing their goals, and don't see, know about, or depend on details of the underlying implementation

Template-templates

```
//: C19:TemplateTemplate.cpp
#include <vector>
#include <iostream>
#include <string>
using namespace std;

// As long as things are simple,
// this approach works fine:
template<typename C>
void print1(C& c) {
  typename C::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
}

// Template-template argument must
// be a class; cannot use typename:
template<typename T, template<typename> class C>
void print2(C<T>& c) {
  copy(c.begin(), c.end(),
    ostream_iterator<T>(cout, " "));
  cout << endl;
}

int main() {
  vector<string> v(5, "Yow!");
  print1(v);
  print2(v);
} ///:~
```

Member function templates

It's also possible to make **apply( )** a *member function template* of the class. That is, a separate template definition from the class' template, and yet a member of the class. This may produce a cleaner syntax:

```
dogs.apply(&Gromit::sit);
```

This is analogous to the act (in Chapter XX) of bringing ordinary functions inside a class.[59]

The definition of the **apply( )** functions turn out to be cleaner, as well, because they are members of the container. To accomplish this, a new container is inherited from one of the existing STL sequence containers and the member function templates are added to the new type. However, for maximum flexibility we'd like to be able to use any of the STL sequence containers, and for this to work a *template-template* must be used, to tell the compiler that a template argument is actually a template, itself, and can thus take a type argument and be instantiated. Here is what it looks like after bringing the **apply( )** functions into the new type as member functions:

```
//: C19:applyMember.h
// applySequence.h modified to use
// member function templates

template<class T, template<typename> class Seq>
class SequenceWithApply : public Seq<T*> {
public:
  // 0 arguments, any type of return value:
  template<class R>
  void apply(R (T::*f)()) {
    iterator it = begin();
    while(it != end()) {
      ((*it)->*f)();
      it++;
    }
```

[59] Check your compiler version information to see if it supports member function templates.

```
  }
  // 1 argument, any type of return value:
  template<class R, class A>
  void apply(R(T::*f)(A), A a) {
    iterator it = begin();
    while(it != end()) {
      ((*it)->*f)(a);
      it++;
    }
  }
  // 2 arguments, any type of return value:
  template<class R, class A1, class A2>
  void apply(R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
    iterator it = begin();
    while(it != end()) {
      ((*it)->*f)(a1, a2);
      it++;
    }
  }
}; ///:~
```

Because they are members, the **apply( )** functions don't need as many arguments, and the **iterator** class doesn't need to be qualified. Also, **begin( )** and **end( )** are now member functions of the new type and so look cleaner as well. However, the basic code is still the same.

You can see how the function calls are also simpler for the client programmer:

```
//: C19:applyGromit2.cpp
// Test applyMember.h
#include "Gromit.h"
#include "applyMember.h"
#include <vector>
#include <iostream>
using namespace std;

int main() {
  SequenceWithApply<Gromit, vector> dogs;
  for(int i = 0; i < 5; i++)
    dogs.push_back(new Gromit(i));
  dogs.apply(&Gromit::speak, 1);
  dogs.apply(&Gromit::eat, 2.0f);
```

```
  dogs.apply(&Gromit::sleep, 'z', 3.0);
  dogs.apply(&Gromit::sit);
} ///:~
```

Conceptually, it reads more sensibly to say that you're calling **apply( )** for the **dogs** container.

Why virtual member template functions are disallowed

Nested template classes

Template specializations

Full specialization

Partial Specialization

A practical example

There's nothing to prevent you from using a class template in any way you'd use an ordinary class. For example, you can easily inherit from a template, and you can create a new template that instantiates and inherits from an existing template. If the **vector** class does everything you want, but you'd also like it to sort itself, you can easily reuse the code and add value to it:

```
//: C19:Sorted.h
// Template specialization
#ifndef SORTED_H
#define SORTED_H
#include <vector>

template<class T>
class Sorted : public std::vector<T> {
public:
  void sort();
};
```

```
template<class T>
void Sorted<T>::sort() { // A bubble sort
  for(int i = size(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(at(j-1) > at(j)) {
        // Swap the two elements:
        T t = at(j-1);
        at(j-1) = at(j);
        at(j) = t;
      }
}

// Partial specialization for pointers:
template<class T>
class Sorted<T*> : public std::vector<T*> {
public:
  void sort();
};

template<class T>
void Sorted<T*>::sort() {
  for(int i = size(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(*at(j-1) > *at(j)) {
        // Swap the two elements:
        T* t = at(j-1);
        at(j-1) = at(j);
        at(j) = t;
      }
}

// Full specialization for char*:
template<>
void Sorted<char*>::sort() {
  for(int i = size(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(strcmp(at(j-1), at(j)) > 0) {
        // Swap the two elements:
        char* t = at(j-1);
        at(j-1) = at(j);
        at(j) = t;
```

```
    }
}
#endif // SORTED_H ///:~
```

The **Sorted** template imposes a restriction on all classes it is instantiated for: They must contain a **>** operator. In **SString** this is added explicitly, but in **Integer** the automatic type conversion **operator int** provides a path to the built-in **>** operator. When a template provides more functionality for you, the trade-off is usually that it puts more requirements on your class. Sometimes you'll have to inherit the contained class to add the required functionality. Notice the value of using an overloaded operator here – the **Integer** class can rely on its underlying implementation to provide the functionality.

The default **Sorted** template only works with objects (including objects of built-in types). However, it won't sort pointers to objects so the partial specialization is necessary. Even then, the code generated by the partial specialization won't sort an array of **char***. To solve this, the full specialization compares the **char*** elements using **strcmp( )** to produce the proper behavior.

Here's a test for **Sorted.h** that uses the unique random number generator introduced earlier in the chapter:

```
//: C19:Sorted.cpp
// Testing template specialization
#include "Sorted.h"
#include "Urand.h"
#include "../arraySize.h"
#include <iostream>
#include <string>
using namespace std;

char* words[] = {
  "is", "running", "big", "dog", "a",
};
char* words2[] = {
  "this", "that", "theother",
};

int main() {
  Sorted<int> is;
  Urand<47> rand;
  for(int i = 0; i < 15; i++)
```

```
  is.push_back(rand());
for(int l = 0; l < is.size(); l++)
  cout << is[l] << ' ';
cout << endl;
is.sort();
for(int l = 0; l < is.size(); l++)
  cout << is[l] << ' ';
cout << endl;

// Uses the template partial specialization:
Sorted<string*> ss;
for(int i = 0; i < asz(words); i++)
  ss.push_back(new string(words[i]));
for(int i = 0; i < ss.size(); i++)
  cout << *ss[i] << ' ';
cout << endl;
ss.sort();
for(int i = 0; i < ss.size(); i++)
  cout << *ss[i] << ' ';
cout << endl;

// Uses the full char* specialization:
Sorted<char*> scp;
for(int i = 0; i < asz(words2); i++)
  scp.push_back(words2[i]);
for(int i = 0; i < scp.size(); i++)
  cout << scp[i] << ' ';
cout << endl;
scp.sort();
for(int i = 0; i < scp.size(); i++)
  cout << scp[i] << ' ';
cout << endl;
} ///:~
```

Each of the template instantiations uses a different version of the template. **Sorted<int>** uses the "ordinary," non-specialized template. **Sorted<string*>** uses the partial specialization for pointers. Lastly, **Sorted<char*>** uses the full specialization for **char***. Note that without this full specialization, you could be fooled into thinking that things were working correctly because the **words** array would still sort out to "a big dog is running" since the partial specialization would end up comparing the first character of each array. However, **words2** would not sort out correctly, and for the desired behavior the full specialization is necessary.

Pointer specialization

Partial ordering of function templates

Design & efficiency

In **Sorted**, every time you call **add( )** the element is inserted and the array is resorted. Here, the horribly inefficient and greatly deprecated (but easy to understand and code) bubble sort is used. This is perfectly appropriate, because it's part of the **private** implementation. During program development, your priorities are to

1. Get the class interfaces correct.
2. Create an accurate implementation as rapidly as possible so you can:
3. Prove your design.

Very often, you will discover problems with the class interface only when you assemble your initial "rough draft" of the working system. You may also discover the need for "helper" classes like containers and iterators during system assembly and during your first-pass implementation. Sometimes it's very difficult to discover these kinds of issues during analysis – your goal in analysis should be to get a big-picture design that can be rapidly implemented and tested. Only after the design has been proven should you spend the time to flesh it out completely and worry about performance issues. If the design fails, or if performance is not a problem, the bubble sort is good enough, and you haven't wasted any time. (Of course, the ideal solution is to use someone else's sorted container; the Standard C++ template library is the first place to look.)

Preventing template bloat

Each time you instantiate a template, the code in the template is generated anew (except for **inline** functions). If some of the functionality of a template does not depend on type, it can be put in a common base class to prevent needless reproduction of that code. For example, in Chapter XX in **InheritStack.cpp** inheritance was used to specify the types that a **Stack** could accept and produce. Here's the templatized version of that code:

```
//: C19:Nobloat.h
// Templatized InheritStack.cpp
#ifndef NOBLOAT_H
```

```
#define NOBLOAT_H
#include "Stack4.h"

template<class T>
class NBStack : public Stack {
public:
  void push(T* str) {
    Stack::push(str);
  }
  T* peek() const {
    return (T*)Stack::peek();
  }
  T* pop() {
    return (T*)Stack::pop();
  }
  ~NBStack();
};

// Defaults to heap objects & ownership:
template<class T>
NBStack<T>::~NBStack() {
  T* top = pop();
  while(top) {
    delete top;
    top = pop();
  }
}
#endif // NOBLOAT_H ///:~
```

As before, the inline functions generate no code and are thus “free.” The functionality is provided by creating the base-class code only once. However, the ownership problem has been solved here by adding a destructor (which *is* type-dependent, and thus must be created by the template). Here, it defaults to ownership. Notice that when the base-class destructor is called, the stack will be empty so no duplicate releases will occur.

Explicit instantiation

At times it is useful to explicitly instantiate a template; that is, to tell the compiler to lay down the code for a specific version of that template even

though you're not creating an object at that point. To do this, you reuse the **template** keyword as follows:

```
template class Bobbin<thread>;
template void sort<char>(char*[]);
```

Here's a version of the **Sorted.cpp** example that explicitly instantiates a template before using it:

```
//: C19:ExplicitInstantiation.cpp
#include "Urand.h"
#include "Sorted.h"
#include <iostream>
using namespace std;

// Explicit instantiation:
template class Sorted<int>;

int main() {
  Sorted<int> is;
  Urand<47> rand1;
  for(int k = 0; k < 15; k++)
    is.push_back(rand1());
  is.sort();
  for(int l = 0; l < is.size(); l++)
    cout << is[l] << endl;
} ///:~
```

In this example, the explicit instantiation doesn't really accomplish anything; the program would work the same without it. Explicit instantiation is only for special cases where extra control is needed.

Explicit specification of template functions

Controlling template instantiation

Normally templates are not instantiated until they are needed. For function templates this just means the point at which you call the

function, but for class templates it's more granular than that: each individual member function of the template is not instantiated until the first point of use. This means that only the member functions you actually use will be instantiated, which is quite important since it allows greater freedom in what the template can be used with. For example:

```
//: C19:DelayedInstantiation.cpp
// Member functions of class templates are not
// instantiated until they're needed.

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

class Y {
public:
  void g() {}
};

template <typename T> class Z {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};

int main() {
  Z<X> zx;
  zx.a(); // Doesn't create Z<X>::b()
  Z<Y> zy;
  zy.b(); // Doesn't create Z<Y>::a()
} ///:~
```

Here, even though the template purports to use both **f( )** and **g( )** member functions of **T**, the fact that the program compiles shows you that it only generates **Z<X>::a( )** when it is explicitly called for **zx** (if **Z<X>::b( )** were also generated at the same time, a compile-time error message would be generated). Similarly, the call to **zy.b( )** doesn't generate **Z<Y>::a( )**. As a result, the **Z** template can be used with **X** and **Y**, whereas if all the member functions were generated when the class was first created it would significantly limit the use of many templates.

The inclusion vs. separation models

The export keyword

Template programming idioms

The “curiously-recurring template”

Traits

Summary

One of the greatest weaknesses of C++ templates will be shown to you when you try to write code that uses templates, especially STL code (introduced in the next two chapters), and start getting compile-time error messages. When you’re not used to it, the quantity of inscrutable text that will be spewed at you by the compiler will be quite overwhelming. After a while you’ll adapt (although it always feels a bit barbaric), and if it’s any consolation, C++ compilers have actually gotten a lot *better* about this – previously they would only give the line where you tried to instantiate the template, and most of them now go to the line in the template definition that caused the problem.

The issue is that *a template implies an interface*. That is, even though the **template** keyword says “I’ll take any type,” the code in a template definition actually requires that certain operators and member functions be supported – that’s the interface. So in reality, a template definition is saying “I’ll take any type that supports this interface.” Things would be much nicer if the compiler could simply say “hey, this type that you’re trying to instantiate the template with doesn’t support that interface – can’t do it.” The Java language has a feature called **interface** that would

be a perfect match for this (Java, however, has no parameterized type mechanism), but it will be many years, if ever, before you will see such a thing in C++ (at this writing the C++ Standard has only just been accepted and it will be a while before all the compilers even achieve compliance). Compilers can only get so good at reporting template instantiation errors, so you'll have to grit your teeth, go to the first line reported as an error and figure it out.

20: STL Containers & Iterators

Container classes are the solution to a specific kind of code reuse problem. They are building blocks used to create object-oriented programs – they make the internals of a program much easier to construct.

A container class describes an object that holds other objects. Container classes are so important that they were considered fundamental to early object-oriented languages. In Smalltalk, for example, programmers think of the language as the program translator together with the class library, and a critical part of that library is the container classes. So it became natural that C++ compiler vendors also include a container class library. You'll note that the **vector** was so useful that it was introduced in its simplest form very early in this book.

Like many other early C++ libraries, early container class libraries followed Smalltalk's *object-based hierarchy*, which worked well for Smalltalk, but turned out to be awkward and difficult to use in C++. Another approach was required.

This chapter attempts to slowly work you into the concepts of the C++ *Standard Template Library* (STL), which is a powerful library of containers (as well as *algorithms*, but these are covered in the following chapter). In the past, I have taught that there is a relatively small subset of elements and ideas that you need to understand in order to get much of the usefulness from the STL. Although this can be true it turns out that

understanding the STL more deeply is important to gain the full power of the library. This chapter and the next probe into the STL containers and algorithms.

Containers and iterators

If you don't know how many objects you're going to need to solve a particular problem, or how long they will last, you also don't know how to store those objects. How can you know how much space to create? You can't, since that information isn't known until run time.

The solution to most problems in object-oriented design seems flippant: you create another type of object. For the storage problem, the new type of object holds other objects, or pointers to objects. Of course, you can do the same thing with an array, but there's more. This new type of object, which is typically referred to in C++ as a *container* (also called a *collection* in some languages), will expand itself whenever necessary to accommodate everything you place inside it. So you don't need to know how many objects you're going to hold in a collection. You just create a collection object and let it take care of the details.

Fortunately, a good OOP language comes with a set of containers as part of the package. In C++, it's the Standard Template Library (STL). In some libraries, a generic container is considered good enough for all needs, and in others (C++ in particular) the library has different types of containers for different needs: a vector for consistent access to all elements, and a linked list for consistent insertion at all elements, for example, so you can choose the particular type that fits your needs. These may include sets, queues, hash tables, trees, stacks, etc.

All containers have some way to put things in and get things out. The way that you place something into a container is fairly obvious. There's a function called "push" or "add" or a similar name. Fetching things out of a container is not always as apparent; if it's an array-like entity such as a vector, you might be able to use an indexing operator or function. But in many situations this doesn't make sense. Also, a single-selection function is restrictive. What if you want to manipulate or compare a group of elements in the container?

The solution is an *iterator*, which is an object whose job is to select the elements within a container and present them to the user of the iterator. As a class, it also provides a level of abstraction. This abstraction can be used to separate the details of the container from the code that's

accessing that container. The container, via the iterator, is abstracted to be simply a sequence. The iterator allows you to traverse that sequence without worrying about the underlying structure – that is, whether it's a vector, a linked list, a stack or something else. This gives you the flexibility to easily change the underlying data structure without disturbing the code in your program.

From the design standpoint, all you really want is a sequence that can be manipulated to solve your problem. If a single type of sequence satisfied all of your needs, there'd be no reason to have different kinds. There are two reasons that you need a choice of containers. First, containers provide different types of interfaces and external behavior. A stack has a different interface and behavior than that of a queue, which is different than that of a set or a list. One of these might provide a more flexible solution to your problem than the other. Second, different containers have different efficiencies for certain operations. The best example is a vector and a list. Both are simple sequences that can have identical interfaces and external behaviors. But certain operations can have radically different costs. Randomly accessing elements in a vector is a constant-time operation; it takes the same amount of time regardless of the element you select. However, in a linked list it is expensive to move through the list to randomly select an element, and it takes longer to find an element if it is further down the list. On the other hand, if you want to insert an element in the middle of a sequence, it's much cheaper in a list than in a vector. These and other operations have different efficiencies depending upon the underlying structure of the sequence. In the design phase, you might start with a list and, when tuning for performance, change to a vector. Because of the abstraction via iterators, you can change from one to the other with minimal impact on your code.

In the end, remember that a container is only a storage cabinet to put objects in. If that cabinet solves all of your needs, it doesn't really matter *how* it is implemented (a basic concept with most types of objects). If you're working in a programming environment that has built-in overhead due to other factors, then the cost difference between a vector and a linked list might not matter. You might need only one type of sequence. You can even imagine the "perfect" container abstraction, which can automatically change its underlying implementation according to the way it is used.

STL reference documentation

You will notice that this chapter does not contain exhaustive documentation describing each of the member functions in each STL container. Although I describe the member functions that I use, I've left the full descriptions to others: there are at least two very good on-line sources of STL documentation in HTML format that you can keep resident on your computer and view with a Web browser whenever you need to look something up. The first is the Dinkumware library (which covers the entire Standard C and C++ library) mentioned at the beginning of this book section (page XXX). The second is the freely-downloadable SGI STL and documentation, freely downloadable at http://www.sgi.com/Technology/STL/. These should provide complete references when you're writing code. In addition, the STL books listed in Appendix XX will provide you with other resources.

The Standard Template Library

The C++ STL[60] is a powerful library intended to satisfy the vast bulk of your needs for containers and algorithms, but in a completely portable fashion. This means that not only are your programs easier to port to other platforms, but that your knowledge itself does not depend on the libraries provided by a particular compiler vendor (and the STL is likely to be more tested and scrutinized than a particular vendor's library). Thus, it will benefit you greatly to look first to the STL for containers and algorithms, *before* looking at vendor-specific solutions.

A fundamental principle of software design is that *all problems can be simplified by introducing an extra level of indirection*. This simplicity is achieved in the STL using *iterators* to perform operations on a data structure while knowing as little as possible about that structure, thus producing data structure independence. With the STL, this means that any operation that can be performed on an array of objects can also be performed on an STL container of objects and vice versa. The STL

[60] Contributed to the C++ Standard by Alexander Stepanov and Meng Lee at Hewlett-Packard.

containers work just as easily with built-in types as they do with user-defined types. If you learn the library, it will work on everything.

The drawback to this independence is that you'll have to take a little time at first getting used to the way things are done in the STL. However, the STL uses a consistent pattern, so once you fit your mind around it, it doesn't change from one STL tool to another.

Consider an example using the STL **set** class. A set will allow only one of each object value to be inserted into itself. Here is a simple **set** created to work with **int**s by providing **int** as the template argument to **set**:

```
//: C20:Intset.cpp
// Simple use of STL set
#include <set>
#include <iostream>
using namespace std;

int main() {
 set<int> intset;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)
      // Try to insert multiple copies:
      intset.insert(j);
  // Print to output:
 copy(intset.begin(), intset.end(),
   ostream_iterator<int>(cout, "\n"));
} ///:~
```

The **insert( )** member does all the work: it tries putting the new element in and rejects it if it's already there. Very often the activities involved in using a set are simply insertion and a test to see whether it contains the element. You can also form a union, intersection, or difference of sets, and test to see if one set is a subset of another.

In this example, the values 0 - 9 are inserted into the set 25 times, and the results are printed out to show that only one of each of the values is actually retained in the set.

The **copy( )** function is actually the instantiation of an STL template function, of which there are many. These template functions are generally referred to as "the STL Algorithms" and will be the subject of the following chapter. However, several of the algorithms are so useful that they will be introduced in this chapter. Here, **copy( )** shows the use of iterators. The **set** member functions **begin( )** and **end( )** produce iterators as their

return values. These are used by **copy( )** as beginning and ending points for its operation, which is simply to move between the boundaries established by the iterators and copy the elements to the third argument, which is also an iterator, but in this case, a special type created for iostreams. This places **int** objects on **cout** and separates them with a newline.

Because of its genericity, **copy( )** is certainly not restricted to printing on a stream. It can be used in virtually any situation: it needs only three iterators to talk to. All of the algorithms follow the form of **copy( )** and simply manipulate iterators (the use of iterators is the "extra level of indirection").

Now consider taking the form of **Intset.cpp** and reshaping it to display a list of the words used in a document. The solution becomes remarkably simple.

```
//: C20:WordSet.cpp
#include "../require.h"
#include <string>
#include <fstream>
#include <iostream>
#include <set>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream source(argv[1]);
  assure(source, argv[1]);
  string word;
  set<string> words;
  while(source >> word)
    words.insert(word);
  copy(words.begin(), words.end(),
    ostream_iterator<string>(cout, "\n"));
  cout << "Number of unique words:"
    << words.size() << endl;
} ///:~
```

The only substantive difference here is that **string** is used instead of **int**. The words are pulled from a file, but everything else is the same as in **Intset.cpp**. The **operator>>** returns a whitespace-separated group of characters each time it is called, until there's no more input from the file. So it approximately breaks an input stream up into words. Each **string** is

placed in the **set** using **insert( )**, and the **copy( )** function is used to display the results. Because of the way **set** is implemented (as a tree), the words are automatically sorted.

Consider how much effort it would be to accomplish the same task in C, or even in C++ without the STL.

The basic concepts

The primary idea in the STL is the *container* (also known as a *collection*), which is just what it sounds like: a place to hold things. You need containers because objects are constantly marching in and out of your program and there must be someplace to put them while they're around. You can't make named local objects because in a typical program you don't know how many, or what type, or the lifetime of the objects you're working with. So you need a container that will expand whenever necessary to fill your needs.

All the containers in the STL hold objects and expand themselves. In addition, they hold your objects in a particular way. The difference between one container and another is the way the objects are held and how the sequence is created. Let's start by looking at the simplest containers.

A **vector** is a linear sequence that allows rapid random access to its elements. However, it's expensive to insert an element in the middle of the sequence, and is also expensive when it allocates additional storage. A **deque** is also a linear sequence, and it allows random access that's nearly as fast as **vector**, but it's significantly faster when it needs to allocate new storage, and you can easily add new elements at either end (**vector** only allows the addition of elements at its tail). A **list** the third type of basic linear sequence, but it's expensive to move around randomly and cheap to insert an element in the middle. Thus **list**, **deque** and **vector** are very similar in their basic functionality (they all hold linear sequences), but different in the cost of their activities. So for your first shot at a program, you could choose any one, and only experiment with the others if you're tuning for efficiency.

Many of the problems you set out to solve will only require a simple linear sequence like a **vector**, **deque** or **list**. All three have a member function **push_back( )** which you use to insert a new element at the back of the sequence (**deque** and **list** also have **push_front( )**).

But now how do you retrieve those elements? With a **vector** or **deque**, it is possible to use the indexing **operator[ ]**, but that doesn't work with **list**. Since it would be nicest to learn a single interface, we'll often use the one defined for all STL containers: the *iterator*.

An iterator is a class that abstracts the process of moving through a sequence. It allows you to select each element of a sequence *without knowing the underlying structure of that sequence*. This is a powerful feature, partly because it allows us to learn a single interface that works with all containers, and partly because it allows containers to be used interchangeably.

One more observation and you're ready for another example. Even though the STL containers hold objects by value (that is, they hold the whole object inside themselves) that's probably not the way you'll generally use them if you're doing object-oriented programming. That's because in OOP, most of the time you'll create objects on the heap with **new** and then *upcast* the address to the base-class type, later manipulating it as a pointer to the base class. The beauty of this is that you don't worry about the specific type of object you're dealing with, which greatly reduces the complexity of your code and increases the maintainability of your program. This process of upcasting is what you try to do in OOP with polymorphism, so you'll usually be using containers of pointers.

Consider the classic "shape" example where shapes have a set of common operations, and you have different types of shapes. Here's what it looks like using the STL **vector** to hold pointers to various types of **Shape** created on the heap:

```
//: C20:Stlshape.cpp
// Simple shapes w/ STL
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw\n"; }
```

```
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw\n"; }
  ~Square() { cout << "~Square\n"; }
};

typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;

int main() {
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin();
      i != shapes.end(); i++)
    (*i)->draw();
  // ... Sometime later:
  for(Iter j = shapes.begin();
      j != shapes.end(); j++)
    delete *j;
} ///:~
```

The creation of **Shape**, **Circle**, **Square** and **Triangle** should be fairly familiar. **Shape** is a pure abstract base class (because of the *pure specifier* **=0**) that defines the interface for all types of **shapes**. The derived classes redefine the **virtual** function **draw( )** to perform the appropriate operation. Now we'd like to create a bunch of different types of **Shape** object, but where to put them? In an STL container, of course. For convenience, this **typedef**:

```
typedef std::vector<Shape*> Container;
```

creates an alias for a **vector** of **Shape***, and this **typedef**:

```
typedef Container::iterator Iter;
```

uses that alias to create another one, for **vector<Shape*>::iterator**. Notice that the **container** type name must be used to produce the appropriate iterator, which is defined as a nested class. Although there are different types of iterators (forward, bidirectional, reverse, etc., which will be explained later) they all have the same basic interface: you can increment them with **++**, you can dereference them to produce the object they're currently selecting, and you can test them to see if they're at the end of the sequence. That's what you'll want to do 90% of the time. And that's what is done in the above example: after creating a container, it's filled with different types of **Shape***. Notice that the upcast happens as the **Circle**, **Square** or **Rectangle** pointer is added to the **shapes** container, which doesn't know about those specific types but instead holds only **Shape***. So as soon as the pointer is added to the container it loses its specific identity and becomes an anonymous **Shape***. This is exactly what we want: toss them all in and let polymorphism sort it out.

The first **for** loop creates an iterator and sets it to the beginning of the sequence by calling the **begin( )** member function for the container. All containers have **begin( )** and **end( )** member functions that produce an iterator selecting, respectively, the beginning of the sequence and one past the end of the sequence. To test to see if you're done, you make sure you're **!=** to the iterator produced by **end( )**. Not **<** or **<=**. The only test that works is **!=**. So it's very common to write a loop like:

```
for(Iter i = shapes.begin(); i != shapes.end(); i++)
```

This says: "take me through every element in the sequence."

What do you do with the iterator to produce the element it's selecting? You dereference it using (what else) the '*****' (which is actually an overloaded operator). What you get back is whatever the container is holding. This container holds **Shape***, so that's what ***i** produces. If you want to send a message to the **Shape**, you must select that message with **->**, so you write the line:

```
(*i)->draw();
```

This calls the **draw( )** function for the **Shape*** the iterator is currently selecting. The parentheses are ugly but necessary to produce the proper order of evaluation. As an alternative, **operator->** is defined so that you can say:

```
i->draw();
```

As they are destroyed or in other cases where the pointers are removed, the STL containers *do not* call **delete** for the pointers they contain. If you

create an object on the heap with **new** and place its pointer in a container, the container can't tell if that pointer is also placed inside another container. So the STL just doesn't do anything about it, and puts the responsibility squarely in your lap. The last lines in the program move through and delete every object in the container so proper cleanup occurs.

It's very interesting to note that you can change the type of container that this program uses with two lines. Instead of including **<vector>**, you include **<list>**, and in the first **typedef** you say:

```
typedef std::list<Shape*> Container;
```

instead of using a **vector**. Everything else goes untouched. This is possible not because of an interface enforced by inheritance (there isn't any inheritance in the STL, which comes as a surprise when you first see it), but because the interface is enforced by a convention adopted by the designers of the STL, precisely so you could perform this kind of interchange. Now you can easily switch between **vector** and **list** and see which one works fastest for your needs.

Containers of strings

In the prior example, at the end of **main( )**, it was necessary to move through the whole list and **delete** all the **Shape** pointers.

```
for(Iter j = shapes.begin();
    j != shapes.end(); j++)
  delete *j;
```

This highlights what could be seen as a flaw in the STL: there's no facility in any of the STL containers to automatically **delete** the pointers they contain, so you must do it by hand. It's as if the assumption of the STL designers was that containers of pointers weren't an interesting problem, although I assert that it is one of the more common things you'll want to do.

Automatically deleting a pointer turns out to be a rather aggressive thing to do because of the *multiple membership* problem. If a container holds a pointer to an object, it's not unlikely that pointer could also be in another container. A pointer to an **Aluminum** object in a list of **Trash** pointers could also reside in a list of **Aluminum** pointers. If that happens, which list is responsible for cleaning up that object – that is, which list "owns" the object?

This question is virtually eliminated if the object rather than a pointer resides in the list. Then it seems clear that when the list is destroyed, the objects it contains must also be destroyed. Here, the STL shines, as you can see when creating a container of **string** objects. The following example stores each incoming line as a **string** in a **vector<string>**:

```
//: C20:StringVector.cpp
// A vector of strings
#include "../require.h"
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  vector<string> strings;
  string line;
  while(getline(in, line))
    strings.push_back(line);
  // Do something to the strings...
  int i = 1;
  vector<string>::iterator w;
  for(w = strings.begin();
      w != strings.end(); w++) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
  }
  // Now send them out:
  copy(strings.begin(), strings.end(),
    ostream_iterator<string>(cout, "\n"));
  // Since they aren't pointers, string
  // objects clean themselves up!
} ///:~
```

Once the **vector<string>** called **strings** is created, each line in the file is read into a **string** and put in the **vector**:

```
while(getline(in, line))
  strings.push_back(line);
```

The operation that's being performed on this file is to add line numbers. A **stringstream** provides easy conversion from an **int** to a **string** of characters representing that **int**.

Assembling **string** objects is quite easy, since **operator+** is overloaded. Sensibly enough, the iterator **w** can be dereferenced to produce a string that can be used as both an rvalue *and* an lvalue:

```
*w = ss.str() + ": " + *w;
```

The fact that you can assign back into the container via the iterator may seem a bit surprising at first, but it's a tribute to the careful design of the STL.

Because the **vector<string>** contains the objects themselves, a number of interesting things take place. First, no cleanup is necessary. Even if you were to put addresses of the **string** objects as pointers into *other* containers, it's clear that **strings** is the "master list" and maintains ownership of the objects.

Second, you are effectively using dynamic object creation, and yet you never use **new** or **delete**! That's because, somehow, it's all taken care of for you by the **vector** (this is non-trivial. You can try to figure it out by looking at the header files for the STL – all the code is there – but it's quite an exercise). Thus your coding is significantly cleaned up.

The limitation of holding objects instead of pointers inside containers is quite severe: you can't upcast from derived types, thus you can't use polymorphism. The problem with upcasting objects by value is that they get sliced and converted until their type is completely changed into the base type, and there's no remnant of the derived type left. It's pretty safe to say that you *never* want to do this.

Inheriting from STL containers

The power of instantly creating a sequence of elements is amazing, and it makes you realize how much time you've spent (or rather, wasted) in the past solving this particular problem. For example, many utility programs involve reading a file into memory, modifying the file and writing it back

out to disk. One might as well take the functionality in **StringVector.cpp** and package it into a class for later reuse.

Now the question is: do you create a member object of type **vector**, or do you inherit? A general guideline is to always prefer composition (member objects) over inheritance, but with the STL this is often not true, because there are so many existing algorithms that work with the STL types that you may want your new type to *be* an STL type. So the list of **string**s should also *be* a **vector**, thus inheritance is desired.

```
//: C20:FileEditor.h
// File editor tool
#ifndef FILEEDITOR_H
#define FILEEDITOR_H
#include <string>
#include <vector>
#include <iostream>

class FileEditor :
public std::vector<std::string> {
public:
  FileEditor(char* filename);
  void write(std::ostream& out = std::cout);
};
#endif // FILEEDITOR_H ///:~
```

Note the careful avoidance of a global **using namespace std** statement here, to prevent the opening of the **std** namespace to every file that includes this header.

The constructor opens the file and reads it into the **FileEditor**, and **write( )** puts the **vector** of **string** onto any **ostream**. Notice in **write( )** that you can have a default argument for a reference.

The implementation is quite simple:

```
//: C20:FileEditor.cpp {O}
#include "FileEditor.h"
#include "../require.h"
#include <fstream>
using namespace std;

FileEditor::FileEditor(char* filename) {
  ifstream in(filename);
  assure(in, filename);
```

```
  string line;
  while(getline(in, line))
    push_back(line);
}

// Could also use copy() here:
void FileEditor::write(ostream& out) {
  for(iterator w = begin();  w != end(); w++)
    out << *w << endl;
} ///:~
```

The functions from **StringVector.cpp** are simply repackaged. Often this is the way classes evolve – you start by creating a program to solve a particular application, then discover some commonly-used functionality within the program that can be turned into a class.

The line numbering program can now be rewritten using **FileEditor**:

```
//: C20:FEditTest.cpp
//{L} FileEditor
// Test the FileEditor tool
#include "FileEditor.h"
#include "../require.h"
#include <sstream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  FileEditor file(argv[1]);
  // Do something to the lines...
  int i = 1;
  FileEditor::iterator w = file.begin();
  while(w != file.end()) {
    ostringstream ss;
    ss << i++;
    *w = ss.str() + ": " + *w;
    w++;
  }
  // Now send them to cout:
  file.write();
} ///:~
```

Now the operation of reading the file is in the constructor:

```
FileEditor file(argv[1]);
```

and writing happens in the single line (which defaults to sending the output to **cout**):

```
file.write();
```

The bulk of the program is involved with actually modifying the file in memory.

A plethora of iterators

As mentioned earlier, the iterator is the abstraction that allows a piece of code to be *generic*, and to work with different types of containers without knowing the underlying structure of those containers. Every container produces iterators. You must always be able to say:

```
ContainerType::iterator
ContainerType::const_iterator
```

to produce the types of the iterators produced by that container. Every container has a **begin( )** method that produces an iterator indicating the beginning of the elements in the container, and an **end( )** method that produces an iterator which is the as the *past-the-end value* of the container. If the container is **const**, **begin( )** and **end( )** produce **const** iterators.

Every iterator can be moved forward to the next element using the **operator++** (an iterator may be able to do more than this, as you shall see, but it must at least support forward movement with **operator++**).

The basic iterator is only guaranteed to be able to perform **==** and **!=** comparisons. Thus, to move an iterator **it** forward without running it off the end you say something like:

```
while(it != pastEnd) {
  // Do something
  it++;
}
```

Where **pastEnd** is the past-the-end value produced by the container's **end( )** member function.

An iterator can be used to produce the element that it is currently selecting within a container by dereferencing the iterator. This can take two forms. If **it** is an iterator and **f( )** is a member function of the objects held in the container that the iterator is pointing within, then you can say either:

```
(*it).f();
```

or

```
it->f();
```

Knowing this, you can create a template that works with any container. Here, the **apply( )** function template calls a member function for every object in the container, using a pointer to member that is passed as an argument:

```
//: C20:Apply.cpp
// Using basic iterators
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

template<class Cont, class PtrMemFun>
void apply(Cont& c, PtrMemFun f) {
  typename Cont::iterator it = c.begin();
  while(it != c.end()) {
    (it->*f)(); // Compact form
    ((*it).*f)(); // Alternate form
    it++;
  }
}

class Z {
  int i;
public:
  Z(int ii) : i(ii) {}
  void g() { i++; }
  friend ostream&
  operator<<(ostream& os, const Z& z) {
    return os << z.i;
  }
};

int main() {
  ostream_iterator<Z> out(cout, " ");
  vector<Z> vz;
  for(int i = 0; i < 10; i++)
    vz.push_back(Z(i));
```

```
  copy(vz.begin(), vz.end(), out);
  cout << endl;
  apply(vz, &Z::g);
  copy(vz.begin(), vz.end(), out);
} ///:~
```

Because **operator->** is defined for STL iterators, it can be used for pointer-to-member dereferencing (in the following chapter you'll learn a more elegant way to handle the problem of applying a member function or ordinary function to every object in a container).

Much of the time, this is all you need to know about iterators – that they are produced by **begin( )** and **end( )**, and that you can use them to move through a container and select elements. Many of the problems that you solve, and the STL algorithms (covered in the next chapter) will allow you to just flail away with the basics of iterators. However, things can at times become more subtle, and in those cases you need to know more about iterators. The rest of this section gives you the details.

Iterators in reversible containers

All containers must produce the basic **iterator**. A container may also be *reversible*, which means that it can produce iterators that move backwards from the end, as well as the iterators that move forward from the beginning.

A reversible container has the methods **rbegin( )** (to produce a **reverse_iterator** selecting the end) and **rend( )** (to produce a **reverse_iterator** indicating "one past the beginning"). If the container is **const** then **rbegin( )** and **rend( )** will produce **const_reverse_iterator**s.

All the basic sequence containers **vector**, **deque** and **list** are reversible containers. The following example uses **vector**, but will work with **deque** and **list** as well:

```
//: C20:Reversible.cpp
// Using reversible containers
#include "../require.h"
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
```

```
using namespace std;

int main() {
  ifstream in("Reversible.cpp");
  assure(in, "Reversible.cpp");
  string line;
  vector<string> lines;
  while(getline(in, line))
    lines.push_back(line);
  vector<string>::reverse_iterator r;
  for(r = lines.rbegin(); r != lines.rend(); r++)
    cout << *r << endl;
} ///:~
```

You move backward through the container using the same syntax as moving forward through a container with an ordinary iterator.

The associative containers **set**, **multiset**, **map** and **multimap** are also reversible. Using iterators with associative containers is a bit different, however, and will be delayed until those containers are more fully introduced.

Iterator categories

The iterators are classified into different “categories” which describe what they are capable of doing. The order in which they are generally described moves from the categories with the most restricted behavior to those with the most powerful behavior.

Input: read-only, one pass

The only predefined implementations of input iterators are **istream_iterator** and **istreambuf_iterator**, to read from an **istream**. As you can imagine, an input iterator can only be dereferenced once for each element that’s selected, just as you can only read a particular portion of an input stream once. They can only move forward. There is a special constructor to define the past-the-end value. In summary, you can dereference it for reading (once only for each value), and move it forward.

Output: write-only, one pass

This is the complement of an input iterator, but for writing rather than reading. The only predefined implementations of output iterators are **ostream_iterator** and **ostreambuf_iterator**, to write to an **ostream**,

and the less-commonly-used **raw_storage_iterator**. Again, these can only be dereferenced once for each written value, and they can only move forward. There is no concept of a terminal past-the-end value for an output iterator. Summarizing, you can dereference it for writing (once only for each value) and move it forward.

Forward: multiple read/write

The forward iterator contains all the functionality of both the input iterator and the output iterator, plus you can dereference an iterator location multiple times, so you can read and write to a value multiple times. As the name implies, you can only move forward. There are no predefined iterators that are only forward iterators.

Bidirectional: operator--

The bidirectional iterator has all the functionality of the forward iterator, and in addition it can be moved backwards one location at a time using **operator--**.

Random-access: like a pointer

Finally, the random-access iterator has all the functionality of the bidirectional iterator plus all the functionality of a pointer (a pointer *is* a random-access iterator). Basically, anything you can do with a pointer you can do with a bidirectional iterator, including indexing with **operator[ ]**, adding integral values to a pointer to move it forward or backward by a number of locations, and comparing one iterator to another with **<**, **>=**, etc.

Is this really important?

Why do you care about this categorization? When you're just using containers in a straightforward way (for example, just hand-coding all the operations you want to perform on the objects in the container) it usually doesn't impact you too much. Things either work or they don't. The iterator categories become important when:

1. You use some of the fancier built-in iterator types that will be demonstrated shortly. Or you graduate to creating your own iterators (this will also be demonstrated, later in this chapter).
2. You use the STL algorithms (the subject of the next chapter). Each of the algorithms have requirements that they place on the iterators that they work with. Knowledge of the iterator categories is even more

important when you create your own reusable algorithm templates, because the iterator category that your algorithm requires determines how flexible the algorithm will be. If you only require the most primitive iterator category (input or output) then your algorithm will work with *everything* (**copy( )** is an example of this).

Predefined iterators

The STL has a predefined set of iterator classes that can be quite handy. For example, you've already seen **reverse_iterator** (produced by calling **rbegin( )** and **rend( )** for all the basic containers).

The *insertion iterators* are necessary because some of the STL algorithms – **copy( )** for example – use the assignment **operator=** in order to place objects in the destination container. This is a problem when you're using the algorithm to *fill* the container rather than to overwrite items that are already in the destination container. That is, when the space isn't already there. What the insert iterators do is change the implementation of the **operator=** so that instead of doing an assignment, it calls a "push" or "insert" function for that container, thus causing it to allocate new space. The constructors for both **back_insert_iterator** and **front_insert_iterator** take a basic sequence container object (**vector**, **deque** or **list**) as their argument and produce an iterator that calls **push_back( )** or **push_front( )**, respectively, to perform assignment. The shorthand functions **back_inserter( )** and **front_inserter( )** produce the same objects with a little less typing. Since all the basic sequence containers support **push_back( )**, you will probably find yourself using **back_inserter( )** with some regularity.

The **insert_iterator** allows you to insert elements in the middle of the sequence, again replacing the meaning of **operator=**, but this time with **insert( )** instead of one of the "push" functions. The **insert( )** member function requires an iterator indicating the place to insert before, so the **insert_iterator** requires this iterator addition to the container object. The shorthand function **inserter( )** produces the same object.

The following example shows the use of the different types of inserters:

```
//: C20:Inserters.cpp
// Different types of iterator inserters
#include <iostream>
#include <vector>
#include <deque>
#include <list>
```

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

int a[] = { 1, 3, 5, 7, 11, 13, 17, 19, 23 };

template<class Cont>
void frontInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(int),
    front_inserter(ci));
  copy(ci.begin(), ci.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
}

template<class Cont>
void backInsertion(Cont& ci) {
  copy(a, a + sizeof(a)/sizeof(int),
    back_inserter(ci));
  copy(ci.begin(), ci.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
}

template<class Cont>
void midInsertion(Cont& ci) {
  typename Cont::iterator it = ci.begin();
  it++; it++; it++;
  copy(a, a + sizeof(a)/(sizeof(int) * 2),
    inserter(ci, it));
  copy(ci.begin(), ci.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
}

int main() {
  deque<int> di;
  list<int>  li;
  vector<int> vi;
  // Can't use a front_inserter() with vector
  frontInsertion(di);
  frontInsertion(li);
  di.clear();
```

```
  li.clear();
  backInsertion(vi);
  backInsertion(di);
  backInsertion(li);
  midInsertion(vi);
  midInsertion(di);
  midInsertion(li);
} ///:~
```

Since **vector** does not support **push_front( )**, it cannot produce a **front_insertion_iterator**. However, you can see that **vector** does support the other two types of insertion (even though, as you shall see later, **insert( )** is not a very efficient operation for **vector**).

IO stream iterators

You've already seen some use of the **ostream_iterator** (an output iterator) in conjuction with **copy( )** to place the contents of a container on an output stream. There is a corresponding **istream_iterator** (an input iterator) which allows you to "iterate" a set of objects of a specified type from an input stream. An important difference between **ostream_iterator** and **istream_iterator** comes from the fact that an output stream doesn't have any concept of an "end," since you can always just keep writing more elements. However, an input stream eventually terminates (for example, when you reach the end of a file) so there needs to be a way to represent that. An **istream_iterator** has two constructors, one that takes an **istream** and produces the iterator you actually read from, and the other which is the default constructor and produces an object which is the past-the-end sentinel. In the following program this object is named **end**:

```
//: C20:StreamIt.cpp
// Iterators for istreams and ostreams
#include "../require.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace std;

int main() {
  ifstream in("StreamIt.cpp");
  assure(in, "StreamIt.cpp");
  istream_iterator<string> init(in), end;
```

```
  ostream_iterator<string> out(cout, "\n");
  vector<string> vs;
  copy(init, end, back_inserter(vs));
  copy(vs.begin(), vs.end(), out);
  *out++ = vs[0];
  *out++ = "That's all, folks!";
} ///:~
```

When **in** runs out of input (in this case when the end of the file is reached) then **init** becomes equivalent to **end** and the **copy( )** terminates.

Because **out** is an **ostream_iterator<string>**, you can simply assign any **string** object to the dereferenced iterator using **operator=** and that **string** will be placed on the output stream, as seen in the two assignments to **out**. Because **out** is defined with a newline as its second argument, these assignments also cause a newline to be inserted along with each assignment.

While it is possible to create an **istream_iterator<char>** and **ostream_iterator<char>**, these actually *parse* the input and thus will for example automatically eat whitespace (spaces, tabs and newlines), which is not desirable if you want to manipulate an exact representation of an **istream**. Instead, you can use the special iterators **istreambuf_iterator** and **ostreambuf_iterator**, which are designed strictly to move characters[61]. Although these are templates, the only template arguments they will accept are either **char** or **wchar_t** (for wide characters). The following example allows you to compare the behavior of the stream iterators vs. the streambuf iterators:

```
//: C20:StreambufIterator.cpp
// istreambuf_iterator & ostreambuf_iterator
#include "../require.h"
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
using namespace std;
```

[61] These were actually created to abstract the “locale” facets away from iostreams, so that locale facets could operate on any sequence of characters, not only iostreams. Locales allow iostreams to easily handle culturally-different formatting (such as representation of money), and are beyond the scope of this book.

```
int main() {
  ifstream in("StreambufIterator.cpp");
  assure(in, "StreambufIterator.cpp");
  // Exact representation of stream:
  istreambuf_iterator<char> isb(in), end;
  ostreambuf_iterator<char> osb(cout);
  while(isb != end)
    *osb++ = *isb++; // Copy 'in' to cout
  cout << endl;
  ifstream in2("StreambufIterator.cpp");
  // Strips white space:
  istream_iterator<char> is(in2), end2;
  ostream_iterator<char> os(cout);
  while(is != end2)
    *os++ = *is++;
  cout << endl;
} ///:~
```

The stream iterators use the parsing defined by **istream::operator>>**, which is probably not
what you want if you are parsing characters directly – it's fairly rare that you would want all the whitespace stripped out of your character stream. You'll virtually always want to use a streambuf iterator when using characters and streams, rather than a stream iterator. In addition, **istream::operator>>** adds significant overhead for each operation, so it is only appropriate for higher-level operations such as parsing floating-point numbers.[62]

Manipulating raw storage

This is a little more esoteric and is generally used in the implementation of other Standard Library functions, but it is nonetheless interesting. The **raw_storage_iterator** is defined in **<algorithm>** and is an output iterator. It is provided to enable algorithms to store their results into uninitialized memory. The interface is quite simple: the constructor takes an output iterator that is pointing to the raw memory (thus it is typically a pointer) and the **operator=** assigns an object into that raw memory. The template parameters are the type of the output iterator pointing to the raw storage, and the type of object that will be stored. Here's an example

[62] I am indebted to Nathan Myers for explaining this to me.

which creates **Noisy** objects (you'll be introduced to the **Noisy** class shortly; it's not necessary to know its details for this example):

```
//: C20:RawStorageIterator.cpp
// Demonstrate the raw_storage_iterator
#include "Noisy.h"
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
  const int quantity = 10;
  // Create raw storage and cast to desired type:
  Noisy* np =
    (Noisy*)new char[quantity * sizeof(Noisy)];
  raw_storage_iterator<Noisy*, Noisy> rsi(np);
  for(int i = 0; i < quantity; i++)
    *rsi++ = Noisy(); // Place objects in storage
  cout << endl;
  copy(np, np + quantity,
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  // Explicit destructor call for cleanup:
  for(int j = 0; j < quantity; j++)
    (&np[j])->~Noisy();
  // Release raw storage:
  delete (char*)np;
} ///:~
```

To make the **raw_storage_iterator** template happy, the raw storage must be of the same type as the objects you're creating. That's why the pointer from the new array of **char** is cast to a **Noisy***. The assignment operator forces the objects into the raw storage using the copy-constructor. Note that the explicit destructor call must be made for proper cleanup, and this also allows the objects to be deleted one at a time during container manipulation.

Basic sequences: vector, list & deque

If you take a step back from the STL containers you'll see that there are really only two types of container: *sequences* (including **vector**, **list**, **deque**, **stack**, **queue**, and **priority_queue**) and *associations* (including **set**, **multiset**, **map** and **multimap**). The sequences keep the objects in whatever sequence that you establish (either by pushing the objects on the end or inserting them in the middle).

Since all the sequence containers have the same basic goal (to maintain your order) they seem relatively interchangeable. However, they differ in the efficiency of their operations, so if you are going to manipulate a sequence in a particular fashion you can choose the appropriate container for those types of manipulations. The "basic" sequence containers are **vector**, **list** and **deque** – these actually have fleshed-out implementations, while **stack**, **queue** and **priority_queue** are built on top of the basic sequences, and represent more specialized uses rather than differences in underlying structure (**stack**, for example, can be implemented using a **deque**, **vector** or **list**).

So far in this book I have been using **vector** as a catch-all container. This was acceptable because I've only used the simplest and safest operations, primarily **push_back( )** and **operator[ ]**. However, when you start making more sophisticated uses of containers it becomes important to know more about their underlying implementations and behavior, so you can make the right choices (and, as you'll see, stay out of trouble).

Basic sequence operations

Using a template, the following example shows the operations that all the basic sequences (**vector**, **deque** or **list**) support. As you shall learn in the sections on the specific sequence containers, not all of these operations make sense for each basic sequence, but they are supported.

```
//: C20:BasicSequenceOperations.cpp
// The operations available for all the
// basic sequence Containers.
#include <iostream>
#include <vector>
#include <deque>
```

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

template<typename Container>
void print(Container& c, char* s = "") {
  cout << s << ":" << endl;
  if(c.empty()) {
    cout << "(empty)" << endl;
    return;
  }
  typename Container::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "size() " << c.size()
    << " max_size() "<< c.max_size()
    << " front() " << c.front()
    << " back() " << c.back() << endl;
}

template<typename ContainerOfInt>
void basicOps(char* s) {
  cout << "------- " << s << " -------" << endl;
  typedef ContainerOfInt Ci;
  Ci c;
  print(c, "c after default constructor");
  Ci c2(10, 1); // 10 elements, values all 1
  print(c2, "c2 after constructor(10,1)");
  int ia[] = { 1, 3, 5, 7, 9 };
  const int iasz = sizeof(ia)/sizeof(*ia);
  // Initialize with begin & end iterators:
  Ci c3(ia, ia + iasz);
  print(c3, "c3 after constructor(iter,iter)");
  Ci c4(c2); // Copy-constructor
  print(c4, "c4 after copy-constructor(c2)");
  c = c2; // Assignment operator
  print(c, "c after operator=c2");
  c.assign(10, 2); // 10 elements, values all 2
  print(c, "c after assign(10, 2)");
  // Assign with begin & end iterators:
  c.assign(ia, ia + iasz);
  print(c, "c after assign(iter, iter)");
```

```
  cout << "c using reverse iterators:" << endl;
  typename Ci::reverse_iterator rit = c.rbegin();
  while(rit != c.rend())
    cout << *rit++ << " ";
  cout << endl;
  c.resize(4);
  print(c, "c after resize(4)");
  c.push_back(47);
  print(c, "c after push_back(47)");
  c.pop_back();
  print(c, "c after pop_back()");
  typename Ci::iterator it = c.begin();
  it++; it++;
  c.insert(it, 74);
  print(c, "c after insert(it, 74)");
  it = c.begin();
  it++;
  c.insert(it, 3, 96);
  print(c, "c after insert(it, 3, 96)");
  it = c.begin();
  it++;
  c.insert(it, c3.begin(), c3.end());
  print(c, "c after insert("
    "it, c3.begin(), c3.end())");
  it = c.begin();
  it++;
  c.erase(it);
  print(c, "c after erase(it)");
  typename Ci::iterator it2 = it = c.begin();
  it++;
  it2++; it2++; it2++; it2++; it2++;
  c.erase(it, it2);
  print(c, "c after erase(it, it2)");
  c.swap(c2);
  print(c, "c after swap(c2)");
  c.clear();
  print(c, "c after clear()");
}

int main() {
  basicOps<vector<int> >("vector");
  basicOps<deque<int> >("deque");
```

```
  basicOps<list<int> >("list");
} ///:~
```

The first function template, **print( )**, demonstrates the basic information you can get from any sequence container: whether it's empty, its current size, the size of the largest possible container, the element at the beginning and the element at the end. You can also see that every container has **begin( )** and **end( )** methods that return iterators.

The **basicOps( )** function tests everything else (and in turn calls **print( )**), including a variety of constructors: default, copy-constructor, quantity and initial value, and beginning and ending iterators. There's an assignment **operator=** and two kinds of **assign( )** member functions, one which takes a quantity and initial value and the other which take a beginning and ending iterator.

All the basic sequence containers are reversible containers, as shown by the use of the **rbegin( )** and **rend( )** member functions. A sequence container can be resized, and the entire contents of the container can be removed with **clear( )**.

Using an iterator to indicate where you want to start inserting into any sequence container, you can **insert( )** a single element, a number of elements that all have the same value, and a group of elements from another container using the beginning and ending iterators of that group.

To **erase( )** a single element from the middle, use an iterator; to **erase( )** a range of elements, use a pair of iterators. Notice that since a **list** only supports bidirectional iterators, all the iterator motion must be performed with increments and decrements (if the containers were limited to **vector** and **deque**, which produce random-access iterators, then **operator+** and **operator-** could have been used to move the iterators in big jumps).

Although both **list** and **deque** support **push_front( )** and **pop_front( )**, **vector** does not, so the only member functions that work with all three are **push_back( )** and **pop_back( )**.

The naming of the member function **swap( )** is a little confusing, since there's also a non-member **swap( )** algorithm that switches two elements of a container. The member **swap( )**, however, swaps *everything* in one container for another (if the containers hold the same type), effectively swapping the containers themselves. There's also a non-member version of this function.

The following sections on the sequence containers discuss the particulars of each type of container.

vector

The **vector** is intentionally made to look like a souped-up array, since it has array-style indexing but also can expand dynamically. **vector** is so fundamentally useful that it was introduced in a very primitive way early in this book, and used quite regularly in previous examples. This section will give a more in-depth look at **vector**.

To achieve maximally-fast indexing and iteration, the **vector** maintains its storage as a single contiguous array of objects. This is a critical point to observe in understanding the behavior of **vector**. It means that indexing and iteration are lighting-fast, being basically the same as indexing and iterating over an array of objects. But it also means that inserting an object anywhere but at the end (that is, appending) is not really an acceptable operation for a **vector**. It also means that when a **vector** runs out of pre-allocated storage, in order to maintain its contiguous array it must allocate a whole new (larger) chunk of storage elsewhere and copy the objects to the new storage. This has a number of unpleasant side effects.

Cost of overflowing allocated storage

A **vector** starts by grabbing a block of storage, as if it's taking a guess at how many objects you plan to put in it. As long as you don't try to put in more objects than can be held in the initial block of storage, everything is very rapid and efficient (note that if you *do* know how many objects to expect, you can pre-allocate storage using **reserve( )**). But eventually you will put in one too many objects and, unbeknownst to you, the **vector** responds by:

1. Allocating a new, bigger piece of storage
2. Copying all the objects from the old storage to the new (using the copy-constructor)
3. Destroying all the old objects (the destructor is called for each one)
4. Releasing the old memory

For complex objects, this copy-construction and destruction can end up being very expensive if you overfill your vector a lot. To see what happens when you're filling a **vector**, here is a class that prints out information about its creations, destructions, assignments and copy-constructions:

```
//: C20:Noisy.h
// A class to track various object activities
#ifndef NOISY_H
#define NOISY_H
#include <iostream>

class Noisy {
  static long create, assign, copycons, destroy;
  long id;
public:
  Noisy() : id(create++) {
    std::cout << "d[" << id << "]";
  }
  Noisy(const Noisy& rv) : id(rv.id) {
    std::cout << "c[" << id << "]";
    copycons++;
  }
  Noisy& operator=(const Noisy& rv) {
    std::cout << "(" << id << ")=[" <<
      rv.id << "]";
    id = rv.id;
    assign++;
    return *this;
  }
  friend bool
  operator<(const Noisy& lv, const Noisy& rv) {
    return lv.id < rv.id;
  }
  friend bool
  operator==(const Noisy& lv, const Noisy& rv) {
    return lv.id == rv.id;
  }
  ~Noisy() {
    std::cout << "~[" << id << "]";
    destroy++;
  }
  friend std::ostream&
  operator<<(std::ostream& os, const Noisy& n) {
```

```
    return os << n.id;
  }
  friend class NoisyReport;
};

struct NoisyGen {
  Noisy operator()() { return Noisy(); }
};

// A singleton. Will automatically report the
// statistics as the program terminates:
class NoisyReport {
  static NoisyReport nr;
  NoisyReport() {} // Private constructor
public:
  ~NoisyReport() {
    std::cout << "\n-------------------\n"
      << "Noisy creations: " << Noisy::create
      << "\nCopy-Constructions: "
      << Noisy::copycons
      << "\nAssignments: " << Noisy::assign
      << "\nDestructions: " << Noisy::destroy
      << std::endl;
  }
};

// Because of these this file can only be used
// in simple test situations. Move them to a
// .cpp file for more complex programs:
long Noisy::create = 0, Noisy::assign = 0,
  Noisy::copycons = 0, Noisy::destroy = 0;
NoisyReport NoisyReport::nr;
#endif // NOISY_H ///:~
```

Each **Noisy** object has its own identifier, and there are **static** variables to keep track of all the creations, assignments (using **operator=**), copy-constructions and destructions. The **id** is initialized using the **create** counter inside the default constructor; the copy-constructor and assignment operator take their **id** values from the rvalue. Of course, with **operator=** the lvalue is already an initialized object so the old value of **id** is printed before it is overwritten with the **id** from the rvalue.

In order to support certain operations like sorting and searching (which are used implicitly by some of the containers), **Noisy** must have an **operator<** and **operator==**. These simply compare the **id** values. The **operator<<** for **ostream** follows the standard form and simply prints the **id**.

NoisyGen produces a function object (since it has an **operator( )**) that is used to automatically generate **Noisy** objects during testing.

NoisyReport is a type of class called a *singleton*, which is a "design pattern" (these are covered more fully in Chapter XX). Here, the goal is to make sure there is one and only one **NoisyReport** object, because it is responsible for printing out the results at program termination. It has a **private** constructor so no one else can make a **NoisyReport** object, and a single static instance of **NoisyReport** called **nr**. The only executable statements are in the destructor, which is called as the program exits and the static destructors are called; this destructor prints out the statistics captured by the **static** variables in **Noisy**.

The one snag to this header file is the inclusion of the definitions for the **static**s at the end. If you include this header in more than one place in your project, you'll get multiple-definition errors at link time. Of course, you can put the **static** definitions in a separate **cpp** file and link it in, but that is less convenient, and since **Noisy** is just intended for quick-and-dirty experiments the header file should be reasonable for most situations.

Using **Noisy.h**, the following program will show the behaviors that occur when a **vector** overflows its currently allocated storage:

```
//: C20:VectorOverflow.cpp
// Shows the copy-construction and destruction
// That occurs when a vector must reallocate
// (It maintains a linear array of elements)
#include "Noisy.h"
#include "../require.h"
#include <vector>
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
```

```
  vector<Noisy> vn;
  Noisy n;
  for(int i = 0; i < size; i++)
    vn.push_back(n);
  cout << "\n cleaning up \n";
} ///:~
```

You can either use the default value of 1000, or use your own value by putting it on the command-line.

When you run this program, you'll see a single default constructor call (for **n**), then a lot of copy-constructor calls, then some destructor calls, then some more copy-constructor calls, and so on. When the vector runs out of space in the linear array of bytes it has allocated, it must (to maintain all the objects in a linear array, which is an essential part of its job) get a bigger piece of storage and move everything over, copying first and then destroying the old objects. You can imagine that if you store a lot of large and complex objects, this process could rapidly become prohibitive.

There are two solutions to this problem. The nicest one requires that you know beforehand how many objects you're going to make. In that case you can use **reserve( )** to tell the vector how much storage to pre-allocate, thus eliminating all the copies and destructions and making everything very fast (especially random access to the objects with **operator[ ]**). Note that the use of **reserve( )** is different from the using the **vector** constructor with an integral first argument; the latter initializes each element using the default copy-constructor.

However, in the more general case you won't know how many objects you'll need. If **vector** reallocations are slowing things down, you can change sequence containers. You could use a **list**, but as you'll see, the **deque** allows speedy insertions at either end of the sequence, and never needs to copy or destroy objects as it expands its storage. The **deque** also allows random access with **operator[ ]**, but it's not quite as fast as **vector**'s **operator[ ]**. So in the case where you're creating all your objects in one part of the program and randomly accessing them in another, you may find yourself filling a **deque**, then creating a **vector** from the **deque** and using the **vector** for rapid indexing. Of course, you don't want to program this way habitually, just be aware of these issues (avoid premature optimization).

There is a darker side to **vector**'s reallocation of memory, however. Because **vector** keeps its objects in a nice, neat array (allowing, for one thing, maximally-fast random access), the iterators used by **vector** are generally just pointers. This is a good thing – of all the sequence

containers, these pointers allow the fastest selection and manipulation. However, consider what happens when you're holding onto an iterator (i.e. a pointer) and then you add the one additional object that causes the **vector** to reallocate storage and move it elsewhere. Your pointer is now pointing off into nowhere:

```
//: C20:VectorCoreDump.cpp
// How to break a program using a vector
#include <vector>
#include <iostream>
using namespace std;

int main() {
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, " ");
  copy(vi.begin(), vi.end(), out);
  vector<int>::iterator i = vi.begin();
  cout << "\n i: " << long(i) << endl;
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  // Force it to move memory (could also just add
  // enough objects):
  vi.resize(vi.capacity() + 1);
  // Now i points to wrong memory:
  cout << "\n i: " << long(i) << endl;
  cout << "vi.begin(): " << long(vi.begin());
  *i = 48;  // Access violation
} ///:~
```

If your program is breaking mysteriously, look for places where you hold onto an iterator while adding more objects to a **vector**. You'll need to get a new iterator after adding elements, or use **operator[ ]** instead for element selections. If you combine the above observation with the awareness of the potential expense of adding new objects to a **vector**, you may conclude that the safest way to use one is to fill it up all at once (ideally, knowing first how many objects you'll need) and then just use it (without adding more objects) elsewhere in the program. This is the way **vector** has been used in the book up to this point.

You may observe that using **vector** as the "basic" container the book in earlier chapters may not be the best choice in all cases. This is a fundamental issue in containers, and in data structures in general: the "best" choice varies according to the way the container is used. The reason **vector** has been the "best" choice up until now is that it looks a lot

like an array, and was thus familiar and easy for you to adopt. But from now on it's also worth thinking about other issues when choosing containers.

Inserting and erasing elements

The **vector** is most efficient if:

1. You **reserve( )** the correct amount of storage at the beginning so the **vector** never has to reallocate.
2. You only add and remove elements from the back end.

It is possible to insert and erase elements from the middle of a **vector** using an iterator, but the following program demonstrates what a bad idea it is:

```
//: C20:VectorInsertAndErase.cpp
// Erasing an element from a vector
#include "Noisy.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  vector<Noisy> v;
  v.reserve(11);
  cout << "11 spaces have been reserved" << endl;
  generate_n(back_inserter(v), 10, NoisyGen());
  ostream_iterator<Noisy> out(cout, " ");
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << "Inserting an element:" << endl;
  vector<Noisy>::iterator it =
    v.begin() + v.size() / 2; // Middle
  v.insert(it, Noisy());
  cout << endl;
  copy(v.begin(), v.end(), out);
  cout << "\nErasing an element:" << endl;
  // Cannot use the previous value of it:
  it = v.begin() + v.size() / 2;
  v.erase(it);
  cout << endl;
```

```
  copy(v.begin(), v.end(), out);
  cout << endl;
} ///:~
```

When you run the program you'll see that the call to **reserve( )** really does only allocate storage – no constructors are called. The **generate_n( )** call is pretty busy: each call to **NoisyGen::operator( )** results in a construction, a copy-construction (into the **vector**) and a destruction of the temporary. But when an object is inserted into the **vector** in the middle, it must shove everything down to maintain the linear array and – since there is enough space – it does this with the assignment operator (if the argument of **reserve( )** is 10 instead of eleven then it would have to reallocate storage). When an object is erased from the **vector**, the assignment operator is once again used to move everything up to cover the place that is being erased (notice that this requires that the assignment operator properly cleans up the lvalue). Lastly, the object on the end of the array is deleted.

You can imagine how enormous the overhead can become if objects are inserted and removed from the middle of a **vector** if the number of elements is large and the objects are complicated. It's obviously a practice to avoid.

deque

The **deque** (double-ended-queue, pronounced "deck") is the basic sequence container optimized for adding and removing elements from either end. It also allows for reasonably fast random access – it has an **operator[ ]** like **vector**. However, it does not have **vector**'s constraint of keeping everything in a single sequential block of memory. Instead, **deque** uses multiple blocks of sequential storage (keeping track of all the blocks and their order in a mapping structure). For this reason the overhead for a **deque** to add or remove elements at either end is very low. In addition, it never needs to copy and destroy contained objects during a new storage allocation (like **vector** does) so it is far more efficient than **vector** if you are adding an unknown quantity of objects. This means that **vector** is the best choice only if you have a pretty good idea of how many objects you need. In addition, many of the programs shown earlier in this book that use **vector** and **push_back( )** might be more efficient with a **deque**. The interface to **deque** is only slightly different from a **vector** (deque has a **push_front( )** and **pop_front( )** while **vector** does not, for example) so converting code from using **vector**

to using **deque** is almost trivial. Consider **StringVector.cpp**, which can be changed to use **deque** by replacing the word “vector” with “deque” everywhere. The following program adds parallel **deque** operations to the **vector** operations in **StringVector.cpp**, and performs timing comparisons:

```
//: C20:StringDeque.cpp
// Converted from StringVector.cpp
#include "../require.h"
#include <string>
#include <deque>
#include <vector>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  vector<string> vstrings;
  deque<string> dstrings;
  string line;
  // Time reading into vector:
  clock_t ticks = clock();
  while(getline(in, line))
    vstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into vector: " << ticks << endl;
  // Repeat for deque:
  ifstream in2(argv[1]);
  assure(in2, argv[1]);
  ticks = clock();
  while(getline(in2, line))
    dstrings.push_back(line);
  ticks = clock() - ticks;
  cout << "Read into deque: " << ticks << endl;
  // Now compare indexing:
  ticks = clock();
  for(int i = 0; i < vstrings.size(); i++) {
```

```
    ostringstream ss;
    ss << i;
    vstrings[i] = ss.str() + ": " + vstrings[i];
  }
  ticks = clock() - ticks;
  cout << "Indexing vector: " << ticks << endl;
  ticks = clock();
  for(int j = 0; j < dstrings.size(); j++) {
    ostringstream ss;
    ss << j;
    dstrings[j] = ss.str() + ": " + dstrings[j];
  }
  ticks = clock() - ticks;
  cout << "Indexing deqeue: " << ticks << endl;
  // Compare iteration
  ofstream tmp1("tmp1.tmp"), tmp2("tmp2.tmp");
  ticks = clock();
  copy(vstrings.begin(), vstrings.end(),
    ostream_iterator<string>(tmp1, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating vector: " << ticks << endl;
  ticks = clock();
  copy(dstrings.begin(), dstrings.end(),
    ostream_iterator<string>(tmp2, "\n"));
  ticks = clock() - ticks;
  cout << "Iterating deqeue: " << ticks << endl;
} ///:~
```

Knowing now what you do about the inefficiency of adding things to **vector** because of storage reallocation, you may expect dramatic differences between the two. However, on a 1.7 Megabyte text file one compiler's program produced the following (measured in platform/compiler specific clock ticks, not seconds):

```
Read into vector: 8350
Read into deque: 7690
Indexing vector: 2360
Indexing deqeue: 2480
Iterating vector: 2470
Iterating deqeue: 2410
```

A different compiler and platform roughly agreed with this. It's not so dramatic, is it? This points out some important issues:

1. We (programmers) are typically very bad at guessing where inefficiencies occur in our programs.
2. Efficiency comes from a combination of effects – here, reading the lines in and converting them to strings may dominate over the cost of the **vector** vs. **deque**.
3. The **string** class is probably fairly well-designed in terms of efficiency.

Of course, this doesn't mean you shouldn't use a **deque** rather than a **vector** when you know that an uncertain number of objects will be pushed onto the end of the container. On the contrary, you should – when you're tuning for performance. But you should also be aware that performance issues are usually not where you think they are, and the only way to know for sure where your bottlenecks are is by testing. Later in this chapter there will be a more "pure" comparison of performance between **vector**, **deque** and **list**.

Converting between sequences

Sometimes you need the behavior or efficiency of one kind of container for one part of your program, and a different container's behavior or efficiency in another part of the program. For example, you may need the efficiency of a **deque** when adding objects to the container but the efficiency of a **vector** when indexing them. Each of the basic sequence containers (**vector**, **deque** and **list**) has a two-iterator constructor (indicating the beginning and ending of the sequence to read from when creating a new object) and an **assign( )** member function to read into an existing container, so you can easily move objects from one sequence container to another.

The following example reads objects into a **deque** and then converts to a **vector**:

```
//: C20:DequeConversion.cpp
// Reading into a Deque, converting to a vector
#include "Noisy.h"
#include <deque>
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdlib>
using namespace std;

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

```
  int size = 25;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> d;
  generate_n(back_inserter(d), size, NoisyGen());
  cout << "\n Converting to a vector(1)" << endl;
  vector<Noisy> v1(d.begin(), d.end());
  cout << "\n Converting to a vector(2)" << endl;
  vector<Noisy> v2;
  v2.reserve(d.size());
  v2.assign(d.begin(), d.end());
  cout << "\n Cleanup" << endl;
} ///:~
```

You can try various sizes, but you should see that it makes no difference – the objects are simply copy-constructed into the new **vector**s. What's interesting is that **v1** does not cause multiple allocations while building the **vector**, no matter how many elements you use. You might initially think that you must follow the process used for **v2** and preallocate the storage to prevent messy reallocations, but the constructor used for **v1** determines the memory need ahead of time so this is unnecessary.

Cost of overflowing allocated storage

It's illuminating to see what happens with a **deque** when it overflows a block of storage, in contrast with **VectorOverflow.cpp**:

```
//: C20:DequeOverflow.cpp
// A deque is much more efficient than a vector
// when pushing back a lot of elements, since it
// doesn't require copying and destroying.
#include "Noisy.h"
#include <deque>
#include <cstdlib>
using namespace std;

int main(int argc, char* argv[]) {
  int size = 1000;
  if(argc >= 2) size = atoi(argv[1]);
  deque<Noisy> dn;
  Noisy n;
  for(int i = 0; i < size; i++)
```

```
    dn.push_back(n);
  cout << "\n cleaning up \n";
} ///:~
```

Here you will never see any destructors before the words “cleaning up” appear. Since the **deque** allocates all its storage in blocks instead of a contiguous array like **vector**, it never needs to move existing storage (thus no additional copy-constructions and destructions occur). It simply allocates a new block. For the same reason, the **deque** can just as efficiently add elements to the *beginning* of the sequence, since if it runs out of storage it (again) just allocates a new block for the beginning. Insertions in the middle of a **deque**, however, could be even messier than for **vector** (but not as costly).

Because a **deque** never moves its storage, a held iterator never becomes invalid when you add new things to either end of a deque, as it was demonstrated to do with **vector** (in **VectorCoreDump.cpp**). However, it’s still possible (albeit harder) to do bad things:

```
//: C20:DequeCoreDump.cpp
// How to break a program using a deque
#include <queue>
#include <iostream>
using namespace std;

int main() {
  deque<int> di(100, 0);
  // No problem iterating from beginning to end,
  // even though it spans multiple blocks:
  copy(di.begin(), di.end(),
    ostream_iterator<int>(cout, " "));
  deque<int>::iterator i = // In the middle:
    di.begin() + di.size() / 2;;
  // Walk the iterator forward as you perform
  // a lot of insertions in the middle:
  for(int j = 0; j < 1000; j++) {
    cout << j << endl;
    di.insert(i++, 1); // Eventually breaks
  }
} ///:~
```

Of course, there are two things here that you wouldn’t normally do with a **deque**: first, elements are inserted in the middle, which **deque** allows but isn’t designed for. Second, calling **insert( )** repeatedly with the same

iterator would not ordinarily cause an access violation, but the iterator is walked forward after each insertion. I'm guessing it eventually walks off the end of a block, but I'm not sure what actually causes the problem.

If you stick to what **deque** is best at – insertions and removals from either end, reasonably rapid traversals and fairly fast random-access using **operator[ ]** – you'll be in good shape.

Checked random-access

Both **vector** and **deque** provide two ways to perform random access of their elements: the **operator[ ]**, which you've seen already, and **at( )**, which checks the boundaries of the container that's being indexed and throws an exception if you go out of bounds. It does cost more to use **at( )**:

```
//: C20:IndexingVsAt.cpp
// Comparing "at()" to operator[]
#include "../require.h"
#include <vector>
#include <deque>
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[]) {
  requireMinArgs(argc, 1);
  long count = 1000;
  int sz = 1000;
  if(argc >= 2) count = atoi(argv[1]);
  if(argc >= 3) sz = atoi(argv[2]);
  vector<int> vi(sz);
  clock_t ticks = clock();
  for(int i1 = 0; i1 < count; i1++)
    for(int j = 0; j < sz; j++)
      vi[j];
  cout << "vector[]" << clock() - ticks << endl;
  ticks = clock();
  for(int i2 = 0; i2 < count; i2++)
    for(int j = 0; j < sz; j++)
      vi.at(j);
  cout << "vector::at()" << clock()-ticks <<endl;
  deque<int> di(sz);
```

```
  ticks = clock();
  for(int i3 = 0; i3 < count; i3++)
    for(int j = 0; j < sz; j++)
      di[j];
  cout << "deque[]" << clock() - ticks << endl;
  ticks = clock();
  for(int i4 = 0; i4 < count; i4++)
    for(int j = 0; j < sz; j++)
      di.at(j);
  cout << "deque::at()" << clock()-ticks <<endl;
  // Demonstrate at() when you go out of bounds:
  di.at(vi.size() + 1);
} ///:~
```

As you'll learn in the exception-handling chapter, different systems may handle the uncaught exception in different ways, but you'll know one way or another that something went wrong with the program when using **at( )**, whereas it's possible to go blundering ahead using **operator[ ]**.

list

A **list** is implemented as a doubly-linked list and is thus designed for rapid insertion and removal of elements in the middle of the sequence (whereas for **vector** and **deque** this is a much more costly operation). A list is so slow when randomly accessing elements that it does not have an **operator[ ]**. It's best used when you're traversing a sequence, in order, from beginning to end (or end to beginning) rather than choosing elements randomly from the middle. Even then the traversal is significantly slower than either a **vector** or a **deque**, but if you aren't doing a lot of traversals that won't be your bottleneck.

Another thing to be aware of with a **list** is the memory overhead of each link, which requires a forward and backward pointer on top of the storage for the actual object. Thus a **list** is a better choice when you have larger objects that you'll be inserting and removing from the middle of the **list**. It's better not to use a **list** if you think you might be traversing it a lot, looking for objects, since the amount of time it takes to get from the beginning of the **list** – which is the only place you can start unless you've already got an iterator to somewhere you know is closer to your destination – to the object of interest is proportional to the number of objects between the beginning and that object.

The objects in a **list** never move after they are created; "moving" a list element means changing the links, but never copying or assigning the actual objects. This means that a held iterator never moves when you add new things to a list as it was demonstrated to do in **vector**. Here's an example using the **Noisy** class:

```
//: C20:ListStability.cpp
// Things don't move around in lists
#include "Noisy.h"
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  list<Noisy> l;
  ostream_iterator<Noisy> out(cout, " ");
  generate_n(back_inserter(l), 25, NoisyGen());
  cout << "\n Printing the list:" << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Reversing the list:" << endl;
  l.reverse();
  copy(l.begin(), l.end(), out);
  cout << "\n Sorting the list:" << endl;
  l.sort();
  copy(l.begin(), l.end(), out);
  cout << "\n Swapping two elements:" << endl;
  list<Noisy>::iterator it1, it2;
  it1 = it2 = l.begin();
  it2++;
  swap(*it1, *it2);
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Using generic reverse(): " << endl;
  reverse(l.begin(), l.end());
  cout << endl;
  copy(l.begin(), l.end(), out);
  cout << "\n Cleanup" << endl;
} ///:~
```

Operations as seemingly radical as reversing and sorting the list require no copying of objects, because instead of moving the objects, the links are simply changed. However, notice that **sort( )** and **reverse( )** are member

functions of **list**, so they have special knowledge of the internals of **list** and can perform the pointer movement instead of copying. On the other hand, the **swap( )** function is a generic algorithm, and doesn't know about **list** in particular and so it uses the copying approach for swapping two elements. There are also generic algorithms for **sort( )** and **reverse( )**, but if you try to use these you'll discover that the generic **reverse( )** performs lots of copying and destruction (so you should never use it with a **list**) and the generic **sort( )** simply doesn't work because it requires random-access iterators that **list** doesn't provide (a definite benefit, since this would certainly be an expensive way to sort compared to **list**'s own **sort( )**). The generic **sort( )** and **reverse( )** should only be used with arrays, **vector**s and **deque**s.

If you have large and complex objects you may want to choose a **list** first, especially if construction, destruction, copy-construction and assignment are expensive and if you are doing things like sorting the objects or otherwise reordering them a lot.

Special list operations

The **list** has some special operations that are built-in to make the best use of the structure of the **list**. You've already seen **reverse( )** and **sort( )**, and here are some of the others in use:

```
//: C20:ListSpecialFunctions.cpp
#include "Noisy.h"
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
ostream_iterator<Noisy> out(cout, " ");

void print(list<Noisy>& ln, char* comment = "") {
  cout << "\n" << comment << ":\n";
  copy(ln.begin(), ln.end(), out);
  cout << endl;
}

int main() {
  typedef list<Noisy> LN;
  LN l1, l2, l3, l4;
  generate_n(back_inserter(l1), 6, NoisyGen());
  generate_n(back_inserter(l2), 6, NoisyGen());
```

```
  generate_n(back_inserter(l3), 6, NoisyGen());
  generate_n(back_inserter(l4), 6, NoisyGen());
  print(l1, "l1"); print(l2, "l2");
  print(l3, "l3"); print(l4, "l4");
  LN::iterator it1 = l1.begin();
  it1++; it1++; it1++;
  l1.splice(it1, l2);
  print(l1, "l1 after splice(it1, l2)");
  print(l2, "l2 after splice(it1, l2)");
  LN::iterator it2 = l3.begin();
  it2++; it2++; it2++;
  l1.splice(it1, l3, it2);
  print(l1, "l1 after splice(it1, l3, it2)");
  LN::iterator it3 = l4.begin(), it4 = l4.end();
  it3++; it4--;
  l1.splice(it1, l4, it3, it4);
  print(l1, "l1 after splice(it1,l4,it3,it4)");
  Noisy n;
  LN l5(3, n);
  generate_n(back_inserter(l5), 4, NoisyGen());
  l5.push_back(n);
  print(l5, "l5 before remove()");
  l5.remove(l5.front());
  print(l5, "l5 after remove()");
  l1.sort(); l5.sort();
  l5.merge(l1);
  print(l5, "l5 after l5.merge(l1)");
  cout << "\n Cleanup" << endl;
} ///:~
```

The **print( )** function is used to display results. After filling four **list**s with **Noisy** objects, one list is spliced into another in three different ways. In the first, the entire list **l2** is spliced into **l1** at the iterator **it1**. Notice that after the splice, **l2** is empty – splicing means removing the elements from the source list. The second splice inserts elements from **l3** starting at **it2** into **l1** starting at **it1**. The third splice starts at **it1** and uses elements from **l4** starting at **it3** and ending at **it4** (the seemingly-redundant mention of the source list is because the elements must be erased from the source list as part of the transfer to the destination list)**.**

The output from the code that demonstrates **remove( )** shows that the list does not have to be sorted in order for all the elements of a particular value to be removed.

Finally, if you **merge( )** one list with another, the merge only works sensibly if the lists have been sorted. What you end up with in that case is a sorted list containing all the elements from both lists (the source list is erased – that is, the elements are *moved* to the destination list).

There's also a **unique( )** member function that removes all duplicates, but only if the **list** has been sorted first:

```
//: C20:UniqueList.cpp
// Testing list's unique() function
#include <list>
#include <iostream>
using namespace std;

int a[] = { 1, 3, 1, 4, 1, 5, 1, 6, 1 };
const int asz = sizeof a / sizeof *a;

int main() {
  // For output:
  ostream_iterator<int> out(cout, " ");
  list<int> li(a, a + asz);
  li.unique();
  // Oops! No duplicates removed:
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Must sort it first:
  li.sort();
  copy(li.begin(), li.end(), out);
  cout << endl;
  // Now unique() will have an effect:
  li.unique();
  copy(li.begin(), li.end(), out);
  cout << endl;
} ///:~
```

The **list** constructor used here takes the starting and past-the-end iterator from another container, and it copies all the elements from that container into itself (a similar constructor is available for all the containers). Here, the "container" is just an array, and the "iterators" are pointers into that array, but because of the design of the STL it works with arrays just as easily as any other container.

If you run this program, you'll see that **unique( )** will only remove *adjacent* duplicate elements, and thus sorting is necessary before calling **unique( )**.

There are four additional **list** member functions that are not demonstrated here: a **remove_if( )** that takes a predicate which is used to decide whether an object should be removed, a **unique( )** that takes a binary predicate to perform uniqueness comparisons, a **merge( )** that takes an additional argument which performs comparisons, and a **sort( )** that takes a comparator (to provide a comparison or override the existing one).

list vs. **set**

Looking at the previous example you may note that if you want a sorted list with no duplicates, a **set** can give you that, right? It's interesting to compare the performance of the two containers:

```
//: C20:ListVsSet.cpp
// Comparing list and set performance
#include <iostream>
#include <list>
#include <set>
#include <algorithm>
#include <ctime>
#include <cstdlib>
using namespace std;

class Obj {
  int a[20];
  int val;
public:
  Obj() : val(rand() % 500) {}
  friend bool
  operator<(const Obj& a, const Obj& b) {
    return a.val < b.val;
  }
  friend bool
  operator==(const Obj& a, const Obj& b) {
    return a.val == b.val;
  }
  friend ostream&
  operator<<(ostream& os, const Obj& a) {
    return os << a.val;
```

```
  }
};

template<class Container>
void print(Container& c) {
  typename Container::iterator it;
  for(it = c.begin(); it != c.end(); it++)
    cout << *it << " ";
  cout << endl;
}

struct ObjGen {
  Obj operator()() { return Obj(); }
};

int main() {
  const int sz = 5000;
  srand(time(0));
  list<Obj> lo;
  clock_t ticks = clock();
  generate_n(back_inserter(lo), sz, ObjGen());
  lo.sort();
  lo.unique();
  cout << "list:" << clock() - ticks << endl;
  set<Obj> so;
  ticks = clock();
  generate_n(inserter(so, so.begin()),
    sz, ObjGen());
  cout << "set:" << clock() - ticks << endl;
  print(lo);
  print(so);
} ///:~
```

When you run the program, you should discover that **set** is much faster than **list**. This is reassuring – after all, it *is* **set**'s primary job description!

Swapping all basic sequences

It turns out that all basic sequences have a member function **swap( )** that's designed to switch one sequence with another (however, this **swap( )** is only defined for sequences of the same type). The member **swap( )** makes use of its knowledge of the internal structure of the particular container in order to be efficient:

```
//: C20:Swapping.cpp
// All basic sequence containers can be swapped
#include "Noisy.h"
#include <list>
#include <vector>
#include <deque>
#include <iostream>
#include <algorithm>
using namespace std;
ostream_iterator<Noisy> out(cout, " ");

template<class Cont>
void print(Cont& c, char* comment = "") {
  cout << "\n" << comment << ": ";
  copy(c.begin(), c.end(), out);
  cout << endl;
}

template<class Cont>
void testSwap(char* cname) {
  Cont c1, c2;
  generate_n(back_inserter(c1), 10, NoisyGen());
  generate_n(back_inserter(c2), 5, NoisyGen());
  cout << "\n" << cname << ":" << endl;
  print(c1, "c1"); print(c2, "c2");
  cout << "\n Swapping the " << cname
    << ":" << endl;
  c1.swap(c2);
  print(c1, "c1"); print(c2, "c2");
}

int main() {
  testSwap<vector<Noisy> >("vector");
  testSwap<deque<Noisy> >("deque");
  testSwap<list<Noisy> >("list");
} ///:~
```

When you run this, you'll discover that each type of sequence container is able to swap one sequence for another without any copying or assignments, even if the sequences are of different sizes. In effect, you're completely swapping the memory of one object for another.

The STL algorithms also contain a **swap( )**, and when this function is applied to two containers of the same type, it will use the member **swap( )** to achieve fast performance. Consequently, if you apply the **sort( )** algorithm to a container of containers, you will find that the performance is very fast – it turns out that fast sorting of a container of containers was a design goal of the STL.

Robustness of lists

To break a **list**, you have to work pretty hard:

```
//: C20:ListRobustness.cpp
// lists are harder to break
#include <list>
#include <iostream>
using namespace std;

int main() {
  list<int> li(100, 0);
  list<int>::iterator i = li.begin();
  for(int j = 0; j < li.size() / 2; j++)
    i++;
  // Walk the iterator forward as you perform
  // a lot of insertions in the middle:
  for(int k = 0; k < 1000; k++)
    li.insert(i++, 1); // No problem
  li.erase(i);
  i++;
  *i = 2; // Oops! It's invalid
} ///:~
```

When the link that the iterator **i** was pointing to was erased, it was unlinked from the list and thus became invalid. Trying to move forward to the “next link” from an invalid link is poorly-formed code. Notice that the operation that broke **deque** in **DequeCoreDump.cpp** is perfectly fine with a **list**.

Performance comparison

To get a better feel for the differences between the sequence containers, it’s illuminating to race them against each other while performing various operations.

```
//: C20:SequencePerformance.cpp
// Comparing the performance of the basic
// sequence containers for various operations
#include <vector>
#include <queue>
#include <list>
#include <iostream>
#include <string>
#include <typeinfo>
#include <ctime>
#include <cstdlib>
using namespace std;

class FixedSize {
  int x[20];
  // Automatic generation of default constructor,
  // copy-constructor and operator=
} fs;

template<class Cont>
struct InsertBack {
  void operator()(Cont& c, long count) {
    for(long i = 0; i < count; i++)
      c.push_back(fs);
  }
  char* testName() { return "InsertBack"; }
};

template<class Cont>
struct InsertFront {
  void operator()(Cont& c, long count) {
    long cnt = count * 10;
    for(long i = 0; i < cnt; i++)
      c.push_front(fs);
  }
  char* testName() { return "InsertFront"; }
};

template<class Cont>
struct InsertMiddle {
  void operator()(Cont& c, long count) {
    typename Cont::iterator it;
```

```
    long cnt = count / 10;
    for(long i = 0; i < cnt; i++) {
      // Must get the iterator every time to keep
      // from causing an access violation with
      // vector. Increment it to put it in the
      // middle of the container:
      it = c.begin();
      it++;
      c.insert(it, fs);
    }
  }
  char* testName() { return "InsertMiddle"; }
};

template<class Cont>
struct RandomAccess { // Not for list
  void operator()(Cont& c, long count) {
    int sz = c.size();
    long cnt = count * 100;
    for(long i = 0; i < cnt; i++)
      c[rand() % sz];
  }
  char* testName() { return "RandomAccess"; }
};

template<class Cont>
struct Traversal {
  void operator()(Cont& c, long count) {
    long cnt = count / 100;
    for(long i = 0; i < cnt; i++) {
      typename Cont::iterator it = c.begin(),
        end = c.end();
      while(it != end) it++;
    }
  }
  char* testName() { return "Traversal"; }
};

template<class Cont>
struct Swap {
  void operator()(Cont& c, long count) {
    int middle = c.size() / 2;
```

```
    typename Cont::iterator it = c.begin(),
      mid = c.begin();
    it++; // Put it in the middle
    for(int x = 0; x < middle + 1; x++)
      mid++;
    long cnt = count * 10;
    for(long i = 0; i < cnt; i++)
      swap(*it, *mid);
  }
  char* testName() { return "Swap"; }
};

template<class Cont>
struct RemoveMiddle {
  void operator()(Cont& c, long count) {
    long cnt = count / 10;
    if(cnt > c.size()) {
      cout << "RemoveMiddle: not enough elements"
        << endl;
      return;
    }
    for(long i = 0; i < cnt; i++) {
      typename Cont::iterator it = c.begin();
      it++;
      c.erase(it);
    }
  }
  char* testName() { return "RemoveMiddle"; }
};

template<class Cont>
struct RemoveBack {
  void operator()(Cont& c, long count) {
    long cnt = count * 10;
    if(cnt > c.size()) {
      cout << "RemoveBack: not enough elements"
        << endl;
      return;
    }
    for(long i = 0; i < cnt; i++)
      c.pop_back();
  }
```

```
  char* testName() { return "RemoveBack"; }
};

template<class Op, class Container>
void measureTime(Op f, Container& c, long count){
  string id(typeid(f).name());
  bool Deque = id.find("deque") != string::npos;
  bool List = id.find("list") != string::npos;
  bool Vector = id.find("vector") !=string::npos;
  string cont = Deque ? "deque" : List ? "list"
    : Vector? "vector" : "unknown";
  cout << f.testName() << " for " << cont << ": ";
  // Standard C library CPU ticks:
  clock_t ticks = clock();
  f(c, count); // Run the test
  ticks = clock() - ticks;
  cout << ticks << endl;
}

typedef deque<FixedSize> DF;
typedef list<FixedSize> LF;
typedef vector<FixedSize> VF;

int main(int argc, char* argv[]) {
  srand(time(0));
  long count = 1000;
  if(argc >= 2) count = atoi(argv[1]);
  DF deq;
  LF lst;
  VF vec, vecres;
  vecres.reserve(count); // Preallocate storage
  measureTime(InsertBack<VF>(), vec, count);
  measureTime(InsertBack<VF>(), vecres, count);
  measureTime(InsertBack<DF>(), deq, count);
  measureTime(InsertBack<LF>(), lst, count);
  // Can't push_front() with a vector:
//! measureTime(InsertFront<VF>(), vec, count);
  measureTime(InsertFront<DF>(), deq, count);
  measureTime(InsertFront<LF>(), lst, count);
  measureTime(InsertMiddle<VF>(), vec, count);
  measureTime(InsertMiddle<DF>(), deq, count);
  measureTime(InsertMiddle<LF>(), lst, count);
```

```
  measureTime(RandomAccess<VF>(), vec, count);
  measureTime(RandomAccess<DF>(), deq, count);
  // Can't operator[] with a list:
//! measureTime(RandomAccess<LF>(), lst, count);
  measureTime(Traversal<VF>(), vec, count);
  measureTime(Traversal<DF>(), deq, count);
  measureTime(Traversal<LF>(), lst, count);
  measureTime(Swap<VF>(), vec, count);
  measureTime(Swap<DF>(), deq, count);
  measureTime(Swap<LF>(), lst, count);
  measureTime(RemoveMiddle<VF>(), vec, count);
  measureTime(RemoveMiddle<DF>(), deq, count);
  measureTime(RemoveMiddle<LF>(), lst, count);
  vec.resize(vec.size() * 10); // Make it bigger
  measureTime(RemoveBack<VF>(), vec, count);
  measureTime(RemoveBack<DF>(), deq, count);
  measureTime(RemoveBack<LF>(), lst, count);
} ///:~
```

This example makes heavy use of templates to eliminate redundancy, save space, guarantee identical code and improve clarity. Each test is represented by a class that is templatized on the container it will operate on. The test itself is inside the **operator( )** which, in each case, takes a reference to the container and a repeat count – this count is not always used exactly as it is, but sometimes increased or decreased to prevent the test from being too short or too long. The repeat count is just a factor, and all tests are compared using the same value.

Each test class also has a member function that returns its name, so that it can easily be printed. You might think that this should be accomplished using run-time type identification, but since the actual name of the class involves a template expansion, this is turns out to be the more direct approach.

The **measureTime( )** function template takes as its first template argument the operation that it's going to test – which is itself a class template selected from the group defined previously in the listing. The template argument **Op** will not only contain the name of the class, but also (decorated into it) the type of the container it's working with. The RTTI **typeid( )** operation allows the name of the class to be extracted as a **char***, which can then be used to create a **string** called **id**. This **string** can be searched using **string::find( )** to look for **deque**, **list** or **vector**. The **bool** variable that corresponds to the matching **string** becomes **true**,

and this is used to properly initialize the **string cont** so the container name can be accurately printed, along with the test name.

Once the type of test and the container being tested has been printed out, the actual test is quite simple. The Standard C library function **clock( )** is used to capture the starting and ending CPU ticks (this is typically more fine-grained than trying to measure seconds). Since **f** is an object of type **Op**, which is a class that has an **operator( )**, the line:

```
f(c, count);
```

is actually calling the **operator( )** for the object **f**.

In **main( )**, you can see that each different type of test is run on each type of container, except for the containers that don't support the particular operation being tested (these are commented out).

When you run the program, you'll get comparative performance numbers for your particular compiler and your particular operating system and platform. Although this is only intended to give you a feel for the various performance features relative to the other sequences, it is not a bad way to get a quick-and-dirty idea of the behavior of your library, and also to compare one library with another.

set

The **set** produces a container that will accept only one of each thing you place in it; it also sorts the elements (sorting isn't intrinsic to the conceptual definition of a set, but the STL **set** stores its elements in a balanced binary tree to provide rapid lookups, thus producing sorted results when you traverse it). The first two examples in this chapter used **set**s.

Consider the problem of creating an index for a book. You might like to start with all the words in the book, but you only want one instance of each word and you want them sorted. Of course, a **set** is perfect for this, and solves the problem effortlessly. However, there's also the problem of punctuation and any other non-alpha characters, which must be stripped off to generate proper words. One solution to this problem is to use the Standard C library function **strtok( )**, which produces tokens (in our case, words) given a set of delimiters to strip out:

```
//: C20:WordList.cpp
// Display a list of words used in a document
#include "../require.h"
```

```
#include <string>
#include <cstring>
#include <set>
#include <iostream>
#include <fstream>
using namespace std;

const char* delimiters =
  " \t;()\"<>:{}[]+-=&*#.,/\\~";

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  set<string> wordlist;
  string line;
  while(getline(in, line)) {
    // Capture individual words:
    char* s = // Cast probably won't crash:
      strtok((char*)line.c_str(), delimiters);
    while(s) {
      // Automatic type conversion:
      wordlist.insert(s);
      s = strtok(0, delimiters);
    }
  }
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
       ostream_iterator<string>(cout, "\n"));
} ///:~
```

strtok( ) takes the starting address of a character buffer (the first argument) and looks for delimiters (the second argument). It replaces the delimiter with a zero, and returns the address of the beginning of the token. If you call it subsequent times with a first argument of zero it will continue extracting tokens from the rest of the string until it finds the end. In this case, the delimiters are those that delimit the keywords and identifiers of C++, so it extracts these keywords and identifiers. Each word is turned into a **string** and placed into the **wordlist** vector, which eventually contains the whole file, broken up into words.

You don't have to use a **set** just to get a sorted sequence. You can use the **sort( )** function (along with a multitude of other functions in the STL) on different STL containers. However, it's likely that **set** will be faster.

Eliminating **strtok( )**

Some programmers consider **strtok( )** to be the poorest design in the Standard C library because it uses a **static** buffer to hold its data between function calls. This means:

1. You can't use **strtok( )** in two places at the same time
2. You can't use **strtok( )** in a multithreaded program
3. You can't use **strtok( )** in a library that might be used in a multithreaded program
4. **strtok( )** modifies the input sequence, which can produce unexpected side effects
5. **strtok( )** depends on reading in "lines", which means you need a buffer big enough for the longest line. This produces both wastefully-sized buffers, and lines longer than the "longest" line. This can also introduce security holes. (Notice that the buffer size problem was eliminated in **WordList.cpp** by using **string** input, but this required a cast so that **strtok( )** could modify the data in the string – a dangerous approach for general-purpose programming).

For all these reasons it seems like a good idea to find an alternative for **strtok( )**. The following example will use an **istreambuf_iterator** (introduced earlier) to move the characters from one place (which happens to be an **istream**) to another (which happens to be a **string**), depending on whether the Standard C library function **isalpha( )** is true:

```
//: C20:WordList2.cpp
// Eliminating strtok() from Wordlist.cpp
#include "../require.h"
#include <string>
#include <cstring>
#include <set>
#include <iostream>
#include <fstream>
#include <iterator>
using namespace std;

int main(int argc, char* argv[]) {
  using namespace std;
```

```
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  istreambuf_iterator<char> p(in), end;
  set<string> wordlist;
  while (p != end) {
    string word;
    insert_iterator<string>
      ii(word, word.begin());
    // Find the first alpha character:
    while(!isalpha(*p) && p != end)
      p++;
    // Copy until the first non-alpha character:
    while (isalpha(*p) && p != end)
      *ii++ = *p++;
    if (word.size() != 0)
      wordlist.insert(word);
  }
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
    ostream_iterator<string>(cout, "\n"));
} ///:~
```

This example was suggested by Nathan Myers, who invented the **istreambuf_iterator** and its relatives. This iterator extracts information character-by-character from a stream. Although the **istreambuf_iterator** template argument might suggest to you that you could extract, for example, **int**s instead of **char**, that's not the case. The argument must be of some character type – a regular **char** or a wide character.

After the file is open, an **istreambuf_iterator** called **p** is attached to the **istream** so characters can be extracted from it. The **set<string>** called **wordlist** will be used to hold the resulting words.

The **while** loop reads words until the end of the input stream is found. This is detected using the default constructor for **istreambuf_iterator** which produces the past-the-end iterator object **end**. Thus, if you want to test to make sure you're not at the end of the stream, you simply say **p != end**.

The second type of iterator that's used here is the **insert_iterator**, which creates an iterator that knows how to insert objects into a container. Here, the "container" is the **string** called **word** which, for the purposes of **insert_iterator**, behaves like a container. The constructor for

insert_iterator requires the container and an iterator indicating where it should start inserting the characters. You could also use a **back_insert_iterator**, which requires that the container have a **push_back( )** (**string** does). There is also a **back_insert_iterator**, but that requires that the container have a **push_back( )**, which **string** does not.

After the **while** loop sets everything up, it begins by looking for the first alpha character, incrementing **start** until that character is found. Then it copies characters from one iterator to the other, stopping when a non-alpha character is found. Each **word**, assuming it is non-empty, is added to **wordlist**.

StreamTokenizer: a more flexible solution

The above program parses its input into strings of words containing only alpha characters, but that's still a special case compared to the generality of **strtok( )**. What we'd like now is an actual replacement for **strtok( )** so we're never tempted to use it. **WordList2.cpp** can be modified to create a class called **StreamTokenizer** that delivers a new token as a **string** whenever you call **next( )**, according to the delimiters you give it upon construction (very similar to **strtok( )**):

```
//: C20:StreamTokenizer.h
// C++ Replacement for Standard C strtok()
#ifndef STREAMTOKENIZER_H
#define STREAMTOKENIZER_H
#include <string>
#include <iostream>
#include <iterator>

class StreamTokenizer {
  typedef std::istreambuf_iterator<char> It;
  It p, end;
  std::string delimiters;
  bool isDelimiter(char c) {
    return
      delimiters.find(c) != std::string::npos;
  }
public:
  StreamTokenizer(std::istream& is,
```

```
    std::string delim = " \t\n;()\"<>:{}[]+-=&*#"
    ".,/\\~!0123456789") : p(is), end(It()),
    delimiters(delim) {}
  std::string next(); // Get next token
};
#endif STREAMTOKENIZER_H ///:~
```

The default delimiters for the **StreamTokenizer** constructor extract words with only alpha characters, as before, but now you can choose different delimiters to parse different tokens. The implementation of **next( )** looks similar to **Wordlist2.cpp**:

```
//: C20:StreamTokenizer.cpp {O}
#include "StreamTokenizer.h"
using namespace std;

string StreamTokenizer::next() {
  string result;
  if(p != end) {
    insert_iterator<string>
      ii(result, result.begin());
    while(isDelimiter(*p) && p != end)
      p++;
    while (!isDelimiter(*p) && p != end)
      *ii++ = *p++;
  }
  return result;
} ///:~
```

The first non-delimiter is found, then characters are copied until a delimiter is found, and the resulting **string** is returned. Here's a test:

```
//: C20:TokenizeTest.cpp
//{L} StreamTokenizer
// Test StreamTokenizer
#include "StreamTokenizer.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
```

```
  ifstream in(argv[1]);
  assure(in, argv[1]);
  StreamTokenizer words(in);
  set<string> wordlist;
  string word;
  while((word = words.next()).size() != 0)
    wordlist.insert(word);
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
    ostream_iterator<string>(cout, "\n"));
} ///:~
```

Now the tool is more reusable than before, but it's still inflexible, because it can only work with an **istream**. This isn't as bad as it first seems, since a **string** can be turned into an **istream** via an **istringstream**. But in the next section we'll come up with the most general, reusable tokenizing tool, and this should give you a feeling of what "reusable" really means, and the effort necessary to create truly reusable code.

A completely reusable tokenizer

Since the STL containers and algorithms all revolve around iterators, the most flexible solution will itself be an iterator. You could think of the **TokenIterator** as an iterator that wraps itself around any other iterator that can produce characters. Because it is designed as an input iterator (the most primitive type of iterator) it can be used with any STL algorithm. Not only is it a useful tool in itself, the **TokenIterator** is also a good example of how you can design your own iterators.[63]

The **TokenIterator** is doubly flexible: first, you can choose the type of iterator that will produce the **char** input. Second, instead of just saying what characters represent the delimiters, **TokenIterator** will use a predicate which is a function object whose **operator( )** takes a **char** and decides if it should be in the token or not. Although the two examples given here have a static concept of what characters belong in a token, you could easily design your own function object to change its state as the characters are read, producing a more sophisticated parser.

The following header file contains the two basic predicates **Isalpha** and **Delimiters**, along with the template for **TokenIterator**:

[63] This is another example coached by Nathan Myers.

```
//: C20:TokenIterator.h
#ifndef TOKENITERATOR_H
#define TOKENITERATOR_H
#include <string>
#include <iterator>
#include <algorithm>
#include <cctype>

struct Isalpha {
  bool operator()(char c) {
    using namespace std; //[[For a compiler bug]]
    return isalpha(c);
  }
};

class Delimiters {
  std::string exclude;
public:
  Delimiters() {}
  Delimiters(const std::string& excl)
    : exclude(excl) {}
  bool operator()(char c) {
    return exclude.find(c) == std::string::npos;
  }
};

template <class InputIter, class Pred = Isalpha>
class TokenIterator: public std::iterator<
  std::input_iterator_tag,std::string,ptrdiff_t>{
  InputIter first;
  InputIter last;
  std::string word;
  Pred predicate;
public:
  TokenIterator(InputIter begin, InputIter end,
    Pred pred = Pred())
    : first(begin), last(end), predicate(pred) {
      ++*this;
  }
  TokenIterator() {} // End sentinel
  // Prefix increment:
  TokenIterator& operator++() {
```

```
    word.resize(0);
    first = std::find_if(first, last, predicate);
    while (first != last && predicate(*first))
      word += *first++;
    return *this;
  }
  // Postfix increment
  class Proxy {
    std::string word;
  public:
    Proxy(const std::string& w) : word(w) {}
    std::string operator*() { return word; }
  };
  Proxy operator++(int) {
    Proxy d(word);
    ++*this;
    return d;
  }
  // Produce the actual value:
  std::string operator*() const { return word; }
  std::string* operator->() const {
    return &(operator*());
  }
  // Compare iterators:
  bool operator==(const TokenIterator&) {
    return word.size() == 0 && first == last;
  }
  bool operator!=(const TokenIterator& rv) {
    return !(*this == rv);
  }
};
#endif // TOKENITERATOR_H ///:~
```

TokenIterator is inherited from the **std::iterator** template. It might appear that there's some kind of functionality that comes with **std::iterator**, but it is purely a way of tagging an iterator so that a container that uses it knows what it's capable of. Here, you can see **input_iterator_tag** as a template argument – this tells anyone who asks that a **TokenIterator** only has the capabilities of an input iterator, and cannot be used with algorithms requiring more sophisticated iterators. Apart from the tagging, **std::iterator** doesn't do anything else, which means you must design all the other functionality in yourself.

TokenIterator may look a little strange at first, because the first constructor requires both a “begin” and “end” iterator as arguments, along with the predicate. Remember that this is a “wrapper” iterator that has no idea of how to tell whether it’s at the end of its input source, so the ending iterator is necessary in the first constructor. The reason for the second (default) constructor is that the STL algorithms (and any algorithms you write) need a **TokenIterator** sentinel to be the past-the-end value. Since all the information necessary to see if the **TokenIterator** has reached the end of its input is collected in the first constructor, this second constructor creates a **TokenIterator** that is merely used as a placeholder in algorithms.

The core of the behavior happens in **operator++**. This erases the current value of **word** using **string::resize( )**, then finds the first character that satisfies the predicate (thus discovering the beginning of the new token) using **find_if( )** (from the STL algorithms, discussed in the following chapter). The resulting iterator is assigned to **first**, thus moving **first** forward to the beginning of the token. Then, as long as the end of the input is not reached and the predicate is satisfied, characters are copied into the word from the input. Finally the new token is returned.

The postfix increment requires a proxy object to hold the value before the increment, so it can be returned (see the operator overloading chapter for more details of this). Producing the actual value is a straightforward **operator***. The only other functions that must be defined for an output iterator are the **operator==** and **operator!=** to indicate whether the **TokenIterator** has reached the end of its input. You can see that the argument for **operator==** is ignored – it only cares about whether it has reached its internal **last** iterator. Notice that **operator!=** is defined in terms of **operator==**.

A good test of **TokenIterator** includes a number of different sources of input characters including a **streambuf_iterator**, a **char***, and a **deque<char>::iterator**. Finally, the original **Wordlist.cpp** problem is solved:

```
//: C20:TokenIteratorTest.cpp
#include "TokenIterator.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <vector>
#include <deque>
#include <set>
```

```
using namespace std;

int main() {
  ifstream in("TokenIteratorTest.cpp");
  assure(in, "TokenIteratorTest.cpp");
  ostream_iterator<string> out(cout, "\n");
  typedef istreambuf_iterator<char> IsbIt;
  IsbIt begin(in), isbEnd;
  Delimiters
    delimiters(" \t\n~;()\"<>:{}[]+-=&*#.,/\\");
  TokenIterator<IsbIt, Delimiters>
    wordIter(begin, isbEnd, delimiters),
    end;
  vector<string> wordlist;
  copy(wordIter, end, back_inserter(wordlist));
  // Output results:
  copy(wordlist.begin(), wordlist.end(), out);
  *out++ = "---------------------------------";
  // Use a char array as the source:
  char* cp =
    "typedef std::istreambuf_iterator<char> It";
  TokenIterator<char*, Delimiters>
    charIter(cp, cp + strlen(cp), delimiters),
    end2;
  vector<string> wordlist2;
  copy(charIter, end2, back_inserter(wordlist2));
  copy(wordlist2.begin(), wordlist2.end(), out);
  *out++ = "---------------------------------";
  // Use a deque<char> as the source:
  ifstream in2("TokenIteratorTest.cpp");
  deque<char> dc;
  copy(IsbIt(in2), IsbIt(), back_inserter(dc));
  TokenIterator<deque<char>::iterator,Delimiters>
    dcIter(dc.begin(), dc.end(), delimiters),
    end3;
  vector<string> wordlist3;
  copy(dcIter, end3, back_inserter(wordlist3));
  copy(wordlist3.begin(), wordlist3.end(), out);
  *out++ = "---------------------------------";
  // Reproduce the Wordlist.cpp example:
  ifstream in3("TokenIteratorTest.cpp");
  TokenIterator<IsbIt, Delimiters>
```

```
    wordIter2(IsbIt(in3), isbEnd, delimiters);
  set<string> wordlist4;
  while(wordIter2 != end)
    wordlist4.insert(*wordIter2++);
  copy(wordlist4.begin(), wordlist4.end(), out);
} ///:~
```

When using an **istreambuf_iterator**, you create one to attach to the **istream** object, and one with the default constructor as the past-the-end marker. Both of these are used to create the **TokenIterator** that will actually produce the tokens; the default constructor produces the faux **TokenIterator** past-the-end sentinel (this is just a placeholder, and as mentioned previously is actually ignored). The **TokenIterator** produces **string**s that are inserted into a container which must, naturally, be a container of **string** – here a **vector<string>** is used in all cases except the last (you could also concatenate the results onto a **string**). Other than that, a **TokenIterator** works like any other input iterator.

stack

The **stack**, along with the **queue** and **priority_queue**, are classified as *adapters*, which means they are implemented using one of the basic sequence containers: **vector**, **list** or **deque**. This, in my opinion, is an unfortunate case of confusing what something does with the details of its underlying implementation – the fact that these are called “adapters” is of primary value only to the creator of the library. When you use them, you generally don’t care that they’re adapters, but instead that they solve your problem. Admittedly there are times when it’s useful to know that you can choose an alternate implementation or build an adapter from an existing container object, but that’s generally one level removed from the adapter’s behavior. So, while you may see it emphasized elsewhere that a particular container is an adapter, I shall only point out that fact when it’s useful. Note that each type of adapter has a default container that it’s built upon, and this default is the most sensible implementation, so in most cases you won’t need to concern yourself with the underlying implementation.

The following example shows **stack<string>** implemented in the three possible ways: the default (which uses **deque**), with a **vector** and with a **list**:

```
//: C20:Stack1.cpp
// Demonstrates the STL stack
```

```
#include "../require.h"
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <vector>
#include <string>
using namespace std;

// Default: deque<string>:
typedef stack<string> Stack1;
// Use a vector<string>:
typedef stack<string, vector<string> > Stack2;
// Use a list<string>:
typedef stack<string, list<string> > Stack3;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack1 textlines; // Try the different versions
  // Read file and store lines in the stack:
  string line;
  while(getline(in, line))
    textlines.push(line + "\n");
  // Print lines from the stack and pop them:
  while(!textlines.empty()) {
    cout << textlines.top();
    textlines.pop();
  }
} ///:~
```

The **top( )** and **pop( )** operations will probably seem non-intuitive if you've used other **stack** classes. When you call **pop( )** it returns void rather than the top element that you might have expected. If you want the top element, you get a reference to it with **top( )**. It turns out this is more efficient, since a traditional **pop( )** would have to return a value rather than a reference, and thus invoke the copy-constructor. When you're using a **stack** (or a **priority_queue**, described later) you can efficiently refer to **top( )** as many times as you want, then discard the top element explicitly using **pop( )** (perhaps if some other term than the familiar "pop" had been used, this would have been a bit clearer).

The **stack** template has a very simple interface, essentially the member functions you see above. It doesn't have sophisticated forms of initialization or access, but if you need that you can use the underlying container that the **stack** is implemented upon. For example, suppose you have a function that expects a **stack** interface but in the rest of your program you need the objects stored in a **list**. The following program stores each line of a file along with the leading number of spaces in that line (you might imagine it as a starting point for performing some kinds of source-code reformatting):

```
//: C20:Stack2.cpp
// Converting a list to a stack
#include "../require.h"
#include <iostream>
#include <fstream>
#include <stack>
#include <list>
#include <string>
using namespace std;

// Expects a stack:
template<class Stk>
void stackOut(Stk& s, ostream& os = cout) {
  while(!s.empty()) {
    os << s.top() << "\n";
    s.pop();
  }
}

class Line {
  string line; // Without leading spaces
  int lspaces; // Number of leading spaces
public:
  Line(string s) : line(s) {
    lspaces = line.find_first_not_of(' ');
    if(lspaces == string::npos)
      lspaces = 0;
    line = line.substr(lspaces);
  }
  friend ostream&
  operator<<(ostream& os, const Line& l) {
    for(int i = 0; i < l.lspaces; i++)
      os << ' ';
```

```
    return os << l.line;
  }
  // Other functions here...
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // File name is argument
  ifstream in(argv[1]);
  assure(in, argv[1]);
  list<Line> lines;
  // Read file and store lines in the list:
  string s;
  while(getline(in, s))
    lines.push_front(s);
  // Turn the list into a stack for printing:
  stack<Line, list<Line> > stk(lines);
  stackOut(stk);
} ///:~
```

The function that requires the **stack** interface just sends each **top( )** object to an **ostream** and then removes it by calling **pop( )**. The **Line** class determines the number of leading spaces, then stores the contents of the line *without* the leading spaces. The **ostream operator<<** re-inserts the leading spaces so the line prints properly, but you can easily change the number of spaces by changing the value of **lspaces** (the member functions to do this are not shown here).

In **main( )**, the input file is read into a **list<Line>**, then a **stack** is wrapped around this list so it can be sent to **stackOut( )**.

You cannot iterate through a **stack**; this emphasizes that you only want to perform **stack** operations when you create a **stack**. You can get equivalent "stack" functionality using a **vector** and its **back( )**, **push_back( )** and **pop_back( )** methods, and then you have all the additional functionality of the **vector**. **Stack1.cpp** can be rewritten to show this:

```
//: C20:Stack3.cpp
// Using a vector as a stack; modified Stack1.cpp
#include "../require.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
```

```
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  vector<string> textlines;
  string line;
  while(getline(in, line))
    textlines.push_back(line + "\n");
  while(!textlines.empty()) {
    cout << textlines.back();
    textlines.pop_back();
  }
} ///:~
```

You'll see this produces the same output as **Stack1.cpp**, but you can now perform **vector** operations as well. Of course, **list** has the additional ability to push things at the front, but it's generally less efficient than using **push_back( )** with **vector**. (In addition, **deque** is usually more efficient than **list** for pushing things at the front).

queue

The **queue** is a restricted form of a **deque** – you can only enter elements at one end, and pull them off the other end. Functionally, you could use a **deque** anywhere you need a **queue**, and you would then also have the additional functionality of the **deque**. The only reason you need to use a **queue** rather than a **deque**, then, is if you want to emphasize that you will only be performing queue-like behavior.

The **queue** is an adapter class like **stack**, in that it is built on top of another sequence container. As you might guess, the ideal implementation for a **queue** is a **deque**, and that is the default template argument for the **queue**; you'll rarely need a different implementation.

Queues are often used when modeling systems where some elements of the system are waiting to be served by other elements in the system. A classic example of this is the "bank-teller problem," where you have customers arriving at random intervals, getting into a line, and then being served by a set of tellers. Since the customers arrive randomly and each take a random amount of time to be served, there's no way to

deterministically know how long the line will be at any time. However, it's possible to simulate the situation and see what happens.

A problem in performing this simulation is the fact that, in effect, each customer and teller should be run by a separate process. What we'd like is a multithreaded environment, then each customer or teller would have their own thread. However, Standard C++ has no model for multithreading so there is no standard solution to this problem. On the other hand, with a little adjustment to the code it's possible to simulate enough multithreading to provide a satisfactory solution to our problem.

Multithreading means you have multiple threads of control running at once, in the same address space (this differs from *multitasking*, where you have different processes each running in their own address space). The trick is that you have fewer CPUs than you do threads (and very often only one CPU) so to give the illusion that each thread has its own CPU there is a *time-slicing* mechanism that says "OK, current thread – you've had enough time. I'm going to stop you and go give time to some other thread." This automatic stopping and starting of threads is called *pre-emptive* and it means you don't need to manage the threading process at all.

An alternative approach is for each thread to voluntarily yield the CPU to the scheduler, which then goes and finds another thread that needs running. This is easier to synthesize, but it still requires a method of "swapping" out one thread and swapping in another (this usually involves saving the stack frame and using the standard C library functions **setjmp( )** and **longjmp( )**; see my article in the (XX) issue of Computer Language magazine for an example). So instead, we'll build the time-slicing into the classes in the system. In this case, it will be the tellers that represent the "threads," (the customers will be passive) so each teller will have an infinite-looping **run( )** method that will execute for a certain number of "time units," and then simply return. By using the ordinary return mechanism, we eliminate the need for any swapping. The resulting program, although small, provides a remarkably reasonable simulation:

```
//: C20:BankTeller.cpp
// Using a queue and simulated multithreading
// To model a bank teller system
#include <iostream>
#include <queue>
#include <list>
#include <cstdlib>
#include <ctime>
```

```
using namespace std;

class Customer {
  int serviceTime;
public:
  Customer() : serviceTime(0) {}
  Customer(int tm) : serviceTime(tm) {}
  int getTime() { return serviceTime; }
  void setTime(int newtime) {
    serviceTime = newtime;
  }
  friend ostream&
  operator<<(ostream& os, const Customer& c) {
    return os << '[' << c.serviceTime << ']';
  }
};

class Teller {
  queue<Customer>& customers;
  Customer current;
  static const int slice = 5;
  int ttime; // Time left in slice
  bool busy; // Is teller serving a customer?
public:
  Teller(queue<Customer>& cq)
    : customers(cq), ttime(0), busy(false) {}
  Teller& operator=(const Teller& rv) {
    customers = rv.customers;
    current = rv.current;
    ttime = rv.ttime;
    busy = rv.busy;
    return *this;
  }
  bool isBusy() { return busy; }
  void run(bool recursion = false) {
    if(!recursion)
      ttime = slice;
    int servtime = current.getTime();
    if(servtime > ttime) {
      servtime -= ttime;
      current.setTime(servtime);
      busy = true; // Still working on current
```

```
      return;
    }
    if(servtime < ttime) {
      ttime -= servtime;
      if(!customers.empty()) {
        current = customers.front();
        customers.pop(); // Remove it
        busy = true;
        run(true); // Recurse
      }
      return;
    }
    if(servtime == ttime) {
      // Done with current, set to empty:
      current = Customer(0);
      busy = false;
      return; // No more time in this slice
    }
  }
};

// Inherit to access protected implementation:
class CustomerQ : public queue<Customer> {
public:
  friend ostream&
  operator<<(ostream& os, const CustomerQ& cd) {
    copy(cd.c.begin(), cd.c.end(),
      ostream_iterator<Customer>(os, ""));
    return os;
  }
};

int main() {
  CustomerQ customers;
  list<Teller> tellers;
  typedef list<Teller>::iterator TellIt;
  tellers.push_back(Teller(customers));
  srand(time(0)); // Seed random number generator
  while(true) {
    // Add a random number of customers to the
    // queue, with random service times:
    for(int i = 0; i < rand() % 5; i++)
```

```
      customers.push(Customer(rand() % 15 + 1));
    cout << '{' << tellers.size() << '}'
      << customers << endl;
    // Have the tellers service the queue:
    for(TellIt i = tellers.begin();
      i != tellers.end(); i++)
      (*i).run();
    cout << '{' << tellers.size() << '}'
      << customers << endl;
    // If line is too long, add another teller:
    if(customers.size() / tellers.size() > 2)
      tellers.push_back(Teller(customers));
    // If line is short enough, remove a teller:
    if(tellers.size() > 1 &&
      customers.size() / tellers.size() < 2)
      for(TellIt i = tellers.begin();
        i != tellers.end(); i++)
        if(!(*i).isBusy()) {
          tellers.erase(i);
          break; // Out of for loop
        }
  }
} ///:~
```

Each customer requires a certain amount of service time, which is the number of time units that a teller must spend on the customer in order to serve that customer's needs. Of course, the amount of service time will be different for each customer, and will be determined randomly. In addition, you won't know how many customers will be arriving in each interval, so this will also be determined randomly.

The **Customer** objects are kept in a **queue<Customer>**, and each **Teller** object keeps a reference to that queue. When a **Teller** object is finished with its current **Customer** object, that **Teller** will get another **Customer** from the queue and begin working on the new **Customer**, reducing the **Customer**'s service time during each time slice that the **Teller** is allotted. All this logic is in the **run( )** member function, which is basically a three-way **if** statement based on whether the amount of time necessary to serve the customer is less than, greater than or equal to the amount of time left in the teller's current time slice. Notice that if the **Teller** has more time after finishing with a **Customer**, it gets a new customer and recurses into itself.

Just as with a **stack**, when you use a **queue**, it's only a **queue** and doesn't have any of the other functionality of the basic sequence containers. This includes the ability to get an iterator in order to step through the **stack**. However, the underlying sequence container (that the **queue** is built upon) is held as a **protected** member inside the **queue**, and the identifier for this member is specified in the C++ Standard as '**c**', which means that you can inherit from **queue** in order to access the underlying implementation. The **CustomerQ** class does exactly that, for the sole purpose of defining an **ostream operator<<** that can iterate through the **queue** and print out its members.

The driver for the simulation is the infinite **while** loop in **main( )**. At the beginning of each pass through the loop, a random number of customers are added, with random service times. Both the number of tellers and the queue contents are displayed so you can see the state of the system. After running each teller, the display is repeated. At this point, the system adapts by comparing the number of customers and the number of tellers; if the line is too long another teller is added and if it is short enough a teller can be removed. It is in this adaptation section of the program that you can experiment with policies regarding the optimal addition and removal of tellers. If this is the only section that you're modifying, you may want to encapsulate policies inside of different objects.

Priority queues

When you **push( )** an object onto a **priority_queue**, that object is sorted into the queue according to a function or function object (you can allow the default **less** template to supply this, or provide one of your own). The **priority_queue** ensures that when you look at the **top( )** element it will be the one with the highest priority. When you're done with it, you call **pop( )** to remove it and bring the next one into place. Thus, the **priority_queue** has nearly the same interface as a **stack**, but it behaves differently.

Like **stack** and **queue**, **priority_queue** is an adapter which is built on top of one of the basic sequences – the default is **vector**.

It's trivial to make a **priority_queue** that works with **int**s:

```
//: C20:PriorityQueue1.cpp
#include <iostream>
#include <queue>
#include <cstdlib>
```

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

int main() {
  priority_queue<int> pqi;
  srand(time(0)); // Seed random number generator
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

This pushes into the **priority_queue** 100 random values from 0 to 24. When you run this program you'll see that duplicates are allowed, and the highest values appear first. To show how you can change the ordering by providing your own function or function object, the following program gives lower-valued numbers the highest priority:

```
//: C20:PriorityQueue2.cpp
// Changing the priority
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;

struct Reverse {
  bool operator()(int x, int y) {
    return y < x;
  }
};

int main() {
  priority_queue<int, vector<int>, Reverse> pqi;
  // Could also say:
  // priority_queue<int, vector<int>,
  //   greater<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  while(!pqi.empty()) {
```

```
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

Although you can easily use the Standard Library **greater** template to produce the predicate, I went to the trouble of creating **Reverse** so you could see how to do it in case you have a more complex scheme for ordering your objects.

If you look at the description for **priority_queue**, you see that the constructor can be handed a "Compare" object, as shown above. If you don't use your own "Compare" object, the default template behavior the **less** template function. You might think (as I did) that it would make sense to leave the template instantiation as **priority_queue<int>**, thus using the default template arguments of **vector<int>** and **less<int>**. Then you could inherit a new class from **less<int>**, redefine **operator( )** and hand an object of that type to the **priority_queue** constructor. I tried this, and got it to compile, but the resulting program produced the same old **less<int>** behavior. The answer lies in the **less< >** template:

```
template <class T>
struct less : binary_function<T, T, bool> {
  // Other stuff...
  bool operator()(const T& x, const T& y) const {
    return x < y;
  }
};
```

The **operator( )** is not **virtual**, so even though the constructor takes your subclass of **less<int>** by reference (thus it doesn't slice it down to a plain **less<int>**), when **operator( )** is called, it is the base-class version that is used. While it is generally reasonable to expect ordinary classes to behave polymorphically, you cannot make this assumption when using the STL.

Of course, a **priority_queue** of **int** is trivial. A more interesting problem is a to-do list, where each object contains a **string** and a primary and secondary priority value:

```
//: C20:PriorityQueue3.cpp
// A more complex use of priority_queue
#include <iostream>
#include <queue>
#include <string>
```

```
using namespace std;

class ToDoItem {
  char primary;
  int secondary;
  string item;
public:
  ToDoItem(string td, char pri ='A', int sec =1)
    : item(td), primary(pri), secondary(sec) {}
  friend bool operator<(
    const ToDoItem& x, const ToDoItem& y) {
    if(x.primary > y.primary)
      return true;
    if(x.primary == y.primary)
      if(x.secondary > y.secondary)
        return true;
    return false;
  }
  friend ostream&
  operator<<(ostream& os, const ToDoItem& td) {
    return os << td.primary << td.secondary
      << ": " << td.item;
  }
};

int main() {
  priority_queue<ToDoItem> toDoList;
  toDoList.push(ToDoItem("Empty trash", 'C', 4));
  toDoList.push(ToDoItem("Feed dog", 'A', 2));
  toDoList.push(ToDoItem("Feed bird", 'B', 7));
  toDoList.push(ToDoItem("Mow lawn", 'C', 3));
  toDoList.push(ToDoItem("Water lawn", 'A', 1));
  toDoList.push(ToDoItem("Feed cat", 'B', 1));
  while(!toDoList.empty()) {
    cout << toDoList.top() << endl;
    toDoList.pop();
  }
} ///:~
```

ToDoItem's **operator<** must be a non-member function for it to work with **less< >**. Other than that, everything happens automatically. The output is:

```
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
```

Note that you cannot iterate through a **priority_queue**. However, it is possible to emulate the behavior of a **priority_queue** using a **vector**, thus allowing you access to that **vector**. You can do this by looking at the implementation of **priority_queue**, which uses **make_heap( )**, **push_heap( )** and **pop_heap( )** (they are the soul of the **priority_queue**; in fact you could say that the heap *is* the priority queue and **priority_queue** is just a wrapper around it). This turns out to be reasonably straightforward, but you might think that a shortcut is possible. Since the container used by **priority_queue** is **protected** (and has the identifier, according to the Standard C++ specification, named **c**) you can inherit a new class which provides access to the underlying implementation:

```
//: C20:PriorityQueue4.cpp
// Manipulating the underlying implementation
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;

class PQI : public priority_queue<int> {
public:
  vector<int>& impl() { return c; }
};

int main() {
  PQI pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.impl().begin(), pqi.impl().end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
```

```
    pqi.pop();
  }
} ///:~
```

However, if you run this program you'll discover that the **vector** doesn't contain the items in the descending order that you get when you call **pop( )**, the order that you want from the priority queue. It would seem that if you want to create a **vector** that is a priority queue, you have to do it by hand, like this:

```
//: C20:PriorityQueue5.cpp
// Building your own priority queue
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(begin(), end(), comp);
  }
  const T& top() { return front(); }
  void push(const T& x) {
    push_back(x);
    push_heap(begin(), end(), comp);
  }
  void pop() {
    pop_heap(begin(), end(), comp);
    pop_back();
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  copy(pqi.begin(), pqi.end(),
    ostream_iterator<int>(cout, " "));
```

```
  cout << endl;
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

But this program behaves in the same way as the previous one! What you are seeing in the underlying **vector** is called a *heap*. This heap represents the tree of the priority queue (stored in the linear structure of the **vector**), but when you iterate through it you do not get a linear priority-queue order. You might think that you can simply call **sort_heap( )**, but that only works once, and then you don't have a heap anymore, but instead a sorted list. This means that to go back to using it as a heap the user must remember to call **make_heap( )** first. This can be encapsulated into your custom priority queue:

```
//: C20:PriorityQueue6.cpp
#include <iostream>
#include <queue>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

template<class T, class Compare>
class PQV : public vector<T> {
  Compare comp;
  bool sorted;
  void assureHeap() {
    if(sorted) {
      // Turn it back into a heap:
      make_heap(begin(), end(), comp);
      sorted = false;
    }
  }
public:
  PQV(Compare cmp = Compare()) : comp(cmp) {
    make_heap(begin(), end(), comp);
    sorted = false;
  }
  const T& top() {
    assureHeap();
```

```
    return front();
  }
  void push(const T& x) {
    assureHeap();
    // Put it at the end:
    push_back(x);
    // Re-adjust the heap:
    push_heap(begin(), end(), comp);
  }
  void pop() {
    assureHeap();
    // Move the top element to the last position:
    pop_heap(begin(), end(), comp);
    // Remove that element:
    pop_back();
  }
  void sort() {
    if(!sorted) {
      sort_heap(begin(), end(), comp);
      reverse(begin(), end());
      sorted = true;
    }
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++) {
    pqi.push(rand() % 25);
    copy(pqi.begin(), pqi.end(),
      ostream_iterator<int>(cout, " "));
    cout << "\n-----\n";
  }
  pqi.sort();
  copy(pqi.begin(), pqi.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n-----\n";
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
```

```
} ///:~
```

If **sorted** is true, then the **vector** is not organized as a heap, but instead as a sorted sequence. **assureHeap( )** guarantees that it's put back into heap form before performing any heap operations on it.

The first **for** loop in **main( )** now has the additional quality that it displays the heap as it's being built.

The only drawback to this solution is that the user must remember to call **sort( )** before viewing it as a sorted sequence (although one could conceivably override all the methods that produce iterators so that they guarantee sorting). Another solution is to build a priority queue that is not a **vector**, but will build you a **vector** whenever you want one:

```
//: C20:PriorityQueue7.cpp
// A priority queue that will hand you a vector
#include <iostream>
#include <queue>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

template<class T, class Compare>
class PQV {
  vector<T> v;
  Compare comp;
public:
  // Don't need to call make_heap(); it's empty:
  PQV(Compare cmp = Compare()) : comp(cmp) {}
  void push(const T& x) {
    // Put it at the end:
    v.push_back(x);
    // Re-adjust the heap:
    push_heap(v.begin(), v.end(), comp);
  }
  void pop() {
    // Move the top element to the last position:
    pop_heap(v.begin(), v.end(), comp);
    // Remove that element:
    v.pop_back();
  }
  const T& top() { return v.front(); }
```

```
  bool empty() const { return v.empty(); }
  int size() const { return v.size(); }
  typedef vector<T> TVec;
  TVec vector() {
    TVec r(v.begin(), v.end());
    // It's already a heap
    sort_heap(r.begin(), r.end(), comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int, less<int> > pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.vector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n----------\n";
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

PQV follows the same form as the STL's **priority_queue**, but has the additional member **vector( )**, which creates a new **vector** that's a copy of the one in **PQV** (which means that it's already a heap), then sorts it (thus it leave's **PQV**'s **vector** untouched), then reverses the order so that traversing the new **vector** produces the same effect as popping the elements from the priority queue.

You may observe that the approach of inheriting from **priority_queue** used in **PriorityQueue4.cpp** could be used with the above technique to produce more succinct code:

```
//: C20:PriorityQueue8.cpp
// A more compact version of PriorityQueue7.cpp
#include <iostream>
#include <queue>
#include <algorithm>
```

```
#include <cstdlib>
#include <ctime>
using namespace std;

template<class T>
class PQV : public priority_queue<T> {
public:
  typedef vector<T> TVec;
  TVec vector() {
    TVec r(c.begin(), c.end());
    // c is already a heap
    sort_heap(r.begin(), r.end(), comp);
    // Put it into priority-queue order:
    reverse(r.begin(), r.end());
    return r;
  }
};

int main() {
  PQV<int> pqi;
  srand(time(0));
  for(int i = 0; i < 100; i++)
    pqi.push(rand() % 25);
  const vector<int>& v = pqi.vector();
  copy(v.begin(), v.end(),
    ostream_iterator<int>(cout, " "));
  cout << "\n----------\n";
  while(!pqi.empty()) {
    cout << pqi.top() << ' ';
    pqi.pop();
  }
} ///:~
```

The brevity of this solution makes it the simplest and most desirable, plus it's guaranteed that the user will not have a **vector** in the unsorted state. The only potential problem is that the **vector( )** member function returns the **vector<T>** by value, which might cause some overhead issues with complex values of the parameter type **T**.

Holding bits

Most of my computer education was in hardware-level design and programming, and I spent my first few years doing embedded systems development. Because C was a language that purported to be "close to the hardware," I have always found it dismaying that there was no native binary representation for numbers. Decimal, of course, and hexadecimal (tolerable only because it's easier to group the bits in your mind), but octal? Ugh. Whenever you read specs for chips you're trying to program, they don't describe the chip registers in octal, or even hexadecimal – they use binary. And yet C won't let you say **0b0101101**, which is the obvious solution for a language close to the hardware.

Although there's still no native binary representation in C++, things have improved with the addition of two classes: **bitset** and **vector<bool>**, both of which are designed to manipulate a group of on-off values. The primary differences between these types are:

1. The **bitset** holds a fixed number of bits. You establish the quantity of bits in the **bitset** template argument. The **vector<bool>** can, like a regular **vector**, expand dynamically to hold any number of **bool** values.
2. The **bitset** is explicitly designed for performance when manipulating bits, and not as a "regular" container. As such, it has no iterators and it's most storage-efficient when it contains an integral number of **long** values. The **vector<bool>**, on the other hand, is a specialization of a **vector**, and so has all the operations of a normal **vector** – the specialization is just designed to be space-efficient for **bool**.

There is no trivial conversion between a **bitset** and a **vector<bool>**, which implies that the two are for very different purposes.

bitset<n>

The template for **bitset** accepts an integral template argument which is the number of bits to represent. Thus, **bitset<10>** is a different type than **bitset<20>**, and you cannot perform comparisons, assignments, etc. between the two.

A **bitset** provides virtually any bit operation that you could ask for, in a very efficient form. However, each **bitset** is made up of an integral number of **long**s (typically 32 bits), so even though it uses no more space than it needs, it always uses at least the size of a **long**. This means you'll

use space most efficiently if you increase the size of your **bitset**s in chunks of the number of bits in a **long**. In addition, the only conversion *from* a **bitset** to a numerical value is to an **unsigned long**, which means that 32 bits (if your **long** is the typical size) is the most flexible form of a **bitset**.

The following example tests almost all the functionality of the **bitset** (the missing operations are redundant or trivial). You'll see the description of each of the bitset outputs to the right of the output so that the bits all line up and you can compare them to the source values. If you still don't understand bitwise operations, running this program should help.

```
//: C20:BitSet.cpp
// Exercising the bitset class
#include <iostream>
#include <bitset>
#include <cstdlib>
#include <ctime>
#include <climits>
#include <string>
using namespace std;
const int sz = 32;
typedef bitset<sz> BS;

template<int bits>
bitset<bits> randBitset() {
  bitset<bits> r(rand());
  for(int i = 0; i < bits/16 - 1; i++) {
    r <<= 16;
    // "OR" together with a new lower 16 bits:
    r |= bitset<bits>(rand());
  }
  return r;
}

int main() {
  srand(time(0));
  cout << "sizeof(bitset<16>) = "
    << sizeof(bitset<16>) << endl;
  cout << "sizeof(bitset<32>) = "
    << sizeof(bitset<32>) << endl;
  cout << "sizeof(bitset<48>) = "
    << sizeof(bitset<48>) << endl;
```

```
  cout << "sizeof(bitset<64>) = "
    << sizeof(bitset<64>) << endl;
  cout << "sizeof(bitset<65>) = "
    << sizeof(bitset<65>) << endl;
  BS a(randBitset<sz>()), b(randBitset<sz>());
  // Converting from a bitset:
  unsigned long ul = a.to_ulong();
  string s = b.to_string();
  // Converting a string to a bitset:
  char* cbits = "111011010110111";
  cout << "char* cbits = " << cbits <<endl;
  cout << BS(cbits) << " [BS(cbits)]" << endl;
  cout << BS(cbits, 2)
    << " [BS(cbits, 2)]" << endl;
  cout << BS(cbits, 2, 11)
    << " [BS(cbits, 2, 11)]" << endl;
  cout << a << " [a]" << endl;
  cout << b << " [b]"<< endl;
  // Bitwise AND:
  cout << (a & b) << " [a & b]" << endl;
  cout << (BS(a) &= b) << " [a &= b]" << endl;
  // Bitwise OR:
  cout << (a | b) << " [a | b]" << endl;
  cout << (BS(a) |= b) << " [a |= b]" << endl;
  // Exclusive OR:
  cout << (a ^ b) << " [a ^ b]" << endl;
  cout << (BS(a) ^= b) << " [a ^= b]" << endl;
  cout << a << " [a]" << endl; // For reference
  // Logical left shift (fill with zeros):
  cout << (BS(a) <<= sz/2)
    << " [a <<= (sz/2)]" << endl;
  cout << (a << sz/2) << endl;
  cout << a << " [a]" << endl; // For reference
  // Logical right shift (fill with zeros):
  cout << (BS(a) >>= sz/2)
    << " [a >>= (sz/2)]" << endl;
  cout << (a >> sz/2) << endl;
  cout << a << " [a]" << endl; // For reference
  cout << BS(a).set() << " [a.set()]" << endl;
  for(int i = 0; i < sz; i++)
    if(!a.test(i)) {
      cout << BS(a).set(i)
```

```
      << " [a.set(" << i <<")]" << endl;
    break; // Just do one example of this
  }
cout << BS(a).reset() << " [a.reset()]"<< endl;
for(int j = 0; j < sz; j++)
  if(a.test(j)) {
    cout << BS(a).reset(j)
      << " [a.reset(" << j <<")]" << endl;
    break; // Just do one example of this
  }
cout << BS(a).flip() << " [a.flip()]" << endl;
cout << ~a << " [~a]" << endl;
cout << a << " [a]" << endl; // For reference
cout << BS(a).flip(1) << " [a.flip(1)]"<< endl;
BS c;
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
  << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
  << (c.none() ? "true" : "false") << endl;
c[1].flip(); c[2].flip();
cout << c << " [c]" << endl;
cout << "c.count() = " << c.count() << endl;
cout << "c.any() = "
  << (c.any() ? "true" : "false") << endl;
cout << "c.none() = "
  << (c.none() ? "true" : "false") << endl;
// Array indexing operations:
c.reset();
for(int k = 0; k < c.size(); k++)
  if(k % 2 == 0)
    c[k].flip();
cout << c << " [c]" << endl;
c.reset();
// Assignment to bool:
for(int ii = 0; ii < c.size(); ii++)
  c[ii] = (rand() % 100) < 25;
cout << c << " [c]" << endl;
// bool test:
if(c[1] == true)
  cout << "c[1] == true";
```

```
  else
    cout << "c[1] == false" << endl;
} ///:~
```

To generate interesting random **bitset**s, the **randBitset( )** function is created. The Standard C **rand( )** function only generates an **int**, so this function demonstrates **operator<<=** by shifting each 16 random bits to the left until the **bitset** (which is templatized in this function for size) is full. The generated number and each new 16 bits is combined using the **operator|=**.

The first thing demonstrated in **main( )** is the unit size of a **bitset**. If it is less than 32 bits, **sizeof** produces 4 (4 bytes = 32 bits), which is the size of a single **long** on most implementations. If it's between 32 and 64, it requires two **long**s, greater than 64 requires 3 **long**s, etc. Thus you make the best use of space if you use a bit quantity that fits in an integral number of **long**s. However, notice there's no extra overhead for the object – it's as if you were hand-coding to use a **long**.

Another clue that **bitset** is optimized for **long**s is that there is a **to_ulong( )** member function that produces the value of the bitset as an **unsigned long**. There are no other numerical conversions from **bitset**, but there is a **to_string( )** conversion that produces a **string** containing ones and zeros, and this can be as long as the actual **bitset**. However, using **bitset<32>** may make your life simpler because of **to_ulong( )**.

There's still no primitive format for binary values, but the next best thing is supported by **bitset**: a **string** of ones and zeros with the least-significant bit (lsb) on the right. The three constructors demonstrated show taking the entire **string** (the **char** array is automatically converted to a **string**), the **string** starting at character 2, and the string from character 2 through 11. You can write to an **ostream** from a **bitset** using **operator<<** and it comes out as ones and zeros. You can also read from an **istream** using **operator>>** (not shown here).

You'll notice that **bitset** only has three non-member operators: *and* (**&**), *or* (**|**) and *exclusive-or* (**^**). Each of these create a new **bitset** as their return value. All of the member operators opt for the more efficient **&=**, **|=**, etc. form where a temporary is not created. However, these forms actually change their lvalue (which is **a** in most of the tests in the above example). To prevent this, I created a temporary to be used as the lvalue by invoking the copy-constructor on **a**; this is why you see the form **BS(a)**. The result of each test is printed out, and occasionally **a** is reprinted so you can easily look at it for reference.

The rest of the example should be self-explanatory when you run it; if not you can find the details in your compiler's documentation or the other documentation mentioned earlier in this chapter.

vector<bool>

vector<bool> is a specialization of the **vector** template. A normal **bool** variable requires at least one byte, but since a **bool** only has two states the ideal implementation of **vector<bool>** is such that each **bool** value only requires one bit. This means the iterator must be specially-defined, and cannot be a **bool***.

The bit-manipulation functions for **vector<bool>** are much more limited than those of **bitset**. The only member function that was added to those already in **vector** is **flip( )**, to invert all the bits; there is no **set( )** or **reset( )** as in **bitset**. When you use **operator[ ]**, you get back an object of type **vector<bool>::reference**, which also has a **flip( )** to invert that individual bit.

```
//: C20:VectorOfBool.cpp
// Demonstrate the vector<bool> specialization
#include <iostream>
#include <sstream>
#include <vector>
#include <bitset>
#include <iterator>
using namespace std;

int main() {
  vector<bool> vb(10, true);
  vector<bool>::iterator it;
  for(it = vb.begin(); it != vb.end(); it++)
    cout << *it;
  cout << endl;
  vb.push_back(false);
  ostream_iterator<bool> out(cout, "");
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  bool ab[] = { true, false, false, true, true,
    true, true, false, false, true };
  // There's a similar constructor:
  vb.assign(ab, ab + sizeof(ab)/sizeof(bool));
  copy(vb.begin(), vb.end(), out);
```

```
  cout << endl;
  vb.flip(); // Flip all bits
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  for(int i = 0; i < vb.size(); i++)
    vb[i] = 0; // (Equivalent to "false")
  vb[4] = true;
  vb[5] = 1;
  vb[7].flip(); // Invert one bit
  copy(vb.begin(), vb.end(), out);
  cout << endl;
  // Convert to a bitset:
  ostringstream os;
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(os, ""));
  bitset<10> bs(os.str());
  cout << "Bitset:\n" << bs << endl;
} ///:~
```

The last part of this example takes a **vector<bool>** and converts it to a **bitset** by first turning it into a **string** of ones and zeros. Of course, you must know the size of the **bitset** at compile-time. You can see that this conversion is not the kind of operation you'll want to do on a regular basis.

Associative containers

The **set**, **map**, **multiset** and **multimap** are called *associative containers* because they associate *keys* with *values*. Well, at least **map**s and **multimap**s associate keys to values, but you can look at a **set** as a **map** that has no values, only keys (and they can in fact be implemented this way), and the same for the relationship between **multiset** and **multimap**. So, because of the structural similarity **set**s and **multiset**s are lumped in with associative containers.

The most important basic operations with associative containers are putting things in, and in the case of a **set**, seeing if something is in the set. In the case of a **map**, you want to first see if a key is in the **map**, and if it exists you want the associated value for that key to be returned. Of course, there are many variations on this theme but that's the fundamental concept. The following example shows these basics:

```
//: C20:AssociativeBasics.cpp
```

```
// Basic operations with sets and maps
#include "Noisy.h"
#include <iostream>
#include <set>
#include <map>
using namespace std;

int main() {
  Noisy na[] = { Noisy(), Noisy(), Noisy(),
    Noisy(), Noisy(), Noisy(), Noisy() };
  // Add elements via constructor:
  set<Noisy> ns(na, na+ sizeof na/sizeof(Noisy));
  // Ordinary insertion:
  Noisy n;
  ns.insert(n);
  cout << endl;
  // Check for set membership:
  cout << "ns.count(n)= " << ns.count(n) << endl;
  if(ns.find(n) != ns.end())
    cout << "n(" << n << ") found in ns" << endl;
  // Print elements:
  copy(ns.begin(), ns.end(),
    ostream_iterator<Noisy>(cout, " "));
  cout << endl;
  cout << "\n-----------\n";
  map<int, Noisy> nm;
  for(int i = 0; i < 10; i++)
    nm[i]; // Automatically makes pairs
  cout << "\n-----------\n";
  for(int j = 0; j < nm.size(); j++)
    cout << "nm[" << j <<"] = " << nm[j] << endl;
  cout << "\n-----------\n";
  nm[10] = n;
  cout << "\n-----------\n";
  nm.insert(make_pair(47, n));
  cout << "\n-----------\n";
  cout << "\n nm.count(10)= "
    << nm.count(10) << endl;
  cout << "nm.count(11)= "
    << nm.count(11) << endl;
  map<int, Noisy>::iterator it = nm.find(6);
  if(it != nm.end())
```

```
    cout << "value:" << (*it).second
      << " found in nm at location 6" << endl;
  for(it = nm.begin(); it != nm.end(); it++)
    cout << (*it).first << ":"
      << (*it).second << ", ";
  cout << "\n----------\n";
} ///:~
```

The **set<Noisy>** object **ns** is created using two iterators into an array of **Noisy** objects, but there is also a default constructor and a copy-constructor, and you can pass in an object that provides an alternate scheme for doing comparisons. Both **set**s and **map**s have an **insert( )** member function to put things in, and there are a couple of different ways to check to see if an object is already in an associative container: **count( )**, when given a key, will tell you how many times that key occurs (this can only be zero or one in a **set** or **map**, but it can be more than one with a **multiset** or **multimap**). The **find( )** member function will produce an iterator indicating the first occurrence (with **set** and **map**, the *only* occurrence) of the key that you give it, or the past-the-end iterator if it can't find the key. The **count( )** and **find( )** member functions exist for all the associative containers, which makes sense. The associative containers also have member functions **lower_bound( )**, **upper-_bound( )** and **equal_range( )**, which actually only make sense for **multiset** and **multimap**, as you shall see (but don't try to figure out how they would be useful for **set** and **map**, since they are designed for dealing with a range of duplicate keys, which those containers don't allow).

Designing an **operator[ ]** always produces a little bit of a dilemma because it's intended to be treated as an array-indexing operation, so people don't tend to think about performing a test before they use it. But what happens if you decide to index out of the bounds of the array? One option, of course, is to throw an exception, but with a **map** "indexing out of the array" could mean that you want an entry there, and that's the way the STL **map** treats it. The first **for** loop after the creation of the **map<int, Noisy> nm** just "looks up" objects using the **operator[ ]**, but this is actually creating new **Noisy** objects! The **map** creates a new key-value pair (using the default constructor for the value) if you look up a value with **operator[ ]** and it isn't there. This means that if you really just want to look something up and not create a new entry, you must use **count( )** (to see if it's there) or **find( )** (to get an iterator to it).

The **for** loop that prints out the values of the container using **operator[ ]** has a number of problems. First, it requires integral keys (which we happen to have in this case). Next and worse, if all the keys are not

sequential, you'll end up counting from 0 to the size of the container, and if there are some spots which don't have key-value pairs you'll automatically create them, and miss some of the higher values of the keys. Finally, if you look at the output from the **for** loop you'll see that things are *very* busy, and it's quite puzzling at first why there are so many constructions and destructions for what appears to be a simple lookup. The answer only becomes clear when you look at the code in the **map** template for **operator[ ]**, which will be something like this:

```
mapped_type& operator[] (const key_type& k) {
  value_type tmp(k,T());
  return (*((insert(tmp)).first)).second;
}
```

Following the trail, you'll find that **map::value_type** is:

```
typedef pair<const Key, T> value_type;
```

Now you need to know what a **pair** is, which can be found in **<utility>**:

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair();
  pair(const T1& x, const T2& y)
    : first(x), second(y) {}
  // Templatized copy-constructor:
  template<class U, class V>
   pair(const pair<U, V> &p);
};
```

It turns out this is a very important (albeit simple) **struct** which is used quite a bit in the STL. All it really does it package together two objects, but it's very useful, especially when you want to return two objects from a function (since a **return** statement only takes one object). There's even a shorthand for creating a pair called **make_pair( )**, which is used in **AssociativeBasics.cpp**.

So to retrace the steps, **map::value_type** is a **pair** of the key and the value of the map – actually, it's a single entry for the map. But notice that **pair** packages its objects by value, which means that copy-constructions are necessary to get the objects into the **pair**. Thus, the creation of **tmp** in **map::operator[ ]** will involve at least a copy-constructor call and

destructor call for each object in the **pair**. Here, we're getting off easy because the key is an **int**. But if you want to really see what kind of activity can result from **map::operator[ ]**, try running this:

```
//: C20:NoisyMap.cpp
// Mapping Noisy to Noisy
#include "Noisy.h"
#include <map>
using namespace std;

int main() {
  map<Noisy, Noisy> mnn;
  Noisy n1, n2;
  cout << "\n--------\n";
  mnn[n1] = n2;
  cout << "\n--------\n";
  cout << mnn[n1] << endl;
  cout << "\n--------\n";
} ///:~
```

You'll see that both the insertion and lookup generate a lot of extra objects, and that's because of the creation of the **tmp** object. If you look back up at **map::operator[ ]** you'll see that the second line calls **insert( )** passing it **tmp** – that is, **operator[ ]** does an insertion every time. The return value of **insert( )** is a different kind of **pair**, where **first** is an iterator pointing to the key-value **pair** that was just inserted, and **second** is a **bool** indicating whether the insertion took place. You can see that **operator[ ]** grabs **first** (the iterator), dereferences it to produce the **pair**, and then returns the **second** which is the value at that location.

So on the upside, **map** has this fancy "make a new entry if one isn't there" behavior, but the downside is that you *always* get a lot of extra object creations and destructions when you use **map::operator[ ]**. Fortunately, **AssociativeBasics.cpp** also demonstrates how to reduce the overhead of insertions and deletions, by not using **operator[ ]** if you don't have to. The **insert( )** member function is slightly more efficient than **operator[ ]**. With a **set** you only hold one object, but with a **map** you hold key-value pairs, so **insert( )** requires a **pair** as its argument. Here's where **make_pair( )** comes in handy, as you can see.

For looking objects up in a **map**, you can use **count( )** to see whether a key is in the map, or you can use **find( )** to produce an iterator pointing directly at the key-value pair. Again, since the **map** contains **pair**s that's what the iterator produces when you dereference it, so you have to select

first and **second**. When you run **AssociativeBasics.cpp** you'll notice that the iterator approach involves no extra object creations or destructions at all. It's not as easy to write or read, though.

If you use a **map** with large, complex objects and discover there's too much overhead when doing lookups and insertions (don't assume this from the beginning – take the easy approach first and use a profiler to discover bottlenecks), then you can use the counted-handle approach shown in Chapter XX so that you are only passing around small, lightweight objects.

Of course, you can also iterate through a **set** or **map** and operate on each of its objects. This will be demonstrated in later examples.

Generators and fillers for associative containers

You've seen how useful the **fill( )**, **fill_n( )**, **generate( )** and **generate_n( )** function templates in **<algorithm>** have been for filling the sequential containers (**vector**, **list** and **deque**) with data. However, these are implemented by using **operator=** to assign values into the sequential containers, and the way that you add objects to associative containers is with their respective **insert( )** member functions. Thus the default "assignment" behavior causes a problem when trying to use the "fill" and "generate" functions with associative containers.

One solution is to duplicate the "fill" and "generate" functions, creating new ones that can be used with associative containers. It turns out that only the **fill_n( )** and **generate_n( )** functions can be duplicated (**fill( )** and **generate( )** copy in between two iterators, which doesn't make sense with associative containers), but the job is fairly easy, since you have the **<algorithm>** header file to work from (and since it contains templates, all the source code is there):

```
//: C20:assocGen.h
// The fill_n() and generate_n() equivalents
// for associative containers.
#ifndef ASSOCGEN_H
#define ASSOCGEN_H

template<class Assoc, class Count, class T>
void
assocFill_n(Assoc& a, Count n, const T& val) {
```

```
  for (; 0 < n; --n)
    a.insert(val);
}

template<class Assoc, class Count, class Gen>
void assocGen_n(Assoc& a, Count n, Gen g) {
  for (; 0 < n; --n)
    a.insert(g());
}
#endif // ASSOCGEN_H ///:~
```

You can see that instead of using iterators, the container class itself is passed (by reference, of course, since you wouldn't want to make a local copy, fill it, and then have it discarded at the end of the scope).

This code demonstrates two valuable lessons. The first lesson is that if the algorithms don't do what you want, copy the nearest thing and modify it. You have the example at hand in the STL header, so most of the work has already been done.

The second lesson is more pointed: if you look long enough, there's probably a way to do it in the STL *without* inventing anything new. The present problem can instead be solved by using an **insert_iterator** (produced by a call to **inserter( )**), which calls **insert( )** to place items in the container instead of **operator=**. This is *not* simply a variation of **front_insert_iterator** (produced by a call to **front_inserter( )**) or **back_insert_iterator** (produced by a call to **back_inserter( )**), since those iterators use **push_front( )** and **push_back( )**, respectively. Each of the insert iterators is different by virtue of the member function it uses for insertion, and **insert( )** is the one we need. Here's a demonstration that shows filling and generating both a **map** and a **set** (of course, it can also be used with **multimap** and **multiset**). First, some templatized, simple generators are created (this may seem like overkill, but you never know when you'll need them; for that reason they're placed in a header file):

```
//: C20:SimpleGenerators.h
// Generic generators, including
// one that creates pairs
#include <iostream>
#include <utility>

// A generator that increments its value:
template<typename T>
```

```
class IncrGen {
  T i;
public:
  IncrGen(T ii) : i (ii) {}
  T operator()() { return i++; }
};

// A generator that produces an STL pair<>:
template<typename T1, typename T2>
class PairGen {
  T1 i;
  T2 j;
public:
  PairGen(T1 ii, T2 jj) : i(ii), j(jj) {}
  std::pair<T1,T2> operator()() {
    return std::pair<T1,T2>(i++, j++);
  }
};

// A generic global operator<<
// for printing any STL pair<>:
template<typename Pair> std::ostream&
operator<<(std::ostream& os, const Pair& p) {
  return os << p.first << "\t"
    << p.second << std::endl;
} ///:~
```

Both generators expect that **T** can be incremented, and they simply use **operator++** to generate new values from whatever you used for initialization. **PairGen** creates an STL **pair** object as its return value, and that's what can be placed into a **map** or **multimap** using **insert( )**.

The last function is a generalization of **operator<<** for **ostream**s, so that any **pair** can be printed, assuming each element of the **pair** supports a stream **operator<<**. As you can see below, this allows the use of **copy( )** to output the **map**:

```
//: C20:AssocInserter.cpp
// Using an insert_iterator so fill_n() and
// generate_n() can be used with associative
// containers
#include "SimpleGenerators.h"
#include <iterator>
#include <iostream>
```

```
#include <algorithm>
#include <set>
#include <map>
using namespace std;

int main() {
  set<int> s;
  fill_n(inserter(s, s.begin()), 10, 47);
  generate_n(inserter(s, s.begin()), 10,
    IncrGen<int>(12));
  copy(s.begin(), s.end(),
    ostream_iterator<int>(cout, "\n"));

  map<int, int> m;
  fill_n(inserter(m, m.begin()), 10,
    make_pair(90,120));
  generate_n(inserter(m, m.begin()), 10,
    PairGen<int, int>(3, 9));
  copy(m.begin(), m.end(),
    ostream_iterator<pair<int,int> >(cout,"\n"));
} ///:~
```

The second argument to **inserter** is an interator, which actually isn't used in the case of associative containers since they maintain their order internally, rather than allowing you to tell them where the element should be inserted. However, an **insert_iterator** can be used with many different types of containers so you must provide the iterator.

Note how the **ostream_iterator** is created to output a **pair**; this wouldn't have worked if the **operator<<** hadn't been created, and since it's a template it is automatically instantiated for **pair<int, int>**.

The magic of maps

An ordinary array uses an integral value to index into a sequential set of elements of some type. A **map** is an *associative array*, which means you associate one object with another in an array-like fashion, but instead of selecting an array element with a number as you do with an ordinary array, you look it up with an object! The example which follows counts the words in a text file, so the index is the **string** object representing the word, and the value being looked up is the object that keeps count of the strings.

In a single-item container like a **vector** or **list**, there's only one thing being held. But in a **map**, you've got two things: the *key* (what you look up by, as in **mapname[key]**) and the *value* that results from the lookup with the key. If you simply want to move through the entire **map** and list each key-value pair, you use an iterator, which when dereferenced produces a **pair** object containing both the key and the value. You access the members of a **pair** by selecting **first** or **second**.

This same philosophy of packaging two items together is also used to insert elements into the map, but the **pair** is created as part of the instantiated **map** and is called **value_type**, containing the key and the value. So one option for inserting a new element is to create a **value_type** object, loading it with the appropriate objects and then calling the **insert( )** member function for the **map**. Instead, the following example makes use of the aforementioned special feature of **map**: if you're trying to find an object by passing in a key to **operator[ ]** and that object doesn't exist, **operator[ ]** will automatically insert a new key-value pair for you, using the default constructor for the value object. With that in mind, consider an implementation of a word counting program:

```
//: C20:WordCount.cpp
//{L} StreamTokenizer
// Count occurrences of words using a map
#include "StreamTokenizer.h"
#include "../require.h"
#include <string>
#include <map>
#include <iostream>
#include <fstream>
using namespace std;

class Count {
  int i;
public:
  Count() : i(0) {}
  void operator++(int) { i++; } // Post-increment
  int& val() { return i; }
};

typedef map<string, Count> WordMap;
typedef WordMap::iterator WMIter;

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

```
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  StreamTokenizer words(in);
  WordMap wordmap;
  string word;
  while((word = words.next()).size() != 0)
    wordmap[word]++;
  for(WMIter w = wordmap.begin();
      w != wordmap.end(); w++)
    cout << (*w).first << ": "
      << (*w).second.val() << endl;
} ///:~
```

The need for the **Count** class is to contain an **int** that's automatically initialized to zero. This is necessary because of the crucial line:

```
wordmap[string(word)]++;
```

This finds the word that has been produced by **StreamTokenizer** and increments the **Count** object associated with that word, which is fine as long as there *is* a key-value pair for that **string**. If there isn't, the **map** automatically inserts a key for the word you're looking up, and a **Count** object, which is initialized to zero by the default constructor. Thus, when it's incremented the **Count** becomes 1.

Printing the entire list requires traversing it with an iterator (there's no **copy( )** shortcut for a **map** unless you want to write an **operator<<** for the **pair** in the map). As previously mentioned, dereferencing this iterator produces a **pair** object, with the **first** member the key and the **second** member the value. In this case **second** is a **Count** object, so its **val( )** member must be called to produce the actual word count.

If you want to find the count for a particular word, you can use the array index operator, like this:

```
cout << "the: " << wordmap["the"].val() << endl;
```

You can see that one of the great advantages of the **map** is the clarity of the syntax; an associative array makes intuitive sense to the reader (note, however, that if "the" isn't already in the **wordmap** a new entry will be created!).

A command-line argument tool

A problem that often comes up in programming is the management of program arguments that you can specify on the command line. Usually you'd like to have a set of defaults that can be changed via the command line. The following tool expects the command line arguments to be in the form **flag1=value1** with no spaces around the '**=**' (so it will be treated as a single argument). The **ProgVal** class simply inherits from **map<string, string>**:

```
//: C20:ProgVals.h
// Program values can be changed by command line
#ifndef PROGVALS_H
#define PROGVALS_H
#include <map>
#include <iostream>
#include <string>

class ProgVals
  : public std::map<std::string, std::string> {
public:
  ProgVals(std::string defaults[][2], int sz);
  void parse(int argc, char* argv[],
    std::string usage, int offset = 1);
  void print(std::ostream& out = std::cout);
};
#endif // PROGVALS_H ///:~
```

The constructor expects an array of **string** pairs (as you'll see, this allows you to initialize it with an array of **char***) and the size of that array. The **parse( )** member function is handed the command-line arguments along with a "usage" string to print if the command line is given incorrectly, and the "offset" which tells it which command-line argument to start with (so you can have non-flag arguments at the beginning of the command line). Finally, **print( )** displays the values. Here is the implementation:

```
//: C20:ProgVals.cpp {O}
#include "ProgVals.h"
using namespace std;

ProgVals::ProgVals(
  std::string defaults[][2], int sz) {
  for(int i = 0; i < sz; i++)
    insert(make_pair(
```

```
    defaults[i][0], defaults[i][1]));
}

void ProgVals::parse(int argc, char* argv[],
  string usage, int offset) {
  // Parse and apply additional
  // command-line arguments:
  for(int i = offset; i < argc; i++) {
    string flag(argv[i]);
    int equal = flag.find('=');
    if(equal == string::npos) {
      cerr << "Command line error: " <<
        argv[i] << endl << usage << endl;
      continue; // Next argument
    }
    string name = flag.substr(0, equal);
    string value = flag.substr(equal + 1);
    if(find(name) == end()) {
      cerr << name << endl << usage << endl;
      continue; // Next argument
    }
    operator[](name) = value;
  }
}

void ProgVals::print(ostream& out) {
  out << "Program values:" << endl;
  for(iterator it = begin(); it != end(); it++)
    out << (*it).first << " = "
        << (*it).second << endl;
} ///:~
```

The constructor uses the STL **make_pair( )** helper function to convert each pair of **char*** into a **pair** object that can be inserted into the **map**. In **parse( )**, each command-line argument is checked for the existence of the telltale '**=**' sign (reporting an error if it isn't there), and then is broken into two strings, the **name** which appears before the '**=**', and the **value** which appears after. The **operator[ ]** is then used to change the existing value to the new one.

Here's an example to test the tool:

```
//: C20:ProgValTest.cpp
//{L} ProgVals
```

```
#include "ProgVals.h"
using namespace std;

string defaults[][2] = {
  { "color", "red" },
  { "size", "medium" },
  { "shape", "rectangular" },
  { "action", "hopping"},
};

const char* usage = "usage:\n"
"ProgValTest [flag1=val1 flag2=val2 ...]\n"
"(Note no space around '=')\n"
"Where the flags can be any of: \n"
"color, size, shape, action \n";

// So it can be used globally:
ProgVals pvals(defaults,
  sizeof defaults / sizeof *defaults);

class Animal {
  string color, size, shape, action;
public:
  Animal(string col, string sz,
    string shp, string act)
    :color(col),size(sz),shape(shp),action(act){}
  // Default constructor uses program default
  // values, possibly change on command line:
  Animal() : color(pvals["color"]),
    size(pvals["size"]), shape(pvals["shape"]),
    action(pvals["action"]) {}
  void print() {
    cout << "color = " << color << endl
      << "size = " << size << endl
      << "shape = " << shape << endl
      << "action = " << action << endl;
  }
  // And of course pvals can be used anywhere
  // else you'd like.
};

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

```
  // Initialize and parse command line values
  // before any code that uses pvals is called:
  pvals.parse(argc, argv, usage);
  pvals.print();
  Animal a;
  cout << "Animal a values:" << endl;
  a.print();
} ///:~
```

This program can create **Animal** objects with different characteristics, and those characteristics can be established with the command line. The default characteristics are given in the two-dimensional array of **char*** called **defaults** and, after the **usage** string you can see a global instance of **ProgVals** called **pvals** is created; this is important because it allows the rest of the code in the program to access the values.

Note that **Animal**'s default constructor uses the values in **pvals** inside its constructor initializer list. When you run the program you can try creating different animal characteristics.

Many command-line programs also use a style of beginning a flag with a hyphen, and sometimes they use single-character flags.

The STL **map** is used in numerous places throughout the rest of this book.

Multimaps and duplicate keys

A **multimap** is a **map** that can contain duplicate keys. At first this may seem like a strange idea, but it can occur surprisingly often. A phone book, for example, can have many entries with the same name.

Suppose you are monitoring wildlife, and you want to keep track of where and when each type of animal is spotted. Thus, you may see many animals of the same kind, all in different locations and at different times. So if the type of animal is the key, you'll need a **multimap**. Here's what it looks like:

```
//: C20:WildLifeMonitor.cpp
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <iostream>
#include <sstream>
#include <ctime>
```

```
using namespace std;

class DataPoint {
  int x, y; // Location coordinates
  time_t time; // Time of Sighting
public:
  DataPoint() : x(0), y(0), time(0) {}
  DataPoint(int xx, int yy, time_t tm) :
    x(xx), y(yy), time(tm) {}
  // Synthesized operator=, copy-constructor OK
  int getX() { return x; }
  int getY() { return y; }
  time_t* getTime() { return &time; }
};

string animal[] = {
  "chipmunk", "beaver", "marmot", "weasel",
  "squirrel", "ptarmigan", "bear", "eagle",
  "hawk", "vole", "deer", "otter", "hummingbird",
};
const int asz = sizeof animal/sizeof *animal;
vector<string> animals(animal, animal + asz);

// All the information is contained in a
// "Sighting," which can be sent to an ostream:
typedef pair<string, DataPoint> Sighting;

ostream&
operator<<(ostream& os, const Sighting& s) {
  return os << s.first << " sighted at x= " <<
    s.second.getX() << ", y= " << s.second.getY()
    << ", time = " << ctime(s.second.getTime());
}

// A generator for Sightings:
class SightingGen {
  vector<string>& animals;
  static const int d = 100;
public:
  SightingGen(vector<string>& an) :
    animals(an) { srand(time(0)); }
  Sighting operator()() {
```

```
    Sighting result;
    int select = rand() % animals.size();
    result.first = animals[select];
    result.second = DataPoint(
      rand() % d, rand() % d, time(0));
    return result;
  }
};

typedef multimap<string, DataPoint> DataMap;
typedef DataMap::iterator DMIter;

int main() {
  DataMap sightings;
  generate_n(
    inserter(sightings, sightings.begin()),
    50, SightingGen(animals));
  // Print everything:
  copy(sightings.begin(), sightings.end(),
    ostream_iterator<Sighting>(cout, ""));
  // Print sightings for selected animal:
  while(true) {
    cout << "select an animal or 'q' to quit: ";
    for(int i = 0; i < animals.size(); i++)
      cout <<'['<< i <<']'<< animals[i] << ' ';
    cout << endl;
    string reply;
    cin >> reply;
    if(reply.at(0) == 'q') return 0;
    istringstream r(reply);
    int i;
    r >> i; // Converts to int
    i %= animals.size();
    // Iterators in "range" denote begin, one
    // past end of matching range:
    pair<DMIter, DMIter> range =
      sightings.equal_range(animals[i]);
    copy(range.first, range.second,
      ostream_iterator<Sighting>(cout, ""));
  }
} ///:~
```

All the data about a sighting is encapsulated into the class **DataPoint**, which is simple enough that it can rely on the synthesized assignment and copy-constructor. It uses the Standard C library time functions to record the time of the sighting.

In the array of **string animal**, notice that the **char*** constructor is automatically used during initialization, which makes initializing an array of **string** quite convenient. Since it's easier to use the animal names in a **vector**, the length of the array is calculated and a **vector<string>** is initialized using the **vector(iterator, iterator)** constructor.

The key-value pairs that make up a **Sighting** are the **string** which names the type of animal, and the **DataPoint** that says where and when it was sighted. The standard **pair** template combines these two types and is typedefed to produce the **Sighting** type. Then an **ostream operator<<** is created for **Sighting**; this will allow you to iterate through a map or multimap of **Sighting**s and print it out.

SightingGen generates random sightings at random data points to use for testing. It has the usual **operator( )** necessary for a function object, but it also has a constructor to capture and store a reference to a **vector<string>**, which is where the aforementioned animal names are stored.

A **DataMap** is a **multimap** of **string**-**DataPoint** pairs, which means it stores **Sighting**s. It is filled with 50 **Sighting**s using **generate_n( )**, and printed out (notice that because there is an **operator<<** that takes a **Sighting**, an **ostream_iterator** can be created). At this point the user is asked to select the animal that they want to see all the sightings for. If you press **'q'** the program will quit, but if you select an animal number, then the **equal_range( )** member function is invoked. This returns an iterator (**DMIter**) to the beginning of the set of matching pairs, and one indicating past-the-end of the set. Since only one object can be returned from a function, **equal_range( )** makes use of **pair**. Since the **range** pair has the beginning and ending iterators of the matching set, those iterators can be used in **copy( )** to print out all the sightings for a particular type of animal.

Multisets

You've seen the **set**, which only allows one object of each value to be inserted. The **multiset** is odd by comparison since it allows more than one object of each value to be inserted. This seems to go against the whole

idea of “setness,” where you can ask “is ‘it’ in this set?” If there can be more than one of ‘it’, then what does that question mean?

With some thought, you can see that it makes no sense to have more than one object of the same value in a set if those duplicate objects are *exactly* the same (with the possible exception of counting occurrences of objects, but as seen earlier in this chapter that can be handled in an alternative, more elegant fashion). Thus each duplicate object will have something that makes it unique from the other duplicates – most likely different state information that is not used in the calculation of the value during the comparison. That is, to the comparison operation, the objects look the same but they actually contain some differing internal state.

Like any STL container that must order its elements, the **multiset** template uses the **less** template by default to determine element ordering. This uses the contained classes’ **operator<**, but you may of course substitute your own comparison function.

Consider a simple class that contains one element that is used in the comparison, and another that is not:

```
//: C20:MultiSet1.cpp
// Demonstration of multiset behavior
#include <iostream>
#include <set>
#include <algorithm>
#include <ctime>
using namespace std;

class X {
  char c; // Used in comparison
  int i; // Not used in comparison
  // Don't need default constructor and operator=
  X();
  X& operator=(const X&);
  // Usually need a copy-constructor (but the
  // synthesized version works here)
public:
  X(char cc, int ii) : c(cc), i(ii) {}
  // Notice no operator== is required
  friend bool operator<(const X& x, const X& y) {
    return x.c < y.c;
  }
  friend ostream& operator<<(ostream& os, X x) {
```

```
    return os << x.c << ":" << x.i;
  }
};

class Xgen {
  static int i;
  // Number of characters to select from:
  static const int span = 6;
public:
  Xgen() { srand(time(0)); }
  X operator()() {
    char c = 'A' + rand() % span;
    return X(c, i++);
  }
};

int Xgen::i = 0;

typedef multiset<X> Xmset;
typedef Xmset::const_iterator Xmit;

int main() {
  Xmset mset;
  // Fill it with X's:
  generate_n(inserter(mset, mset.begin()),
    25, Xgen());
  // Initialize a regular set from mset:
  set<X> unique(mset.begin(), mset.end());
  copy(unique.begin(), unique.end(),
    ostream_iterator<X>(cout, " "));
  cout << "\n----\n";
  // Iterate over the unique values:
  for(set<X>::iterator i = unique.begin();
      i != unique.end(); i++) {
    pair<Xmit, Xmit> p = mset.equal_range(*i);
    copy(p.first, p.second,
      ostream_iterator<X>(cout, " "));
    cout << endl;
  }
} ///:~
```

In **X**, all the comparisons are made with the **char c**. The comparison is performed with **operator<**, which is all that is necessary for the

multiset, since in this example the default **less** comparison object is used. The class **Xgen** is used to randomly generate **X** objects, but the comparison value is restricted to the span from **'A**' to '**E**'. In **main( )**, a **multiset<X>** is created and filled with 25 **X** objects using **Xgen**, guaranteeing that there will be duplicate keys. So that we know what the unique values are, a regular **set<X>** is created from the **multiset** (using the **iterator, iterator** constructor). These values are displayed, then each one is used to produce the **equal_range( )** in the **multiset** (**equal_range( )** has the same meaning here as it does with **multimap**: all the elements with matching keys). Each set of matching keys is then printed.

As a second example, a (possibly) more elegant version of **WordCount.cpp** can be created using **multiset**:

```
//: C20:MultiSetWordCount.cpp
//{L} StreamTokenizer
// Count occurrences of words using a multiset
#include "StreamTokenizer.h"
#include "../require.h"
#include <string>
#include <set>
#include <fstream>
#include <iterator>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  StreamTokenizer words(in);
  multiset<string> wordmset;
  string word;
  while((word = words.next()).size() != 0)
    wordmset.insert(word);
  typedef multiset<string>::iterator MSit;
  MSit it = wordmset.begin();
  while(it != wordmset.end()) {
    pair<MSit, MSit> p=wordmset.equal_range(*it);
    int count = distance(p.first, p.second);
    cout << *it << ": " << count << endl;
    it = p.second; // Move to the next word
  }
```

```
} ///:~
```

The setup in **main( )** is identical to **WordCount.cpp**, but then each word is simply inserted into the **multiset<string>**. An iterator is created and initialized to the beginning of the **multiset**; dereferencing this iterator produces the current word. **equal_range( )** produces the starting and ending iterators of the word that's currently selected, and the STL algorithm **distance( )** (which is in **<iterator>**) is used to count the number of elements in that range. Then the iterator **it** is moved forward to the end of the range, which puts it at the next word. Although if you're unfamiliar with the **multiset** this code can seem more complex, the density of it and the lack of need for supporting classes like **Count** has a lot of appeal.

In the end, is this really a "set," or should it be called something else? An alternative is the generic "bag" that has been defined in some container libraries, since a bag holds anything at all without discrimination – including duplicate objects. This is close, but it doesn't quite fit since a bag has no specification about how elements should be ordered, while a **multiset** (which requires that all duplicate elements be adjacent to each other) is even more restrictive than the concept of a set, which could use a hashing function to order its elements, in which case they would not be in sorted order. Besides, if you wanted to store a bunch of objects without any special criterions, you'd probably just use a **vector**, **deque** or **list**.

Combining STL containers

When using a thesaurus, you have a word and you want to know all the words that are similar. When you look up a word, then, you want a list of words as the result. Here, the "multi" containers (**multimap** or **multiset**) are not appropriate. The solution is to combine containers, which is easily done using the STL. Here, we need a tool that turns out to be a powerful general concept, which is a **map** of **vector**:

```
//: C20:Thesaurus.cpp
// A map of vectors
#include <map>
#include <vector>
#include <string>
#include <iostream>
```

```
#include <algorithm>
#include <ctime>
using namespace std;

typedef map<string, vector<string> > Thesaurus;
typedef pair<string, vector<string> > TEntry;
typedef Thesaurus::iterator TIter;

ostream& operator<<(ostream& os,const TEntry& t){
  os << t.first << ": ";
  copy(t.second.begin(), t.second.end(),
    ostream_iterator<string>(os, " "));
  return os;
}

// A generator for thesaurus test entries:
class ThesaurusGen {
  static const string letters;
  static int count;
public:
  int maxSize() { return letters.size(); }
  ThesaurusGen() { srand(time(0)); }
  TEntry operator()() {
    TEntry result;
    if(count >= maxSize()) count = 0;
    result.first = letters[count++];
    int entries = (rand() % 5) + 2;
    for(int i = 0; i < entries; i++) {
      int choice = rand() % maxSize();
      char cbuf[2] = { 0 };
      cbuf[0] = letters[choice];
      result.second.push_back(cbuf);
    }
    return result;
  }
};

int ThesaurusGen::count = 0;
const string ThesaurusGen::letters("ABCDEFGHIJKL"
  "MNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");

int main() {
```

```
  Thesaurus thesaurus;
  // Fill with 10 entries:
  generate_n(
    inserter(thesaurus, thesaurus.begin()),
    10, ThesaurusGen());
  // Print everything:
  copy(thesaurus.begin(), thesaurus.end(),
    ostream_iterator<TEntry>(cout, "\n"));
  // Ask for a "word" to look up:
  while(true) {
    cout << "Select a \"word\", 0 to quit: ";
    for(TIter it = thesaurus.begin();
      it != thesaurus.end(); it++)
      cout << (*it).first << ' ';
    cout << endl;
    string reply;
    cin >> reply;
    if(reply.at(0) == '0') return 0; // Quit
    if(thesaurus.find(reply) == thesaurus.end())
      continue; // Not in list, try again
    vector<string>& v = thesaurus[reply];
    copy(v.begin(), v.end(),
      ostream_iterator<string>(cout, " "));
    cout << endl;
  }
} ///:~
```

A **Thesaurus** maps a **string** (the word) to a **vector<string>** (the synonyms). A **TEntry** is a single entry in a **Thesaurus**. By creating an **ostream operator<<** for a **TEntry**, a single entry from the **Thesaurus** can easily be printed (and the whole **Thesaurus** can easily be printed with **copy( )**). The **ThesaurusGen** creates “words” (which are just single letters) and “synonyms” for those words (which are just other randomly-chosen single letters) to be used as thesaurus entries. It randomly chooses the number of synonym entries to make, but there must be at least two. All the letters are chosen by indexing into a **static string** that is part of **ThesaurusGen**.

In **main( )**, a **Thesaurus** is created, filled with 10 entries and printed using the **copy( )** algorithm. Then the user is requested to choose a “word” to look up by typing the letter of that word. The **find( )** member function is used to find whether the entry exists in the **map** (remember, you don’t want to use **operator[ ]** or it will automatically make a new

entry if it doesn't find a match!). If so, **operator[ ]** is used to fetch out the **vector<string>** which is displayed.

Because templates make the expression of powerful concepts easy, you can take this concept much further, creating a **map** of **vector**s containing **map**s, etc. For that matter, you can combine any of the STL containers this way.

Cleaning up containers of pointers

In **Stlshape.cpp**, the pointers did not clean themselves up automatically. It would be convenient to be able to do this easily, rather than writing out the code each time. Here is a function template that will clean up the pointers in any sequence container; note that it is placed in the book's root directory for easy access:

```
//: :purge.h
// Delete pointers in an STL sequence container
#ifndef PURGE_H
#define PURGE_H
#include <algorithm>

template<class Seq> void purge(Seq& c) {
  typename Seq::iterator i;
  for(i = c.begin(); i != c.end(); i++) {
    delete *i;
    *i = 0;
  }
}

// Iterator version:
template<class InpIt>
void purge(InpIt begin, InpIt end) {
  while(begin != end) {
    delete *begin;
    *begin = 0;
    begin++;
  }
}
#endif // PURGE_H ///:~
```

In the first version of **purge( )**, note that **typename** is absolutely necessary; indeed this is exactly the case that the keyword was added for: **Seq** is a template argument, and **iterator** is something that is nested within that template. So what does **Seq::iterator** refer to? The **typename** keyword specifies that it refers to a type, and not something else.

While the container version of purge must work with an STL-style container, the iterator version of **purge( )** will work with any range, including an array.

Here is **Stlshape.cpp**, modified to use the **purge( )** function:

```
//: C20:Stlshape2.cpp
// Stlshape.cpp with the purge() function
#include "../purge.h"
#include <vector>
#include <iostream>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {};
};

class Circle : public Shape {
public:
  void draw() { cout << "Circle::draw\n"; }
  ~Circle() { cout << "~Circle\n"; }
};

class Triangle : public Shape {
public:
  void draw() { cout << "Triangle::draw\n"; }
  ~Triangle() { cout << "~Triangle\n"; }
};

class Square : public Shape {
public:
  void draw() { cout << "Square::draw\n"; }
  ~Square() { cout << "~Square\n"; }
};
```

```
typedef std::vector<Shape*> Container;
typedef Container::iterator Iter;

int main() {
  Container shapes;
  shapes.push_back(new Circle);
  shapes.push_back(new Square);
  shapes.push_back(new Triangle);
  for(Iter i = shapes.begin();
      i != shapes.end(); i++)
    (*i)->draw();
  purge(shapes);
} ///:~
```

When using **purge( )**, you must be careful to consider ownership issues – if an object pointer is held in more than one container, then you must be sure not to delete it twice, and you don't want to destroy the object in the first container before the second one is finished with it. Purging the same container twice is not a problem, because **purge( )** sets the pointer to zero once it deletes that pointer, and calling **delete** for a zero pointer is a safe operation.

Creating your own containers

With the STL as a foundation, it's possible to create your own containers. Assuming you follow the same model of providing iterators, your new container will behave as if it were a built-in STL container.

Consider the "ring" data structure, which is a circular sequence container. If you reach the end, it just wraps around to the beginning. This can be implemented on top of a **list** as follows:

```
//: C20:Ring.cpp
// Making a "ring" data structure from the STL
#include <iostream>
#include <list>
#include <string>
using namespace std;

template<class T>
```

```
class Ring {
  list<T> lst;
public:
  // Declaration necessary so the following
  // 'friend' statement sees this 'iterator'
  // instead of std::iterator:
  class iterator;
  friend class iterator;
  class iterator : public std::iterator<
    std::bidirectional_iterator_tag,T,ptrdiff_t>{
    list<T>::iterator it;
    list<T>* r;
  public:
    // "typename" necessary to resolve nesting:
    iterator(list<T>& lst,
      const typename list<T>::iterator& i)
      : r(&lst), it(i) {}
    bool operator==(const iterator& x) const {
      return it == x.it;
    }
    bool operator!=(const iterator& x) const {
      return !(*this == x);
    }
    list<T>::reference operator*() const {
      return *it;
    }
    iterator& operator++() {
      ++it;
      if(it == r->end())
        it = r->begin();
      return *this;
    }
    iterator operator++(int) {
      iterator tmp = *this;
      ++*this;
      return tmp;
    }
    iterator& operator--() {
      if(it == r->begin())
        it = r->end();
      --it;
      return *this;
```

```
    }
    iterator operator--(int) {
      iterator tmp = *this;
      --*this;
      return tmp;
    }
    iterator insert(const T& x){
      return iterator(*r, r->insert(it, x));
    }
    iterator erase() {
      return iterator(*r, r->erase(it));
    }
  };
  void push_back(const T& x) {
    lst.push_back(x);
  }
  iterator begin() {
    return iterator(lst, lst.begin());
  }
 int size() { return lst.size(); }
};

int main() {
  Ring<string> rs;
  rs.push_back("one");
  rs.push_back("two");
  rs.push_back("three");
  rs.push_back("four");
  rs.push_back("five");
  Ring<string>::iterator it = rs.begin();
  it++; it++;
  it.insert("six");
  it = rs.begin();
  // Twice around the ring:
  for(int i = 0; i < rs.size() * 2; i++)
    cout << *it++ << endl;
} ///:~
```

You can see that the iterator is where most of the coding is done. The **Ring iterator** must know how to loop back to the beginning, so it must keep a reference to the **list** of its "parent" **Ring** object in order to know if it's at the end and how to get back to the beginning.

You'll notice that the interface for **Ring** is quite limited; in particular there is no **end( )**, since a ring just keeps looping. This means that you won't be able to use a **Ring** in any STL algorithms that require a past-the-end iterator – which is many of them. (It turns out that adding this feature is a non-trivial exercise). Although this can seem limiting, consider **stack**, **queue** and **priority_queue**, which don't produce any iterators at all!

Freely-available STL extensions

Although the STL containers may provide all the functionality you'll ever need, they are not complete. For example, the standard implementations of **set** and **map** use trees, and although these are reasonably fast they may not be fast enough for your needs. In the C++ Standards Committee it was generally agreed that hashed implementations of **set** and **map** should have been included in Standard C++, however there was not considered to be enough time to add these components, and thus they were left out.

Fortunately, there are freely-available alternatives. One of the nice things about the STL is that it establishes a basic model for creating STL-like classes, so anything built using the same model is easy to understand if you are already familiar with the STL.

The SGI STL (freely available at http://www.sgi.com/Technology/STL/) is one of the most robust implementations of the STL, and can be used to replace your compiler's STL if that is found wanting. In addition they've added a number of extensions including **hash_set**, **hash_multiset**, **hash_map**, **hash_multimap**, **slist** (a singly-linked list) and **rope** (a variant of **string** optimized for very large strings and fast concatenation and substring operations).

Let's consider a performance comparison between a tree-based **map** and the SGI **hash_map**. To keep things simple, the mappings will be from **int** to **int**:

```
//: C20:MapVsHashMap.cpp
// The hash_map header is not part of the
// Standard C++ STL. It is an extension that
// is only available as part of the SGI STL:
#include <hash_map>
#include <iostream>
```

```
#include <map>
#include <ctime>
using namespace std;

int main(){
  hash_map<int, int> hm;
  map<int, int> m;
  clock_t ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.insert(make_pair(j,j));
  cout << "map insertions: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.insert(make_pair(j,j));
  cout << "hash_map insertions: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m[j];
  cout << "map::operator[] lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm[j];
  cout << "hash_map::operator[] lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      m.find(j);
  cout << "map::find() lookups: "
    << clock() - ticks << endl;
  ticks = clock();
  for(int i = 0; i < 100; i++)
    for(int j = 0; j < 1000; j++)
      hm.find(j);
  cout << "hash_map::find() lookups: "
```

```
    << clock() - ticks << endl;
} ///:~
```

The performance test I ran showed a speed improvement of roughly 4:1 for the **hash_map** over the **map** in all operations (and as expected, **find( )** is slightly faster than **operator[ ]** for lookups for both types of map). If a profiler shows a bottleneck in your **map**, you should consider a **hash_map**.

Summary

The goal of this chapter was not just to introduce the STL containers in some considerable depth (of course, not every detail could be covered here, but you should have enough now that you can look up further information in the other resources). My higher hope is that this chapter has made you grasp the incredible power available in the STL, and shown you how much faster and more efficient your programming activities can be by using and understanding the STL.

The fact that I could not escape from introducing some of the STL algorithms in this chapter suggests how useful they can be. In the next chapter you'll get a much more focused look at the algorithms.

Exercises

1. Create a **set<char>**, then open a file (whose name is provided on the command line) and read that file in a **char** at a time, placing each **char** in the **set**. Print the results and observe the organization, and whether there are any letters in the alphabet that are not used in that particular file.
2. Create a kind of "hangman" game. Create a class that contains a **char** and a **bool** to indicate whether that **char** has been guessed yet. Randomly select a word from a file, and read it into a **vector** of your new type. Repeatedly ask the user for a character guess, and after each guess display the characters in the word that have been guessed, and underscores for the characters that haven't. Allow a way for the user to guess the whole word. Decrement a value for each guess, and if the user can get the whole word before the value goes to zero, they win.

3. Modify **WordCount.cpp** so that it uses **insert( )** instead of **operator[ ]** to insert elements in the **map**.
4. Modify **WordCount.cpp** so that it uses a **multimap** instead of a **map**.
5. Create a generator that produces random **int** values between 0 and 20. Use this to fill a **multiset<int>**. Count the occurrences of each value, following the example given in **MultiSetWordCount.cpp**.
6. Change **StlShape.cpp** so that it uses a **deque** instead of a **vector**.
7. Modify **Reversible.cpp** so it works with **deque** and **list** instead of **vector**.
8. Modify **Progvals.h** and **ProgVals.cpp** so that they expect leading hyphens to distinguish command-line arguments.
9. Create a second version of **Progvals.h** and **ProgVals.cpp** that uses a **set** instead of a **map** to manage single-character flags on the command line (such as **-a -b -c** etc) and also allows the characters to be ganged up behind a single hyphen (such as **-abc**).
10. Use a **stack<int>** and build a Fibonacci sequence on the stack. The program's command line should take the number of Fibonacci elements desired, and you should have a loop that looks at the last two elements on the stack and pushes a new one for every pass through the loop.
11. Open a text file whose name is provided on the command line. Read the file a word at a time (hint: use **>>**) and use a **multiset<string>** to create a word count for each word.
12. Modify **BankTeller.cpp** so that the policy that decides when a teller is added or removed is encapsulated inside a class.
13. Create two classes **A** and **B** (feel free to choose more interesting names). Create a **multimap<A, B>** and fill it with key-value pairs, ensuring that there are some duplicate keys. Use **equal_range( )** to discover and print a range of objects with duplicate keys. Note you may have to add some functions in **A** and/or **B** to make this program work.
14. Perform the above exercise for a **multiset<A>**.
15. Create a class that has an **operator<** and an **ostream& operator<<**. The class should contain a priority number. Create a generator for your class that makes a random priority number. Fill a **priority_queue** using your

generator, then pull the elements out to show they are in the proper order.

16. Rewrite **Ring.cpp** so it uses a deque instead of a list for its underlying implementation.
17. Modify **Ring.cpp** so that the underlying implementation can be chosen using a template argument (let that template argument default to **list**).
18. Open a file and read it into a single **string**. Turn the **string** into a **stringstream**. Read tokens from the **stringstream** into a **list<string>** using a **TokenIterator**.
19. Compare the performance of **stack** based on whether it is implemented with **vector**, **deque** or **list**.
20. Create an iterator class called **BitBucket** that just absorbs whatever you send to it without writing it anywhere.
21. Create a template that implements a singly-linked list called **SList**. Provide a default constructor, **begin( )** and **end( )** functions (thus you must create the appropriate nested iterator), **insert( )**, **erase( )** and a destructor.
22. (More challenging) Create a little command language. Each command can simply print its name and its arguments, but you may also want to make it perform other activities like run programs. The commands will be read from a file that you pass as an command-line argument, or from standard input if no file is given. Each command is on a single line, and lines beginning with '**#**' are comments. A line begins with the one-word command itself, followed by any number of arguments. Commands and arguments are separated by spaces. Use a **map** that maps **string** objects (the name of the command) to object pointers. The object pointers point to objects of a base class **Command** that has a virtual **execute(string args)** function, where **args** contains all the arguments for that command (**execute( )** will parse its own arguments from **args**). Each different type of command is represented by a class that is inherited from **Command**.
23. Add features to the above exercise so that you can have labels, **if-then** statements, and the ability to jump program execution to a label.

21: STL Algorithms

The other half of the STL is the algorithms, which are templatized functions designed to work with the containers (or, as you will see, anything that can behave like a container, including arrays and **string** objects).

The STL was originally designed around the algorithms. The goal was that you use algorithms for almost every piece of code that you write. In this sense it was a bit of an experiment, and only time will tell how well it works. The real test will be in how easy or difficult it is for the average programmer to adapt. At the end of this chapter you'll be able to decide for yourself whether you find the algorithms addictive or too confusing to remember. If you're like me, you'll resist them at first but then tend to use them more and more.

Before you make your judgement, however, there's one other thing to consider. The STL algorithms provide a *vocabulary* with which to describe solutions. That is, once you become familiar with the algorithms you'll have a new set of words with which to discuss what you're doing, and these words are at a higher level than what you've had before. You don't have to say "this loop moves through and assigns from here to there … oh, I see, it's copying!" Instead, you say **copy( )**. This is the kind of thing we've been doing in computer programming from the beginning – creating more dense ways to express *what* we're doing and spending less time saying *how* we're doing it. Whether the STL algorithms and *generic programming* are a great success in accomplishing this remains to be seen, but that is certainly the objective.

Function objects

A concept that is used heavily in the STL algorithms is the *function object*, which was introduced in the previous chapter. A function object has an overloaded **operator( )**, and the result is that a template function can't tell whether you've handed it a pointer to a function or an object that has an **operator( )**; all the template function knows is that it can attach an argument list to the object *as if* it were a pointer to a function:

```
//: C21:FuncObject.cpp
// Simple function objects
#include <iostream>
using namespace std;

template<class UnaryFunc, class T>
void callFunc(T& x, UnaryFunc f) {
  f(x);
}

void g(int& x) {
  x = 47;
}

struct UFunc {
  void operator()(int& x) {
    x = 48;
  }
};

int main() {
  int y = 0;
  callFunc(y, g);
  cout << y << endl;
  y = 0;
  callFunc(y, UFunc());
  cout << y << endl;
} ///:~
```

The template **callFunc( )** says "give me an **f** and an **x**, and I'll write the code **f(x)**." In **main( )**, you can see that it doesn't matter if **f** is a pointer to a function (as in the case of **g( )**), or if it's a function object (which is created as a temporary object by the expression **UFunc( )**). Notice you

can only accomplish this genericity with a template function; a non-template function is too particular about its argument types to allow such a thing. The STL algorithms use this flexibility to take either a function pointer or a function object, but you'll usually find that creating a function object is more powerful and flexible.

The function object is actually a variation on the theme of a *callback*, which is described in the design patterns chapter. A callback allows you to vary the behavior of a function or object by passing, as an argument, a way to execute some other piece of code. Here, we are handing **callFunc( )** a pointer to a function or a function object.

The following descriptions of function objects should not only make that topic clear, but also give you an introduction to the way the STL algorithms work.

Classification of function objects

Just as the STL classifies iterators (based on their capabilities), it also classifies function objects based on the number of arguments that their **operator( )** takes and the kind of value returned by that operator (of course, this is also true for function pointers when you treat them as function objects). The classification of function objects in the STL is based on whether the **operator( )** takes zero, one or two arguments, and if it returns a **bool** or non-**bool** value.

Generator: Takes no arguments, and returns a value of the desired type. A **RandomNumberGenerator** is a special case.

UnaryFunction: Takes a single argument of any type and returns a value which may be of a different type.

BinaryFunction: Takes two arguments of any two types and returns a value of any type.

A special case of the unary and binary functions is the *predicate*, which simply means a function that returns a **bool**. A predicate is a function you use to make a **true**/**false** decision.

Predicate: This can also be called a **UnaryPredicate**. It takes a single argument of any type and returns a **bool**.

BinaryPredicate: Takes two arguments of any two types and returns a **bool**.

StrictWeakOrdering: A binary predicate that says that if you have two objects and neither one is less than the other, they can be regarded as equivalent to each other.

In addition, there are sometimes qualifications on object types that are passed to an algorithm. These qualifications are given in the template argument type identifier name:

LessThanComparable: A class that has a less-than **operator<**.

Assignable: A class that has an assignment **operator=** for its own type.

EqualityComparable: A class that has an equivalence **operator==** for its own type.

Automatic creation of function objects

The STL has, in the header file **<functional>**, a set of templates that will automatically create function objects for you. These generated function objects are admittedly simple, but the goal is to provide very basic functionality that will allow you to compose more complicated function objects, and in many situations this is all you'll need. Also, you'll see that there are some *function object adapters* that allow you to take the simple function objects and make them slightly more complicated.

Here are the templates that generate function objects, along with the expressions that they effect.

Name	Type	Result produced by generated function object
plus	BinaryFunction	arg1 + arg2
minus	BinaryFunction	arg1 - arg2
multiplies	BinaryFunction	arg1 * arg2
divides	BinaryFunction	arg1 / arg2
modulus	BinaryFunction	arg1 % arg2

Name	Type	Result produced by generated function object
negate	UnaryFunction	- arg1
equal_to	BinaryPredicate	arg1 == arg2
not_equal_to	BinaryPredicate	arg1 != arg2
greater	BinaryPredicate	arg1 > arg2
less	BinaryPredicate	arg1 < arg2
greater_equal	BinaryPredicate	arg1 >= arg2
less_equal	BinaryPredicate	arg1 <= arg2
logical_and	BinaryPredicate	arg1 && arg2
logical_or	BinaryPredicate	arg1 || arg2
logical_not	UnaryPredicate	!arg1
not1( )	Unary Logical	!(UnaryPredicate(arg1))
not2( )	Binary Logical	!(BinaryPredicate(arg1, arg2))

The following example provides simple tests for each of the built-in basic function object templates. This way, you can see how to use each one, along with their resulting behavior.

```
//: C21:FunctionObjects.cpp
// Using the predefined function object templates
// in the Standard C++ library
// This will be defined shortly:
```

```
#include "Generators.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

template<typename T>
void print(vector<T>& v, char* msg = "") {
  if(*msg != 0)
    cout << msg << ":" << endl;
  copy(v.begin(), v.end(),
    ostream_iterator<T>(cout, " "));
  cout << endl;
}

template<typename Contain, typename UnaryFunc>
void testUnary(Contain& source, Contain& dest,
  UnaryFunc f) {
  transform(source.begin(), source.end(),
    dest.begin(), f);
}

template<typename Contain1, typename Contain2,
  typename BinaryFunc>
void testBinary(Contain1& src1, Contain1& src2,
  Contain2& dest, BinaryFunc f) {
  transform(src1.begin(), src1.end(),
    src2.begin(), dest.begin(), f);
}

// Executes the expression, then stringizes the
// expression into the print statement:
#define T(EXPR) EXPR; print(r, "After " #EXPR);
// For Boolean tests:
#define B(EXPR) EXPR; print(br,"After " #EXPR);

// Boolean random generator:
struct BRand {
  BRand() { srand(time(0)); }
  bool operator()() {
    return rand() > RAND_MAX / 2;
```

```
  }
};

int main() {
  const int sz = 10;
  const int max = 50;
  vector<int> x(sz), y(sz), r(sz);
  // An integer random number generator:
  URandGen urg(max);
  generate_n(x.begin(), sz, urg);
  generate_n(y.begin(), sz, urg);
  // Add one to each to guarantee nonzero divide:
  transform(y.begin(), y.end(), y.begin(),
    bind2nd(plus<int>(), 1));
  // Guarantee one pair of elements is ==:
  x[0] = y[0];
  print(x, "x");
  print(y, "y");
  // Operate on each element pair of x & y,
  // putting the result into r:
  T(testBinary(x, y, r, plus<int>()));
  T(testBinary(x, y, r, minus<int>()));
  T(testBinary(x, y, r, multiplies<int>()));
  T(testBinary(x, y, r, divides<int>()));
  T(testBinary(x, y, r, modulus<int>()));
  T(testUnary(x, r, negate<int>()));
  vector<bool> br(sz); // For Boolean results
  B(testBinary(x, y, br, equal_to<int>()));
  B(testBinary(x, y, br, not_equal_to<int>()));
  B(testBinary(x, y, br, greater<int>()));
  B(testBinary(x, y, br, less<int>()));
  B(testBinary(x, y, br, greater_equal<int>()));
  B(testBinary(x, y, br, less_equal<int>()));
  B(testBinary(x, y, br,
    not2(greater_equal<int>())));
  B(testBinary(x,y,br,not2(less_equal<int>())));
  vector<bool> b1(sz), b2(sz);
  generate_n(b1.begin(), sz, BRand());
  generate_n(b2.begin(), sz, BRand());
  print(b1, "b1");
  print(b2, "b2");
  B(testBinary(b1, b2, br, logical_and<int>()));
```

```
  B(testBinary(b1, b2, br, logical_or<int>()));
  B(testUnary(b1, br, logical_not<int>()));
  B(testUnary(b1, br, not1(logical_not<int>())));
} ///:~
```

To keep this example small, some tools are created. The **print( )** template is designed to print any **vector<T>**, along with an optional message. Since **print( )** uses the STL **copy( )** algorithm to send objects to **cout** via an **ostream_iterator**, the **ostream_iterator** must know the type of object it is printing, and therefore the **print( )** template must know this type also. However, you'll see in **main( )** that the compiler can deduce the type of **T** when you hand it a **vector<T>**, so you don't have to hand it the template argument explicitly; you just say **print(x)** to print the **vector<T> x**.

The next two template functions automate the process of testing the various function object templates. There are two since the function objects are either unary or binary. In **testUnary( )**, you pass a source and destination vector, and a unary function object to apply to the source vector to produce the destination vector. In **testBinary( )**, there are two source vectors which are fed to a binary function to produce the destination vector. In both cases, the template functions simply turn around and call the **transform( )** algorithm, although the tests could certainly be more complex.

For each test, you want to see a string describing what the test is, followed by the results of the test. To automate this, the preprocessor comes in handy; the **T( )** and **B( )** macros each take the expression you want to execute. They call that expression, then call **print( )**, passing it the result vector (they assume the expression changes a vector named **r** and **br**, respectively), and to produce the message the expression is "string-ized" using the preprocessor. So that way you see the code of the expression that is executed followed by the result vector.

The last little tool is a generator object that creates random **bool** values. To do this, it gets a random number from **rand( )** and tests to see if it's greater than **RAND_MAX/2**. If the random numbers are evenly distributed, this should happen half the time.

In **main( )**, three **vector<int>** are created: **x** and **y** for source values, and **r** for results. To initialize **x** and **y** with random values no greater than 50, a generator of type **URandGen** is used; this will be defined shortly. Since there is one operation where elements of **x** are divided by elements of **y**, we must ensure that there are no zero values of **y**. This is accomplished using the **transform( )** algorithm, taking the source values

from **y** and putting the results back into **y**. The function object for this is created with the expression:

```
bind2nd(plus<int>(), 1)
```

This uses the **plus** function object that adds two objects together. It is thus a binary function which requires two arguments; we only want to pass it one argument (the element from **y**) and have the other argument be the value 1. A "binder" does the trick (we will look at these next). The binder in this case says "make a new function object which is the **plus** function object with the second argument fixed at 1."

Another of the tests in the program compares the elements in the two vectors for equality, so it is interesting to guarantee that at least one pair of elements is equivalent; in this case element zero is chosen.

Once the two vectors are printed, **T( )** is used to test each of the function objects that produces a numerical value, and then **B( )** is used to test each function object that produces a Boolean result. The result is placed into a **vector<bool>**, and when this vector is printed it produces a '**1**' for a true value and a '**0**' for a false value.

Binders

It's common to want to take a binary function object and to "bind" one of its arguments to a constant value. After binding, you get a unary function object.

For example, suppose you want to find integers that are less than a particular value, say 20. Sensibly enough, the STL algorithms have a function called **find_if( )** that will search through a sequence; however, **find_if( )** requires a unary predicate to tell it if this is what you're looking for. This unary predicate can of course be some function object that you have written by hand, but it can also be created using the built-in function object templates. In this case, the **less** template will work, but that produces a binary predicate, so we need some way of forming a unary predicate. The binder templates (which work with any binary function object, not just binary predicates) give you two choices:

```
bind1st(const BinaryFunction& op, const T& t);
bind2nd(const BinaryFunction& op, const T& t);
```

Both bind **t** to one of the arguments of **op**, but **bind1st( )** binds **t** to the first argument, and **bind2nd( )** binds **t** to the second argument. With **less**, the function object that provides the solution to our exercise is:

```
bind2nd(less<int>(), 20);
```

This produces a new function object that returns true if its argument is less than 20. Here it is, used with **find_if( )**:

```
//: C21:Binder1.cpp
// Using STL "binders"
#include "Generators.h"
#include "copy_if.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

int main() {
  const int sz = 10;
  const int max = 40;
  vector<int> a(sz), r;
  URandGen urg(max);
  ostream_iterator<int> out(cout, " ");
  generate_n(a.begin(), sz, urg);
  copy(a.begin(), a.end(), out);
  int* d = find_if(a.begin(), a.end(),
    bind2nd(less<int>(), 20));
  cout << "\n *d = " << *d << endl;
  // copy_if() is not in the Standard C++ library
  // but is defined later in the chapter:
  copy_if(a.begin(), a.end(), back_inserter(r),
    bind2nd(less<int>(), 20));
  copy(r.begin(), r.end(), out);
  cout << endl;
} ///:~
```

The **vector<int> a** is filled with random numbers between 0 and **max**. **find_if( )** finds the first element in **a** that satisfies the predicate (that is, which is less than 20) and returns an iterator to it (here, the type of the iterator is actually just **int*** although I could have been more precise and said **vector<int>::iterator** instead).

A more interesting algorithm to use is **copy_if( )**, which isn't part of the STL but is defined at the end of this chapter. This algorithm only copies an element from the source to the destination if that element satisfies a predicate. So the resulting vector will only contain elements that are less than 20.

Here's a second example, using a **vector<string>** and replacing strings that satisfy particular conditions:

```
//: C21:Binder2.cpp
// More binders
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main() {
  ostream_iterator<string> out(cout, " ");
  vector<string> v, r;
  v.push_back("Hi");
  v.push_back("Hi");
  v.push_back("Hey");
  v.push_back("Hee");
  v.push_back("Hi");
  copy(v.begin(), v.end(), out);
  cout << endl;
  // Replace each "Hi" with "Ho":
  replace_copy_if(v.begin(), v.end(),
    back_inserter(r),
    bind2nd(equal_to<string>(), "Hi"), "Ho");
  copy(r.begin(), r.end(), out);
  cout << endl;
  // Replace anything that's not "Hi" with "Ho":
  replace_if(v.begin(), v.end(),
    not1(bind2nd(equal_to<string>(),"Hi")),"Ho");
  copy(v.begin(), v.end(), out);
  cout << endl;
} ///:~
```

This uses another pair of STL algorithms. The first, **replace_copy_if( )**, copies each element from a source range to a destination range, performing replacements on those that satisfy a particular unary predicate. The second, **replace_if( )**, doesn't do any copying but instead performs the replacements directly into the original range.

A binder doesn't have to produce a unary predicate; it can also create a unary function (that is, a function that returns something other than **bool**). For example, suppose you'd like to multiply every element in a

vector by 10. Using a binder with the **transform( )** algorithm does the trick:

```
//: C21:Binder3.cpp
// Binders aren't limited to producing predicates
#include "Generators.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

int main() {
  ostream_iterator<int> out(cout, " ");
  vector<int> v(15);
  generate(v.begin(), v.end(), URandGen(20));
  copy(v.begin(), v.end(), out);
  cout << endl;
  transform(v.begin(), v.end(), v.begin(),
    bind2nd(multiplies<int>(), 10));
  copy(v.begin(), v.end(), out);
  cout << endl;
} ///:~
```

Since the third argument to **transform( )** is the same as the first, the resulting elements are copied back into the source vector. The function object created by **bind2nd( )** in this case produces an **int** result.

The “bound” argument to a binder cannot be a function object, but it does not have to be a compile-time constant. For example:

```
//: C21:Binder4.cpp
// The bound argument does not have
// to be a compile-time constant
#include "copy_if.h"
#include "PrintSequence.h"
#include "../require.h"
#include <iostream>
#include <algorithm>
#include <functional>
#include <cstdlib>
using namespace std;

int boundedRand() { return rand() % 100; }
```

```
int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "usage: Binder4 int");
  const int sz = 20;
  int a[20], b[20] = {0};
  generate(a, a + sz, boundedRand);
  int* end = copy_if(a, a + sz, b,
    bind2nd(greater<int>(), atoi(argv[1])));
  // Sort for easier viewing:
  sort(a, a + sz);
  sort(b, end);
  print(a, a + sz, "array a", " ");
  print(b, end, "values greater than yours"," ");
} ///:~
```

Here, an array is filled with random numbers between 0 and 100, and the user provides a value on the command line. In the **copy_if( )** call, you can see that the bound argument to **bind2nd( )** is the result of the function call **atoi( )** (from **<cstdlib>**).

Function pointer adapters

Any place in an STL algorithm where a function object is required, it's very conceivable that you'd like to use a function pointer instead. Actually, you *can* use an ordinary function pointer – that's how the STL was designed, so that a "function object" can actually be anything that can be dereferenced using an argument list. For example, the **rand( )** random number generator can be passed to **generate( )** or **generate_n( )** as a function pointer, like this:

```
//: C21:RandGenTest.cpp
// A little test of the random number generator
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
  const int sz = 10000;
  int v[sz];
  srand(time(0)); // Seed the random generator
```

```
  for(int i = 0; i < 300; i++) {
    // Using a naked pointer to function:
    generate(v, v + sz, std::rand);
    int count = count_if(v, v + sz,
      bind2nd(greater<int>(), RAND_MAX/2));
    cout << (((double)count)/((double)sz)) * 100
      << ' ';
  }
} ///:~
```

The “iterators” in this case are just the starting and past-the-end pointers for the array **v**, and the generator is just a pointer to the standard library **rand( )** function. The program repeatedly generates a group of random numbers, then it uses the STL algorithm **count_if( )** and a predicate that tells whether a particular element is greater than **RAND_MAX/2**. The result is the number of elements that match this criterion; this is divided by the total number of elements and multiplied by 100 to produce the percentage of elements greater than the midpoint. If the random number generator is reasonable, this value should hover at around 50% (of course, there are many other tests to determine if the random number generator is reasonable).

The **ptr_fun( )** adapters take a pointer to a function and turn it into a function object. They are not designed for a function that takes no arguments, like the one above (that is, a generator). Instead, they are for unary functions and binary functions. However, these could also be simply passed as if they were function objects, so the **ptr_fun( )** adapters might at first appear to be redundant. Here’s an example where using **ptr_fun( )** and simply passing the address of the function both produce the same results:

```
//: C21:PtrFun1.cpp
// Using ptr_fun() for single-argument functions
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

char* n[] = { "01.23", "91.370", "56.661",
  "023.230", "19.959", "1.0", "3.14159" };
const int nsz = sizeof n / sizeof *n;

template<typename InputIter>
```

```
void print(InputIter first, InputIter last) {
  while(first != last)
    cout << *first++ << "\t";
  cout << endl;
}

int main() {
  print(n, n + nsz);
  vector<double> vd;
  transform(n, n + nsz, back_inserter(vd), atof);
  print(vd.begin(), vd.end());
  transform(n,n + nsz,vd.begin(), ptr_fun(atof));
  print(vd.begin(), vd.end());
} ///:~
```

The goal of this program is to convert an array of **char*** which are ASCII representations of floating-point numbers into a **vector<double>**. After defining this array and the **print( )** template (which encapsulates the act of printing a range of elements), you can see **transform( )** used with **atof( )** as a "naked" pointer to a function, and then a second time with **atof** passed to **ptr_fun( )**. The results are the same. So why bother with **ptr_fun( )**? Well, the actual effect of **ptr_fun( )** is to create a function object with an **operator( )**. This function object can then be passed to other template adapters, such as binders, to create new function objects. As you'll see a bit later, the SGI extensions to the STL contain a number of other function templates to enable this, but in the Standard C++ STL there are only the **bind1st( )** and **bind2nd( )** function templates, and these expect binary function objects as their first arguments. In the above example, only the **ptr_fun( )** for a unary function is used, and that doesn't work with the binders. So **ptr_fun( )** used with a unary function in Standard C++ really is redundant (note that Gnu g++ uses the SGI STL).

With a binary function and a binder, things can be a little more interesting. This program produces the squares of the input vector **d**:

```
//: C21:PtrFun2.cpp
// Using ptr_fun() for two-argument functions
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
#include <cmath>
using namespace std;
```

```
double d[] = { 01.23, 91.370, 56.661,
  023.230, 19.959, 1.0, 3.14159 };
const int dsz = sizeof d / sizeof *d;

int main() {
  vector<double> vd;
  transform(d, d + dsz, back_inserter(vd),
    bind2nd(ptr_fun(pow), 2.0));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, " "));
  cout << endl;
} ///:~
```

Here, **ptr_fun( )** is indispensable; **bind2nd( )** *must* have a function object as its first argument and a pointer to function won't cut it.

A trickier problem is that of converting a member function into a function object suitable for using in the STL algorithms. As a simple example, suppose we have the "shape" problem and would like to apply the **draw( )** member function to each pointer in a coFntainer of **Shape**:

```
//: C21:MemFun1.cpp
// Applying pointers to member functions
#include "../purge.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  virtual void draw() {
    cout << "Circle::Draw()" << endl;
  }
  ~Circle() {
    cout << "Circle::~Circle()" << endl;
```

```
  }
};

class Square : public Shape {
public:
  virtual void draw() {
    cout << "Square::Draw()" << endl;
  }
  ~Square() {
    cout << "Square::~Square()" << endl;
  }
};

int main() {
  vector<Shape*> vs;
  vs.push_back(new Circle);
  vs.push_back(new Square);
  for_each(vs.begin(), vs.end(),
    mem_fun(&Shape::draw));
  purge(vs);
} ///:~
```

The **for_each( )** function does just what it sounds like it does: passes each element in the range determined by the first two (iterator) arguments to the function object which is its third argument. In this case we want the function object to be created from one of the member functions of the class itself, and so the function object's "argument" becomes the pointer to the object that the member function is called for. To produce such a function object, the **mem_fun( )** template takes a pointer to member as its argument.

The **mem_fun( )** functions are for producing function objects that are called using a pointer to the object that the member function is called for, while **mem_fun_ref( )** is used for calling the member function directly for an object. One set of overloads of both **mem_fun( )** and **mem_fun_ref( )** are for member functions that take zero arguments and one argument, and this is multiplied by two to handle **const** vs. non-**const** member functions. However, templates and overloading takes care of sorting all of that out; all you need to remember is when to use **mem_fun( )** vs. **mem_fun_ref( )**.

Suppose you have a container of objects (not pointers) and you want to call a member function that takes an argument. The argument you pass

should come from a second container of objects. To accomplish this, the second overloaded form of the **transform( )** algorithm is used:

```
//: C21:MemFun2.cpp
// Applying pointers to member functions
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;

class Angle {
  int degrees;
public:
  Angle(int deg) : degrees(deg) {}
  int mul(int times) {
    return degrees *= times;
  }
};

int main() {
  vector<Angle> va;
  for(int i = 0; i < 50; i += 10)
    va.push_back(Angle(i));
  int x[] = { 1, 2, 3, 4, 5 };
  transform(va.begin(), va.end(), x,
    ostream_iterator<int>(cout, " "),
    mem_fun_ref(&Angle::mul));
  cout << endl;
} ///:~
```

Because the container is holding objects, **mem_fun_ref( )** must be used with the pointer-to-member function. This version of **transform( )** takes the start and end point of the first range (where the objects live), the starting point of second range which holds the arguments to the member function, the destination iterator which in this case is standard output, and the function object to call for each object; this function object is created with **mem_fun_ref( )** and the desired pointer to member. Notice the **transform( )** and **for_each( )** template functions are incomplete; **transform( )** requires that the function it calls return a value and there is no **for_each( )** that passes two arguments to the function it calls. Thus, you cannot call a member function that returns **void** and takes an argument using **transform( )** or **for_each( )**.

Any member function works, including those in the Standard libraries. For example, suppose you'd like to read a file and search for blank lines; you can use the **string::empty( )** member function like this:

```
//: C21:FindBlanks.cpp
// Demonstrate mem_fun_ref() with string::empty()
#include "../require.h"
#include <algorithm>
#include <list>
#include <string>
#include <fstream>
#include <functional>
using namespace std;

typedef list<string>::iterator LSI;

LSI blank(LSI begin, LSI end) {
  return find_if(begin, end,
    mem_fun_ref(&string::empty));
}

int main(int argc, char* argv[]) {
 requireArgs(argc, 1);
 ifstream in(argv[1]);
 assure(in, argv[1]);
 list<string> ls;
 string s;
 while(getline(in, s))
  ls.push_back(s);
 LSI lsi = blank(ls.begin(), ls.end());
 while(lsi != ls.end()) {
  *lsi = "A BLANK LINE";
  lsi = blank(lsi, ls.end());
 }
 string f(argv[1]);
 f += ".out";
 ofstream out(f.c_str());
 copy(ls.begin(), ls.end(),
  ostream_iterator<string>(out, "\n"));
} ///:~
```

The **blank( )** function uses **find_if( )** to locate the first blank line in the given range using **mem_fun_ref( )** with **string::empty( )**. After the file

is opened and read into the **list**, **blank( )** is called repeated times to find every blank line in the file. Notice that subsequent calls to **blank( )** use the current version of the iterator so it moves forward to the next one. Each time a blank line is found, it is replaced with the characters "A BLANK LINE." All you have to do to accomplish this is dereference the iterator, and you select the current **string**.

SGI extensions

The SGI STL (mentioned at the end of the previous chapter) also includes additional function object templates, which allow you to write expressions that create even more complicated function objects. Consider a more involved program which converts strings of digits into floating point numbers, like **PtrFun2.cpp** but more general. First, here's a generator that creates strings of integers that represent floating-point values (including an embedded decimal point):

```
//: C21:NumStringGen.h
// A random number generator that produces
// strings representing floating-point numbers
#ifndef NUMSTRINGGEN_H
#define NUMSTRINGGEN_H
#include <string>
#include <cstdlib>
#include <ctime>

class NumStringGen {
  const int sz; // Number of digits to make
public:
  NumStringGen(int ssz = 5) : sz(ssz) {
    std::srand(std::time(0));
  }
  std::string operator()() {
    static char n[] = "0123456789";
    const int nsz = 10;
    std::string r(sz, ' ');
    for(int i = 0; i < sz; i++)
      if(i == sz/2)
        r[i] = '.'; // Insert a decimal point
      else
        r[i] = n[std::rand() % nsz];
    return r;
  }
```

```
};
#endif // NUMSTRINGGEN_H ///:~
```

You tell it how big the **string**s should be when you create the **NumStringGen** object. The random number generator is used to select digits, and a decimal point is placed in the middle.

The following program (which works with the Standard C++ STL without the SGI extensions) uses **NumStringGen** to fill a **vector<string>**. However, to use the Standard C library function **atof( )** to convert the strings to floating-point numbers, the **string** objects must first be turned into **char** pointers, since there is no automatic type conversion from **string** to **char***. The **transform( )** algorithm can be used with **mem_fun_ref( )** and **string::c_str( )** to convert all the **string**s to **char***, and then these can be transformed using **atof**:

```
//: C21:MemFun3.cpp
// Using mem_fun()
#include "NumStringGen.h"
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main() {
  const int sz = 9;
  vector<string> vs(sz);
  // Fill it with random number strings:
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  const char* vcp[sz];
  transform(vs.begin(), vs.end(), vcp,
    mem_fun_ref(&string::c_str));
  vector<double> vd;
  transform(vcp,vcp + sz,back_inserter(vd),
    std::atof);
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

The SGI extensions to the STL contain a number of additional function object templates that accomplish more detailed activities than the Standard C++ function object templates, including **identity** (returns its argument unchanged), **project1st** and **project2nd** (to take two arguments and return the first or second one, respectively), **select1st** and **select2nd** (to take a **pair** object and return the first or second element, respectively), and the "compose" function templates.

If you're using the SGI extensions, you can make the above program denser using one of the two "compose" function templates. The first, **compose1(f1, f2)**, takes the two function objects **f1** and **f2** as its arguments. It produces a function object that takes a single argument, passes it to **f2**, then takes the result of the call to **f2** and passes it to **f1**. The result of **f1** is returned. By using **compose1( )**, the process of converting the **string** objects to **char***, then converting the **char*** to a floating-point number can be combined into a single operation, like this:

```
//: C21:MemFun4.cpp
// Using the SGI STL compose1 function
#include "NumStringGen.h"
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main() {
  const int sz = 9;
  vector<string> vs(sz);
  // Fill it with random number strings:
  generate(vs.begin(), vs.end(), NumStringGen());
  copy(vs.begin(), vs.end(),
    ostream_iterator<string>(cout, "\t"));
  cout << endl;
  vector<double> vd;
  transform(vs.begin(), vs.end(), back_inserter(vd),
    compose1(ptr_fun(atof),
      mem_fun_ref(&string::c_str)));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

You can see there's only a single call to **transform( )** now, and no intermediate holder for the **char** pointers.

The second "compose" function is **compose2( )**, which takes three function objects as its arguments. The first function object is binary (it takes two arguments), and its arguments are the results of the second and third function objects, respectively. The function object that results from **compose2( )** expects one argument, and it feeds that argument to the second and third function objects. Here is an example:

```
//: C21:Compose2.cpp
// Using the SGI STL compose2() function
#include "copy_if.h"
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
  srand(time(0));
  vector<int> v(100);
  generate(v.begin(), v.end(), rand);
  transform(v.begin(), v.end(), v.begin(),
    bind2nd(divides<int>(), RAND_MAX/100));
  vector<int> r;
  copy_if(v.begin(), v.end(), back_inserter(r),
    compose2(logical_and<bool>(),
      bind2nd(greater_equal<int>(), 30),
      bind2nd(less_equal<int>(), 40)));
  sort(r.begin(), r.end());
  copy(r.begin(), r.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
} ///:~
```

The **vector<int> v** is first filled with random numbers. To cut these down to size, the **transform( )** algorithm is used to divide each value by **RAND_MAX/100**, which will force the values to be between 0 and 100 (making them more readable). The **copy_if( )** algorithm defined later in this chapter is then used, along with a composed function object, to copy all the elements that are greater than or equal to 30 and less than or

equal to 40 into the destination **vector<int> r**. Just to show how easy it is, **r** is sorted, and then displayed.

The arguments of **compose2( )** say, in effect:

```
(x >= 30) && (x <= 40)
```

You could also take the function object that comes from a **compose1( )** or **compose2( )** call and pass it into another "compose" expression … but this could rapidly get very difficult to decipher.

Instead of all this composing and transforming, you can write your own function objects (*without* using the SGI extensions) as follows:

```
//: C21:NoCompose.cpp
// Writing out the function objects explicitly
#include "copy_if.h"
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
#include <functional>
#include <cstdlib>
#include <ctime>
using namespace std;

class Rgen {
  const int max;
public:
  Rgen(int mx = 100) : max(RAND_MAX/mx) {
    srand(time(0));
  }
  int operator()() { return rand() / max; }
};

class BoundTest {
  int top, bottom;
public:
  BoundTest(int b, int t) : bottom(b), top(t) {}
  bool operator()(int arg) {
    return (arg >= bottom) && (arg <= top);
  }
};

int main() {
```

```
  vector<int> v(100);
  generate(v.begin(), v.end(), Rgen());
  vector<int> r;
  copy_if(v.begin(), v.end(), back_inserter(r),
    BoundTest(30, 40));
  sort(r.begin(), r.end());
  copy(r.begin(), r.end(),
    ostream_iterator<int>(cout, " "));
  cout << endl;
} ///:~
```

There are a few more lines of code, but you can't deny that it's much clearer and easier to understand, and therefore to maintain.

We can thus observe two drawbacks to the SGI extensions to the STL. The first is simply that it's an extension; yes, you can download and use them for free so the barriers to entry are low, but your company may be conservative and decide that if it's not in Standard C++, they don't want to use it. The second drawback is complexity. Once you get familiar and comfortable with the idea of composing complicated functions from simple ones you can visually parse complicated expressions and figure out what they mean. However, my guess is that most people will find anything more than what you can do with the Standard, non-extended STL function object notation to be overwhelming. At some point on the complexity curve you have to bite the bullet and write a regular class to produce your function object, and that point might as well be the point where you can't use the Standard C++ STL. A stand-alone class for a function object is going to be much more readable and maintainable than a complicated function-composition expression (although my sense of adventure does lure me into wanting to experiment more with the SGI extensions…).

As a final note, you can't compose generators; you can only create function objects whose **operator( )** requires one or two arguments.

A catalog of STL algorithms

This section provides a quick reference for when you're searching for the appropriate algorithm. I leave the full exploration of all the STL algorithms to other references (see the end of this chapter, and Appendix XX), along with the more intimate details of complexity, performance, etc. My goal

here is for you to become rapidly comfortable and facile with the algorithms, and I will assume you will look into the more specialized references if you need more depth of detail.

Although you will often see the algorithms described using their full template declaration syntax, I am not doing that here because you already know they are templates, and it's quite easy to see what the template arguments are from the function declarations. The type names for the arguments provide descriptions for the types of iterators required. I think you'll find this form is easier to read, while you can quickly find the full declaration in the template header file if for some reason you feel the need.

The names of the iterator classes describe the iterator type they must conform to. The iterator types were described in the previous chapter, but here is a summary:

> **InputIterator**. You (or rather, the STL algorithm and any algorithms you write that use **InputIterator**s) can increment this with **operator++** and dereference it with **operator*** to *read* the value (and *only* read the value), but you can only read each value once. **InputIterator**s can be tested with **operator==** and **operator!=**. That's all. Because an **InputIterator** is so limited, it can be used with **istream**s (via **istream_iterator**).
>
> **OutputIterator**. This can be incremented with **operator++**, and dereferenced with **operator*** to *write* the value (and *only* write the value), but you can only dereference/write each value once. **OutputIterator**s cannot be tested with **operator==** and **operator!=**, however, because you assume that you can just keep sending elements to the destination and that you don't have to see if the destination's end marker has been reached. That is, the container that an **OutputIterator** references can take an infinite number of objects, so no end-checking is necessary. This requirement is important so that an **OutputIterator** can be used with **ostream**s (via **ostream_iterator**), but you'll also commonly use the "insert" iterators **insert_iterator**, **front_insert_iterator** and **back_insert_iterator** (generated by the helper templates **inserter( )**, **front_inserter( )** and **back_inserter( )**).
>
> With both **InputIterator** and **OutputIterator**, you cannot have multiple iterators pointing to different parts of the same range. Just think in terms of iterators to support **istream**s and **ostream**s, and **InputIterator** and **OutputIterator** will make perfect sense. Also note that **InputIterator** and **OutputIterator** put the weakest

restrictions on the types of iterators they will accept, which means that you can use any "more sophisticated" type of iterator when you see **InputIterator** or **OutputIterator** used as STL algorithm template arguments.

ForwardIterator. **InputIterator** and **OutputIterator** are the most restricted, which means they'll work with the largest number of actual iterators. However, there are some operations for which they are too restricted; you can only read from an **InputIterator** and write to an **OutputIterator**, so you can't use them to read and modify a range, for example, and you can't have more than one active iterator on a particular range, or dereference such an iterator more than once. With a **ForwardIterator** these restrictions are relaxed; you can still only move forward using **operator++**, but you can both write and read and you can write/read multiple times in each location. A **ForwardIterator** is much more like a regular pointer, whereas **InputIterator** and **OutputIterator** are a bit strange by comparison.

BidirectionalIterator. Effectively, this is a **ForwardIterator** that can also go backward. That is, a **BidirectionalIterator** supports all the operations that a **ForwardIterator** does, but in addition it has an **operator--**.

RandomAccessIterator. An iterator that is random access supports all the same operations that a regular pointer does: you can add and subtract integral values to move it forward and backward by jumps (rather than just one element at a time), you can subscript it with **operator[ ]**, you can subtract one iterator from another, and iterators can be compared to see which is greater using **operator<**, **operator>**, etc. If you're implementing a sorting routine or something similar, random access iterators are necessary to be able to create an efficient algorithm.

The names used for the template parameter types consist of the above iterator types (sometimes with a '1' or '2' appended to distinguish different template arguments), and may also include other arguments, often function objects.

When describing the group of elements that an operation is performed on, mathematical "range" notation will often be used. In this, the square bracket means "includes the end point" while the parenthesis means "does not include the end point." When using iterators, a range is determined by the iterator pointing to the initial element, and the "past-the-end" iterator, pointing past the last element. Since the past-the-end element is never

used, the range determined by a pair of iterators can thus be expressed as **[first, last)**, where **first** is the iterator pointing to the initial element and **last** is the past-the-end iterator.

Most books and discussions of the STL algorithms arrange them according to side effects: nonmutating algorithms don't change the elements in the range, mutating algorithms do change the elements, etc. These descriptions are based more on the underlying behavior or implementation of the algorithm – that is, the designer's perspective. In practice, I don't find this a very useful categorization so I shall instead organize them according to the problem you want to solve: are you searching for an element or set of elements, performing an operation on each element, counting elements, replacing elements, etc. This should help you find the one you want more easily.

Note that all the algorithms are in the **namespace std**. If you do not see a different header such as **<utility>** or **<numerics>** above the function declarations, that means it appears in **<algorithm>**.

Support tools for example creation

It's useful to create some basic tools with which to test the algorithms.

Displaying a range is something that will be done constantly, so here is a templatized function that allows you to print any sequence, regardless of the type that's in that sequence:

```
//: C21:PrintSequence.h
// Prints the contents of any sequence
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <iostream>

template<typename InputIter>
void print(InputIter first, InputIter last,
  char* nm = "", char* sep = "\n",
  std::ostream& os = std::cout) {
  if(*nm != '\0') // Only if you provide a string
    os << nm << ": " << sep; // is this printed
  while(first != last)
    os << *first++ << sep;
  os << std::endl;
```

```
}

// Use template-templates to allow type deduction
// of the typename T:
template<typename T, template<typename> class C>
void print(C<T>& c, char* nm = "",
  char* sep = "\n",
  std::ostream& os = std::cout) {
  if(*nm != '\0') // Only if you provide a string
    os << nm << ": " << sep; // is this printed
  std::copy(c.begin(), c.end(),
    std::ostream_iterator<T>(os, " "));
  cout << endl;
}
#endif // PRINTSEQUENCE_H ///:~
```

There are two forms here, one that requires you to give an explicit range (this allows you to print an array or a sub-sequence) and one that prints any of the STL containers, which provides notational convenience when printing the entire contents of that container. The second form performs template type deduction to determine the type of **T** so it can be used in the **copy( )** algorithm. That trick wouldn't work with the first form, so the **copy( )** algorithm is avoided and the copying is just done by hand (this could have been done with the second form as well, but it's instructive to see a template-template in use). Because of this, you never need to specify the type that you're printing when you call either template function.

The default is to print to **cout** with newlines as separators, but you can change that. You may also provide a message to print at the head of the output.

Next, it's useful to have some generators (classes with an **operator( )** that returns values of the appropriate type) which allow a sequence to be rapidly filled with different values.

```
//: C21:Generators.h
// Different ways to fill sequences
#ifndef GENERATORS_H
#define GENERATORS_H
#include <set>
#include <cstdlib>
#include <cstring>
#include <ctime>
```

```
// A generator that can skip over numbers:
class SkipGen {
  int i;
  int skp;
public:
  SkipGen(int start = 0, int skip = 1)
    : i(start), skp(skip) {}
  int operator()() {
    int r = i;
    i += skp;
    return r;
  }
};

// Generate unique random numbers from 0 to mod:
class URandGen {
  std::set<int> used;
  int modulus;
public:
  URandGen(int mod) : modulus(mod) {
    std::srand(std::time(0));
  }
  int operator()() {
    while(true) {
      int i = (int)std::rand() % modulus;
      if(used.find(i) == used.end()) {
        used.insert(i);
        return i;
      }
    }
  }
};

// Produces random characters:
class CharGen {
  static const char* source;
  static const int len;
public:
  CharGen() { std::srand(std::time(0)); }
  char operator()() {
    return source[std::rand() % len];
```

```
  }
};

// Statics created here for convenience, but
// will cause problems if multiply included:
const char* CharGen::source = "ABCDEFGHIJK"
  "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
const int CharGen::len = std::strlen(source);
#endif // GENERATORS_H ///:~
```

To create some interesting values, the **SkipGen** generator skips by the value **skp** each time its **operator( )** is called. You can initialize both the start value and the skip value in the constructor.

URandGen ('U' for "unique") is a generator for random **int**s between 0 and **mod**, with the additional constraint that each value can only be produced once (thus you must be careful not to use up all the values). This is easily accomplished with a **set**.

CharGen generates **char**s and can be used to fill up a **string** (when treating a **string** as a sequence container). You'll note that the one member function that any generator implements is **operator( )** (with no arguments). This is what is called by the "generate" functions.

The use of the generators and the **print( )** functions is shown in the following section.

Finally, a number of the STL algorithms that move elements of a sequence around distinguish between "stable" and "unstable" reordering of a sequence. This refers to preserving the original order of the elements for those elements that are equivalent but not identical. For example, consider a sequence **{ c(1), b(1), c(2), a(1), b(2), a(2) }**. These elements are tested for equivalence based on their letters, but their numbers indicate how they first appeared in the sequence. If you sort (for example) this sequence using an unstable sort, there's no guarantee of any particular order among equivalent letters, so you could end up with **{ a(2), a(1), b(1), b(2), c(2), c(1) }**. However, if you used a stable sort, it guarantees you will get **{ a(1), a(2), b(1), b(2), c(1), c(2) }**.

To demonstrate the stability versus instability of algorithms that reorder a sequence, we need some way to keep track of how the elements originally appeared. The following is a kind of **string** object that keeps track of the order in which that particular object originally appeared, using a **static map** that maps **NString**s to **Counter**s. Each **NString** then contains an

occurrence field that indicates the order in which this **NString** was discovered:

```
//: C21:NString.h
// A "numbered string" that indicates which
// occurrence this is of a particular word
#ifndef NSTRING_H
#define NSTRING_H
#include <string>
#include <map>
#include <iostream>

class NString {
  std::string s;
  int occurrence;
  struct Counter {
    int i;
    Counter() : i(0) {}
    Counter& operator++(int) {
      i++;
      return *this;
    } // Post-incr
    operator int() { return i; }
  };
  // Keep track of the number of occurrences:
  typedef std::map<std::string, Counter> csmap;
  static csmap occurMap;
public:
  NString() : occurrence(0) {}
  NString(const std::string& x)
    : s(x), occurrence(occurMap[s]++) {}
  NString(const char* x)
    : s(x), occurrence(occurMap[s]++) {}
  // The synthesized operator= and
  // copy-constructor are OK here
  friend std::ostream& operator<<(
    std::ostream& os, const NString& ns) {
    return os << ns.s << " ["
      << ns.occurrence << "]";
  }
  // Need this for sorting. Notice it only
  // compares strings, not occurrences:
  friend bool
```

```
  operator<(const NString& l, const NString& r) {
    return l.s < r.s;
  }
  // For sorting with greater<NString>:
  friend bool
  operator>(const NString& l, const NString& r) {
    return l.s > r.s;
  }
  // To get at the string directly:
  operator const std::string&() const {return s;}
};

// Allocate static member object. Done here for
// brevity, but should actually be done in a
// separate cpp file:
NString::csmap NString::occurMap;
#endif // NSTRING_H ///:~
```

In the constructors (one that takes a **string**, one that takes a **char***), the simple-looking initialization **occurrence(occurMap[s]++)** performs all the work of maintaining and assigning the occurrence counts (see the demonstration of the **map** class in the previous chapter for more details).

To do an ordinary ascending sort, the only operator that's necessary is **NString::operator<( )**, however to sort in reverse order the **operator>( )** is also provided so that the **greater** template can be used.

As this is just a demonstration class I am getting away with the convenience of putting the definition of the static member **occurMap** in the header file, but this will break down if the header file is included in more than one place, so you should normally relegate all **static** definitions to **cpp** files.

Filling & generating

These algorithms allow you to automatically fill a range with a particular value, or to generate a set of values for a particular range (these were introduced in the previous chapter). The "fill" functions insert a single value multiple times into the container, while the "generate" functions use an object called a *generator* (described earlier) to create the values to insert into the container.

void fill(ForwardIterator first, ForwardIterator last, const T& value);
void fill_n(OutputIterator first, Size n, const T& value);

fill( ) assigns **value** to every element in the range **[first, last)**. **fill_n( )** assigns **value** to **n** elements starting at **first**.

void generate(ForwardIterator first, ForwardIterator last, Generator gen);
void generate_n(OutputIterator first, Size n, Generator gen);

generate( ) makes a call to **gen( )** for each element in the range **[first, last)**, presumably to produce a different value for each element. **generate_n( )** calls **gen( ) n** times and assigns each result to **n** elements starting at **first**.

Example

The following example fills and generates into **vector**s. It also shows the use of **print( )**:

```
//: C21:FillGenerateTest.cpp
// Demonstrates "fill" and "generate"
#include "Generators.h"
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

int main() {
  vector<string> v1(5);
  fill(v1.begin(), v1.end(), "howdy");
  print(v1, "v1", " ");
  vector<string> v2;
  fill_n(back_inserter(v2), 7, "bye");
  print(v2.begin(), v2.end(), "v2");
  vector<int> v3(10);
  generate(v3.begin(), v3.end(), SkipGen(4,5));
  print(v3, "v3", " ");
  vector<int> v4;
  generate_n(back_inserter(v4),15, URandGen(30));
  print(v4, "v4", " ");
} ///:~
```

A **vector<string>** is created with a pre-defined size. Since storage has already been created for all the **string** objects in the **vector**, **fill( )** can use its assignment operator to assign a copy of "howdy" to each space in the **vector**. To print the result, the second form of **print( )** is used which simply needs a container (you don't have to give the first and last iterators). Also, the default newline separator is replaced with a space.

The second **vector<string> v2** is not given an initial size so **back_inserter** must be used to force new elements in instead of trying to assign to existing locations. Just as an example, the other **print( )** is used which requires a range.

The **generate( )** and **generate_n( )** functions have the same form as the "fill" functions except that they use a generator instead of a constant value; here, both generators are demonstrated.

Counting

All containers have a method **size( )** that will tell you how many elements they hold. The following two algorithms count objects only if they satisfy certain criteria.

IntegralValue count(InputIterator first, InputIterator last, const EqualityComparable& value);

Produces the number of elements in **[first, last)** that are equivalent to **value** (when tested using **operator==**).

IntegralValue count_if(InputIterator first, InputIterator last, Predicate pred);

Produces the number of elements in **[first, last)** which each cause **pred** to return **true**.

Example

Here, a **vector<char> v** is filled with random characters (including some duplicates). A **set<char>** is initialized from **v**, so it holds only one of each letter represented in **v**. This **set** is used to count all the instances of all the different characters, which are then displayed:

```
//: C21:Counting.cpp
// The counting algorithms
#include "PrintSequence.h"
#include "Generators.h"
#include <vector>
```

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

int main() {
  vector<char> v;
  generate_n(back_inserter(v), 50, CharGen());
  print(v, "v", "");
  // Create a set of the characters in v:
  set<char> cs(v.begin(), v.end());
  set<char>::iterator it = cs.begin();
  while(it != cs.end()) {
    int n = count(v.begin(), v.end(), *it);
    cout << *it << ": " << n << ", ";
    it++;
  }
  int lc = count_if(v.begin(), v.end(),
    bind2nd(greater<char>(), 'a'));
  cout << "\nLowercase letters: " << lc << endl;
  sort(v.begin(), v.end());
  print(v, "sorted", "");
} ///:~
```

The **count_if( )** algorithm is demonstrated by counting all the lowercase letters; the predicate is created using the **bind2nd( )** and **greater** function object templates.

Manipulating sequences

These algorithms allow you to move sequences around.

OutputIterator copy(InputIterator, first InputIterator last, OutputIterator destination);

Using assignment, copies from **[first, last)** to **destination**, incrementing **destination** after each assignment. Works with almost any type of source range and almost any kind of destination. Because assignment is used, you cannot directly insert elements into an empty container or at the end of a container, but instead you must wrap the **destination** iterator in an **insert_iterator** (typically by using **back_inserter( )**, or **inserter( )** in the case of an associative container).

The copy algorithm is used in many examples in this book.

BidirectionalIterator2 copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 destinationEnd);

Like **copy( )**, but performs the actual copying of the elements in reverse order. That is, the resulting sequence is the same, it's just that the copy happens in a different way. The source range **[first, last)** is copied to the destination, but the first destination element is **destinationEnd - 1**. This iterator is then decremented after each assignment. The space in the destination range must already exist (to allow assignment), and the destination range cannot be within the source range.

---

void reverse(BidirectionalIterator first, BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
OutputIterator destination);

Both forms of this function reverse the range **[first, last)**. **reverse( )** reverses the range in place, while **reverse_copy( )** leaves the original range alone and copies the reversed elements into **destination**, returning the past-the-end iterator of the resulting range.

ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);

Exchanges the contents of two ranges of equal size, by moving from the beginning to the end of each range and swapping each set of elements.

---

void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
ForwardIterator last, OutputIterator destination);

Swaps the two ranges **[first, middle)** and **[middle, last)**. With **rotate( )**, the swap is performed in place, and with **rotate_copy( )** the original range is untouched and the rotated version is copied into **destination**, returning the past-the-end iterator of the resulting range. Note that while **swap_ranges( )** requires that the two ranges be exactly the same size, the "rotate" functions do not.

---

bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last,

StrictWeakOrdering binary_pred);
bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last);
bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last,
StrictWeakOrdering binary_pred);

A *permutation* is one unique ordering of a set of elements. If you have **n** unique elements, then there are **n!** (**n** factorial) distinct possible combinations of those elements. All these combinations can be conceptually sorted into a sequence using a lexicographical ordering, and thus produce a concept of a "next" and "previous" permutation. Therefore, whatever the current ordering of elements in the range, there is a distinct "next" and "previous" permutation in the sequence of permutations.

The **next_permutation( )** and **prev_permutation( )** functions re-arrange the elements into their next or previous permutation, and if successful return **true**. If there are no more "next" permutations, it means that the elements are in sorted order so **next_permutation( )** returns **false**. If there are no more "previous" permutations, it means that the elements are in descending sorted order so **previous_permutation( )** returns **false**.

The versions of the functions which have a **StrictWeakOrdering** argument perform the comparisons using **binary_pred** instead of **operator<**.

---

void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last
RandomNumberGenerator& rand);

This function randomly rearranges the elements in the range. It yields uniformly distributed results. The first form uses an internal random number generator and the second uses a user-supplied random-number generator.

BidirectionalIterator partition(BidirectionalIterator first,
BidirectionalIterator last,
Predicate pred);
BidirectionalIterator stable_partition(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);

The "partition" functions use **pred** to organize the elements in the range **[first, last)** so they are before or after the partition (a point in the

range). The partition point is given by the returned iterator. If **pred(*i)** is **true** (where **i** is the iterator pointing to a particular element), then that element will be placed before the partition point, otherwise it will be placed after the partition point.

With **partition( )**, the order of the elements is after the function call is not specified, but with **stable_parition( )** the relative order of the elements before and after the partition point will be the same as before the partitioning process.

Example

This gives a basic demonstration of sequence manipulation:

```
//: C21:Manipulations.cpp
// Shows basic manipulations
#include "PrintSequence.h"
#include "NString.h"
#include "Generators.h"
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() {
  vector<int> v1(10);
  // Simple counting:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1, "v1", " ");
  vector<int> v2(v1.size());
  copy_backward(v1.begin(), v1.end(), v2.end());
  print(v2, "copy_backward", " ");
  reverse_copy(v1.begin(), v1.end(), v2.begin());
  print(v2, "reverse_copy", " ");
  reverse(v1.begin(), v1.end());
  print(v1, "reverse", " ");
  int half = v1.size() / 2;
  // Ranges must be exactly the same size:
  swap_ranges(v1.begin(), v1.begin() + half,
    v1.begin() + half);
  print(v1, "swap_ranges", " ");
  // Start with fresh sequence:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1, "v1", " ");
```

```
int third = v1.size() / 3;
for(int i = 0; i < 10; i++) {
  rotate(v1.begin(), v1.begin() + third,
    v1.end());
  print(v1, "rotate", " ");
}
cout << "Second rotate example:" << endl;
char c[] = "aabbccddeeffgghhiijj";
const char csz = strlen(c);
for(int i = 0; i < 10; i++) {
  rotate(c, c + 2, c + csz);
  print(c, c + csz, "", "");
}
cout << "All n! permutations of abcd:" << endl;
int nf = 4 * 3 * 2 * 1;
char p[] = "abcd";
for(int i = 0; i < nf; i++) {
  next_permutation(p, p + 4);
  print(p, p + 4, "", "");
}
cout << "Using prev_permutation:" << endl;
for(int i = 0; i < nf; i++) {
  prev_permutation(p, p + 4);
  print(p, p + 4, "", "");
}
cout << "random_shuffling a word:" << endl;
string s("hello");
cout << s << endl;
for(int i = 0; i < 5; i++) {
  random_shuffle(s.begin(), s.end());
  cout << s << endl;
}
NString sa[] = { "a", "b", "c", "d", "a", "b",
  "c", "d", "a", "b", "c", "d", "a", "b", "c"};
const int sasz = sizeof sa / sizeof *sa;
vector<NString> ns(sa, sa + sasz);
print(ns, "ns", " ");
vector<NString>::iterator it =
  partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
cout << "Partition point: " << *it << endl;
print(ns, "", " ");
```

```
  // Reload vector:
  copy (sa, sa + sasz, ns.begin());
  it = stable_partition(ns.begin(), ns.end(),
    bind2nd(greater<NString>(), "b"));
  cout << "Stable partition" << endl;
  cout << "Partition point: " << *it << endl;
  print(ns, "", " ");
} ///:~
```

The best way to see the results of the above program is to run it (you'll probably want to redirect the output to a file).

The **vector<int> v1** is initially loaded with a simple ascending sequence and printed. You'll see that the effect of **copy_backward( )** (which copies into **v2**, which is the same size as **v1**) is the same as an ordinary copy. Again, **copy_backward( )** does the same thing as **copy( )**, it just performs the operations in backward order.

reverse_copy( ), however, actually does created a reversed copy, while **reverse( )** performs the reversal in place. Next, **swap_ranges( )** swaps the upper half of the reversed sequence with the lower half. Of course, the ranges could be smaller subsets of the entire vector, as long as they are of equivalent size.

After re-creating the ascending sequence, **rotate( )** is demonstrated by rotating one third of **v1** multiple times. A second **rotate( )** example uses characters and just rotates two characters at a time. This also demonstrates the flexibility of both the STL algorithms and the **print( )** template, since they can both be used with arrays of **char** as easily as with anything else.

To demonstrate **next_permutation( )** and **prev_permutation( )**, a set of four characters "abcd" is permuted through all **n!** (**n** factorial) possible combinations. You'll see from the output that the permutations move through a strictly-defined order (that is, permuting is a deterministic process).

A quick-and-dirty demonstration of **random_shuffle( )** is to apply it to a **string** and see what words result. Because a **string** object has **begin( )** and **end( )** member functions that return the appropriate iterators, it too may be easily used with many of the STL algorithms. Of course, an array of **char** could also have been used.

Finally, the **partition( )** and **stable_partition( )** are demonstrated, using an array of **NString**. You'll note that the aggregate initialization

expression uses **char** arrays, but **NString** has a **char*** constructor which is automatically used.

When partitioning a sequence, you need a predicate which will determine whether the object belongs above or below the partition point. This takes a single argument and returns **true** (the object is above the partition point) or **false** (it isn't). I could have written a separate function or function object to do this, but for something simple, like "the object is greater than 'b'", why not use the built-in function object templates? The expression is:

```
bind2nd(greater<NString>(), "b")
```

And to understand it, you need to pick it apart from the middle outward. First,

```
greater<NString>()
```

produces a binary function object which compares its first and second arguments:

```
return first > second;
```

and returns a **bool**. But we don't want a binary predicate, and we want to compare against the constant value "**b**." So **bind2nd( )** says: create a new function object which only takes one argument, by taking this **greater<NString>( )** function and forcing the second argument to always be "**b**." The first argument (the only argument) will be the one from the vector **ns**.

You'll see from the output that with the unstable partition, the objects are correctly above and below the partition point, but in no particular order, whereas with the stable partition their original order is maintained.

Searching & replacing

All of these algorithms are used for searching for one or more objects within a range defined by the first two iterator arguments.

InputIterator find(InputIterator first, InputIterator last, const EqualityComparable& value);

Searches for **value** within a range of elements. Returns an iterator in the range **[first, last)** that points to the first occurrence of **value**. If **value** isn't in the range, then **find( )** returns **last**. This is a *linear search*, that is, it starts at the beginning and looks at each sequential element without making any assumptions about the way the elements are ordered. In

contrast, a **binary_search( )** (defined later) works on a sorted sequence and can thus be much faster.

InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

Just like **find( )**, **find_if( )** performs a linear search through the range. However, instead of searching for **value**, **find_if( )** looks for an element such that the **Predicate pred** returns **true** when applied to that element. Returns **last** if no such element can be found.

ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
BinaryPredicate binary_pred);

Like **find( )**, performs a linear search through the range, but instead of looking for only one element it searches for two elements that are right next to each other. The first form of the function looks for two elements that are equivalent (via **operator==**). The second form looks for two adjacent elements that, when passed together to **binary_pred**, produce a **true** result. If two adjacent elements cannot be found, **last** is returned.

ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate binary_pred);

Like **find( )**, performs a linear search through the range. The first form finds the first element in the first range that is equivalent to any of the elements in the second range. The second form finds the first element in the first range that produces **true** when passed to **binary_pred** along with any of the elements in the second range. When a **BinaryPredicate** is used with two ranges in the algorithms, the element from the first range becomes the first argument to **binary_pred**, and the element from the second range becomes the second argument.

ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 search(ForwardIterator1 first1,

ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2 BinaryPredicate binary_pred);

Attempts to find the entire range **[first2, last2)** within the range **[first1, last1)**. That is, it checks to see if the second range occurs (in the exact order of the second range) within the first range, and if so returns an iterator pointing to the place in the first range where the second range begins. Returns **last1** if no subset can be found. The first form performs its test using **operator==**, while the second checks to see if each pair of objects being compared causes **binary_pred** to return **true**.

ForwardIterator1 find_end(ForwardIterator1 first1,
ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);
ForwardIterator1 find_end(ForwardIterator1 first1,
ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate binary_pred);

The forms and arguments are just like **search( )** in that it looks for the second range within the first range, but while **search( )** looks for the first occurrence of the second range, **find_end( )** looks for the *last* occurrence of the second range within the first.

ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
Size count, const T& value);
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
Size count, const T& value, BinaryPredicate binary_pred);

Looks for a group of **count** consecutive values in **[first, last)** that are all equal to **value** (in the first form) or that all cause a return value of **true** when passed into **binary_pred** along with **value** (in the second form). Returns **last** if such a group cannot be found.

ForwardIterator min_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last,
BinaryPredicate binary_pred);

Returns an iterator pointing to the first occurrence of the smallest value in the range (there may be multiple occurrences of the smallest value). Returns **last** if the range is empty. The first version performs comparisons

with **operator<** and the value **r** returned is such that
***e < *r**
is false for every element **e** in the range. The second version compares using **binary_pred** and the value **r** returned is such that **binary_pred (*e, *r)** is false for every element **e** in the range.

**ForwardIterator max_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last,
BinaryPredicate binary_pred);**

Returns an iterator pointing to the first occurrence of the largest value in the range (there may be multiple occurrences of the largest value). Returns **last** if the range is empty. The first version performs comparisons with **operator<** and the value **r** returned is such that
***r < *e**
is false for every element **e** in the range. The second version compares using **binary_pred** and the value **r** returned is such that **binary_pred (*r, *e)** is false for every element **e** in the range.

**void replace(ForwardIterator first, ForwardIterator last,
const T& old_value, const T& new_value);
void replace_if(ForwardIterator first, ForwardIterator last,
Predicate pred, const T& new_value);
OutputIterator replace_copy(InputIterator first, InputIterator last,
OutputIterator result, const T& old_value, const T& new_value);
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
OutputIterator result, Predicate pred, const T& new_value);**

Each of the "replace" forms moves through the range **[first, last)**, finding values that match a criterion and replacing them with **new_value**. Both **replace( )** and **replace_copy( )** simply look for **old_value** to replace, while **replace_if( )** and **replace_copy_if( )** look for values that satisfy the predicate **pred**. The "copy" versions of the functions do not modify the original range but instead make a copy with the replacements into **result** (incrementing **result** after each assignment).

Example

To provide easy viewing of the results, this example will manipulate **vector**s of **int**. Again, not every possible version of each algorithm will be shown (some that should be obvious have been omitted).

```
//: C21:SearchReplace.cpp
// The STL search and replace algorithms
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

struct PlusOne {
  bool operator()(int i, int j) {
    return j == i + 1;
  }
};

class MulMoreThan {
  int value;
public:
  MulMoreThan(int val) : value(val) {}
  bool operator()(int v, int m) {
    return v * m > value;
  }
};

int main() {
  int a[] = { 1, 2, 3, 4, 5, 6, 6, 7, 7, 7,
    8, 8, 8, 8, 11, 11, 11, 11, 11 };
  const int asz = sizeof a / sizeof *a;
  vector<int> v(a, a + asz);
  print(v, "v", " ");
  vector<int>::iterator it =
    find(v.begin(), v.end(), 4);
  cout << "find: " << *it << endl;
  it = find_if(v.begin(), v.end(),
    bind2nd(greater<int>(), 8));
  cout << "find_if: " << *it << endl;
  it = adjacent_find(v.begin(), v.end());
  while(it != v.end()) {
    cout << "adjacent_find: " << *it
      << ", " << *(it + 1) << endl;
    it = adjacent_find(it + 2, v.end());
  }
  it = adjacent_find(v.begin(), v.end(),
```

```
  PlusOne());
while(it != v.end()) {
  cout << "adjacent_find PlusOne: " << *it
    << ", " << *(it + 1) << endl;
  it = adjacent_find(it + 1, v.end(),
    PlusOne());
}
int b[] = { 8, 11 };
const int bsz = sizeof b / sizeof *b;
print(b, b + bsz, "b", " ");
it = find_first_of(v.begin(), v.end(),
  b, b + bsz);
print(it, it + bsz, "find_first_of", " ");
it = find_first_of(v.begin(), v.end(),
  b, b + bsz, PlusOne());
print(it,it + bsz,"find_first_of PlusOne"," ");
it = search(v.begin(), v.end(), b, b + bsz);
print(it, it + bsz, "search", " ");
int c[] = { 5, 6, 7 };
const int csz = sizeof c / sizeof *c;
print(c, c + csz, "c", " ");
it = search(v.begin(), v.end(),
  c, c + csz, PlusOne());
print(it, it + csz,"search PlusOne", " ");
int d[] = { 11, 11, 11 };
const int dsz = sizeof d / sizeof *d;
print(d, d + dsz, "d", " ");
it = find_end(v.begin(), v.end(), d, d + dsz);
print(it, v.end(),"find_end", " ");
int e[] = { 9, 9 };
print(e, e + 2, "e", " ");
it = find_end(v.begin(), v.end(),
  e, e + 2, PlusOne());
print(it, v.end(),"find_end PlusOne"," ");
it = search_n(v.begin(), v.end(), 3, 7);
print(it, it + 3, "search_n 3, 7", " ");
it = search_n(v.begin(), v.end(),
  6, 15, MulMoreThan(100));
print(it, it + 6,
  "search_n 6, 15, MulMoreThan(100)", " ");
cout << "min_element: " <<
  *min_element(v.begin(), v.end()) << endl;
```

```
  cout << "max_element: " <<
    *max_element(v.begin(), v.end()) << endl;
  vector<int> v2;
  replace_copy(v.begin(), v.end(),
    back_inserter(v2), 8, 47);
  print(v2, "replace_copy 8 -> 47", " ");
  replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
  print(v, "replace_if >= 7 -> -1", " ");
} ///:~
```

The example begins with two predicates: **PlusOne** which is a binary predicate that returns **true** if the second argument is equivalent to one plus the first argument, and **MulMoreThan** which returns **true** if the first argument times the second argument is greater than a value stored in the object. These binary predicates are used as tests in the example.

In **main( )**, an array **a** is created and fed to the constructor for **vector<int> v**. This vector will be used as the target for the search and replace activities, and you'll note that there are duplicate elements – these will be discovered by some of the search/replace routines.

The first test demonstrates **find( )**, discovering the value 4 in **v**. The return value is the iterator pointing to the first instance of 4, or the end of the input range (**v.end( )**) if the search value is not found.

find_if( ) uses a predicate to determine if it has discovered the correct element. In the above example, this predicate is created on the fly using **greater<int>** (that is, "see if the first **int** argument is greater than the second") and **bind2nd( )** to fix the second argument to 8. Thus, it returns true if the value in **v** is greater than 8.

Since there are a number of cases in **v** where two identical objects appear next to each other, the test of **adjacent_find( )** is designed to find them all. It starts looking from the beginning and then drops into a **while** loop, making sure that the iterator **it** has not reached the end of the input sequence (which would mean that no more matches can be found). For each match it finds, the loop prints out the matches and then performs the next **adjacent_find( )**, this time using **it + 2** as the first argument (this way, it moves past the two elements that it already found).

You might look at the **while** loop and think that you can do it a bit more cleverly, to wit:

```
  while(it != v.end()) {
    cout << "adjacent_find: " << *it++
```

```
      << ", " << *it++ << endl;
    it = adjacent_find(it, v.end());
  }
```

Of course, this is exactly what I tried at first. However, I did not get the output I expected, on any compiler. This is because there is no guarantee about when the increments occur in the above expression. A bit of a disturbing discovery, I know, but the situation is best avoided now that you're aware of it.

The next test uses **adjacent_find( )** with the **PlusOne** predicate, which discovers all the places where the next number in the sequence **v** changes from the previous by one. The same **while** approach is used to find all the cases.

find_first_of( ) requires a second range of objects for which to hunt; this is provided in the array **b**. Notice that, because the first range and the second range in **find_first_of( )** are controlled by separate template arguments, those ranges can refer to two different types of containers, as seen here. The second form of **find_first_of( )** is also tested, using **PlusOne**.

search( ) finds exactly the second range inside the first one, with the elements in the same order. The second form of **search( )** uses a predicate, which is typically just something that defines equivalence, but it also opens some interesting possibilities – here, the **PlusOne** predicate causes the range **{ 4, 5, 6 }** to be found.

The **find_end( )** test discovers the *last* occurrence of the entire sequence **{ 11, 11, 11 }**. To show that it has in fact found the last occurrence, the rest of **v** starting from **it** is printed.

The first **search_n( )** test looks for 3 copies of the value 7, which it finds and prints. When using the second version of **search_n( )**, the predicate is ordinarily meant to be used to determine equivalence between two elements, but I've taken some liberties and used a function object that multiplies the value in the sequence by (in this case) 15 and checks to see if it's greater than 100. That is, the **search_n( )** test above says "find me 6 consecutive values which, when multiplied by 15, each produce a number greater than 100." Not exactly what you normally expect to do, but it might give you some ideas the next time you have an odd searching problem.

min_element( ) and **max_element( )** are straightforward; the only thing that's a bit odd is that it looks like the function is being dereferenced

with a '*****'. Actually, the returned iterator is being dereferenced to produce the value for printing.

To test replacements, **replace_copy( )** is used first (so it doesn't modify the original vector) to replace all values of 8 with the value 47. Notice the use of **back_inserter( )** with the empty vector **v2**. To demonstrate **replace_if( )**, a function object is created using the standard template **greater_equal** along with **bind2nd** to replace all the values that are greater than or equal to 7 with the value -1.

Comparing ranges

These algorithms provide ways to compare two ranges. At first glance, the operations they perform seem very close to the **search( )** function above. However, **search( )** tells you where the second sequence appears within the first, while **equal( )** and **lexicographical_compare( )** simply tell you whether or not two sequences are exactly identical (using different comparison algorithms). On the other hand, **mismatch( )** does tell you where the two sequences go out of sync, but those sequences must be exactly the same length.

```
bool equal(InputIterator first1, InputIterator last1, InputIterator
first2);
bool equal(InputIterator first1, InputIterator last1, InputIterator
first2
   BinaryPredicate binary_pred);
```

In both of these functions, the first range is the typical one, **[first1, last1)**. The second range starts at **first2**, but there is no "last2" because its length is determined by the length of the first range. The **equal( )** function returns true if both ranges are exactly the same (the same elements in the same order); in the first case, the **operator==** is used to perform the comparison and in the second case **binary_pred** is used to decide if two elements are the same.

```
bool lexicographical_compare(InputIterator1 first1,
InputIterator1 last1
   InputIterator2 first2, InputIterator2 last2);
bool lexicographical_compare(InputIterator1 first1,
InputIterator1 last1
   InputIterator2 first2, InputIterator2 last2, BinaryPredicate
binary_pred);
```

These two functions determine if the first range is “lexicographically less” than the second (they return **true** if range 1 is less than range 2, and false otherwise. Lexicographical equality, or “dictionary” comparison, means that the comparison is done the same way we establish the order of strings in a dictionary, one element at a time. The first elements determine the result if these elements are different, but if they’re equal the algorithm moves on to the next elements and looks at those, and so on. until it finds a mismatch. At that point it looks at the elements, and if the element from range 1 is less than the element from range two, then **lexicographical_compare( )** returns **true**, otherwise it returns **false**. If it gets all the way through one range or the other (the ranges may be different lengths for this algorithm) without finding an inequality, then range 1 is *not* less than range 2 so the function returns **false**.

If the two ranges are different lengths, a missing element in one range acts as one that “precedes” an element that exists in the other range. So {‘a’, ‘b’} lexicographically precedes {‘a’, ‘b’, ‘a’ }.

In the first version of the function, **operator<** is used to perform the comparisons, and in the second version **binary_pred** is used.

---

pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2);
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred);

As in **equal( )**, the length of both ranges is exactly the same, so only the first iterator in the second range is necessary, and the length of the first range is used as the length of the second range. Whereas **equal( )** just tells you whether or not the two ranges are the same, **mismatch( )** tells you where they begin to differ. To accomplish this, you must be told (1) the element in the first range where the mismatch occurred and (2) the element in the second range where the mismatch occurred. These two iterators are packaged together into a **pair** object and returned. If no mismatch occurs, the return value is **last1** combined with the past-the-end iterator of the second range.

As in **equal( )**, the first function tests for equality using **operator==** while the second one uses **binary_pred**.

Example

Because the standard C++ **string** class is built like a container (it has **begin( )** and **end( )** member functions which produce objects of type **string::iterator**), it can be used to conveniently create ranges of characters to test with the STL comparison algorithms. However, you should note that **string** has a fairly complete set of native operations, so you should look at the **string** class before using the STL algorithms to perform operations.

```
//: C21:Comparison.cpp
// The STL range comparison algorithms
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
#include <functional>
#include <string>
using namespace std;

int main() {
  // strings provide a convenient way to create
  // ranges of characters, but you should
  // normally look for native string operations:
  string s1("This is a test");
  string s2("This is a Test");
  cout << "s1: " << s1 << endl
    << "s2: " << s2 << endl;
  cout << "compare s1 & s1: "
    << equal(s1.begin(), s1.end(), s1.begin())
    << endl;
  cout << "compare s1 & s2: "
    << equal(s1.begin(), s1.end(), s2.begin())
    << endl;
  cout << "lexicographical_compare s1 & s1: " <<
    lexicographical_compare(s1.begin(), s1.end(),
      s1.begin(), s1.end()) <<  endl;
  cout << "lexicographical_compare s1 & s2: " <<
    lexicographical_compare(s1.begin(), s1.end(),
      s2.begin(), s2.end()) << endl;
  cout << "lexicographical_compare s2 & s1: " <<
    lexicographical_compare(s2.begin(), s2.end(),
      s1.begin(), s1.end()) << endl;
  cout << "lexicographical_compare shortened "
```

```
    "s1 & full-length s2: " << endl;
  string s3(s1);
  while(s3.length() != 0) {
    bool result = lexicographical_compare(
      s3.begin(), s3.end(), s2.begin(),s2.end());
    cout << s3 << endl << s2 << ", result = "
      << result << endl;
    if(result == true) break;
    s3 = s3.substr(0, s3.length() - 1);
  }
  pair<string::iterator, string::iterator> p =
    mismatch(s1.begin(), s1.end(), s2.begin());
  print(p.first, s1.end(), "p.first", "");
  print(p.second, s2.end(), "p.second","");
} ///:~
```

Note that the only difference between **s1** and **s2** is the capital 'T' in **s2**'s "Test." Comparing **s1** and **s1** for equality yields **true**, as expected, while **s1** and **s2** are not equal because of the capital 'T'.

To understand the output of the **lexicographical_compare( )** tests, you must remember two things: first, the comparison is performed character-by-character, and second that capital letters "precede" lowercase letters. In the first test, **s1** is compared to **s1**. These are exactly equivalent, thus one is *not* lexicographically less than the other (which is what the comparison is looking for) and thus the result is **false**. The second test is asking "does **s1** precede **s2**?" When the comparison gets to the 't' in "test", it discovers that the lowercase 't' in **s1** is "greater" than the uppercase 'T' in **s2**, so the answer is again **false**. However, if we test to see whether **s2** precedes **s1**, the answer is **true**.

To further examine lexicographical comparison, the next test in the above example compares **s1** with **s2** again (which returned **false** before). But this time it repeats the comparison, trimming one character off the end of **s1** (which is first copied into **s3**) each time through the loop until the test evaluates to **true**. What you'll see is that, as soon as the uppercase 'T' is trimmed off of **s3** (the copy of **s1**), then the characters, which are exactly equal up to that point, no longer count and the fact that **s3** is shorter than **s2** is what makes it lexicographically precede **s2**.

The final test uses **mismatch( )**. In order to capture the return value, you must first create the appropriate **pair p**, constructing the template using the iterator type from the first range and the iterator type from the second range (in this case, both **string::iterator**s). To print the results,

the iterator for the mismatch in the first range is **p.first**, and for the second range is **p.second**. In both cases, the range is printed from the mismatch iterator to the end of the range so you can see exactly where the iterator points.

Removing elements

Because of the genericity of the STL, the concept of removal is a bit constrained. Since elements can only be "removed" via iterators, and iterators can point to arrays, vectors, lists, etc., it is not safe or reasonable to actually try to destroy the elements that are being removed, and to change the size of the input range **[first, last)** (an array, for example, cannot have its size changed). So instead, what the STL "remove" functions do is rearrange the sequence so that the "removed" elements are at the end of the sequence, and the "un-removed" elements are at the beginning of the sequence (in the same order that they were before, minus the removed elements – that is, this is a *stable* operation). Then the function will return an iterator to the "new last" element of the sequence, which is the end of the sequence without the removed elements and the beginning of the sequence of the removed elements. In other words, if **new_last** is the iterator that is returned from the "remove" function, then **[first, new_last)** is the sequence without any of the removed elements, and **[new_last, last)** is the sequence of removed elements.

If you are simply using your sequence, including the removed elements, with more STL algorithms, you can just use **new_last** as the new past-the-end iterator. However, if you're using a resizable container **c** (not an array) and you actually want to eliminate the removed elements from the container you can use **erase( )** to do so, for example:

```
c.erase(remove(c.begin(), c.end(), value), c.end());
```

The return value of **remove( )** is the **new_last** iterator, so **erase( )** will delete all the removed elements from **c**.

The iterators in **[new_last, last)** are dereferenceable but the element values are undefined and should not be used.

ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
Predicate pred);
OutputIterator remove_copy(InputIterator first, InputIterator

last,
OutputIterator result, const T& value);
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
OutputIterator result, Predicate pred);

Each of the "remove" forms moves through the range **[first, last)**, finding values that match a removal criterion and copying the un-removed elements over the removed elements (thus effectively removing them). The original order of the un-removed elements is maintained. The return value is an iterator pointing past the end of the range that contains none of the removed elements. The values that this iterator points to are unspecified.

The "if" versions pass each element to **pred( )** to determine whether it should be removed or not (if **pred( )** returns **true**, the element is removed). The "copy" versions do not modify the original sequence, but instead copy the un-removed values into a range beginning at **result**, and return an iterator indicating the past-the-end value of this new range.

ForwardIterator unique(ForwardIterator first, ForwardIterator last);
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
BinaryPredicate binary_pred);
OutputIterator unique_copy(InputIterator first, InputIterator last,
OutputIterator result);
OutputIterator unique_copy(InputIterator first, InputIterator last,
OutputIterator result, BinaryPredicate binary_pred);

Each of the "unique" functions moves through the range **[first, last)**, finding adjacent values that are equivalent (that is, duplicates) and "removing" the duplicate elements by copying over them. The original order of the un-removed elements is maintained. The return value is an iterator pointing past the end of the range that has the adjacent duplicates removed.

Because only duplicates that are adjacent are removed, it's likely that you'll want to call **sort( )** before calling a "unique" algorithm, since that will guarantee that *all* the duplicates are removed.

The versions containing **binary_pred** call, for each iterator value **i** in the input range:

```
binary_pred(*i, *(i-1));
```

and if the result is true then ***(i-1)** is considered a duplicate.

The “copy” versions do not modify the original sequence, but instead copy the un-removed values into a range beginning at **result**, and return an iterator indicating the past-the-end value of this new range.

Example

This example gives a visual demonstration of the way the “remove” and “unique” functions work.

```
//: C21:Removing.cpp
// The removing algorithms
#include "PrintSequence.h"
#include "Generators.h"
#include <vector>
#include <algorithm>
#include <cctype>
using namespace std;

struct IsUpper {
  bool operator()(char c) {
    return isupper(c);
  }
};

int main() {
  vector<char> v(50);
  generate(v.begin(), v.end(), CharGen());
  print(v, "v", "");
  // Create a set of the characters in v:
  set<char> cs(v.begin(), v.end());
  set<char>::iterator it = cs.begin();
  vector<char>::iterator cit;
  // Step through and remove everything:
  while(it != cs.end()) {
    cit = remove(v.begin(), v.end(), *it);
    cout << *it << "[" << *cit << "] ";
    print(v, "", "");
    it++;
  }
  generate(v.begin(), v.end(), CharGen());
  print(v, "v", "");
  cit = remove_if(v.begin(), v.end(), IsUpper());
```

```
  print(v.begin(), cit, "after remove_if", "");
  // Copying versions are not shown for remove
  // and remove_if.
  sort(v.begin(), cit);
  print(v.begin(), cit, "sorted", "");
  vector<char> v2;
  unique_copy(v.begin(), cit, back_inserter(v2));
  print(v2, "unique_copy", "");
  // Same behavior:
  cit = unique(v.begin(), cit, equal_to<char>());
  print(v.begin(), cit, "unique", "");
} ///:~
```

The **vector<char> v** is filled with randomly-generated characters and then copied into a **set**. Each element of the **set** is used in a **remove** statement, but the entire **vector v** is printed out each time so you can see what happens to the rest of the range, after the resulting endpoint (which is stored in **cit**).

To demonstrate **remove_if( )**, the address of the Standard C library function **isupper( )** (in **<cctype>** is called inside of the function object class **IsUpper**, an object of which is passed as the predicate for **remove_if( )**. This only returns **true** if a character is uppercase, so only lowercase characters will remain. Here, the end of the range is used in the call to **print( )** so only the remaining elements will appear. The copying versions of **remove( )** and **remove_if( )** are not shown because they are a simple variation on the non-copying versions which you should be able to use without an example.

The range of lowercase letters is sorted in preparation for testing the "unique" functions (the "unique" functions are not undefined if the range isn't sorted, but it's probably not what you want). First, **unique_copy( )** puts the unique elements into a new **vector** using the default element comparison, and then the form of **unique( )** that takes a predicate is used; the predicate used is the built-in function object **equal_to( )**, which produces the same results as the default element comparison.

Sorting and operations on sorted ranges

There is a significant category of STL algorithms which require that the range they operate on be in sorted order.

There is actually only one “sort” algorithm used in the STL. This algorithm is presumably the fastest one, but the implementor has fairly broad latitude. However, it comes packaged in various flavors depending on whether the sort should be stable, partial or just the regular sort. Oddly enough, only the partial sort has a copying version; otherwise you’ll need to make your own copy before sorting if that’s what you want. If you are working with a very large number of items you may be better off transferring them to an array (or at least a **vector**, which uses an array internally) rather than using them in some of the STL containers.

Once your sequence is sorted, there are many operations you can perform on that sequence, from simply locating an element or group of elements to merging with another sorted sequence or manipulating sequences as mathematical sets.

Each algorithm involved with sorting or operations on sorted sequences has two versions of each function, the first that uses the object’s own **operator<** to perform the comparison, and the second that uses an additional **StrictWeakOrdering** object’s **operator( )(a, b)** to compare two objects for **a < b**. Other than this there are no differences, so the distinction will not be pointed out in the description of each algorithm.

Sorting

One STL container (**list**) has its own built-in **sort( )** function which is almost certainly going to be faster than the generic sort presented here (especially since the **list** sort just swaps pointers rather than copying entire objects around). This means that you’ll only want to use the sort functions here if (a) you’re working with an array or a sequence container that doesn’t have a **sort( )** function or (b) you want to use one of the other sorting flavors, like a partial or stable sort, which aren’t supported by **list**’s **sort( )**.

---

void sort(RandomAccessIterator first, RandomAccessIterator last);
void sort(RandomAccessIterator first, RandomAccessIterator last, StrictWeakOrdering binary_pred);

Sorts **[first, last)** into ascending order. The second form allows a comparator object to determine the order.

---

void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,

RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Sorts **[first, last)** into ascending order, preserving the original ordering of equivalent elements (this is important if elements can be equivalent but not identical). The second form allows a comparator object to determine the order.

---

void partial_sort(RandomAccessIterator first,
RandomAccessIterator middle, RandomAccessIterator last);
void partial_sort(RandomAccessIterator first,
RandomAccessIterator middle, RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Sorts the number of elements from **[first, last)** that can be placed in the range **[first, middle)**. The rest of the elements end up in **[middle, last)**, and have no guaranteed order. The second form allows a comparator object to determine the order.

RandomAccessIterator partial_sort_copy(InputIterator first,
InputIterator last,
RandomAccessIterator result_first, RandomAccessIterator
result_last);
RandomAccessIterator partial_sort_copy(InputIterator first,
InputIterator last, RandomAccessIterator result_first,
RandomAccessIterator result_last, StrictWeakOrdering
binary_pred);

Sorts the number of elements from **[first, last)** that can be placed in the range **[result_first, result_last)**, and copies those elements into **[result_first, result_last)**. If the range **[first, last)** is smaller than **[result_first, result_last)**, then the smaller number of elements is used. The second form allows a comparator object to determine the order.

---

void nth_element(RandomAccessIterator first,
RandomAccessIterator nth, RandomAccessIterator last);
void nth_element(RandomAccessIterator first,
RandomAccessIterator nth, RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Just like **partial_sort( )**, **nth_element( )** partially orders a range of elements. However, it's much "less ordered" than **partial_sort( )**. The only thing that **nth_element( )** guarantees is that whatever *location* you choose will become a dividing point. All the elements in the range **[first, nth)** will be less than (they could also be equivalent to) whatever element ends up at location **nth** and all the elements in the range **(nth, last]** will

be greater than whatever element ends up location **nth**. However, neither range is in any particular order, unlike **partial_sort( )** which has the first range in sorted order.

If all you need is this very weak ordering (if, for example, you're determining medians, percentiles and that sort of thing) this algorithm is faster than **partial_sort( )**.

Example

The **StreamTokenizer** class from the previous chapter is used to break a file into words, and each word is turned into an **NString** and added to a **deque<NString>**. Once the input file is completely read, a **vector<NString>** is created from the contents of the **deque**. The **vector** is then used to demonstrate the sorting algorithms:

```
//: C21:SortTest.cpp
//{L} ../C20/StreamTokenizer
// Test different kinds of sorting
#include "../C20/StreamTokenizer.h"
#include "NString.h"
#include "PrintSequence.h"
#include "Generators.h"
#include "../require.h"
#include <algorithm>
#include <fstream>
#include <queue>
#include <vector>
#include <cctype>
using namespace std;

// For sorting NStrings and ignore string case:
struct NoCase {
  bool operator()(
    const NString& x, const NString& y) {
/* Somthing's wrong with this approach but I
   can't seem to see it. It would be much faster:
    const string& lv = x;
    const string& rv = y;
    int len = min(lv.size(), rv.size());
    for(int i = 0; i < len; i++)
      if(tolower(lv[i]) < tolower(rv[i]))
        return true;
    return false;
```

```
  }
*/
    // Brute force: copy, force to lowercase:
    string lv(x);
    string rv(y);
    lcase(lv);
    lcase(rv);
    return lv < rv;
  }
  void lcase(string& s) {
    int n = s.size();
    for(int i = 0; i < n; i++)
      s[i] = tolower(s[i]);
  }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  StreamTokenizer words(in);
  deque<NString> nstr;
  string word;
  while((word = words.next()).size() != 0)
    nstr.push_back(NString(word));
  print(nstr);
  // Create a vector from the contents of nstr:
  vector<NString> v(nstr.begin(), nstr.end());
  sort(v.begin(), v.end());
  print(v, "sort");
  // Use an additional comparator object:
  sort(v.begin(), v.end(), NoCase());
  print(v, "sort NoCase");
  copy(nstr.begin(), nstr.end(), v.begin());
  stable_sort(v.begin(), v.end());
  print(v, "stable_sort");
  // Use an additional comparator object:
  stable_sort(v.begin(), v.end(),
    greater<NString>());
  print(v, "stable_sort greater");
  copy(nstr.begin(), nstr.end(), v.begin());
  // Partial sorts. The additional comparator
```

```
  // versions are obvious and not shown here.
  partial_sort(v.begin(),
    v.begin() + v.size()/2, v.end());
  print(v, "partial_sort");
  // Create a vector with a preallocated size:
  vector<NString> v2(v.size()/2);
  partial_sort_copy(v.begin(), v.end(),
    v2.begin(), v2.end());
  print(v2, "partial_sort_copy");
  // Finally, the weakest form of ordering:
  vector<int> v3(20);
  generate(v3.begin(), v3.end(), URandGen(50));
  print(v3, "v3 before nth_element");
  int n = 10;
  vector<int>::iterator vit = v3.begin() + n;
  nth_element(v3.begin(), vit, v3.end());
  cout << "After ordering with nth = " << n
    << ", nth element is " << v3[n] << endl;
  print(v3, "v3 after nth_element");
} ///:~
```

The first class is a binary predicate used to compare two **NString** objects while ignoring the case of the **string**s. You can pass the object into the various sort routines to produce an alphabetic sort (rather than the default lexicographic sort, which has all the capital letters in one group, followed by all the lowercase letters).

As an example, try the source code for the above file as input. Because the occurrence numbers are printed along with the strings you can distinguish between an ordinary sort and a stable sort, and you can also see what happens during a partial sort (the remaining unsorted elements are in no particular order). There is no “partial stable sort.”

You’ll notice that the use of the second “comparator” forms of the functions are not exhaustively tested in the above example, but the use of a comparator is the same as in the first part of the example.

The test of **nth_element** does not use the **NString** objects because it’s simpler to see what’s going on if **int**s are used. Notice that, whatever the nth element turns out to be (which will vary from one run to another because of **URandGen**), the elements before that are less, and after that are greater, but the elements have no particular order other than that. Because of **URandGen**, there are no duplicates but if you use a generator

that allows duplicates you can see that the elements before the nth element will be less than or equal to the nth element.

Locating elements in sorted ranges

Once a range is sorted, there are a group of operations that can be used to find elements within those ranges. In the following functions, there are always two forms, one that assumes the intrinsic **operator<** has been used to perform the sort, and the second that must be used if some other comparison function object has been used to perform the sort. You must use the same comparison for locating elements as you do to perform the sort, otherwise the results are undefined. In addition, if you try to use these functions on unsorted ranges the results will be undefined.

---

bool binary_search(ForwardIterator first, ForwardIterator last, const T& value);
bool binary_search(ForwardIterator first, ForwardIterator last, const T& value,
StrictWeakOrdering binary_pred);

Tells you whether **value** appears in the sorted range **[first, last)**.

ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
const T& value);
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
const T& value, StrictWeakOrdering binary_pred);

Returns an iterator indicating the first occurrence of **value** in the sorted range **[first, last)**. Returns **last** if **value** is not found.

ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, StrictWeakOrdering binary_pred);

Returns an iterator indicating one past the last occurrence of **value** in the sorted range **[first, last)**. Returns **last** if **value** is not found.

---

pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last,
const T& value);
pair<ForwardIterator, ForwardIterator>

---

equal_range(ForwardIterator first, ForwardIterator last, const T& value, StrictWeakOrdering binary_pred);

Essentially combines **lower_bound( )** and **upper_bound( )** to return a **pair** indicating the first and one-past-the-last occurrences of **value** in the sorted range **[first, last)**. Both iterators indicate **last** if **value** is not found.

Example

Here, we can use the approach from the previous example:

```
//: C21:SortedSearchTest.cpp
//{L} ../C20/StreamTokenizer
// Test searching in sorted ranges
#include "../C20/StreamTokenizer.h"
#include "PrintSequence.h"
#include "NString.h"
#include "../require.h"
#include <algorithm>
#include <fstream>
#include <queue>
#include <vector>
using namespace std;

int main() {
  ifstream in("SortedSearchTest.cpp");
  assure(in, "SortedSearchTest.cpp");
  StreamTokenizer words(in);
  deque<NString> dstr;
  string word;
  while((word = words.next()).size() != 0)
    dstr.push_back(NString(word));
  vector<NString> v(dstr.begin(), dstr.end());
  sort(v.begin(), v.end());
  print(v, "sorted");
  typedef vector<NString>::iterator sit;
  sit it, it2;
  string f("include");
  cout << "binary search: "
    << binary_search(v.begin(), v.end(), f)
    << endl;
  it = lower_bound(v.begin(), v.end(), f);
  it2 = upper_bound(v.begin(), v.end(), f);
```

```
  print(it, it2, "found range");
  pair<sit, sit> ip =
    equal_range(v.begin(), v.end(), f);
  print(ip.first, ip.second,
    "equal_range");
} ///:~
```

The input is forced to be the source code for this file because the word "include" will be used for a find string (since "include" appears many times). The file is tokenized into words that are placed into a **deque** (a better container when you don't know how much storage to allocate), and left unsorted in the **deque**. The **deque** is copied into a **vector** via the appropriate constructor, and the **vector** is sorted and printed.

The **binary_search( )** function only tells you if the object is there or not; **lower_bound( )** and **upper_bound( )** produce iterators to the beginning and ending positions where the matching objects appear. The same effect can be produced more succinctly using **equal_range( )** (as shown in the previous chapter, with **multimap** and **multiset**).

Merging sorted ranges

As before, the first form of each function assumes the intrinsic **operator<** has been used to perform the sort. The second form must be used if some other comparison function object has been used to perform the sort. You must use the same comparison for locating elements as you do to perform the sort, otherwise the results are undefined. In addition, if you try to use these functions on unsorted ranges the results will be undefined.

OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
** InputIterator2 first2, InputIterator2 last2, OutputIterator result);**
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
** InputIterator2 first2, InputIterator2 last2, OutputIterator result,**
** StrictWeakOrdering binary_pred);**

Copies elements from **[first1, last1)** and **[first2, last2)** into **result**, such that the resulting range is sorted in ascending order. This is a stable operation.

---

void inplace_merge(BidirectionalIterator first,
** BidirectionalIterator middle, BidirectionalIterator last);**
void inplace_merge(BidirectionalIterator first,

BidirectionalIterator middle, BidirectionalIterator last, StrictWeakOrdering binary_pred);

This assumes that **[first, middle)** and **[middle, last)** are each sorted ranges. The two ranges are merged so that the resulting range **[first, last)** contains the combined ranges in sorted order.

Example

It's easier to see what goes on with merging if **int**s are used; the following example also emphasizes how the algorithms (and my own **print** template) work with arrays as well as containers.

```
//: C21:MergeTest.cpp
// Test merging in sorted ranges
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  const int sz = 15;
  int a[sz*2] = {0};
  // Both ranges go in the same array:
  generate(a, a + sz, SkipGen(0, 2));
  generate(a + sz, a + sz*2, SkipGen(1, 3));
  print(a, a + sz, "range1", " ");
  print(a + sz, a + sz*2, "range2", " ");
  int b[sz*2] = {0}; // Initialize all to zero
  merge(a, a + sz, a + sz, a + sz*2, b);
  print(b, b + sz*2, "merge", " ");
  // set_union is a merge that removes duplicates
  set_union(a, a + sz, a + sz, a + sz*2, b);
  print(b, b + sz*2, "set_union", " ");
  inplace_merge(a, a + sz, a + sz*2);
  print(a, a + sz*2, "inplace_merge", " ");
} ///:~
```

In **main( )**, instead of creating two separate arrays both ranges will be created end-to-end in the same array **a** (this will come in handy for the **inplace_merge**). The first call to **merge( )** places the result in a different array, **b**. For comparison, **set_union( )** is also called, which has the same signature and similar behavior, except that it removes the duplicates. Finally, **inplace_merge( )** is used to combine both parts of **a**.

Set operations on sorted ranges

Once ranges have been sorted, you can perform mathematical set operations on them.

---

bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
bool includes (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
StrictWeakOrdering binary_pred);

Returns **true** if **[first2, last2)** is a subset of **[first1, last1)**. Neither range is required to hold only unique elements, but if **[first2, last2)** holds **n** elements of a particular value, then **[first1, last1)** must also hold **n** elements if the result is to be **true**.

OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result);
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result,
StrictWeakOrdering binary_pred);

Creates the mathematical union of two sorted ranges in the **result** range, returning the end of the output range. Neither input range is required to hold only unique elements, but if a particular value appears multiple times in both input sets, then the resulting set will contain the larger number of identical values.

OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result);
OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result,
StrictWeakOrdering binary_pred);

Produces, in **result**, the intersection of the two input sets, returning the end of the output range. That is, the set of values that appear in both input sets. Neither input range is required to hold only unique elements,

but if a particular value appears multiple times in both input sets, then the resulting set will contain the smaller number of identical values.

OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result,
StrictWeakOrdering binary_pred);

Produces, in **result**, the mathematical set difference, returning the end of the output range. All the elements that are in **[first1, last1)** but not in **[first2, last2)** are placed in the result set. Neither input range is required to hold only unique elements, but if a particular value appears multiple times in both input sets (**n** times in set 1 and **m** times in set 2), then the resulting set will contain **max(n-m, 0)** copies of that value.

OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);
OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, StrictWeakOrdering binary_pred);

Constructs, in **result**, the set containing:

- All the elements in set 1 that are not in set 2
- All the elements in set 2 that are not in set 1.

Neither input range is required to hold only unique elements, but if a particular value appears multiple times in both input sets (**n** times in set 1 and **m** times in set 2), then the resulting set will contain **abs(n-m)** copies of that value, where **abs( )** is the absolute value. The return value is the end of the output range

Example

It's easiest to see the set operations demonstrated using simple vectors of characters, so you view the sets more easily. These characters are randomly generated and then sorted, but the duplicates are not removed so you can see what the set operations do when duplicates are involved.

```
//: C21:SetOperations.cpp
// Set operations on sorted ranges
```

```
#include <vector>
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  vector<char> v(50), v2(50);
  CharGen g;
  generate(v.begin(), v.end(), g);
  generate(v2.begin(), v2.end(), g);
  sort(v.begin(), v.end());
  sort(v2.begin(), v2.end());
  print(v, "v", "");
  print(v2, "v2", "");
  bool b = includes(v.begin(), v.end(),
    v.begin() + v.size()/2, v.end());
  cout << "includes: " <<
    (b ? "true" : "false") << endl;
  vector<char> v3, v4, v5, v6;
  set_union(v.begin(), v.end(),
    v2.begin(), v2.end(), back_inserter(v3));
  print(v3, "set_union", "");
  set_intersection(v.begin(), v.end(),
    v2.begin(), v2.end(), back_inserter(v4));
  print(v4, "set_intersection", "");
  set_difference(v.begin(), v.end(),
    v2.begin(), v2.end(), back_inserter(v5));
  print(v5, "set_difference", "");
  set_symmetric_difference(v.begin(), v.end(),
    v2.begin(), v2.end(), back_inserter(v6));
  print(v6, "set_symmetric_difference","");
} ///:~
```

After **v** and **v2** are generated, sorted and printed, the **includes( )** algorithm is tested by seeing if the entire range of **v** contains the last half of **v**, which of course it does so the result should always be true. The vectors **v3**, **v4**, **v5** and **v6** are created to hold the output of **set_union( )**, **set_intersection( )**, **set_difference( )** and **set_symmetric_difference( )**, and the results of each are displayed so you can ponder them and convince yourself that the algorithms do indeed work as promised.

Heap operations

The heap operations in the STL are primarily concerned with the creation of the STL **priority_queue**, which provides efficient access to the "largest" element, whatever "largest" happens to mean for your program. These were discussed in some detail in the previous chapter, and you can find an example there.

As with the "sort" operations, there are two versions of each function, the first that uses the object's own **operator<** to perform the comparison, the second that uses an additional **StrictWeakOrdering** object's **operator( )(a, b)** to compare two objects for **a < b**.

---

void make_heap(RandomAccessIterator first,
RandomAccessIterator last);
void make_heap(RandomAccessIterator first,
RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Turns an arbitrary range into a heap. A heap is just a range that is organized in a particular way.

---

void push_heap(RandomAccessIterator first,
RandomAccessIterator last);
void push_heap(RandomAccessIterator first,
RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Adds the element *(**last-1)** to the heap determined by the range **[first, last-1)**. Yes, it seems like an odd way to do things but remember that the **priority_queue** container presents the nice interface to a heap, as shown in the previous chapter.

---

void pop_heap(RandomAccessIterator first,
RandomAccessIterator last);
void pop_heap(RandomAccessIterator first,
RandomAccessIterator last,
StrictWeakOrdering binary_pred);

Places the largest element (which is actually in ***first**, before the operation, because of the way heaps are defined) into the position ***(last-1)** *and* reorganizes the remaining range so that it's still in heap order. If you simply grabbed ***first**, the next element would not be the next-largest element so you must use **pop_heap( )** if you want to maintain the heap in its proper priority-queue order.

void sort_heap(RandomAccessIterator first,
RandomAccessIterator last);
void sort_heap(RandomAccessIterator first,
RandomAccessIterator last,
StrictWeakOrdering binary_pred);

This could be thought of as the complement of **make_heap( )**, since it takes a range that is in heap order and turns it into ordinary sorted order, so it is no longer a heap. That means that if you call **sort_heap( )** you can no longer use **push_heap( )** or **pop_heap( )** on that range (rather, you can use those functions but they won't do anything sensible). This is not a stable sort.

Applying an operation to each element in a range

These algorithms move through the entire range and perform an operation on each element. They differ in what they do with the results of that operation: **for_each( )** discards the return value of the operation (but returns the function object that has been applied to each element), while **transform( )** places the results of each operation into a destination sequence (which can be the original sequence).

UnaryFunction for_each(InputIterator first, InputIterator last,
UnaryFunction f);

Applies the function object **f** to each element in **[first, last)**, discarding the return value from each individual application of **f**. If **f** is just a function pointer then you are typically not interested in the return value, but if **f** is an object that maintains some internal state it can capture the combined return value of being applied to the range. The final return value of **for_each( )** is **f**.

OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryFunction f);
OutputIterator transform(InputIterator1 first, InputIterator1 last,
InputIterator2 first2, OutputIterator result, BinaryFunction f);

Like **for_each( )**, applies a function object **f** to each element in the range **[first, last)**. However, instead of discarding the result of each function call, **transform( )** copies the result (using **operator=**) into ***result**, incrementing **result** after each copy (the sequence pointed to by **result**

must have enough storage, otherwise you should use an inserter to force insertions instead of assignments).

The first form of **transform( )** simply calls **f( )** and passes it each object from the input range as an argument. The second form passes an object from the first input range and one from the second input range as the two arguments to the binary function **f** (note the length of the second input range is determined by the length of the first). The return value in both cases is the past-the-end iterator for the resulting output range.

Examples

Since much of what you do with objects in a container is to apply an operation to all of those objects, these are fairly important algorithms and merit several illustrations.

First, consider **for_each( )**. This sweeps through the range, pulling out each element and passing it as an argument as it calls whatever function object it's been given. Thus **for_each( )** performs operations that you might normally write out by hand. In **StIshape.cpp**, for example:

```
for(Iter j = shapes.begin();
    j != shapes.end(); j++)
  delete *j;
```

If you look in your compiler's header file at the template defining **for_each( )**, you'll see something like this:

```
template <class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last,
                  Function f) {
  while (first != last) f(*first++);
  return f;
}
```

Function f looks at first like it must be a pointer to a function which takes, as an argument, an object of whatever **InputIterator** selects. However, the above template actually only says that you must be able to call **f** using parentheses and an argument. This is true for a function pointer, but it's also true for a function object – any class that defines the appropriate **operator( )**. The following example shows several different ways this template can be expanded. First, we need a class that keeps track of its objects so we can know that it's being properly destroyed:

```
//: C21:Counted.h
// An object that keeps track of itself
```

```
#ifndef COUNTED_H
#define COUNTED_H
#include <vector>
#include <iostream>

class Counted {
  static int count;
  char* ident;
public:
  Counted(char* id) : ident(id) { count++; }
  ~Counted() {
    std::cout << ident << " count = "
      << --count << std::endl;
  }
};

int Counted::count = 0;

class CountedVector :
  public std::vector<Counted*> {
public:
  CountedVector(char* id) {
    for(int i = 0; i < 5; i++)
      push_back(new Counted(id));
  }
};
#endif // COUNTED_H ///:~
```

The **class Counted** keeps a static count of how many **Counted** objects have been created, and tells you as they are destroyed. In addition, each **Counted** keeps a **char*** identifier to make tracking the output easier.

The **CountedVector** is inherited from **vector<Counted*>**, and in the constructor it creates some **Counted** objects, handing each one your desired **char***. The **CountedVector** makes testing quite simple, as you'll see.

```
//: C21:ForEach.cpp
// Use of STL for_each() algorithm
#include "Counted.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
// Simple function:
void destroy(Counted* fp) { delete fp; }

// Function object:
template<class T>
class DeleteT {
public:
  void operator()(T* x) { delete x; }
};

// Template function:
template <class T>
void wipe(T* x) { delete x; }

int main() {
  CountedVector A("one");
  for_each(A.begin(), A.end(), destroy);
  CountedVector B("two");
  for_each(B.begin(),B.end(),DeleteT<Counted>());
  CountedVector C("three");
  for_each(C.begin(), C.end(), wipe<Counted>);
} ///:~
```

In **main( )**, the first approach is the simple pointer-to-function, which works but has the drawback that you must write a new **Destroy** function for each different type. The obvious solution is to make a template, which is shown in the second approach with a templatized function object. On the other hand, approach three also makes sense: template functions work as well.

Since this is obviously something you might want to do a lot, why not create an algorithm to **delete** all the pointers in a container? This was accomplished with the **purge( )** template created in the previous chapter. However, that used explicitly-written code; here, we could use **transform( )**. The value of **transform( )** over **for_each( )** is that **transform( )** assigns the result of calling the function object into a resulting range, which can actually be the input range. That case means a literal transformation for the input range, since each element would be a modification of its previous value. In the above example this would be especially useful since it's more appropriate to assign each pointer to the safe value of zero after calling **delete** for that pointer. **Transform( )** can easily do this:

```
//: C21:Transform.cpp
// Use of STL transform() algorithm
#include "Counted.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template<class T>
T* deleteP(T* x) { delete x; return 0; }

template<class T> struct Deleter {
  T* operator()(T* x) { delete x; return 0; }
};

int main() {
  CountedVector cv("one");
  transform(cv.begin(), cv.end(), cv.begin(),
    deleteP<Counted>);
  CountedVector cv2("two");
  transform(cv2.begin(), cv2.end(), cv2.begin(),
    Deleter<Counted>());
} ///:~
```

This shows both approaches: using a template function or a templatized function object. After the call to **transform( )**, the vector contains zero pointers, which is safer since any duplicate **delete**s will have no effect.

One thing you cannot do is **delete** every pointer in a collection without wrapping the call to **delete** inside a function or an object. That is, you don't want to say something like this:

```
for_each(a.begin(), a.end(), ptr_fun(operator delete));
```

You can say it, but what you'll get is a sequence of calls to the function that releases the storage. You will not get the effect of calling **delete** for each pointer in **a**, however; the destructor will not be called. This is typically not what you want, so you will need wrap your calls to **delete**.

In the previous example of **for_each( )**, the return value of the algorithm was ignored. This return value is the function that is passed in to **for_each( )**. If the function is just a pointer to a function, then the return value is not very useful, but if it is a function object, then that function object may have internal member data that it uses to accumulate information about all the objects that it sees during **for_each( )**.

For example, consider a simple model of inventory. Each **Inventory** object has the type of product it represents (here, single characters will be used for product names), the quantity of that product and the price of each item:

```
//: C21:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
#include <ctime>

class Inventory {
  char item;
  int quantity;
  int value;
public:
  Inventory(char it, int quant, int val)
    : item(it), quantity(quant), value(val) {}
  // Synthesized operator= & copy-constructor OK
  char getItem() const { return item; }
  int getQuantity() const { return quantity; }
  void setQuantity(int q) { quantity = q; }
  int getValue() const { return value; }
  void setValue(int val) { value = val; }
  friend std::ostream& operator<<(
    std::ostream& os, const Inventory& inv) {
    return os << inv.item << ": "
      << "quantity " << inv.quantity
      << ", value " << inv.value;
  }
};

// A generator:
struct InvenGen {
  InvenGen() { std::srand(std::time(0)); }
  Inventory operator()() {
    static char c = 'a';
    int q = std::rand() % 100;
    int v = std::rand() % 500;
    return Inventory(c++, q, v);
  }
};
```

```
#endif // INVENTORY_H ///:~
```

There are member functions to get the item name, and to get and set quantity and value. An **operator<<** prints the **Inventory** object to an **ostream**. There's also a generator that creates objects that have sequentially-labeled items and random quantities and values. Note the use of the return value optimization in **operator( )**.

To find out the total number of items and total value, you can create a function object to use with **for_each( )** that has data members to hold the totals:

```
//: C21:CalcInventory.cpp
// More use of for_each()
#include "Inventory.h"
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
using namespace std;

// To calculate inventory totals:
class InvAccum {
  int quantity;
  int value;
public:
  InvAccum() : quantity(0), value(0) {}
  void operator()(const Inventory& inv) {
    quantity += inv.getQuantity();
    value += inv.getQuantity() * inv.getValue();
  }
  friend ostream&
  operator<<(ostream& os, const InvAccum& ia) {
    return os << "total quantity: "
      << ia.quantity
      << ", total value: " << ia.value;
  }
};

int main() {
  vector<Inventory> vi;
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi, "vi");
  InvAccum ia = for_each(vi.begin(),vi.end(),
    InvAccum());
```

```
  cout << ia << endl;
} ///:~
```

InvAccum's **operator( )** takes a single argument, as required by **for_each( )**. As **for_each( )** moves through its range, it takes each object in that range and passes it to **InvAccum::operator( )**, which performs calculations and saves the result. At the end of this process, **for_each( )** returns the **InvAccum** object which you can then examine; in this case it is simply printed.

You can do most things to the **Inventory** objects using **for_each( )**. For example, if you wanted to increase all the prices by 10%, **for_each( )** could do this handily. But you'll notice that the **Inventory** objects have no way to change the **item** value. The programmers who designed **Inventory** thought this was a good idea, after all, why would you want to change the name of an item? But marketing has decided that they want a "new, improved" look by changing all the item names to uppercase; they've done studies and determined that the new names will boost sales (well, marketing has to have *something* to do …). So **for_each( )** will not work here, but **transform( )** will:

```
//: C21:TransformNames.cpp
// More use of transform()
#include "Inventory.h"
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
#include <cctype>
using namespace std;

struct NewImproved {
  Inventory operator()(const Inventory& inv) {
    return Inventory(toupper(inv.getItem()),
      inv.getQuantity(), inv.getValue());
  }
};

int main() {
  vector<Inventory> vi;
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi, "vi");
  transform(vi.begin(), vi.end(), vi.begin(),
    NewImproved());
  print(vi, "vi");
```

```
} ///:~
```

Notice that the resulting range is the same as the input range, that is, the transformation is performed in-place.

Now suppose that the sales department needs to generate special price lists with different discounts for each item. The original list must stay the same, and there need to be any number of generated special lists. Sales will give you a separate list of discounts for each new list. To solve this problem we can use the second version of **transform( )**:

```
//: C21:SpecialList.cpp
// Using the second version of transform()
#include "Inventory.h"
#include "PrintSequence.h"
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

struct Discounter {
  Inventory operator()(const Inventory& inv,
    float discount) {
    return Inventory(inv.getItem(),
      inv.getQuantity(),
      inv.getValue() * (1 - discount));
  }
};

struct DiscGen {
  DiscGen() { srand(time(0)); }
  float operator()() {
    float r = float(rand() % 10);
    return r / 100.0;
  }
};

int main() {
  vector<Inventory> vi;
  generate_n(back_inserter(vi), 15, InvenGen());
  print(vi, "vi");
  vector<float> disc;
  generate_n(back_inserter(disc), 15, DiscGen());
```

```
  print(disc, "Discounts:");
  vector<Inventory> discounted;
  transform(vi.begin(),vi.end(), disc.begin(),
    back_inserter(discounted), Discounter());
  print(discounted, "discounted");
} ///:~
```

Discounter is a function object that, given an **Inventory** object and a discount percentage, produces a new **Inventory** with the discounted price. **DiscGen** just generates random discount values between 1 and 10 percent to use for testing. In **main( )**, two **vector**s are created, one for **Inventory** and one for discounts. These are passed to **transform( )** along with a **Discounter** object, and **transform( )** fills a new **vector<Inventory>** called **discounted**.

Numeric algorithms

These algorithms are all tucked into the header **<numeric>**, since they are primarily useful for performing numerical calculations.

<numeric>
T accumulate(InputIterator first, InputIterator last, T result);
T accumulate(InputIterator first, InputIterator last, T result,
BinaryFunction f);

The first form is a generalized summation; for each element pointed to by an iterator **i** in **[first, last)**, it performs the operation **result = result + *i**, where **result** is of type **T**. However, the second form is more general; it applies the function **f(result, *i)** on each element ***i** in the range from beginning to end. The value **result** is initialized in both cases by **resultI**, and if the range is empty then **resultI** is returned.

Note the similarity between the second form of **transform( )** and the second form of **accumulate( )**.

<numeric>
T inner_product(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, T init);
T inner_product(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, T init
BinaryFunction1 op1, BinaryFunction2 op2);

Calculates a generalized inner product of the two ranges **[first1, last1)** and **[first2, first2 + (last1 - first1))**. The return value is produced by multiplying the element from the first sequence by the “parallel” element

in the second sequence, and then adding it to the sum. So if you have two sequences {1, 1, 2, 2} and {1, 2, 3, 4} the inner product becomes:

```
(1*1) + (1*2) + (2*3) + (2*4)
```

Which is 17. The **init** argument is the initial value for the inner product; this is probably zero but may be anything and is especially important for an empty first sequence, because then it becomes the default return value. The second sequence must have at least as many elements as the first.

While the first form is very specifically mathematical, the second form is simply a multiple application of functions and could conceivably be used in many other situations. The **op1** function is used in place of addition, and **op2** is used instead of multiplication. Thus, if you applied the second version of **inner_product( )** to the above sequence, the result would be the following operations:

```
init = op1(init, op2(1,1));
init = op1(init, op2(1,2));
init = op1(init, op2(2,3));
init = op1(init, op2(2,4));
```

Thus it's similar to **transform( )** but two operations are performed instead of one.

<numeric>
OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);
OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryFunction op);

Calculates a generalized partial sum. This means that a new sequence is created, beginning at **result**, where each element is the sum of all the elements up to the currently selected element in **[first, last)**. For example, if the original sequence is **{1, 1, 2, 2, 3}** then the generated sequence is **{1, 1 + 1, 1 + 1 + 2, 1 + 1 + 1 + 2 + 2, 1 + 1 + 1 + 2 + 2 + 3}**, that is, **{1, 2, 4, 6, 9}**.

In the second version, the binary function **op** is used instead of the **+** operator to take all the "summation" up to that point and combine it with the new value. For example, if you use **multiplies<int>( )** as the object for the above sequence, the output is **{1, 1, 2, 4, 12}**. Note that the first output value is always the same as the first input value.

The return value is the end of the output range **[result, result + (last - first))**.

<numeric>
OutputIterator adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result);
OutputIterator adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result, BinaryFunction op);

Calculates the differences of adjacent elements throughout the range **[first, last)**. This means that in the new sequence, the value is the value of the difference of the current element and the previous element in the original sequence (the first value is the same). For example, if the original sequence is **{1, 1, 2, 2, 3}**, the resulting sequence is **{1, 1 – 1, 2 – 1, 2 – 2, 3 – 2}**, that is: **{1, 0, 1, 0, 1}**.

The second form uses the binary function **op** instead of the **–** operator to perform the "differencing." For example, if you use **multiplies<int>( )** as the function object for the above sequence, the output is **{1, 1, 2, 4, 6}**.

The return value is the end of the output range **[result, result + (last - first))**.

Example

This program tests all the algorithms in **<numeric>** in both forms, on integer arrays. You'll notice that in the test of the form where you supply the function or functions, the function objects used are the ones that produce the same result as form one so the results produced will be exactly the same. This should also demonstrate a bit more clearly the operations that are going on, and how to substitute your own operations.

```
//: C21:NumericTest.cpp
#include "PrintSequence.h"
#include <numeric>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main() {
  int a[] = { 1, 1, 2, 2, 3, 5, 7, 9, 11, 13 };
  const int asz = sizeof a / sizeof a[0];
  print(a, a + asz, "a", " ");
  int r = accumulate(a, a + asz, 0);
```

```
  cout << "accumulate 1: " << r << endl;
  // Should produce the same result:
  r = accumulate(a, a + asz, 0, plus<int>());
  cout << "accumulate 2: " << r << endl;
  int b[] = { 1, 2, 3, 4, 1, 2, 3, 4, 1, 2 };
  print(b, b + sizeof b / sizeof b[0], "b", " ");
  r = inner_product(a, a + asz, b, 0);
  cout << "inner_product 1: " << r << endl;
  // Should produce the same result:
  r = inner_product(a, a + asz, b, 0,
    plus<int>(), multiplies<int>());
  cout << "inner_product 2: " << r << endl;
  int* it = partial_sum(a, a + asz, b);
  print(b, it, "partial_sum 1", " ");
  // Should produce the same result:
  it = partial_sum(a, a + asz, b, plus<int>());
  print(b, it, "partial_sum 2", " ");
  it = adjacent_difference(a, a + asz, b);
  print(b, it, "adjacent_difference 1"," ");
  // Should produce the same result:
  it = adjacent_difference(a, a + asz, b,
    minus<int>());
  print(b, it, "adjacent_difference 2"," ");
} ///:~
```

Note that the return value of **inner_product( )** and **partial_sum( )** is the past-the-end iterator for the resulting sequence, so it is used as the second iterator in the **print( )** function.

Since the second form of each function allows you to provide your own function object, only the first form of the functions is purely “numeric.” You could conceivably do some things that are not intuitively numeric with something like **inner_product( )**.

General utilities

Finally, here are some basic tools that are used with the other algorithms; you may or may not use them directly yourself.

<utility>
struct pair;
make_pair( );

This was described and used in the previous chapter and in this one. A **pair** is simply a way to package two objects (which may be of different types) together into a single object. This is typically used when you need to return more than one object from a function, but it can also be used to create a container that holds **pair** objects, or to pass more than one object as a single argument. You access the elements by saying **p.first** and **p.second**, where **p** is the **pair** object. The function **equal_range( )**, described in the last chapter and in this one, returns its result as a **pair** of iterators. You can **insert( )** a **pair** directly into a **map** or **multimap**; a **pair** is the **value_type** for those containers.

If you want to create a **pair**, you typically use the template function **make_pair( )** rather than explicitly constructing a **pair** object.

<iterator>
distance(InputIterator first, InputIterator last);

Tells you the number of elements between **first** and **last**. More precisely, it returns an integral value that tells you the number of times **first** must be incremented before it is equal to **last**. No dereferencing of the iterators occurs during this process.

<iterator>
void advance(InputIterator& i, Distance n);

Moves the iterator **i** forward by the value of **n** (the iterator can also be moved backward for negative values of **n** if the iterator is also a bidirectional iterator). This algorithm is aware of bidirectional iterators, and will use the most efficient approach.

<iterator>
back_insert_iterator<Container> back_inserter(Container& x);
front_insert_iterator<Container> front_inserter(Container& x);
insert_iterator<Container> inserter(Container& x, Iterator i);

These functions are used to create iterators for the given containers that will insert elements into the container, rather than overwrite the existing elements in the container using **operator=** (which is the default behavior). Each type of iterator uses a different operation for insertion: **back_insert_iterator** uses **push_back( )**, **front_insert_iterator** uses **push_front( )** and **insert_iterator** uses **insert( )** (and thus it can be used with the associative containers, while the other two can be used with sequence containers). These were shown in some detail in the previous chapter, and also used in this chapter.

const LessThanComparable& min(const LessThanComparable& a, const LessThanComparable& b);
const T& min(const T& a, const T& b, BinaryPredicate binary_pred);

Returns the lesser of its two arguments, or the first argument if the two are equivalent. The first version performs comparisons using **operator<** and the second passes both arguments to **binary_pred** to perform the comparison.

const LessThanComparable& max(const LessThanComparable& a, const LessThanComparable& b);
const T& max(const T& a, const T& b, BinaryPredicate binary_pred);

Exactly like **min( )**, but returns the greater of its two arguments.

void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

Exchanges the values of **a** and **b** using assignment. Note that all container classes use specialized versions of **swap( )** that are typically more efficient than this general version.

iter_swap( ) is a backwards-compatible remnant in the standard; you can just use **swap( )**.

Creating your own STL-style algorithms

Once you become comfortable with the STL algorithm style, you can begin to create your own STL-style algorithms. Because these will conform to the format of all the other algorithms in the STL, they're easy to use for programmers who are familiar with the STL, and thus become a way to "extend the STL vocabulary."

The easiest way to approach the problem is to go to the **<algorithm>** header file and find something similar to what you need, and modify that (virtually all STL implementations provide the code for the templates directly in the header files). For example, an algorithm that stands out by its absence is **copy_if( )** (the closest approximation is **partition( )**), which was used in **Binder1.cpp** at the beginning of this chapter, and in

several other examples in this chapter. This will only copy an element if it satisfies a predicate. Here's an implementation:

```
//: C21:copy_if.h
// Roll your own STL-style algorithm
#ifndef COPY_IF_H
#define COPY_IF_H

template<typename ForwardIter,
  typename OutputIter, typename UnaryPred>
OutputIter copy_if(ForwardIter begin, ForwardIter end,
  OutputIter dest, UnaryPred f) {
  while(begin != end) {
    if(f(*begin))
      *dest++ = *begin;
    begin++;
  }
  return dest;
}
#endif // COPY_IF_H ///:~
```

The return value is the past-the-end iterator for the destination sequence (the copied sequence).

Now that you're comfortable with the ideas of the various iterator types, the actual implementation is quite straightforward. You can imagine creating an entire additional library of your own useful algorithms that follow the format of the STL.

Summary

The goal of this chapter, and the previous one, was to give you a programmer's-depth understanding of the containers and algorithms in the Standard Template Library. That is, to make you aware of and comfortable enough with the STL that you begin to use it on a regular basis (or at least, to think of using it so you can come back here and hunt for the appropriate solution). It is powerful not only because it's a reasonably complete library of tools, but also because it provides a vocabulary for thinking about problem solutions, and because it is a framework for creating additional tools.

Although this chapter and the last did show some examples of creating your own tools, I did not go into the full depth of the theory of the STL

that is necessary to completely understand all the STL nooks and crannies to allow you to create tools more sophisticated than those shown here. I did not do this partially because of space limitations, but mostly because it is beyond the charter of this book; my goal here is to give you practical understanding that will affect your day-to-day programming skills.

There are a number of books dedicated solely to the STL (these are listed in the appendices), but the two that I learned the most from, in terms of the theory necessary for tool creation, were first, *Generic Programming and the STL* by Matthew H. Austern, Addison-Wesley 1999 (this also covers all the SGI extensions, which Austern was instrumental in creating), and second (older and somewhat out of date, but still quite valuable), *C++ Programmer's Guide to the Standard Template Library* by Mark Nelson, IDG press 1995.

Exercises

1. Create a generator that returns the current value of **clock( )** (in **<ctime>**). Create a **list<clock_t>** and fill it with your generator using **generate_n( )**. Remove any duplicates in the list and print it to **cout** using **copy( )**.
2. Modify **Stlshape.cpp** from chapter XXX so that it uses **transform( )** to delete all its objects.
3. Using **transform( )** and **toupper( )** (in **<cctype>**) write a single function call that will convert a **string** to all uppercase letters.
4. Create a **Sum** function object template that will accumulate all the values in a range when used with **for_each( )**.
5. Write an anagram generator that takes a word as a command-line argument and produces all possible permutations of the letters.
6. Write a "sentence anagram generator" that takes a sentence as a command-line argument and produces all possible permutations of the words in the sentence (it leaves the words alone, just moves them around).
7. Create a class hierarchy with a base class **B** and a derived class **D**. Put a **virtual** member function **void f( )** in **B** such that it will print a message indicating that **B**'s **f( )** has been called, and redefine this function for **D** to print a different message. Create a **deque<B*>** and fill it with **B** and **D**

objects. Use **for_each( )** to call **f( )** for each of the objects in your **deque**.

8. Modify **FunctionObjects.cpp** so that it uses **float** instead of **int**.
9. Modify **FunctionObjects.cpp** so that it templatizes the main body of tests so you can choose which type you're going to test (you'll have to pull most of **main( )** out into a separate template function).
10. Using **transform( )**, **toupper( )** and **tolower( )** (in **<ccytpe>**), create two functions such that the first takes a **string** object and returns that **string** with all the letters in uppercase, and the second returns a **string** with all the letters in lowercase.
11. Create a container of containers of **Noisy** objects, and sort them. Now write a template for your sorting test (to use with the three basic sequence containers), and compare the performance of the different container types.
12. Write a program that takes as a command line argument the name of a text file. Open this file and read it a word at a time (hint: use **>>**). Store each word into a **deque<string>**. Force all the words to lowercase, sort them, remove all the duplicates and print the results.
13. Write a program that finds all the words that are in common between two input files, using **set_intersection( )**. Change it to show the words that are not in common, using **set_symmetric_difference( )**.
14. Create a program that, given an integer on the command line, creates a "factorial table" of all the factorials up to and including the number on the command line. To do this, write a generator to fill a **vector<int>**, then use **partial_sum( )** with a standard function object.
15. Modify **CalcInventory.cpp** so that it will find all the objects that have a quantity that's less than a certain amount. Provide this amount as a command-line argument, and use **copy_if( )** and **bind2nd( )** to create the collection of values less than the target value.
16. Create template function objects that perform bitwise operations for **&**, **|**, **^** and **~**. Test these with a **bitset**.
17. Fill a **vector<double>** with numbers representing angles in radians. Using function object composition, take the sine of all the elements in your vector (see **<cmath>**).

18. Create a **map** which is a cosine table where the keys are the angles in degrees and the values are the cosines. Use **transform( )** with **cos( )** (in **<cmath>**) to fill the table.
19. Write a program to compare the speed of sorting a **list** using **list::sort( )** vs. using **std::sort( )** (the STL algorithm version of **sort( )**). Hint: see the timing examples in the previous chapter.
20. Create and test a **logical_xor** function object template to implement a logical exclusive-*or*.
21. Create an STL-style algorithm **transform_if( )** following the first form of **transform( )** which only performs transformations on objects that satisfy a unary predicate.
22. Create an STL-style algorithm which is an overloaded version of **for_each( )** that follows the second form of **transform( )** and takes two input ranges so it can pass the objects of the second input range a to a binary function which it applies to each object of the first range.
23. Create a **Matrix** class which is made from a **vector<vector<int> >**. Provide it with a **friend ostream& operator<<(ostream&, const Matrix&)** to display the matrix. Create the following using the STL algorithms where possible (you may need to look up the mathematical meanings of the matrix operations if you don't remember them): **operator+(const Matrix&, const Matrix&)** for **Matrix** addition, **operator*(const Matrix&, const vector<int>&)** for multiplying a matrix by a vector, and **operator*(const Matrix&, const Matrix&)** for matrix multiplication. Demonstrate each.
24. Templatize the **Matrix** class and associated operations from the previous example so they will work with any appropriate type.

Part 3: Advanced Topics

22: Multiple inheritance

The basic concept of multiple inheritance (MI) sounds simple enough.

[[[Notes:

1. Demo of use of MI, using Greenhouse example and different company's greenhouse controller equipment.
2. Introduce concept of interfaces; toys and "tuckable" interface

]]]

You create a new type by inheriting from more than one base class. The syntax is exactly what you'd expect, and as long as the inheritance diagrams are simple, MI is simple as well.

However, MI can introduce a number of ambiguities and strange situations, which are covered in this chapter. But first, it helps to get a perspective on the subject.

Perspective

Before C++, the most successful object-oriented language was Smalltalk. Smalltalk was created from the ground up as an OO language. It is often referred to as *pure,* whereas C++, because it was built on top of C, is called *hybrid*. One of the design decisions made with Smalltalk was that all classes would be derived in a single hierarchy, rooted in a single base class (called **Object** – this is the model for the *object-based hierarchy*). You cannot create a new class in Smalltalk without inheriting it from an existing class, which is why it takes a certain amount of time to become productive in Smalltalk – you must learn the class library before you can

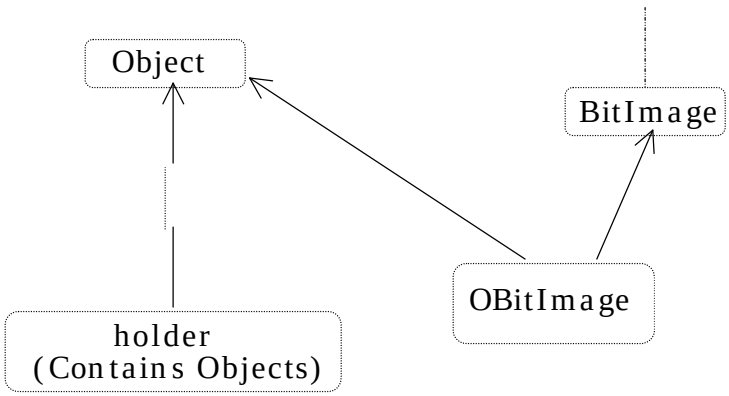
start making new classes. So the Smalltalk class hierarchy is always a single monolithic tree.

Classes in Smalltalk usually have a number of things in common, and always have *some* things in common (the characteristics and behaviors of **Object**), so you almost never run into a situation where you need to inherit from more than one base class. However, with C++ you can create as many hierarchy trees as you want. Therefore, for logical completeness the language must be able to combine more than one class at a time – thus the need for multiple inheritance.

However, this was not a crystal-clear case of a feature that no one could live without, and there was (and still is) a lot of disagreement about whether MI is really essential in C++. MI was added in AT&T **cfront** release 2.0 and was the first significant change to the language. Since then, a number of other features have been added (notably templates) that change the way we think about programming and place MI in a much less important role. You can think of MI as a "minor" language feature that shouldn't be involved in your daily design decisions.

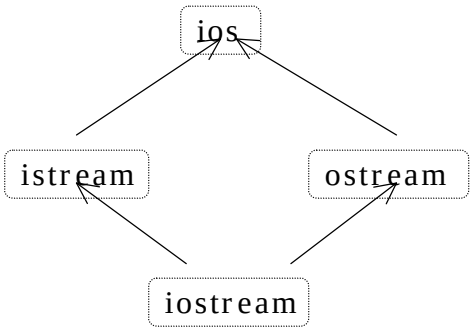
One of the most pressing issues that drove MI involved containers. Suppose you want to create a container that everyone can easily use. One approach is to use **void*** as the type inside the container, as with **PStash** and **Stack**. The Smalltalk approach, however, is to make a container that holds **Object**s. (Remember that **Object** is the base type of the entire Smalltalk hierarchy.) Because everything in Smalltalk is ultimately derived from **Object**, any container that holds **Object**s can hold anything, so this approach works nicely.

Now consider the situation in C++. Suppose vendor **A** creates an object-based hierarchy that includes a useful set of containers including one you want to use called **Holder**. Now you come across vendor **B**'s class hierarchy that contains some other class that is important to you, a **BitImage** class, for example, which holds graphic images. The only way to make a **Holder** of **BitImage**s is to inherit a new class from both **Object**, so it can be held in the **Holder**, and **BitImage**:



This was seen as an important reason for MI, and a number of class libraries were built on this model. However, as you saw in Chapter XX, the addition of templates has changed the way containers are created, so this situation isn't a driving issue for MI.

The other reason you may need MI is logical, related to design. Unlike the above situation, where you don't have control of the base classes, in this one you do, and you intentionally use MI to make the design more flexible or useful. (At least, you may believe this to be the case.) An example of this is in the original iostream library design:

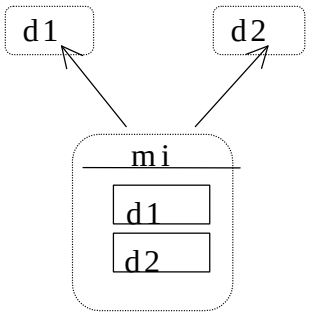


Both **istream** and **ostream** are useful classes by themselves, but they can also be inherited into a class that combines both their characteristics and behaviors.

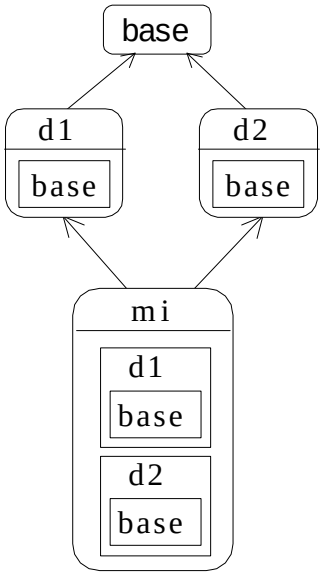
Regardless of what motivates you to use MI, a number of problems arise in the process, and you need to understand them to use it.

Duplicate subobjects

When you inherit from a base class, you get a copy of all the data members of that base class in your derived class. This copy is referred to as a *subobject*. If you multiply inherit from class **d1** and class **d2** into class **mi**, class **mi** contains one subobject of **d1** and one of **d2**. So your **mi** object looks like this:



Now consider what happens if **d1** and **d2** both inherit from the same base class, called **Base**:



In the above diagram, both **d1** and **d2** contain a subobject of **Base**, so **mi** contains *two* subobjects of **Base**. Because of the path produced in the diagram, this is sometimes called a “diamond” in the inheritance

hierarchy. Without diamonds, multiple inheritance is quite straightforward, but as soon as a diamond appears, trouble starts because you have duplicate subobjects in your new class. This takes up extra space, which may or may not be a problem depending on your design. But it also introduces an ambiguity.

Ambiguous upcasting

What happens, in the above diagram, if you want to cast a pointer to an **mi** to a pointer to a **Base**? There are two subobjects of type **Base**, so which address does the cast produce? Here's the diagram in code:

```
//: C22:MultipleInheritance1.cpp
// MI & ambiguity
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class MBase {
public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};

class D1 : public MBase {
public:
  char* vf() const { return "D1"; }
};

class D2 : public MBase {
public:
  char* vf() const { return "D2"; }
};

// Causes error: ambiguous override of vf():
//! class MI : public D1, public D2 {};

int main() {
  vector<MBase*> b;
  b.push_back(new D1);
  b.push_back(new D2);
```

```
  // Cannot upcast: which subobject?:
//!  b.push_back(new mi);
  for(int i = 0; i < b.size(); i++)
    cout << b[i]->vf() << endl;
  purge(b);
} ///:~
```

Two problems occur here. First, you cannot even create the class **mi** because doing so would cause a clash between the two definitions of **vf( )** in **D1** and **D2**.

Second, in the array definition for **b[ ]** this code attempts to create a **new mi** and upcast the address to a **MBase***. The compiler won't accept this because it has no way of knowing whether you want to use **D1**'s subobject **MBase** or **D2**'s subobject **MBase** for the resulting address.

virtual base classes

To solve the first problem, you must explicitly disambiguate the function **vf( )** by writing a redefinition in the class **mi**.

The solution to the second problem is a language extension: The meaning of the **virtual** keyword is overloaded. If you inherit a base class as **virtual**, only one subobject of that class will ever appear as a base class. Virtual base classes are implemented by the compiler with pointer magic in a way suggesting the implementation of ordinary virtual functions.

Because only one subobject of a virtual base class will ever appear during multiple inheritance, there is no ambiguity during upcasting. Here's an example:

```
//: C22:MultipleInheritance2.cpp
// Virtual base classes
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class MBase {
public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};
```

```
class D1 : virtual public MBase {
public:
  char* vf() const { return "D1"; }
};

class D2 : virtual public MBase {
public:
  char* vf() const { return "D2"; }
};

// MUST explicitly disambiguate vf():
class MI : public D1, public D2 {
public:
  char* vf() const { return D1::vf();}
};

int main() {
  vector<MBase*> b;
  b.push_back(new D1);
  b.push_back(new D2);
  b.push_back(new MI); // OK
  for(int i = 0; i < b.size(); i++)
    cout << b[i]->vf() << endl;
  purge(b);
} ///:~
```

The compiler now accepts the upcast, but notice that you must still explicitly disambiguate the function **vf( )** in **MI**; otherwise the compiler wouldn't know which version to use.

The "most derived" class and virtual base initialization

The use of virtual base classes isn't quite as simple as that. The above example uses the (compiler-synthesized) default constructor. If the virtual base has a constructor, things become a bit strange. To understand this, you need a new term: *most-derived* class.

The most-derived class is the one you're currently in, and is particularly important when you're thinking about constructors. In the previous example, **MBase** is the most-derived class inside the **MBase** constructor.

Inside the **D1** constructor, **D1** is the most-derived class, and inside the **MI** constructor, **MI** is the most-derived class.

When you are using a virtual base class, the most-derived constructor is responsible for initializing that virtual base class. That means any class, no matter how far away it is from the virtual base, is responsible for initializing it. Here's an example:

```
//: C22:MultipleInheritance3.cpp
// Virtual base initialization
// Virtual base classes must always be
// Initialized by the "most-derived" class
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class MBase {
public:
  MBase(int) {}
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};

class D1 : virtual public MBase {
public:
  D1() : MBase(1) {}
  char* vf() const { return "D1"; }
};

class D2 : virtual public MBase {
public:
  D2() : MBase(2) {}
  char* vf() const { return "D2"; }
};

class MI : public D1, public D2 {
public:
  MI() : MBase(3) {}
  char* vf() const {
    return D1::vf(); // MUST disambiguate
  }
};
```

```
class X : public MI {
public:
  // You must ALWAYS init the virtual base:
  X() : MBase(4) {}
};

int main() {
  vector<MBase*> b;
  b.push_back(new D1);
  b.push_back(new D2);
  b.push_back(new MI); // OK
  b.push_back(new X);
  for(int i = 0; i < b.size(); i++)
    cout << b[i]->vf() << endl;
  purge(b);
} ///:~
```

As you would expect, both **D1** and **D2** must initialize **MBase** in their constructor. But so must **MI** and **X**, even though they are more than one layer away! That's because each one in turn becomes the most-derived class. The compiler can't know whether to use **D1**'s initialization of **MBase** or to use **D2**'s version. Thus you are always forced to do it in the most-derived class. Note that only the single selected virtual base constructor is called.

"Tying off" virtual bases with a default constructor

Forcing the most-derived class to initialize a virtual base that may be buried deep in the class hierarchy can seem like a tedious and confusing task to put upon the user of your class. It's better to make this invisible, which is done by creating a default constructor for the virtual base class, like this:

```
//: C22:MultipleInheritance4.cpp
// "Tying off" virtual bases
// so you don't have to worry about them
// in derived classes
#include "../purge.h"
#include <iostream>
#include <vector>
```

```
using namespace std;

class MBase {
public:
 // Default constructor removes responsibility:
  MBase(int = 0) {}
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};

class D1 : virtual public MBase {
public:
  D1() : MBase(1) {}
  char* vf() const { return "D1"; }
};

class D2 : virtual public MBase {
public:
  D2() : MBase(2) {}
  char* vf() const { return "D2"; }
};

class MI : public D1, public D2 {
public:
  MI() {} // Calls default constructor for MBase
  char* vf() const {
    return D1::vf(); // MUST disambiguate
  }
};

class X : public MI {
public:
  X() {} // Calls default constructor for MBase
};

int main() {
  vector<MBase*> b;
  b.push_back(new D1);
  b.push_back(new D2);
  b.push_back(new MI); // OK
  b.push_back(new X);
  for(int i = 0; i < b.size(); i++)
```

```
    cout << b[i]->vf() << endl;
  purge(b);
} ///:~
```

If you can always arrange for a virtual base class to have a default constructor, you'll make things much easier for anyone who inherits from that class.

Overhead

The term "pointer magic" has been used to describe the way virtual inheritance is implemented. You can see the physical overhead of virtual inheritance with the following program:

```
//: C22:Overhead.cpp
// Virtual base class overhead
#include <fstream>
using namespace std;
ofstream out("overhead.out");

class MBase {
public:
  virtual void f() const {};
  virtual ~MBase() {}
};

class NonVirtualInheritance
  : public MBase {};

class VirtualInheritance
  : virtual public MBase {};

class VirtualInheritance2
  : virtual public MBase {};

class MI
  : public VirtualInheritance,
    public VirtualInheritance2 {};

#define WRITE(ARG) \
out << #ARG << " = " << ARG << endl;
```

```
int main() {
  MBase b;
  WRITE(sizeof(b));
  NonVirtualInheritance nonv_inheritance;
  WRITE(sizeof(nonv_inheritance));
  VirtualInheritance v_inheritance;
  WRITE(sizeof(v_inheritance));
  MI mi;
  WRITE(sizeof(mi));
} ///:~
```

Each of these classes only contains a single byte, and the "core size" is that byte. Because all these classes contain virtual functions, you expect the object size to be bigger than the core size by a pointer (at least – your compiler may also pad extra bytes into an object for alignment). The results are a bit surprising (these are from one particular compiler; yours may do it differently):

```
sizeof(b) = 2
sizeof(nonv_inheritance) = 2
sizeof(v_inheritance) = 6
sizeof(MI) = 12
```

Both **b** and **nonv_inheritance** contain the extra pointer, as expected. But when virtual inheritance is added, it would appear that the VPTR plus *two extra pointers* are added! By the time the multiple inheritance is performed, the object appears to contain five extra pointers (however, one of these is probably a second VPTR for the second multiply inherited subobject).

The curious can certainly probe into your particular implementation and look at the assembly language for member selection to determine exactly what these extra bytes are for, and the cost of member selection with multiple inheritance[64]. The rest of you have probably seen enough to guess that quite a bit more goes on with virtual multiple inheritance, so it should be used sparingly (or avoided) when efficiency is an issue.

[64] See also Jan Gray, *"C++ Under the Hood"*, a chapter in *Black Belt C++* (edited by Bruce Eckel, M&T Press, 1995).

Upcasting

When you embed subobjects of a class inside a new class, whether you do it by creating member objects or through inheritance, each subobject is placed within the new object by the compiler. Of course, each subobject has its own **this** pointer, and as long as you're dealing with member objects, everything is quite straightforward. But as soon as multiple inheritance is introduced, a funny thing occurs: An object can have more than one **this** pointer because the object represents more than one type during upcasting. The following example demonstrates this point:

```
//: C22:Mithis.cpp
// MI and the "this" pointer
#include <fstream>
using namespace std;
ofstream out("mithis.out");

class Base1 {
  char c[0x10];
public:
  void printthis1() {
    out << "Base1 this = " << this << endl;
  }
};

class Base2 {
  char c[0x10];
public:
  void printthis2() {
    out << "Base2 this = " << this << endl;
  }
};

class Member1 {
  char c[0x10];
public:
  void printthism1() {
    out << "Member1 this = " << this << endl;
  }
};

class Member2 {
```

```
  char c[0x10];
public:
  void printthism2() {
    out << "Member2 this = " << this << endl;
  }
};

class MI : public Base1, public Base2 {
  Member1 m1;
  Member2 m2;
public:
  void printthis() {
    out << "MI this = " << this << endl;
    printthis1();
    printthis2();
    m1.printthism1();
    m2.printthism2();
  }
};

int main() {
  MI mi;
  out << "sizeof(mi) = "
    << hex << sizeof(mi) << " hex" << endl;
  mi.printthis();
  // A second demonstration:
  Base1* b1 = &mi; // Upcast
  Base2* b2 = &mi; // Upcast
  out << "Base 1 pointer = " << b1 << endl;
  out << "Base 2 pointer = " << b2 << endl;
} ///:~
```

The arrays of bytes inside each class are created with hexadecimal sizes, so the output addresses (which are printed in hex) are easy to read. Each class has a function that prints its **this** pointer, and these classes are assembled with both multiple inheritance and composition into the class **MI,** which prints its own address and the addresses of all the other subobjects. This function is called in **main( )**. You can clearly see that you get two different **this** pointers for the same object. The address of the **MI** object is taken and upcast to the two different types. Here's the output:[65]

[65] For easy readability the code was generated for a small-model Intel processor.

```
sizeof(mi) = 40 hex
mi this = 0x223e
Base1 this = 0x223e
Base2 this = 0x224e
Member1 this = 0x225e
Member2 this = 0x226e
Base 1 pointer = 0x223e
Base 2 pointer = 0x224e
```

Although object layouts vary from compiler to compiler and are not specified in Standard C++, this one is fairly typical. The starting address of the object corresponds to the address of the first class in the base-class list. Then the second inherited class is placed, followed by the member objects in order of declaration.

When the upcast to the **Base1** and **Base2** pointers occur, you can see that, even though they're ostensibly pointing to the same object, they must actually have different **this** pointers, so the proper starting address can be passed to the member functions of each subobject. The only way things can work correctly is if this implicit upcasting takes place when you call a member function for a multiply inherited subobject.

Persistence

Normally this isn't a problem, because you want to call member functions that are concerned with that subobject of the multiply inherited object. However, if your member function needs to know the true starting address of the object, multiple inheritance causes problems. Ironically, this happens in one of the situations where multiple inheritance seems to be useful: *persistence*.

The lifetime of a local object is the scope in which it is defined. The lifetime of a global object is the lifetime of the program. A *persistent object* lives between invocations of a program: You can normally think of it as existing on disk instead of in memory. One definition of an object-oriented database is "a collection of persistent objects."

To implement persistence, you must move a persistent object from disk into memory in order to call functions for it, and later store it to disk before the program expires. Four issues arise when storing an object on disk:

1. The object must be converted from its representation in memory to a series of bytes on disk.

2. Because the values of any pointers in memory won't have meaning the next time the program is invoked, these pointers must be converted to something meaningful.

3. What the pointers *point to* must also be stored and retrieved.

4. When restoring an object from disk, the virtual pointers in the object must be respected.

Because the object must be converted back and forth between a layout in memory and a serial representation on disk, the process is called *serialization* (to write an object to disk) and *deserialization* (to restore an object from disk). Although it would be very convenient, these processes require too much overhead to support directly in the language. Class libraries will often build in support for serialization and deserialization by adding special member functions and placing requirements on new classes. (Usually some sort of **serialize( )** function must be written for each new class.) Also, persistence is generally not automatic; you must usually explicitly write and read the objects.

MI-based persistence

Consider sidestepping the pointer issues for now and creating a class that installs persistence into simple objects using multiple inheritance. By inheriting the **persistence** class along with your new class, you automatically create classes that can be read from and written to disk. Although this sounds great, the use of multiple inheritance introduces a pitfall, as seen in the following example.

```
//: C22:Persist1.cpp
// Simple persistence with MI
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

class Persistent {
  int objSize; // Size of stored object
public:
  Persistent(int sz) : objSize(sz) {}
  void write(ostream& out) const {
    out.write((char*)this, objSize);
  }
  void read(istream& in) {
```

```
    in.read((char*)this, objSize);
  }
};

class Data {
  float f[3];
public:
  Data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << "   ";
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class WData1 : public Persistent, public Data {
public:
  WData1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : Data(f0, f1, f2),
    Persistent(sizeof(WData1)) {}
};

class WData2 : public Data, public Persistent {
public:
  WData2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : Data(f0, f1, f2),
    Persistent(sizeof(WData2)) {}
};

int main() {
  {
    ofstream f1("f1.dat"), f2("f2.dat");
    assure(f1, "f1.dat"); assure(f2, "f2.dat");
    WData1 d1(1.1, 2.2, 3.3);
    WData2 d2(4.4, 5.5, 6.6);
    d1.print("d1 before storage");
```

```
    d2.print("d2 before storage");
    d1.write(f1);
    d2.write(f2);
  } // Closes files
  ifstream f1("f1.dat"), f2("f2.dat");
  assure(f1, "f1.dat"); assure(f2, "f2.dat");
  WData1 d1;
  WData2 d2;
  d1.read(f1);
  d2.read(f2);
  d1.print("d1 after storage");
  d2.print("d2 after storage");
} ///:~
```

In this very simple version, the **Persistent::read( )** and **Persistent::write( )** functions take the **this** pointer and call **iostream read( )** and **write( )** functions. (Note that any type of **iostream** can be used). A more sophisticated **Persistent** class would call a **virtual write( )** function for each subobject.

With the language features covered so far in the book, the number of bytes in the object cannot be known by the **Persistent** class so it is inserted as a constructor argument. (In Chapter XX, *run-time type identification* shows how you can find the exact type of an object given only a base pointer; once you have the exact type you can find out the correct size with the **sizeof** operator.)

The **Data** class contains no pointers or VPTR, so there is no danger in simply writing it to disk and reading it back again. And it works fine in class **WData1** when, in **main( )**, it's written to file F1.DAT and later read back again. However, when **Persistent** is second in the inheritance list of **WData2**, the **this** pointer for **Persistent** is offset to the end of the object, so it reads and writes past the end of the object. This not only produces garbage when reading the object from the file, it's dangerous because it walks over any storage that occurs after the object.

This problem occurs in multiple inheritance any time a class must produce the **this** pointer for the actual object from a subobject's **this** pointer. Of course, if you know your compiler always lays out objects in order of declaration in the inheritance list, you can ensure that you always put the critical class at the beginning of the list (assuming there's only one critical class). However, such a class may exist in the inheritance hierarchy of another class and you may unwittingly put it in the wrong place during multiple inheritance. Fortunately, using run-time type identification (the

subject of Chapter XX) will produce the proper pointer to the actual object, even if multiple inheritance is used.

Improved persistence

A more practical approach to persistence, and one you will see employed more often, is to create virtual functions in the base class for reading and writing and then require the creator of any new class that must be streamed to redefine these functions. The argument to the function is the stream object to write to or read from.[66] Then the creator of the class, who knows best how the new parts should be read or written, is responsible for making the correct function calls. This doesn't have the "magical" quality of the previous example, and it requires more coding and knowledge on the part of the user, but it works and doesn't break when pointers are present:

```
//: C22:Persist2.cpp
// Improved MI persistence
#include "../require.h"
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

class Persistent {
public:
  virtual void write(ostream& out) const = 0;
  virtual void read(istream& in) = 0;
  virtual ~Persistent() {}
};

class Data {
protected:
  float f[3];
public:
  Data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
```

[66] Sometimes there's only a single function for streaming, and the argument contains information about whether you're reading or writing.

```
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class WData1 : public Persistent, public Data {
public:
  WData1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : Data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " "
      << f[1] << " " << f[2] << " ";
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class WData2 : public Data, public Persistent {
public:
  WData2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : Data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " "
      << f[1] << " " << f[2] << " ";
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class Conglomerate : public Data,
public Persistent {
  char* name; // Contains a pointer
  WData1 d1;
  WData2 d2;
public:
```

```
  Conglomerate(const char* nm = "",
    float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0, float f3 = 0.0,
    float f4 = 0.0, float f5 = 0.0,
    float f6 = 0.0, float f7 = 0.0,
    float f8= 0.0) : Data(f0, f1, f2),
    d1(f3, f4, f5), d2(f6, f7, f8) {
    name = new char[strlen(nm) + 1];
    strcpy(name, nm);
  }
  void write(ostream& out) const {
    int i = strlen(name) + 1;
    out << i << " "; // Store size of string
    out << name << endl;
    d1.write(out);
    d2.write(out);
    out << f[0] << " " << f[1] << " " << f[2];
  }
  // Must read in same order as write:
  void read(istream& in) {
    delete []name; // Remove old storage
    int i;
    in >> i >> ws; // Get int, strip whitespace
    name = new char[i];
    in.getline(name, i);
    d1.read(in);
    d2.read(in);
    in >> f[0] >> f[1] >> f[2];
  }
  void print() const {
    Data::print(name);
    d1.print();
    d2.print();
  }
};

int main() {
  {
    ofstream data("data.dat");
    assure(data, "data.dat");
    Conglomerate C("This is Conglomerate C",
      1.1, 2.2, 3.3, 4.4, 5.5,
```

```
      6.6, 7.7, 8.8, 9.9);
    cout << "C before storage" << endl;
    C.print();
    C.write(data);
  } // Closes file
  ifstream data("data.dat");
  assure(data, "data.dat");
  Conglomerate C;
  C.read(data);
  cout << "after storage: " << endl;
  C.print();
} ///:~
```

The pure virtual functions in **Persistent** must be redefined in the derived classes to perform the proper reading and writing. If you already knew that **Data** would be persistent, you could inherit directly from **Persistent** and redefine the functions there, thus eliminating the need for multiple inheritance. This example is based on the idea that you don't own the code for **Data**, that it was created elsewhere and may be part of another class hierarchy so you don't have control over its inheritance. However, for this scheme to work correctly you must have access to the underlying implementation so it can be stored; thus the use of **protected**.

The classes **WData1** and **WData2** use familiar iostream inserters and extractors to store and retrieve the **protected** data in **Data** to and from the iostream object. In **write( )**, you can see that spaces are added after each floating point number is written; these are necessary to allow parsing of the data on input.

The class **Conglomerate** not only inherits from **Data**, it also has member objects of type **WData1** and **WData2**, as well as a pointer to a character string. In addition, all the classes that inherit from **Persistent** also contain a VPTR, so this example shows the kind of problem you'll actually encounter when using persistence.

When you create **write( )** and **read( )** function pairs, the **read( )** must exactly mirror what happens during the **write( )**, so **read( )** pulls the bits off the disk the same way they were placed there by **write( )**. Here, the first problem that's tackled is the **char***, which points to a string of any length. The size of the string is calculated and stored on disk as an **int** (followed by a space to enable parsing) to allow the **read( )** function to allocate the correct amount of storage.

When you have subobjects that have **read( )** and **write( )** member functions, all you need to do is call those functions in the new **read( )** and

write( ) functions. This is followed by direct storage of the members in the base class.

People have gone to great lengths to automate persistence, for example, by creating modified preprocessors to support a "persistent" keyword to be applied when defining a class. One can imagine a more elegant approach than the one shown here for implementing persistence, but it has the advantage that it works under all implementations of C++, doesn't require special language extensions, and is relatively bulletproof.

Avoiding MI

The need for multiple inheritance in **Persist2.cpp** is contrived, based on the concept that you don't have control of some of the code in the project. Upon examination of the example, you can see that MI can be easily avoided by using member objects of type **Data**, and putting the virtual **read( )**and **write( )** members inside **Data** or **WData1** and **WData2** rather than in a separate class. There are many situations like this one where multiple inheritance may be avoided; the language feature is included for unusual, special-case situations that would otherwise be difficult or impossible to handle. But when the question of whether to use multiple inheritance comes up, you should ask two questions:

1. Do I need to show the public interfaces of both these classes, or could one class be embedded with some of its interface produced with member functions in the new class?
2. Do I need to upcast to both of the base classes? (This applies when you have more than two base classes, of course.)

If you can't answer "no" to both questions, you can avoid using MI and should probably do so.

One situation to watch for is when one class only needs to be upcast as a function argument. In that case, the class can be embedded and an automatic type conversion operator provided in your new class to produce a reference to the embedded object. Any time you use an object of your new class as an argument to a function that expects the embedded object, the type conversion operator is used. However, type conversion can't be used for normal member selection; that requires inheritance.

Repairing an interface

One of the best arguments for multiple inheritance involves code that's out of your control. Suppose you've acquired a library that consists of a header file and compiled member functions, but no source code for member functions. This library is a class hierarchy with virtual functions, and it contains some global functions that take pointers to the base class of the library; that is, it uses the library objects polymorphically. Now suppose you build an application around this library, and write your own code that uses the base class polymorphically.

Later in the development of the project or sometime during its maintenance, you discover that the base-class interface provided by the vendor is incomplete: A function may be nonvirtual and you need it to be virtual, or a virtual function is completely missing in the interface, but essential to the solution of your problem. If you had the source code, you could go back and put it in. But you don't, and you have a lot of existing code that depends on the original interface. Here, multiple inheritance is the perfect solution.

For example, here's the header file for a library you acquire:

```
//: C22:Vendor.h
// Vendor-supplied class header
// You only get this & the compiled Vendor.obj
#ifndef VENDOR_H
#define VENDOR_H

class Vendor {
public:
  virtual void v() const;
  void f() const;
  ~Vendor();
};

class Vendor1 : public Vendor {
public:
  void v() const;
  void f() const;
  ~Vendor1();
};

void A(const Vendor&);
```

```
void B(const Vendor&);
// Etc.
#endif // VENDOR_H ///:~
```

Assume the library is much bigger, with more derived classes and a larger interface. Notice that it also includes the functions **A( )** and **B( )**, which take a base pointer and treat it polymorphically. Here's the implementation file for the library:

```
//: C22:Vendor.cpp {O}
// Implementation of VENDOR.H
// This is compiled and unavailable to you
#include "Vendor.h"
#include <fstream>
using namespace std;

extern ofstream out; // For trace info

void Vendor::v() const {
  out << "Vendor::v()\n";
}

void Vendor::f() const {
  out << "Vendor::f()\n";
}

Vendor::~Vendor() {
  out << "~Vendor()\n";
}

void Vendor1::v() const {
  out << "Vendor1::v()\n";
}

void Vendor1::f() const {
  out << "Vendor1::f()\n";
}

Vendor1::~Vendor1() {
  out << "~Vendor1()\n";
}

void A(const Vendor& V) {
```

```
  // ...
  V.v();
  V.f();
  //..
}

void B(const Vendor& V) {
  // ...
  V.v();
  V.f();
  //..
} ///:~
```

In your project, this source code is unavailable to you. Instead, you get a compiled file as **Vendor.obj** or **Vendor.lib** (or the equivalent for your system).

The problem occurs in the use of this library. First, the destructor isn't virtual. This is actually a design error on the part of the library creator. In addition, **f( )** was not made virtual; assume the library creator decided it wouldn't need to be. And you discover that the interface to the base class is missing a function essential to the solution of your problem. Also suppose you've already written a fair amount of code using the existing interface (not to mention the functions **A( )** and **B( ),** which are out of your control), and you don't want to change it.

To repair the problem, create your own class interface and multiply inherit a new set of derived classes from your interface and from the existing classes:

```
//: C22:Paste.cpp
//{L} Vendor
// Fixing a mess with MI
#include "Vendor.h"
#include <fstream>
using namespace std;

ofstream out("paste.out");

class MyBase { // Repair Vendor interface
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // New interface function:
```

```
  virtual void g() const = 0;
  virtual ~MyBase() { out << "~MyBase()\n"; }
};

class Paste1 : public MyBase, public Vendor1 {
public:
  void v() const {
    out << "Paste1::v()\n";
    Vendor1::v();
  }
  void f() const {
    out << "Paste1::f()\n";
    Vendor1::f();
  }
  void g() const {
    out << "Paste1::g()\n";
  }
  ~Paste1() { out << "~Paste1()\n"; }
};

int main() {
  Paste1& p1p = *new Paste1;
  MyBase& mp = p1p; // Upcast
  out << "calling f()\n";
  mp.f();  // Right behavior
  out << "calling g()\n";
  mp.g(); // New behavior
  out << "calling A(p1p)\n";
  A(p1p); // Same old behavior
  out << "calling B(p1p)\n";
  B(p1p);  // Same old behavior
  out << "delete mp\n";
  // Deleting a reference to a heap object:
  delete &mp; // Right behavior
} ///:~
```

In **MyBase** (which does *not* use MI), both **f( )** and the destructor are now virtual, and a new virtual function **g( )** has been added to the interface. Now each of the derived classes in the original library must be recreated, mixing in the new interface with MI. The functions **Paste1::v( )** and **Paste1::f( )**need to call only the original base-class versions of their functions. But now, if you upcast to **MyBase** as in **main( )**

```
MyBase* mp = p1p; // Upcast
```

any function calls made through **mp** will be polymorphic, including **delete**. Also, the new interface function **g( )** can be called through **mp**. Here's the output of the program:

```
calling f()
Paste1::f()
Vendor1::f()
calling g()
Paste1::g()
calling A(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
calling B(p1p)
Paste1::v()
Vendor1::v()
Vendor::f()
delete mp
~Paste1()
~Vendor1()
~Vendor()
~MyBase()
```

The original library functions **A( )** and **B( )** still work the same (assuming the new **v( )** calls its base-class version). The destructor is now virtual and exhibits the correct behavior.

Although this is a messy example, it does occur in practice and it's a good demonstration of where multiple inheritance is clearly necessary: You must be able to upcast to both base classes.

Summary

The reason MI exists in C++ and not in other OOP languages is that C++ is a hybrid language and couldn't enforce a single monolithic class hierarchy the way Smalltalk does. Instead, C++ allows many inheritance trees to be formed, so sometimes you may need to combine the interfaces from two or more trees into a new class.

If no "diamonds" appear in your class hierarchy, MI is fairly simple (although identical function signatures in base classes must be resolved). If a diamond appears, then you must deal with the problems of duplicate

subobjects by introducing virtual base classes. This not only adds confusion, but the underlying representation becomes more complex and less efficient.

Multiple inheritance has been called the "goto of the 90's".[67] This seems appropriate because, like a goto, MI is best avoided in normal programming, but can occasionally be very useful. It's a "minor" but more advanced feature of C++, designed to solve problems that arise in special situations. If you find yourself using it often, you may want to take a look at your reasoning. A good Occam's Razor is to ask, "Must I upcast to all of the base classes?" If not, your life will be easier if you embed instances of all the classes you *don't* need to upcast to.

Exercises

1. These exercises will take you step-by-step through the traps of MI. Create a base class **X** with a single constructor that takes an **int** argument and a member function **f( ),** that takes no arguments and returns **void**. Now inherit **X** into **Y** and **Z**, creating constructors for each of them that takes a single **int** argument. Now multiply inherit **Y** and **Z** into **A**. Create an object of class **A**, and call **f( )** for that object. Fix the problem with explicit disambiguation.
2. Starting with the results of exercise 1, create a pointer to an **X** called **px**, and assign to it the address of the object of type **A** you created before. Fix the problem using a virtual base class. Now fix **X** so you no longer have to call the constructor for **X** inside **A**.
3. Starting with the results of exercise 2, remove the explicit disambiguation for **f( )**, and see if you can call **f( )** through **px**. Trace it to see which function gets called. Fix the problem so the correct function will be called in a class hierarchy.

[67] A phrase coined by Zack Urlocker.

23: Exception handling

Improved error recovery is one of the most powerful ways you can increase the robustness of your code.

Unfortunately, it's almost accepted practice to ignore error conditions, as if we're in a state of denial about errors. Some of the reason is no doubt the tediousness and code bloat of checking for many errors. For example, **printf( )** returns the number of characters that were successfully printed, but virtually no one checks this value. The proliferation of code alone would be disgusting, not to mention the difficulty it would add in reading the code.

The problem with C's approach to error handling could be thought of as one of coupling – the user of a function must tie the error-handling code so closely to that function that it becomes too ungainly and awkward to use.

One of the major features in C++ is *exception handling*, which is a better way of thinking about and handling errors. With exception handling,

1. Error-handling code is not nearly so tedious to write, and it doesn't become mixed up with your "normal" code. You write the code you *want* to happen; later in a separate section you write the code to cope with the problems. If you make multiple calls to a function, you handle the errors from that function once, in one place.
2. Errors cannot be ignored. If a function needs to send an error message to the caller of that function, it "throws" an object representing that error out of the function. If the caller doesn't "catch" the error and handle it, it goes to the next enclosing scope, and so on until *someone* catches the error.

This chapter examines C's approach to error handling (such as it is), why it did not work very well for C, and why it won't work at all for C++. Then you'll learn about **try**, **throw**, and **catch**, the C++ keywords that support exception handling.

Error handling in C

In most of the examples in this book, **assert( )** was used as it was intended: for debugging during development with code that could be disabled with **#define NDEBUG** for the shipping product. Runtime error checking uses the **require.h** functions developed in Chapter XX. These were a convenient way to say, "There's a problem here you'll probably want to handle with some more sophisticated code, but you don't need to be distracted by it in this example." The **require.h** functions may be enough for small programs, but for complicated products you may need to write more sophisticated error-handling code.

Error handling is quite straightforward in situations where you check some condition and you know exactly what to do because you have all the necessary information in that context. Of course, you just handle the error at that point. These are ordinary errors and not the subject of this chapter.

The problem occurs when you *don't* have enough information in that context, and you need to pass the error information into a larger context where that information does exist. There are three typical approaches in C to handle this situation.

1. Return error information from the function or, if the return value cannot be used this way, set a global error condition flag. (Standard C provides **errno** and **perror( )** to support this.) As mentioned before, the programmer may simply ignore the error information because tedious and obfuscating error checking must occur with each function call. In addition, returning from a function that hits an exceptional condition may not make sense.
2. Use the little-known Standard C library signal-handling system, implemented with the **signal( )** function (to determine what happens when the event occurs) and **raise( )** (to generate an event). Again, this has high coupling because it requires the user of any library that generates signals to understand and install the appropriate

signal-handling mechanism; also in large projects the signal numbers from different libraries may clash with each other.

3. Use the nonlocal goto functions in the Standard C library: **setjmp( )** and **longjmp( )**. With **setjmp( )** you save a known good state in the program, and if you get into trouble, **longjmp( )** will restore that state. Again, there is high coupling between the place where the state is stored and the place where the error occurs.

When considering error-handling schemes with C++, there's an additional very critical problem: The C techniques of signals and setjmp/longjmp do not call destructors, so objects aren't properly cleaned up. This makes it virtually impossible to effectively recover from an exceptional condition because you'll always leave objects behind that haven't been cleaned up and that can no longer be accessed. The following example demonstrates this with setjmp/longjmp:

```
//: C23:Nonlocal.cpp
// setjmp() & longjmp()
#include <iostream>
#include <csetjmp>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

jmp_buf kansas;

void oz() {
  Rainbow rb;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  longjmp(kansas, 47);
}

int main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins...\n";
    oz();
```

```
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
} ///:~
```

setjmp( ) is an odd function because if you call it directly, it stores all the relevant information about the current processor state in the **jmp_buf** and returns zero. In that case it has the behavior of an ordinary function. However, if you call **longjmp( )** using the same **jmp_buf**, it's as if you're returning from **setjmp( )** again – you pop right out the back end of the **setjmp( )**. This time, the value returned is the second argument to **longjmp( )**, so you can detect that you're actually coming back from a **longjmp( )**. You can imagine that with many different **jmp_buf**s, you could pop around to many different places in the program. The difference between a local **goto** (with a label) and this nonlocal goto is that you can go *anywhere* with setjmp/longjmp (with some restrictions not discussed here).

The problem with C++ is that **longjmp( )** doesn't respect objects; in particular it doesn't call destructors when it jumps out of a scope.[68] Destructor calls are essential, so this approach won't work with C++.

Throwing an exception

If you encounter an exceptional situation in your code – that is, one where you don't have enough information in the current context to decide what to do – you can send information about the error into a larger context by creating an object containing that information and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
throw myerror("something bad happened");
```

myerror is an ordinary class, which takes a **char*** as its argument. You can use any type when you throw (including built-in types), but often you'll use special types created just for throwing exceptions.

[68] You may be surprised when you run the example – some C++ compilers have extended **longjmp( )** to clean up objects on the stack. This is nonportable behavior.

The keyword **throw** causes a number of relatively magical things to happen. First it creates an object that isn't there under normal program execution, and of course the constructor is called for that object. Then the object is, in effect, "returned" from the function, even though that object type isn't normally what the function is designed to return. A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take the analogy too far – you can also exit from ordinary scopes by throwing an exception. But a value is returned, and the function or scope exits.

Any similarity to function returns ends there because *where* you return to is someplace completely different than for a normal function call. (You end up in an appropriate exception handler that may be miles away from where the exception was thrown.) In addition, only objects that were successfully created at the time of the exception are destroyed (unlike a normal function return that assumes all the objects in the scope must be destroyed). Of course, the exception object itself is also properly cleaned up at the appropriate point.

In addition, you can throw as many different types of objects as you want. Typically, you'll throw a different type for each different type of error. The idea is to store the information in the object and the *type* of object, so someone in the bigger context can figure out what to do with your exception.

Catching an exception

If a function throws an exception, it must assume that exception is caught and dealt with. As mentioned before, one of the advantages of C++ exception handling is that it allows you to concentrate on the problem you're actually trying to solve in one place, and then deal with the errors from that code in another place.

The **try** block

If you're inside a function and you throw an exception (or a called function throws an exception), that function will exit in the process of throwing. If you don't want a **throw** to leave a function, you can set up a special block within the function where you try to solve your actual programming problem (and potentially generate exceptions). This is called the *try block* because you try your various function calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // Code that may generate exceptions
}
```

If you were carefully checking for errors without using exception handling, you'd have to surround every function call with setup and test code, even if you call the same function several times. With exception handling, you put everything in a try block without error checking. This means your code is a lot easier to write and easier to read because the goal of the code is not confused with the error checking.

Exception handlers

Of course, the thrown exception must end up someplace. This is the *exception handler*, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
// code that may generate exceptions
} catch(type1 id1) {
  // handle exceptions of type1
} catch(type2 id2) {
  // handle exceptions of type2
}
// etc...
```

Each catch clause (exception handler) is like a little function that takes a single argument of one particular type. The identifier (**id1**, **id2**, and so on) may be used inside the handler, just like a function argument, although sometimes there is no identifier because it's not needed in the handler – the exception type gives you enough information to deal with it.

The handlers must appear directly after the try block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. (The search for handlers stops once the catch clause is finished.) Only the matching catch clause executes; it's not like a **switch** statement where you need a **break** after each **case** to prevent the remaining ones from executing.

Notice that, within the try block, a number of different function calls might generate the same exception, but you only need one handler.

Termination vs. resumption

There are two basic models in exception-handling theory. In *termination* (which is what C++ supports) you assume the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don't *want* to come back.

The alternative is called *resumption*. It means the exception handler is expected to do something to rectify the situation, and then the faulting function is retried, presuming success the second time. If you want resumption, you still hope to continue execution after the exception is handled, so your exception is more like a function call – which is how you should set up situations in C++ where you want resumption-like behavior (that is, don't throw an exception; call a function that fixes the problem). Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it seems it isn't quite so useful in practice. One reason may be the distance that can occur between the exception and its handler; it's one thing to terminate to a handler that's far away, but to jump to that handler and then back again may be too conceptually difficult for large systems where the exception can be generated from many points.

The exception specification

You're not required to inform the person using your function what exceptions you might throw. However, this is considered very uncivilized because it means he cannot be sure what code to write to catch all potential exceptions. Of course, if he has your source code, he can hunt through and look for **throw** statements, but very often a library doesn't come with sources. C++ provides a syntax to allow you to politely tell the user what exceptions this function throws, so the user may handle them. This is the *exception specification* and it's part of the function declaration, appearing after the argument list.

The exception specification reuses the keyword **throw**, followed by a parenthesized list of all the potential exception types. So your function declaration may look like

```
void f() throw(toobig, toosmall, divzero);
```

With exceptions, the traditional function declaration

```
void f();
```

means that any type of exception may be thrown from the function. If you say

```
void f() throw();
```

it means that no exceptions are thrown from a function.

For good coding policy, good documentation, and ease-of-use for the function caller, you should always use an exception specification when you write a function that throws exceptions.

unexpected( )

If your exception specification claims you're going to throw a certain set of exceptions and then you throw something that isn't in that set, what's the penalty? The special function **unexpected( )** is called when you throw something other than what appears in the exception specification.

set_unexpected( )

unexpected( ) is implemented with a pointer to a function, so you can change its behavior. You do so with a function called **set_unexpected( )** which, like **set_new_handler( )**, takes the address of a function with no arguments and **void** return value. Also, it returns the previous value of the **unexpected( )** pointer so you can save it and restore it later. To use **set_unexpected( )**, you must include the header file **<exception>**. Here's an example that shows a simple use of all the features discussed so far in the chapter:

```
//: C23:Except.cpp
// Basic exceptions
// Exception specifications & unexpected()
#include <exception>
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

class Up {};
class Fit {};
void g();

void f(int i) throw (Up, Fit) {
```

```
  switch(i) {
    case 1: throw Up();
    case 2: throw Fit();
  }
  g();
}

// void g() {} // Version 1
void g() { throw 47; } // Version 2
// (Can throw built-in types)

void my_unexpected() {
  cout << "unexpected exception thrown";
  exit(1);
}

int main() {
  set_unexpected(my_unexpected);
  // (ignores return value)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(Up) {
      cout << "Up caught" << endl;
    } catch(Fit) {
      cout << "Fit caught" << endl;
    }
} ///:~
```

The classes **Up** and **Fit** are created solely to throw as exceptions. Often exception classes will be this small, but sometimes they contain additional information so that the handlers can query them.

f( ) is a function that promises in its exception specification to throw only exceptions of type **Up** and **Fit**, and from looking at the function definition this seems plausible. Version one of **g( )**, called by **f( )**, doesn't throw any exceptions so this is true. But then someone changes **g( )** so it throws exceptions and the new **g( )** is linked in with **f( )**. Now **f( )** begins to throw a new exception, unbeknown to the creator of **f( )**. Thus the exception specification is violated.

The **my_unexpected( )** function has no arguments or return value, following the proper form for a custom **unexpected( )** function. It simply prints a message so you can see it has been called, then exits the

program. Your new **unexpected( )** function must not return (that is, you can write the code that way but it's an error). However, it can throw another exception (you can even rethrow the same exception), or call **exit( )** or **abort( )**. If **unexpected( )** throws an exception, the search for the handler starts at the function call that threw the unexpected exception. (This behavior is unique to **unexpected( )**.)

Although the **new_handler( )** function pointer can be null and the system will do something sensible, the **unexpected( )** function pointer should never be null. The default value is **terminate( )** (mentioned later), but whenever you use exceptions and specifications you should write your own **unexpected( )** to log the error and either rethrow it, throw something new, or terminate the program.

In **main( )**, the **try** block is within a **for** loop so all the possibilities are exercised. Note that this is a way to achieve something like resumption – nest the **try** block inside a **for**, **while**, **do**, or **if** and cause any exceptions to attempt to repair the problem; then attempt the **try** block again.

Only the **Up** and **Fit** exceptions are caught because those are the only ones the programmer of **f( )** said would be thrown. Version two of **g( )** causes **my_unexpected( )** to be called because **f( )** then throws an **int**. (You can throw any type, including a built-in type.)

In the call to **set_unexpected( )**, the return value is ignored, but it can also be saved in a pointer to function and restored later.

Better exception specifications?

You may feel the existing exception specification rules aren't very safe, and that

```
void f();
```

should mean that no exceptions are thrown from this function. If the programmer wants to throw any type of exception, you may think he or she *should* have to say

```
void f() throw(...); // Not in C++
```

This would surely be an improvement because function declarations would be more explicit. Unfortunately you can't always know by looking at the code in a function whether an exception will be thrown – it could happen because of a memory allocation, for example. Worse, existing functions written before exception handling was introduced may find themselves inadvertently throwing exceptions because of the functions they call

(which may be linked into new, exception-throwing versions). Thus, the ambiguity, so

```
void f();
```

means “Maybe I’ll throw an exception, maybe I won’t.” This ambiguity is necessary to avoid hindering code evolution.

Catching any exception

As mentioned, if your function has no exception specification, *any* type of exception can be thrown. One solution to this problem is to create a handler that *catches* any type of exception. You do this using the ellipses in the argument list (á la C):

```
catch(...) {
  cout << "an exception was thrown" << endl;
}
```

This will catch any exception, so you’ll want to put it at the *end* of your list of handlers to avoid pre-empting any that follow it.

The ellipses give you no possibility to have an argument or to know anything about the type of the exception. It’s a catch-all.

Rethrowing an exception

Sometimes you’ll want to rethrow the exception that you just caught, particularly when you use the ellipses to catch any exception because there’s no information available about the exception. This is accomplished by saying **throw** with no argument:

```
catch(...) {
  cout << "an exception was thrown" << endl;
  throw;
}
```

Any further **catch** clauses for the same **try** block are still ignored – the **throw** causes the exception to go to the exception handlers in the next-higher context. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type is able to extract all the information from that object.

Uncaught exceptions

If none of the exception handlers following a particular **try** block matches an exception, that exception moves to the next-higher context, that is, the function or **try** block surrounding the **try** block that failed to catch the exception. (The location of this higher-context **try** block is not always obvious at first glance.) This process continues until, at some level, a handler matches the exception. At that point, the exception is considered "caught," and no further searching occurs.

If no handler at any level catches the exception, it is "uncaught" or "unhandled." An uncaught exception also occurs if a new exception is thrown before an existing exception reaches its handler – the most common reason for this is that the constructor for the exception object itself causes a new exception.

terminate( )

If an exception is uncaught, the special function **terminate( )** is automatically called. Like **unexpected( )**, terminate is actually a pointer to a function. Its default value is the Standard C library function **abort( )**, which immediately exits the program with no calls to the normal termination functions (which means that destructors for global and static objects might not be called).

No cleanups occur for an uncaught exception; that is, no destructors are called. If you don't wrap your code (including, if necessary, all the code in **main( )**) in a try block followed by handlers and ending with a default handler (**catch(...)**) to catch all exceptions, then you will take your lumps. An uncaught exception should be thought of as a programming error.

set_terminate( )

You can install your own **terminate( )** function using the standard **set_terminate( )** function, which returns a pointer to the **terminate( )** function you are replacing, so you can restore it later if you want. Your custom **terminate( )** must take no arguments and have a **void** return value. In addition, any **terminate( )** handler you install must not return or throw an exception, but instead must call some sort of program-termination function. If **terminate( )** is called, it means the problem is unrecoverable.

Like **unexpected( )**, the **terminate( )** function pointer should never be null.

Here's an example showing the use of **set_terminate( )**. Here, the return value is saved and restored so the **terminate( )** function can be used to help isolate the section of code where the uncaught exception is occurring:

```
//: C23:Trmnator.cpp
// Use of set_terminate()
// Also shows uncaught exceptions
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

void terminator() {
  cout << "I'll be back!" << endl;
  abort();
}

void (*old_terminate)()
  = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try{
    Botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
} ///:~
```

The definition of **old_terminate** looks a bit confusing at first: It not only creates a pointer to a function, but it initializes that pointer to the return value of **set_terminate( )**. Even though you may be familiar with seeing

a semicolon right after a pointer-to-function definition, it's just another kind of variable and may be initialized when it is defined.

The class **Botch** not only throws an exception inside **f( )**, but also in its destructor. This is one of the situations that causes a call to **terminate( )**, as you can see in **main( )**. Even though the exception handler says **catch(...)**, which would seem to catch everything and leave no cause for **terminate( )** to be called, **terminate( )** is called anyway, because in the process of cleaning up the objects on the stack to handle one exception, the **Botch** destructor is called, and that generates a second exception, forcing a call to **terminate( )**. Thus, a destructor that throws an exception or causes one to be thrown is a design error.

Function-level try blocks

```
//: C23:FunctionTryBlock.cpp
// Function-level try blocks
#include <iostream>
using namespace std;

int main() try {
    throw "main";
} catch(const char* msg) {
  cout << msg << endl;
} ///:~
```

Cleaning up

Part of the magic of exception handling is that you can pop from normal program flow into the appropriate exception handler. This wouldn't be very useful, however, if things weren't cleaned up properly as the exception was thrown. C++ exception handling guarantees that as you leave a scope, all objects in that scope *whose constructors have been completed* will have destructors called.

Here's an example that demonstrates that constructors that aren't completed don't have the associated destructors called. It also shows what happens when an exception is thrown in the middle of the creation of an array of objects, and an **unexpected( )** function that rethrows the unexpected exception:

```
//: C23:Cleanup.cpp
// Exceptions clean up objects
#include <fstream>
#include <exception>
#include <cstring>
using namespace std;
ofstream out("cleanup.out");

class Noisy {
  static int i;
  int objnum;
  static const int sz = 40;
  char name[sz];
public:
  Noisy(const char* nm="array elem") throw(int){
    objnum = i++;
    memset(name, 0, sz);
    strncpy(name, nm, sz - 1);
    out << "constructing Noisy " << objnum
      << " name [" << name << "]" << endl;
    if(objnum == 5) throw int(5);
    // Not in exception specification:
    if(*nm == 'z') throw char('z');
  }
  ~Noisy() {
    out << "destructing Noisy " << objnum
      << " name [" << name << "]" << endl;
  }
  void* operator new[](size_t sz) {
    out << "Noisy::new[]" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    out << "Noisy::delete[]" << endl;
    ::delete []p;
  }
};

int Noisy::i = 0;

void unexpected_rethrow() {
  out << "inside unexpected_rethrow()" << endl;
```

```
  throw; // Rethrow same exception
}

int main() {
  set_unexpected(unexpected_rethrow);
  try {
    Noisy n1("before array");
    // Throws exception:
    Noisy* array = new Noisy[7];
    Noisy n2("after array");
  } catch(int i) {
    out << "caught " << i << endl;
  }
  out << "testing unexpected:" << endl;
  try {
    Noisy n3("before unexpected");
    Noisy n4("z");
    Noisy n5("after unexpected");
  } catch(char c) {
    out << "caught " << c << endl;
  }
} ///:~
```

The class **Noisy** keeps track of objects so you can trace program progress. It keeps a count of the number of objects created with a **static** data member **i**, and the number of the particular object with **objnum**, and a character buffer called **name** to hold an identifier. This buffer is first set to zeroes. Then the constructor argument is copied in. (Note that a default argument string is used to indicate array elements, so this constructor also acts as a default constructor.) Because the Standard C library function **strncpy( )**stops copying after a null terminator *or* the number of characters specified by its third argument, the number of characters copied in is one minus the size of the buffer, so the last character is always zero, and a print statement will never run off the end of the buffer.

There are two cases where a **throw** can occur in the constructor. The first case happens if this is the fifth object created (not a real exception condition, but demonstrates an exception thrown during array construction). The type thrown is **int**, which is the type promised in the exception specification. The second case, also contrived, happens if the first character of the argument string is **'z'**, in which case a **char** is thrown. Because **char** is not listed in the exception specification, this will cause a call to **unexpected( )**.

The array versions of **new** and **delete** are overloaded for the class, so you can see when they're called.

The function **unexpected_rethrow( )** prints a message and rethrows the same exception. It is installed as the **unexpected( )** function in the first line of **main( )**. Then some objects of type **Noisy** are created in a **try** block, but the array causes an exception to be thrown, so the object **n2** is never created. You can see the results in the output of the program:

```
constructing Noisy 0 name [before array]
Noisy::new[]
constructing Noisy 1 name [array elem]
constructing Noisy 2 name [array elem]
constructing Noisy 3 name [array elem]
constructing Noisy 4 name [array elem]
constructing Noisy 5 name [array elem]
destructing Noisy 4 name [array elem]
destructing Noisy 3 name [array elem]
destructing Noisy 2 name [array elem]
destructing Noisy 1 name [array elem]
Noisy::delete[]
destructing Noisy 0 name [before array]
caught 5
testing unexpected:
constructing Noisy 6 name [before unexpected]
constructing Noisy 7 name [z]
inside unexpected_rethrow()
destructing Noisy 6 name [before unexpected]
caught z
```

Four array elements are successfully created, but in the middle of the constructor for the fifth one, an exception is thrown. Because the fifth constructor never completes, only the destructors for objects 1–4 are called.

The storage for the array is allocated separately with a single call to the global **new**. Notice that even though **delete** is never explicitly called anywhere in the program, the exception-handling system knows it must call **delete** to properly release the storage. This behavior happens only with "normal" versions of **operator new**. If you use the placement syntax described in Chapter XX, the exception-handling mechanism will not call **delete** for that object because then it might release memory that was not allocated on the heap.

Finally, object **n1** is destroyed, but not object **n2** because it was never created.

In the section testing **unexpected_rethrow( )**, the **n3** object is created, and the constructor of **n4** is begun. But before it can complete, an exception is thrown. This exception is of type **char**, which violates the exception specification, so the **unexpected( )** function is called (which is **unexpected_rethrow( )**, in this case). This rethrows the same exception, which is expected this time, because **unexpected_rethrow( )** can throw any type of exception. The search begins right after the constructor for **n4**, and the **char** exception handler catches it (after destroying **n3**, the only successfully created object). Thus, the effect of **unexpected_rethrow( )** is to take any unexpected exception and make it expected; used this way it provides a filter to allow you to track the appearance of unexpected exceptions and pass them through.

Constructors

When writing code with exceptions, it's particularly important that you always be asking, "If an exception occurs, will this be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a problem: If an exception is thrown before a constructor is completed, the associated destructor will not be called for that object. This means you must be especially diligent while writing your constructor.

The general difficulty is allocating resources in constructors. If an exception occurs in the constructor, the destructor doesn't get a chance to deallocate the resource. This problem occurs most often with "naked" pointers. For example,

```
//: C23:Nudep.cpp
// Naked pointers
#include <fstream>
#include <cstdlib>
using namespace std;
ofstream out("nudep.out");

class Cat {
public:
  Cat() { out << "Cat()" << endl; }
  ~Cat() { out << "~Cat()" << endl; }
};
```

```
class Dog {
public:
  void* operator new(size_t sz) {
    out << "allocating a Dog" << endl;
    throw int(47);
  }
  void operator delete(void* p) {
    out << "deallocating a Dog" << endl;
    ::delete p;
  }
};

class UseResources {
  Cat* bp;
  Dog* op;
public:
  UseResources(int count = 1) {
    out << "UseResources()" << endl;
    bp = new Cat[count];
    op = new Dog;
  }
  ~UseResources() {
    out << "~UseResources()" << endl;
    delete []bp; // Array delete
    delete op;
  }
};

int main() {
  try {
    UseResources ur(3);
  } catch(int) {
    out << "inside handler" << endl;
  }
} ///:~
```

The output is the following:

```
UseResources()
Cat()
Cat()
Cat()
allocating a Dog
```

```
inside handler
```

The **UseResources** constructor is entered, and the **Cat** constructor is successfully completed for the array objects. However, inside **Dog::operator new**, an exception is thrown (as an example of an out-of-memory error). Suddenly, you end up inside the handler, *without* the **UseResources** destructor being called. This is correct because the **UseResources** constructor was unable to finish, but it means the **Cat** object that was successfully created on the heap is never destroyed.

Making everything an object

To prevent this, guard against these “raw” resource allocations by placing the allocations inside their own objects with their own constructors and destructors. This way, each allocation becomes atomic, as an object, and if it fails, the other resource allocation objects are properly cleaned up. Templates are an excellent way to modify the above example:

```
//: C23:Wrapped.cpp
// Safe, atomic pointers
#include <fstream>
#include <cstdlib>
using namespace std;
ofstream out("wrapped.out");

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class PWrap {
  T* ptr;
public:
  class RangeError {}; // Exception class
  PWrap() {
    ptr = new T[sz];
    out << "PWrap constructor" << endl;
  }
  ~PWrap() {
    delete []ptr;
    out << "PWrap destructor" << endl;
  }
  T& operator[](int i) throw(RangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw RangeError();
  }
};
```

```
class Cat {
public:
  Cat() { out << "Cat()" << endl; }
  ~Cat() { out << "~Cat()" << endl; }
  void g() {}
};

class Dog {
public:
  void* operator new[](size_t sz) {
    out << "allocating an Dog" << endl;
    throw int(47);
  }
  void operator delete[](void* p) {
    out << "deallocating an Dog" << endl;
    ::delete p;
  }
};

class UseResources {
  PWrap<Cat, 3> Bonk;
  PWrap<Dog> Og;
public:
  UseResources() : Bonk(), Og() {
    out << "UseResources()" << endl;
  }
  ~UseResources() {
    out << "~UseResources()" << endl;
  }
  void f() { Bonk[1].g(); }
};

int main() {
  try {
    UseResources ur;
  } catch(int) {
    out << "inside handler" << endl;
  } catch(...) {
    out << "inside catch(...)" << endl;
  }
} ///:~
```

The difference is the use of the template to wrap the pointers and make them into objects. The constructors for these objects are called *before* the body of the **UseResources** constructor, and any of these constructors that complete before an exception is thrown will have their associated destructors called.

The **PWrap** template shows a more typical use of exceptions than you've seen so far: A nested class called **RangeError** is created to use in **operator[ ]** if its argument is out of range. Because **operator[ ]** returns a reference it cannot return zero. (There are no null references.) This is a true exceptional condition – you don't know what to do in the current context, and you can't return an improbable value. In this example, **RangeError** is very simple and assumes all the necessary information is in the class name, but you may also want to add a member that contains the value of the index, if that is useful.

Now the output is

```
Cat()
Cat()
Cat()
PWrap constructor
allocating a Dog
~Cat()
~Cat()
~Cat()
PWrap destructor
inside handler
```

Again, the storage allocation for **Dog** throws an exception, but this time the array of **Cat** objects is properly cleaned up, so there is no memory leak.

Exception matching

When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. An object or reference to a derived-class object will match a handler for the base class. (However, if the handler is for an object rather than a reference, the exception object is "sliced" as it is

passed to the handler; this does no damage but loses all the derived-type information.) If a pointer is thrown, standard pointer conversions are used to match the exception. However, no automatic type conversions are used to convert one exception type to another in the process of matching. For example,

```
//: C23:Autoexcp.cpp
// No matching conversions
#include <iostream>
using namespace std;

class Except1 {};
class Except2 {
public:
  Except2(Except1&) {}
};

void f() { throw Except1(); }

int main() {
  try { f();
  } catch (Except2) {
    cout << "inside catch(Except2)" << endl;
  } catch (Except1) {
    cout << "inside catch(Except1)" << endl;
  }
} ///:~
```

Even though you might think the first handler could be used by converting an **Except1** object into an **Except2** using the constructor conversion, the system will not perform such a conversion during exception handling, and you'll end up at the **Except1** handler.

The following example shows how a base-class handler can catch a derived-class exception:

```
//: C23:Basexcpt.cpp
// Exception hierarchies
#include <iostream>
using namespace std;

class X {
public:
  class Trouble {};
```

```
  class Small : public Trouble {};
  class Big : public Trouble {};
  void f() { throw Big(); }
};

int main() {
  X x;
  try {
    x.f();
  } catch(X::Trouble) {
    cout << "caught Trouble" << endl;
  // Hidden by previous handler:
  } catch(X::Small) {
    cout << "caught Small Trouble" << endl;
  } catch(X::Big) {
    cout << "caught Big Trouble" << endl;
  }
} ///:~
```

Here, the exception-handling mechanism will always match a **Trouble** object, *or anything derived from* **Trouble**, to the first handler. That means the second and third handlers are never called because the first one captures them all. It makes more sense to catch the derived types first and put the base type at the end to catch anything less specific (or a derived class introduced later in the development cycle).

In addition, if **Small** and **Big** represent larger objects than the base class **Trouble** (which is often true because you regularly add data members to derived classes), then those objects are sliced to fit into the first handler. Of course, in this example it isn't important because there are no additional members in the derived classes and there are no argument identifiers in the handlers anyway. You'll usually want to use reference arguments rather than objects in your handlers to avoid slicing off information.

Standard exceptions

The set of exceptions used with the Standard C++ library are also available for your own use. Generally it's easier and faster to start with a standard exception class than to try to define your own. If the standard class doesn't do what you need, you can derive from it.

The following tables describe the standard exceptions:

exception	The base class for all the exceptions thrown by the C++ standard library. You can ask **what( )** and get a result that can be displayed as a character representation.
logic_error	Derived from **exception**. Reports program logic errors, which could presumably be detected before the program executes.
runtime_error	Derived from **exception**. Reports runtime errors, which can presumably be detected only when the program executes.

The iostream exception class **ios::failure** is also derived from **exception**, but it has no further subclasses.

The classes in both of the following tables can be used as they are, or they can act as base classes to derive your own more specific types of exceptions.

Exception classes derived from **logic_error**	
domain_error	Reports violations of a precondition.
invalid_argument	Indicates an invalid argument to the function it's thrown from.
length_error	Indicates an attempt to produce an object whose length is greater than or equal to NPOS (the largest representable value of type **size_t**).
out_of_range	Reports an out-of-range argument.
bad_cast	Thrown for executing an invalid **dynamic_cast** expression in run-time type identification (see Chapter XX).
bad_typeid	Reports a null pointer **p** in an

Exception classes derived from **logic_error**	
	expression **typeid(*p)**. (Again, a run-time type identification feature in Chapter XX).

Exception classes derived from **runtime_error**	
range_error	Reports violation of a postcondition.
overflow_error	Reports an arithmetic overflow.
bad_alloc	Reports a failure to allocate storage.

Programming with exceptions

For most programmers, especially C programmers, exceptions are not available in their existing language and take a bit of adjustment. Here are some guidelines for programming with exceptions.

When to avoid exceptions

Exceptions aren't the answer to all problems. In fact, if you simply go looking for something to pound with your new hammer, you'll cause trouble. The following sections point out situations where exceptions are *not* warranted.

Not for asynchronous events

The Standard C **signal( )** system, and any similar system, handles asynchronous events: events that happen outside the scope of the program, and thus events the program cannot anticipate. C++ exceptions cannot be used to handle asynchronous events because the exception and its handler are on the same call stack. That is, exceptions rely on scoping, whereas asynchronous events must be handled by completely separate

code that is not part of the normal program flow (typically, interrupt service routines or event loops).

This is not to say that asynchronous events cannot be *associated* with exceptions. But the interrupt handler should do its job as quickly as possible and then return. Later, at some well-defined point in the program, an exception might be thrown *based on* the interrupt.

Not for ordinary error conditions

If you have enough information to handle an error, it's not an exception. You should take care of it in the current context rather than throwing an exception to a larger context.

Also, C++ exceptions are not thrown for machine-level events like divide-by-zero. It's assumed these are dealt with by some other mechanism, like the operating system or hardware. That way, C++ exceptions can be reasonably efficient, and their use is isolated to program-level exceptional conditions.

Not for flow-of-control

An exception looks somewhat like an alternate return mechanism and somewhat like a **switch** statement, so you can be tempted to use them for other than their original intent. This is a bad idea, partly because the exception-handling system is significantly less efficient than normal program execution; exceptions are a rare event, so the normal program shouldn't pay for them. Also, exceptions from anything other than error conditions are quite confusing to the user of your class or function.

You're not forced to use exceptions

Some programs are quite simple, many utilities, for example. You may only need to take input and perform some processing. In these programs you might attempt to allocate memory and fail, or try to open a file and fail, and so on. It is acceptable in these programs to use **assert( )** or to print a message and **abort( )** the program, allowing the system to clean up the mess, rather than to work very hard to catch all exceptions and recover all the resources yourself. Basically, if you don't need to use exceptions, you don't have to.

New exceptions, old code

Another situation that arises is the modification of an existing program that doesn't use exceptions. You may introduce a library that *does* use

exceptions and wonder if you need to modify all your code throughout the program. Assuming you have an acceptable error-handling scheme already in place, the most sensible thing to do here is surround the largest block that uses the new library (this may be all the code in **main( )**) with a **try** block, followed by a **catch(...)** and basic error message. You can refine this to whatever degree necessary by adding more specific handlers, but, in any case, the code you're forced to add can be minimal.

You can also isolate your exception-generating code in a try block and write handlers to convert the exceptions into your existing error-handling scheme.

It's truly important to think about exceptions when you're creating a library for someone else to use, and you can't know how they need to respond to critical error conditions.

Typical uses of exceptions

Do use exceptions to

4. Fix the problem and call the function (which caused the exception) again.
5. Patch things up and continue without retrying the function.
6. Calculate some alternative result instead of what the function was supposed to produce.
7. Do whatever you can in the current context and rethrow the *same* exception to a higher context.
8. Do whatever you can in the current context and throw a *different* exception to a higher context.
9. Terminate the program.
10. Wrap functions (especially C library functions) that use ordinary error schemes so they produce exceptions instead.
11. Simplify. If your exception scheme makes things more complicated, then it is painful and annoying to use.
12. Make your library and program safer. This is a short-term investment (for debugging) and a long-term investment (for application robustness).

Always use exception specifications

The exception specification is like a function prototype: It tells the user to write exception-handling code and what exceptions to handle. It tells the compiler the exceptions that may come out of this function.

Of course, you can't always anticipate by looking at the code what exceptions will arise from a particular function. Sometimes the functions it calls produce an unexpected exception, and sometimes an old function that didn't throw an exception is replaced with a new one that does, and you'll get a call to **unexpected( )**. Anytime you use exception specifications or call functions that do, you should create your own **unexpected( )** function that logs a message and rethrows the same exception.

Start with standard exceptions

Check out the Standard C++ library exceptions before creating your own. If a standard exception does what you need, chances are it's a lot easier for your user to understand and handle.

If the exception type you want isn't part of the standard library, try to derive one from an existing standard **exception**. It's nice for your users if they can always write their code to expect the **what( )** function defined in the **exception( )** class interface.

Nest your own exceptions

If you create exceptions for your particular class, it's a very good idea to nest the exception classes inside your class to provide a clear message to the reader that this exception is used only for your class. In addition, it prevents the pollution of the namespace.

You can nest your exceptions even if you're deriving them from C++ standard exceptions.

Use exception hierarchies

Exception hierarchies provide a valuable way to classify the different types of critical errors that may be encountered with your class or library. This gives helpful information to users, assists them in organizing their code, and gives them the option of ignoring all the specific types of exceptions and just catching the base-class type. Also, any exceptions added later by inheriting from the same base class will not force all existing code to be rewritten – the base-class handler will catch the new exception.

Of course, the Standard C++ exceptions are a good example of an exception hierarchy, and one that you can use to build upon.

Multiple inheritance

You'll remember from Chapter XX that the only *essential* place for MI is if you need to upcast a pointer to your object into two different base classes – that is, if you need polymorphic behavior with both of those base classes. It turns out that exception hierarchies are a useful place for multiple inheritance because a base-class handler from any of the roots of the multiply inherited exception class can handle the exception.

Catch by reference, not by value

If you throw an object of a derived class and it is caught *by value* in a handler for an object of the base class, that object is "sliced" – that is, the derived-class elements are cut off and you'll end up with the base-class object being passed. Chances are this is not what you want because the object will behave like a base-class object and not the derived class object it really is (or rather, was – before it was sliced). Here's an example:

```
//: C23:Catchref.cpp
// Why catch by reference?
#include <iostream>
using namespace std;

class Base {
public:
  virtual void what() {
    cout << "Base" << endl;
  }
};

class Derived : public Base {
public:
  void what() {
    cout << "Derived" << endl;
  }
};

void f() { throw Derived(); }

int main() {
  try {
```

```
    f();
  } catch(Base b) {
    b.what();
  }
  try {
    f();
  } catch(Base& b) {
    b.what();
  }
} ///:~
```

The output is

```
Base
Derived
```

because, when the object is caught by value, it is *turned into* a **Base** object (by the copy-constructor) and must behave that way in all situations, whereas when it's caught by reference, only the address is passed and the object isn't truncated, so it behaves like what it really is, a **Derived** in this case.

Although you can also throw and catch pointers, by doing so you introduce more coupling – the thrower and the catcher must agree on how the exception object is allocated and cleaned up. This is a problem because the exception itself may have occurred from heap exhaustion. If you throw exception objects, the exception-handling system takes care of all storage.

Throw exceptions in constructors

Because a constructor has no return value, you've previously had two choices to report an error during construction:

13. Set a nonlocal flag and hope the user checks it.
14. Return an incompletely created object and hope the user checks it.

This is a serious problem because C programmers have come to rely on an implied guarantee that object creation is always successful, which is not unreasonable in C where types are so primitive. But continuing execution after construction fails in a C++ program is a guaranteed disaster, so constructors are one of the most important places to throw exceptions – now you have a safe, effective way to handle constructor errors. However,

you must also pay attention to pointers inside objects and the way cleanup occurs when an exception is thrown inside a constructor.

Don’t cause exceptions in destructors

Because destructors are called in the process of throwing other exceptions, you’ll never want to throw an exception in a destructor or cause another exception to be thrown by some action you perform in the destructor. If this happens, it means that a new exception may be thrown *before* the catch-clause for an existing exception is reached, which will cause a call to **terminate( )**.

This means that if you call any functions inside a destructor that may throw exceptions, those calls should be within a **try** block in the destructor, and the destructor must handle all exceptions itself. None must escape from the destructor.

Avoid naked pointers

See **Wrapped.cpp**. A naked pointer usually means vulnerability in the constructor if resources are allocated for that pointer. A pointer doesn’t have a destructor, so those resources won’t be released if an exception is thrown in the constructor.

Overhead

Of course it costs something for this new feature; when an exception is thrown there’s considerable runtime overhead. This is the reason you never want to use exceptions as part of your normal flow-of-control, no matter how tempting and clever it may seem. Exceptions should occur only rarely, so the overhead is piled on the exception and not on the normally executing code. One of the important design goals for exception handling was that it could be implemented with no impact on execution speed when it *wasn’t* used; that is, as long as you don’t throw an exception, your code runs as fast as it would without exception handling. Whether or not this is actually true depends on the particular compiler implementation you’re using.

Exception handling also causes extra information to be put on the stack by the compiler, to aid in stack unwinding.

Exception objects are properly passed around like any other objects, except that they can be passed into and out of what can be thought of as a special “exception scope” (which may just be the global scope). That’s

how they go from one place to another. When the exception handler is finished, the exception objects are properly destroyed.

Summary

Error recovery is a fundamental concern for every program you write, and it's especially important in C++, where one of the goals is to create program components for others to use. To create a robust system, each component must be robust.

The goals for exception handling in C++ are to simplify the creation of large, reliable programs using less code than currently possible, with more confidence that your application doesn't have an unhandled error. This is accomplished with little or no performance penalty, and with low impact on existing code.

Basic exceptions are not terribly difficult to learn, and you should begin using them in your programs as soon as you can. Exceptions are one of those features that provide immediate and significant benefits to your project.

Exercises

1. Create a class with member functions that throw exceptions. Within this class, make a nested class to use as an exception object. It takes a single **char*** as its argument; this represents a description string. Create a member function that throws this exception. (State this in the function's exception specification.) Write a try block that calls this function and a catch clause that handles the exception by printing out its description string.
2. Rewrite the **Stash** class from Chapter XX so it throws out-of-range exceptions for **operator[]**.
3. Write a generic **main( )** that takes all exceptions and reports them as errors.
4. Create a class with its own **operator new**. This operator should allocate 10 objects, and on the 11th "run out of memory" and throw an exception. Also add a static member function that reclaims this memory. Now create a **main( )** with a **try** block and a **catch** clause that calls the memory-restoration routine. Put these inside a **while** loop, to

demonstrate recovering from an exception and continuing execution.

5. Create a destructor that throws an exception, and write code to prove to yourself that this is a bad idea by showing that if a new exception is thrown before the handler for the existing one is reached, **terminate( )** is called.
6. Prove to yourself that all exception objects (the ones that are thrown) are properly destroyed.
7. Prove to yourself that if you create an exception object on the heap and throw the pointer to that object, it will *not* be cleaned up.
8. (Advanced). Track the creation and passing of an exception using a class with a constructor and copy-constructor that announce themselves and provide as much information as possible about how the object is being created (and in the case of the copy-constructor, what object it's being created from). Set up an interesting situation, throw an object of your new type, and analyze the result.

24: Run-time type identification

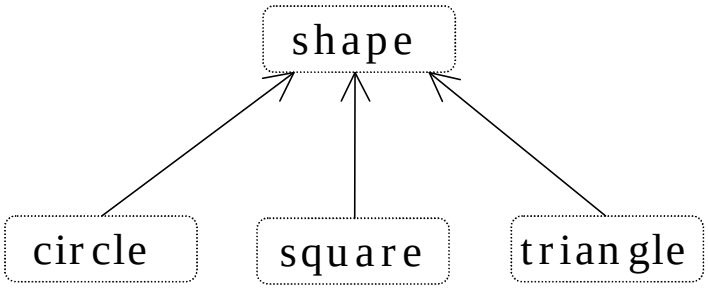
Run-time type identification (RTTI) lets you find the exact type of an object when you have only a pointer or reference to the base type.

This can be thought of as a “secondary” feature in C++, a pragmatism to help out when you get into messy situations. Normally, you’ll want to intentionally ignore the exact type of an object and let the virtual function mechanism implement the correct behavior for that type. But occasionally it’s useful to know the exact type of an object for which you only have a base pointer. Often this information allows you to perform a special-case operation more efficiently or prevent a base-class interface from becoming ungainly. It happens enough that most class libraries contain virtual functions to produce run-time type information. When exception handling was added to C++, it required the exact type information about objects. It became an easy next step to build access to that information into the language.

This chapter explains what RTTI is for and how to use it. In addition, it explains the why and how of the new C++ cast syntax, which has the same appearance as RTTI.

The “Shape” example

This is an example of a class hierarchy that uses polymorphism. The generic type is the base class **Shape**, and the specific derived types are **Circle**, **Square**, and **Triangle**:



This is a typical class-hierarchy diagram, with the base class at the top and the derived classes growing downward. The normal goal in object-oriented programming is for the bulk of your code to manipulate pointers to the base type (**Shape**, in this case) so if you decide to extend the program by adding a new class (**rhomboid**, derived from **Shape**, for example), the bulk of the code is not affected. In this example, the virtual function in the **Shape** interface is **draw( )**, so the intent is for the client programmer to call **draw( )** through a generic **Shape** pointer. **draw( )** is redefined in all the derived classes, and because it is a virtual function, the proper behavior will occur even though it is called through a generic **Shape** pointer.

Thus, you generally create a specific object (**Circle**, **Square**, or **Triangle**), take its address and cast it to a **Shape*** (forgetting the specific type of the object), and use that anonymous pointer in the rest of the program. Historically, diagrams are drawn as seen above, so the act of casting from a more derived type to a base type is called *upcasting*.

What is RTTI?

But what if you have a special programming problem that’s easiest to solve if you know the exact type of a generic pointer? For example, suppose you want to allow your users to highlight all the shapes of any particular type by turning them purple. This way, they can find all the triangles on the screen by highlighting them. Your natural first approach may be to try a virtual function like **TurnColorIfYouAreA( )**, which

allows enumerated arguments of some type **color** and of **Shape::Circle**, **Shape::Square**, or **Shape::Triangle**.

To solve this sort of problem, most class library designers put virtual functions in the base class to return type information about the specific object at runtime. You may have seen library member functions with names like **isA( )** and **typeOf( )**. These are vendor-defined RTTI functions. Using these functions, as you go through the list you can say, "If you're a triangle, turn purple."

When exception handling was added to C++, the implementation required that some run-time type information be put into the virtual function tables. This meant that with a small language extension the programmer could also get the run-time type information about an object. All library vendors were adding their own RTTI anyway, so it was included in the language.

RTTI, like exceptions, depends on type information residing in the virtual function table. If you try to use RTTI on a class that has no virtual functions, you'll get unexpected results.

Two syntaxes for RTTI

There are two different ways to use RTTI. The first acts like **sizeof( )** because it looks like a function, but it's actually implemented by the compiler. **typeid( )** takes an argument that's an object, a reference, or a pointer and returns a reference to a global **const** object of type **typeinfo**. These can be compared to each other with the **operator==** and **operator!=**, and you can also ask for the **name( )** of the type, which returns a string representation of the type name. Note that if you hand **typeid( )** a **Shape***, it will say that the type is **Shape***, so if you want to know the exact type it is pointing to, you must dereference the pointer. For example, if **s** is a **Shape***,

```
cout << typeid(*s).name() << endl;
```

will print out the type of the object **s** points to.

You can also ask a **typeinfo** object if it precedes another **typeinfo** object in the implementation-defined "collation sequence," using **before(typeinfo&)**, which returns true or false. When you say,

```
if(typeid(me).before(typeid(you))) // ...
```

you're asking if **me** occurs before **you** in the collation sequence.

The second syntax for RTTI is called a "type-safe downcast." The reason for the term "downcast" is (again) the historical arrangement of the class hierarchy diagram. If casting a **Circle*** to a **Shape*** is an upcast, then casting a **Shape*** to a **Circle*** is a downcast. However, you know a **Circle*** is also a **Shape***,and the compiler freely allows an upcast assignment, but you *don't* know that a **Shape*** is necessarily a **Circle***, so the compiler doesn't allow you to perform a downcast assignment without using an explicit cast. You can of course force your way through using ordinary C-style casts or a C++ **static_cast** (described at the end of this chapter), which says, "I hope this is actually a **Circle***, and I'm going to pretend it is." Without some explicit knowledge that it *is* in fact a **Circle**, this is a totally dangerous thing to do. A common approach in vendor-defined RTTI is to create some function that attempts to assign (for this example) a **Shape*** to a **Circle***, checking the type in the process. If this function returns the address, it was successful; if it returns null, you didn't have a **Circle***.

The C++ RTTI typesafe-downcast follows this "attempt-to-cast" function form, but it uses (very logically) the template syntax to produce the special function **dynamic_cast**. So the example becomes

```
Shape* sp = new Circle;
Circle* cp = dynamic_cast<Circle*>(sp);
if(cp) cout << "cast successful";
```

The template argument for **dynamic_cast** is the type you want the function to produce, and this is the return value for the function. The function argument is what you are trying to cast from.

Normally you might be hunting for one type (triangles to turn purple, for instance), but the following example fragment can be used if you want to count the number of various shapes.

```
Circle* cp = dynamic_cast<Circle*>(sh);
Square* sp = dynamic_cast<Square*>(sh);
Triangle* tp = dynamic_cast<Triangle*>(sh);
```

Of course this is contrived – you'd probably put a **static** data member in each type and increment it in the constructor. You would do something like that *if* you had control of the source code for the class and could change it. Here's an example that counts shapes using both the **static** member approach and **dynamic_cast**:

```
//: C24:Rtshapes.cpp
// Counting shapes
#include "../purge.h"
```

```
#include <iostream>
#include <ctime>
#include <typeinfo>
#include <vector>
using namespace std;

class Shape {
protected:
  static int count;
public:
  Shape() { count++; }
  virtual ~Shape() { count--; }
  virtual void draw() const = 0;
  static int quantity() { return count; }
};

int Shape::count = 0;

class SRectangle : public Shape {
  void operator=(SRectangle&); // Disallow
protected:
  static int count;
public:
  SRectangle() { count++; }
  SRectangle(const SRectangle&) { count++;}
  ~SRectangle() { count--; }
  void draw() const {
    cout << "SRectangle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int SRectangle::count = 0;

class SEllipse : public Shape {
  void operator=(SEllipse&); // Disallow
protected:
  static int count;
public:
  SEllipse() { count++; }
  SEllipse(const SEllipse&) { count++; }
  ~SEllipse() { count--; }
```

```
  void draw() const {
    cout << "SEllipse::draw()" << endl;
  }
  static int quantity() { return count; }
};

int SEllipse::count = 0;

class SCircle : public SEllipse {
  void operator=(SCircle&); // Disallow
protected:
  static int count;
public:
  SCircle() { count++; }
  SCircle(const SCircle&) { count++; }
  ~SCircle() { count--; }
  void draw() const {
    cout << "SCircle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int SCircle::count = 0;

int main() {
  vector<Shape*> shapes;
  srand(time(0)); // Seed random number generator
  const int mod = 12;
  // Create a random quantity of each type:
  for(int i = 0; i < rand() % mod; i++)
    shapes.push_back(new SRectangle);
  for(int j = 0; j < rand() % mod; j++)
    shapes.push_back(new SEllipse);
  for(int k = 0; k < rand() % mod; k++)
    shapes.push_back(new SCircle);
  int nCircles = 0;
  int nEllipses = 0;
  int nRects = 0;
  int nShapes = 0;
  for(int u = 0; u < shapes.size(); u++) {
    shapes[u]->draw();
    if(dynamic_cast<SCircle*>(shapes[u]))
```

```
      nCircles++;
    if(dynamic_cast<SEllipse*>(shapes[u]))
      nEllipses++;
    if(dynamic_cast<SRectangle*>(shapes[u]))
      nRects++;
    if(dynamic_cast<Shape*>(shapes[u]))
      nShapes++;
  }
  cout << endl << endl
    << "Circles = " << nCircles << endl
    << "Ellipses = " << nEllipses << endl
    << "Rectangles = " << nRects << endl
    << "Shapes = " << nShapes << endl
    << endl
    << "SCircle::quantity() = "
    << SCircle::quantity() << endl
    << "SEllipse::quantity() = "
    << SEllipse::quantity() << endl
    << "SRectangle::quantity() = "
    << SRectangle::quantity() << endl
    << "Shape::quantity() = "
    << Shape::quantity() << endl;
  purge(shapes);
} ///:~
```

Both types work for this example, but the **static** member approach can be used only if you own the code and have installed the **static** members and functions (or if a vendor provides them for you). In addition, the syntax for RTTI may then be different from one class to another.

Syntax specifics

This section looks at the details of how the two forms of RTTI work, and how they differ.

typeid( ) with built-in types

For consistency, the **typeid( )** operator works with built-in types. So the following expressions are true:

```
//: C24:TypeidAndBuiltins.cpp
#include <cassert>
```

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

int main() {
  assert(typeid(47) == typeid(int));
  assert(typeid(0) == typeid(int));
  int i;
  assert(typeid(i) == typeid(int));
  assert(typeid(&i) == typeid(int*));
} ///:~
```

Producing the proper type name

typeid( ) must work properly in all situations. For example, the following class contains a nested class:

```
//: C24:RTTIandNesting.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class One {
  class Nested {};
  Nested* n;
public:
  One() : n(new Nested) {}
  ~One() { delete n; }
  Nested* nested() { return n; }
};

int main() {
  One o;
  cout << typeid(*o.nested()).name() << endl;
} ///:~
```

The **typeinfo::name( )** member function will still produce the proper class name; the result is **One::Nested**.

Nonpolymorphic types

Although **typeid( )** works with nonpolymorphic types (those that don't have a virtual function in the base class), the information you get this way is dubious. For the following class hierarchy,

```
//: C24:RTTIWithoutPolymorphism.cpp
#include <cassert>
#include <typeinfo>
using namespace std;

class X {
 int i;
public:
  // ...
};

class Y : public X {
  int j;
public:
  // ...
};

int main() {
  X* xp = new Y;
  assert(typeid(*xp) == typeid(X));
  assert(typeid(*xp) != typeid(Y));
} ///:~
```

If you create an object of the derived type and upcast it,

```
X* xp = new Y;
```

The **typeid( )** operator will produce results, but not the ones you might expect. Because there's no polymorphism, the static type information is used:

```
typeid(*xp) == typeid(X)
typeid(*xp) != typeid(Y)
```

RTTI is intended for use only with polymorphic classes.

Casting to intermediate levels

dynamic_cast can detect both exact types and, in an inheritance hierarchy with multiple levels, intermediate types. For example,

```
//: C24:DynamicCast.cpp
// Using the standard dynamic_cast operation
#include <cassert>
#include <typeinfo>
```

```
using namespace std;

class D1 {
public:
  virtual void func() {}
  virtual ~D1() {}
};

class D2 {
public:
  virtual void bar() {}
};

class MI : public D1, public D2 {};
class Mi2 : public MI {};

int main() {
  D2* d2 = new Mi2;
  Mi2* mi2 = dynamic_cast<Mi2*>(d2);
  MI* mi = dynamic_cast<MI*>(d2);
  D1* d1 = dynamic_cast<D1*>(d2);
  assert(typeid(d2) != typeid(Mi2*));
  assert(typeid(d2) == typeid(D2*));
} ///:~
```

This has the extra complication of multiple inheritance. If you create an **mi2** and upcast it to the root (in this case, one of the two possible roots is chosen), then the **dynamic_cast** back to either of the derived levels **MI** or **mi2** is successful.

You can even cast from one root to the other:

```
  D1* d1 = dynamic_cast<D1*>(d2);
```

This is successful because **D2** is actually pointing to an **mi2** object, which contains a subobject of type **d1**.

Casting to intermediate levels brings up an interesting difference between **dynamic_cast** and **typeid( )**. **typeid( )** always produces a reference to a **typeinfo** object that describes the *exact* type of the object. Thus it doesn't give you intermediate-level information. In the following expression (which is true), **typeid( )** doesn't see **d2** as a pointer to the derived type, like **dynamic_cast** does:

```
typeid(d2) != typeid(Mi2*)
```

The type of **D2** is simply the exact type of the pointer:

```
typeid(d2) == typeid(D2*)
```

void pointers

Run-time type identification doesn't work with **void** pointers:

```
//: C24:Voidrtti.cpp
// RTTI & void pointers
#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
  virtual void happy() {}
  virtual void joy() {}
  virtual ~Stimpy() {}
};

int main() {
  void* v = new Stimpy;
  // Error:
//!  Stimpy* s = dynamic_cast<Stimpy*>(v);
  // Error:
//!  cout << typeid(*v).name() << endl;
} ///:~
```

A **void*** truly means "no type information at all."

Using RTTI with templates

Templates generate many different class names, and sometimes you'd like to print out information about what class you're in. RTTI provides a convenient way to do this. The following example revisits the code in Chapter XX to print out the order of constructor and destructor calls without using a preprocessor macro:

```
//: C24:ConstructorOrder.cpp
// Order of constructor calls
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
template<int id> class Announce {
public:
  Announce() {
    cout << typeid(*this).name()
         << " constructor " << endl;
  }
  ~Announce() {
    cout << typeid(*this).name()
         << " destructor " << endl;
  }
};

class X : public Announce<0> {
  Announce<1> m1;
  Announce<2> m2;
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

int main() { X x; } ///:~
```

The **<typeinfo>** header must be included to call any member functions for the **typeinfo** object returned by **typeid( )**. The template uses a constant **int** to differentiate one class from another, but class arguments will work as well. Inside both the constructor and destructor, RTTI information is used to produce the name of the class to print. The class **X** uses both inheritance and composition to create a class that has an interesting order of constructor and destructor calls.

This technique is often useful in situations when you're trying to understand how the language works.

References

RTTI must adjust somewhat to work with references. The contrast between pointers and references occurs because a reference is always dereferenced for you by the compiler, whereas a pointer's type *or* the type it points to may be examined. Here's an example:

```
//: C24:RTTIwithReferences.cpp
#include <cassert>
```

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

class B {
public:
  virtual float f() { return 1.0;}
  virtual ~B() {}
};

class D : public B { /* ... */ };

int main() {
  B* p = new D;
  B& r = *p;
  assert(typeid(p) == typeid(B*));
  assert(typeid(p) != typeid(D*));
  assert(typeid(r) == typeid(D));
  assert(typeid(*p) == typeid(D));
  assert(typeid(*p) != typeid(B));
  assert(typeid(&r) == typeid(B*));
  assert(typeid(&r) != typeid(D*));
  assert(typeid(r.f()) == typeid(float));
} ///:~
```

Whereas the type of pointer that **typeid( )** sees is the base type and not the derived type, the type it sees for the reference is the derived type:

```
typeid(p) == typeid(B*)
typeid(p) != typeid(D*)
typeid(r) == typeid(D)
```

Conversely, what the pointer points to is the derived type and not the base type, and taking the address of the reference produces the base type and not the derived type:

```
typeid(*p) == typeid(D)
typeid(*p) != typeid(B)
typeid(&r) == typeid(B*)
typeid(&r) != typeid(D*)
```

Expressions may also be used with the **typeid( )** operator because they have a type as well:

```
typeid(r.f()) == typeid(float)
```

Exceptions

When you perform a **dynamic_cast** to a reference, the result must be assigned to a reference. But what happens if the cast fails? There are no null references, so this is the perfect place to throw an exception; the Standard C++ exception type is **bad_cast**, but in the following example the ellipses are used to catch any exception:

```
//: C24:RTTIwithExceptions.cpp
#include <typeinfo>
#include <iostream>
using namespace std;
class X { public: virtual ~X(){} };
class B { public: virtual ~B(){} };
class D : public B {};

int main() {
  D d;
  B & b = d; // Upcast to reference
  try {
    X& xr = dynamic_cast<X&>(b);
  } catch(...) {
    cout << "dynamic_cast<X&>(b) failed"
         << endl;
  }
  X* xp = 0;
  try {
    typeid(*xp); // Throws exception
  } catch(bad_typeid) {
    cout << "Bad typeid() expression" << endl;
  }
} ///:~
```

The failure, of course, is because **b** doesn't actually point to an **X** object. If an exception was not thrown here, then **xr** would be unbound, and the guarantee that all objects or references are constructed storage would be broken.

An exception is also thrown if you try to dereference a null pointer in the process of calling **typeid( )**. The Standard C++ exception is called **bad_typeid**.

Here (unlike the reference example above) you can avoid the exception by checking for a nonzero pointer value before attempting the operation; this is the preferred practice.

Multiple inheritance

Of course, the RTTI mechanisms must work properly with all the complexities of multiple inheritance, including **virtual** base classes:

```
//: C24:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
  virtual void f() {}
  virtual ~BB() {}
};
class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
  BB* bbp = new MI; // Upcast
  // Proper name detection:
  cout << typeid(*bbp).name() << endl;
  // Dynamic_cast works properly:
  MI* mip = dynamic_cast<MI*>(bbp);
  // Can't force old-style cast:
  //! MI* mip2 = (MI*)bbp; // Compile error
} ///:~
```

typeid( ) properly detects the name of the actual object, even through the **virtual** base class pointer. The **dynamic_cast** also works correctly. But the compiler won't even allow you to try to force a cast the old way:

```
MI* mip = (MI*)bbp; // Compile-time error
```

It knows this is never the right thing to do, so it requires that you use a **dynamic_cast**.

Sensible uses for RTTI

Because it allows you to discover type information from an anonymous polymorphic pointer, RTTI is ripe for misuse by the novice because RTTI may make sense before virtual functions do. For many people coming from a procedural background, it's very difficult not to organize their programs into sets of **switch** statements. They could accomplish this with RTTI and thus lose the very important value of polymorphism in code development and maintenance. The intent of C++ is that you use virtual functions throughout your code, and you only use RTTI when you must.

However, using virtual functions as they are intended requires that you have control of the base-class definition because at some point in the extension of your program you may discover the base class doesn't include the virtual function you need. If the base class comes from a library or is otherwise controlled by someone else, a solution to the problem is RTTI: You can inherit a new type and add your extra member function. Elsewhere in the code you can detect your particular type and call that member function. This doesn't destroy the polymorphism and extensibility of the program, because adding a new type will not require you to hunt for switch statements. However, when you add new code in your main body that requires your new feature, you'll have to detect your particular type.

Putting a feature in a base class might mean that, for the benefit of one particular class, all the other classes derived from that base require some meaningless stub of a virtual function. This makes the interface less clear and annoys those who must redefine pure virtual functions when they derive from that base class. For example, suppose that in the **Wind5.cpp** program in Chapter XX you wanted to clear the spit valves of all the instruments in your orchestra that had them. One option is to put a **virtual ClearSpitValve( )** function in the base class **Instrument**, but this is confusing because it implies that **Percussion** and **electronic** instruments also have spit valves. RTTI provides a much more reasonable solution in this case because you can place the function in the specific class (**Wind** in this case) where it's appropriate.

Finally, RTTI will sometimes solve efficiency problems. If your code uses polymorphism in a nice way, but it turns out that one of your objects reacts to this general-purpose code in a horribly inefficient way, you can pick that type out using RTTI and write case-specific code to improve the efficiency.

Revisiting the trash recycler

Here's the trash recycling simulation from Chapter XX, rewritten to use RTTI instead of building the information into the class hierarchy:

```
//: C24:Recycle2.cpp
// Chapter XX example w/ RTTI
#include "../purge.h"
#include <fstream>
#include <vector>
#include <typeinfo>
#include <cstdlib>
#include <ctime>
using namespace std;
ofstream out("recycle2.out");

class Trash {
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() { out << "~Trash()\n"; }
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float Aluminum::val = 1.67;

class Paper : public Trash {
  static float val;
public:
  Paper(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(int newval) {
```

```
    val = newval;
  }
};

float Paper::val = 0.10;

class Glass : public Trash {
  static float val;
public:
  Glass(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float Glass::val = 0.23;

// Sums up the value of the Trash in a bin:
template<class Container> void
sumValue(Container& bin, ostream& os) {
  typename Container::iterator tally =
    bin.begin();
  float val = 0;
  while(tally != bin.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of "
       << typeid(*tally).name()
       << " = " << (*tally)->weight() << endl;
    tally++;
  }
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Seed random number generator
  vector<Trash*> bin;
  // Fill up the Trash bin:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push_back(new Aluminum(rand() % 100));
```

```
      break;
    case 1 :
      bin.push_back(new Paper(rand() % 100));
      break;
    case 2 :
      bin.push_back(new Glass(rand() % 100));
      break;
  }
// Note difference w/ chapter 14: Bins hold
// exact type of object, not base type:
vector<Glass*> glassBin;
vector<Paper*> paperBin;
vector<Aluminum*> alBin;
vector<Trash*>::iterator sorter = bin.begin();
// Sort the Trash:
while(sorter != bin.end()) {
  Aluminum* ap =
    dynamic_cast<Aluminum*>(*sorter);
  Paper* pp =
    dynamic_cast<Paper*>(*sorter);
  Glass* gp =
    dynamic_cast<Glass*>(*sorter);
  if(ap) alBin.push_back(ap);
  if(pp) paperBin.push_back(pp);
  if(gp) glassBin.push_back(gp);
  sorter++;
}
sumValue(alBin, out);
sumValue(paperBin, out);
sumValue(glassBin, out);
sumValue(bin, out);
purge(bin);
} ///:~
```

The nature of this problem is that the trash is thrown unclassified into a single bin, so the specific type information is lost. But later, the specific type information must be recovered to properly sort the trash, and so RTTI is used. In Chapter XX, an RTTI system was inserted into the class hierarchy, but as you can see here, it's more convenient to use C++'s built-in RTTI.

Mechanism & overhead of RTTI

Typically, RTTI is implemented by placing an additional pointer in the VTABLE. This pointer points to the **typeinfo** structure for that particular type. (Only one instance of the **typeinfo** structure is created for each new class.) So the effect of a **typeid( )** expression is quite simple: The VPTR is used to fetch the **typeinfo** pointer, and a reference to the resulting **typeinfo** structure is produced. Also, this is a deterministic process – you always know how long it's going to take.

For a **dynamic_cast<destination*>(source_pointer)**, most cases are quite straightforward: **source_pointer**'s RTTI information is retrieved, and RTTI information for the type **destination*** is fetched. Then a library routine determines whether **source_pointer**'s type is of type **destination*** or a base class of **destination***. The pointer it returns may be slightly adjusted because of multiple inheritance if the base type isn't the first base of the derived class. The situation is (of course) more complicated with multiple inheritance where a base type may appear more than once in an inheritance hierarchy and where virtual base classes are used.

Because the library routine used for **dynamic_cast** must check through a list of base classes, the overhead for **dynamic_cast** is higher than **typeid( )** (but of course you get different information, which may be essential to your solution), and it's nondeterministic because it may take more time to discover a base class than a derived class. In addition, **dynamic_cast** allows you to compare any type to any other type; you aren't restricted to comparing types within the same hierarchy. This adds extra overhead to the library routine used by **dynamic_cast**.

Creating your own RTTI

If your compiler doesn't yet support RTTI, you can build it into your class libraries quite easily. This makes sense because RTTI was added to the language after observing that virtually all class libraries had some form of it anyway (and it was relatively "free" after exception handling was added because exceptions require exact knowledge of type information).

Essentially, RTTI requires only a virtual function to identify the exact type of the class, and a function to take a pointer to the base type and cast it down to the more derived type; this function must produce a pointer to the more derived type. (You may also wish to handle references.) There are a number of approaches to implement your own RTTI, but all require a unique identifier for each class and a virtual function to produce type information. The following uses a **static** member function called **dynacast( )** that calls a type information function **dynamic_type( )**. Both functions must be defined for each new derivation:

```
//: C24:Selfrtti.cpp
// Your own RTTI system
#include "../purge.h"
#include <iostream>
#include <vector>
using namespace std;

class Security {
protected:
  static const int baseID = 1000;
public:
  virtual int dynamic_type(int id) {
    if(id == baseID) return 1;
    return 0;
  }
};

class Stock : public Security {
protected:
  static const int typeID = baseID + 1;
public:
  int dynamic_type(int id) {
    if(id == typeID) return 1;
    return Security::dynamic_type(id);
  }
  static Stock* dynacast(Security* s) {
    if(s->dynamic_type(typeID))
      return (Stock*)s;
    return 0;
  }
};

class Bond : public Security {
```

```
protected:
  static const int typeID = baseID + 2 ;
public:
  int dynamic_type(int id) {
    if(id == typeID) return 1;
    return Security::dynamic_type(id);
  }
  static Bond* dynacast(Security* s) {
    if(s->dynamic_type(typeID))
      return (Bond*)s;
    return 0;
  }
};

class Commodity : public Security {
protected:
  static const int typeID = baseID + 3;
public:
  int dynamic_type(int id) {
    if(id == typeID) return 1;
    return Security::dynamic_type(id);
  }
  static Commodity* dynacast(Security* s) {
    if(s->dynamic_type(typeID))
      return (Commodity*)s;
    return 0;
  }
  void special() {
    cout << "special Commodity function\n";
  }
};

class Metal : public Commodity {
protected:
  static const int typeID = baseID + 4;
public:
  int dynamic_type(int id) {
    if(id == typeID) return 1;
    return Commodity::dynamic_type(id);
  }
  static Metal* dynacast(Security* s) {
    if(s->dynamic_type(typeID))
```

```
      return (Metal*)s;
    return 0;
  }
};

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Commodity);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  vector<Security*>::iterator it =
    portfolio.begin();
  while(it != portfolio.end()) {
    Commodity* cm = Commodity::dynacast(*it);
    if(cm) cm->special();
    else cout << "not a Commodity" << endl;
    it++;
  }
  cout << "cast from intermediate pointer:\n";
  Security* sp = new Metal;
  Commodity* cp = Commodity::dynacast(sp);
  if(cp) cout << "it's a Commodity\n";
  Metal* mp = Metal::dynacast(sp);
  if(mp) cout << "it's a Metal too!\n";
  purge(portfolio);
} ///:~
```

Each subclass must create its own **typeID**, redefine the **virtual dynamic_type( )** function to return that **typeID**, and define a **static** member called **dynacast( )**, which takes the base pointer (or a pointer at any level in a deeper hierarchy – in that case, the pointer is simply upcast).

In the classes derived from **Security**, you can see that each defines its own **typeID** enumeration by adding to **baseID**. It's essential that **baseID** be directly accessible in the derived class because the **enum** must be evaluated at compile-time, so the usual approach of reading private data with an **inline** function would fail. This is a good example of the need for the **protected** mechanism.

The **enum baseID** establishes a base identifier for all types derived from **Security**. That way, if an identifier clash ever occurs, you can change all the identifiers by changing the base value. (However, because this

scheme doesn't compare different inheritance trees, an identifier clash is unlikely). In all the classes, the class identifier number is **protected**, so it's directly available to derived classes but not to the end user.

This example illustrates what built-in RTTI must cope with. Not only must you be able to determine the exact type, you must also be able to find out whether your exact type is *derived from* the type you're looking for. For example, **Metal** is derived from **Commodity**, which has a function called **special( )**, so if you have a **Metal** object you can call **special( )** for it. If **dynamic_type( )** told you only the exact type of the object, you could ask it if a **Metal** were a **Commodity**, and it would say "no," which is untrue. Therefore, the system must be set up so it will properly cast to intermediate types in a hierarchy as well as exact types.

The **dynacast( )** function determines the type information by calling the **virtual dynamic_type( )** function for the **Security** pointer it's passed. This function takes an argument of the **typeID** for the class you're trying to cast to. It's a virtual function, so the function body is the one for the exact type of the object. Each **dynamic_type( )** function first checks to see if the identifier it was passed is an exact match for its own type. If that isn't true, it must check to see if it matches a base type; this is accomplished by making a call to the base class **dynamic_type( )**. Just like a recursive function call, each **dynamic_type( )** checks against its own identifier. If it doesn't find a match, it returns the result of calling the base class **dynamic_type( )**. When the root of the hierarchy is reached, zero is returned to indicate no match was found.

If **dynamic_type( )** returns one (for "true") the object pointed to is either the exact type you're asking about or derived from that type, and **dynacast( )** takes the **Security** pointer and casts it to the desired type. If the return value is false, **dynacast( )** returns zero to indicate the cast was unsuccessful. In this way it works just like the C++ **dynamic_cast** operator.

The C++ **dynamic_cast** operator does one more thing the above scheme can't do: It compares types from one inheritance hierarchy to another, completely separate inheritance hierarchy. This adds generality to the system for those unusual cases where you want to compare across hierarchies, but it also adds some complexity and overhead.

You can easily imagine how to create a DYNAMIC_CAST macro that uses the above scheme and allows an easier transition to the built-in **dynamic_cast** operator.

Explicit cast syntax

Whenever you use a cast, you're breaking the type system. [69] You're telling the compiler that even though you know an object is a certain type, you're going to pretend it is a different type. This is an inherently dangerous activity, and a clear source of errors.

Unfortunately, each cast is different: the name of the pretender type surrounded by parentheses. So if you are given a piece of code that isn't working correctly and you know you want to examine all casts to see if they're the source of the errors, how can you guarantee that you find all the casts? In a C program, you can't. For one thing, the C compiler doesn't always require a cast (it's possible to assign dissimilar types *through* a void pointer without being forced to use a cast), and the casts all look different, so you can't know if you've searched for every one.

To solve this problem, C++ provides a consistent casting syntax using four reserved words: **dynamic_cast** (the subject of the first part of this chapter), **const_cast**, **static_cast,** and **reinterpret_cast**. This window of opportunity opened up when the need for **dynamic_cast** arose – the meaning of the existing cast syntax was already far too overloaded to support any additional functionality.

By using these casts instead of the **(newtype)** syntax, you can easily search for all the casts in any program. To support existing code, most compilers have various levels of error/warning generation that can be turned on and off. But if you turn on full errors for the explicit cast syntax, you can be guaranteed that you'll find all the places in your project where casts occur, which will make bug-hunting much easier.

The following table describes the different forms of casting:

static_cast	For "well-behaved" and "reasonably well-behaved" casts, including things you might now do without a cast (e.g., an upcast or automatic type conversion).

[69] See Josée Lajoie , "The new cast notation and the bool data type," C++ Report, September, 1994 pp. 46-51.

const_cast	To cast away **const** and/or **volatile**.
dynamic_cast	For type-safe downcasting (described earlier in the chapter).
reinterpret_cast	To cast to a completely different meaning. The key is that you'll need to cast back to the original type to use it safely. The type you cast to is typically used only for bit twiddling or some other mysterious purpose. This is the most dangerous of all the casts.

The three explicit casts will be described more completely in the following sections.

static_cast

A **static_cast** is used for all conversions that are well-defined. These include "safe" conversions that the compiler would allow you to do without a cast and less-safe conversions that are nonetheless well-defined. The types of conversions covered by **static_cast** include typical castless conversions, narrowing (information-losing) conversions, forcing a conversion from a **void***, implicit type conversions, and static navigation of class hierarchies:

```
//: C24:Statcast.cpp
// Examples of static_cast

class Base { /* ... */ };
class Derived : public Base {
public:
  // ...
  // Automatic type conversion:
  operator int() { return 1; }
};

void func(int) {}

class Other {};

int main() {
```

```
int i = 0x7fff; // Max pos value = 32767
long l;
float f;
// (1) typical castless conversions:
l = i;
f = i;
// Also works:
l = static_cast<long>(i);
f = static_cast<float>(i);

// (2) narrowing conversions:
i = l; // May lose digits
i = f; // May lose info
// Says "I know," eliminates warnings:
i = static_cast<int>(l);
i = static_cast<int>(f);
char c = static_cast<char>(i);

// (3) forcing a conversion from void* :
void* vp = &i;
// Old way produces a dangerous conversion:
float* fp = (float*)vp;
// The new way is equally dangerous:
fp = static_cast<float*>(vp);

// (4) implicit type conversions, normally
// Performed by the compiler:
Derived d;
Base* bp = &d; // Upcast: normal and OK
bp = static_cast<Base*>(&d); // More explicit
int x = d; // Automatic type conversion
x = static_cast<int>(d); // More explicit
func(d); // Automatic type conversion
func(static_cast<int>(d)); // More explicit

// (5) Static Navigation of class hierarchies:
Derived* dp = static_cast<Derived*>(bp);
// ONLY an efficiency hack. dynamic_cast is
// Always safer. However:
// Other* op = static_cast<Other*>(bp);
// Conveniently gives an error message, while
Other* op2 = (Other*)bp;
```

```
  // Does not.
} ///:~
```

In Section (1), you see the kinds of conversions you're used to doing in C, with or without a cast. Promoting from an **int** to a **long** or **float** is not a problem because the latter can always hold every value that an **int** can contain. Although it's unnecessary, you can use **static_cast** to highlight these promotions.

Converting back the other way is shown in (2). Here, you can lose data because an **int** is not as "wide" as a **long** or a **float** – it won't hold numbers of the same size. Thus these are called "narrowing conversions." The compiler will still perform these, but will often give you a warning. You can eliminate this warning and indicate that you really did mean it using a cast.

Assigning from a **void*** is not allowed without a cast in C++ (unlike C), as seen in (3). This is dangerous and requires that a programmer know what he's doing. The **static_cast**, at least, is easier to locate than the old standard cast when you're hunting for bugs.

Section (4) shows the kinds of implicit type conversions that are normally performed automatically by the compiler. These are automatic and require no casting, but again **static_cast** highlights the action in case you want to make it clear what's happening or hunt for it later.

If a class hierarchy has no **virtual** functions or if you have other information that allows you to safely downcast, it's slightly faster to do the downcast statically than with **dynamic_cast**, as shown in (5). In addition, **static_cast** won't allow you to cast out of the hierarchy, as the traditional cast will, so it's safer. However, statically navigating class hierarchies is always risky and you should use **dynamic_cast** unless you have a special situation.

const_cast

If you want to convert from a **const** to a non**const** or from a **volatile** to a non**volatile**, you use **const_cast**. This is the *only* conversion allowed with **const_cast**; if any other conversion is involved it must be done separately or you'll get a compile-time error.

```
//: C24:Constcst.cpp
// Const casts

int main() {
```

```
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
}

class X {
  int i;
// mutable int i; // A better approach
public:
  void f() const {
    // Casting away const-ness:
    (const_cast<X*>(this))->i = 1;
  }
}; ///:~
```

If you take the address of a **const** object, you produce a pointer to a **const**, and this cannot be assigned to a non**const** pointer without a cast. The old-style cast will accomplish this, but the **const_cast** is the appropriate one to use. The same holds true for **volatile**.

If you want to change a class member inside a **const** member function, the traditional approach is to cast away **const**ness by saying **(X*)this**. You can still cast away **const**ness using the better **const_cast**, but a superior approach is to make that particular data member **mutable**, so it's clear in the class definition, and not hidden away in the member function definitions, that the member may change in a **const** member function.

reinterpret_cast

This is the least safe of the casting mechanisms, and the one most likely to point to bugs. At the very least, your compiler should contain switches to allow you to force the use of **const_cast** and **reinterpret_cast**, which will locate the most unsafe of the casts.

A **reinterpret_cast** pretends that an object is just a bit pattern that can be treated (for some dark purpose) as if it were an entirely different type of object. This is the low-level bit twiddling that C is notorious for. You'll virtually always need to **reinterpret_cast** back to the original type before doing anything else with it.

```
//: C24:Reinterp.cpp
// Reinterpret_cast
// Example depends on VPTR location,
// Which may differ between compilers.
#include <cstring>
#include <fstream>
using namespace std;
ofstream out("reinterp.out");

class X {
  static const int sz = 5 ;
  int a[sz];
public:
  X() { memset(a, 0, sz * sizeof(int)); }
  virtual void f() {}
  // Size of all the data members:
  int membsize() { return sizeof(a); }
  friend ostream&
    operator<<(ostream& os, const X& x) {
      for(int i = 0; i < sz; i++)
        os << x.a[i] << ' ';
      return os;
  }
  virtual ~X() {}
};

int main() {
  X x;
  out << x << endl; // Initialized to zeroes
  int* xp = reinterpret_cast<int*>(&x);
  xp[1] = 47;
  out << x << endl; // Oops!

  X x2;
  const int vptr_size= sizeof(X) - x2.membsize();
  long l = reinterpret_cast<long>(&x2);
  // *IF* the VPTR is first in the object:
  l += vptr_size; // Move past VPTR
  xp = reinterpret_cast<int*>(l);
  xp[1] = 47;
  out << x2 << endl;
} ///:~
```

The **class X** contains some data and a **virtual** member function. In **main( )**, an **X** object is printed out to show that it gets initialized to zero, and then its address is cast to an **int*** using a **reinterpret_cast**. Pretending it's an **int***, the object is indexed into as if it were an array and (in theory) element one is set to 47. But here's the output:[70]

```
0 0 0 0 0
47 0 0 0 0
```

Clearly, it's not safe to assume that the data in the object begins at the starting address of the object. In fact, this compiler puts the VPTR at the beginning of the object, so if **xp[0]** had been selected instead of **xp[1]**, it would have trashed the VPTR.

To fix the problem, the size of the VPTR is calculated by subtracting the size of the data members from the size of the object. Then the address of the object is cast (again, with **reinterpret_cast**) to a **long**, and the starting address of the actual data is established, *assuming* the VPTR is placed at the beginning of the object. The resulting number is cast back to an **int*** and the indexing now produces the desired result:

```
0 47 0 0 0
```

Of course, this is inadvisable and nonportable programming. That's the kind of thing that a **reinterpret_cast** indicates, but it's available when you decide you have to use it.

Summary

RTTI is a convenient extra feature, a bit of icing on the cake. Although normally you upcast a pointer to a base class and then use the generic interface of that base class (via virtual functions), occasionally you get into a corner where things can be more effective if you know the exact type of the object pointed to by the base pointer, and that's what RTTI provides. Because some form of virtual-function-based RTTI has appeared in almost all class libraries, this is a useful feature because it means

1. You don't have to build it into your own libraries.
2. You don't have to worry whether it will be built into someone else's library.

[70] For this particular compiler. Yours will probably be different.

3. You don't have the extra programming overhead of maintaining an RTTI scheme during inheritance.
4. The syntax is consistent, so you don't have to figure out a new one for each library.

While RTTI is a convenience, like most features in C++ it can be misused by either a naive or determined programmer. The most common misuse may come from the programmer who doesn't understand virtual functions and uses RTTI to do type-check coding instead. The philosophy of C++ seems to be to provide you with powerful tools and guard for type violations and integrity, but if you want to deliberately misuse or get around a language feature, there's nothing to stop you. Sometimes a slight burn is the fastest way to gain experience.

The explicit cast syntax will be a big help during debugging because casting opens a hole into your type system and allows errors to slip in. The explicit cast syntax will allow you to more easily locate these error entryways.

Exercises

1. Use RTTI to assist in program debugging by printing out the exact name of a template using **typeid( )**. Instantiate the template for various types and see what the results are.
2. Implement the function **TurnColorIfYouAreA( )** described earlier in this chapter using RTTI.
3. Modify the **Instrument** hierarchy from Chapter XX by first copying **Wind5.cpp** to a new location. Now add a **virtual ClearSpitValve( )** function to the **Wind** class, and redefine it for all the classes inherited from **Wind**. Instantiate a **TStash** to hold **Instrument** pointers and fill it up with various types of **Instrument** objects created using **new**. Now use RTTI to move through the container looking for objects in class **Wind**, or derived from **Wind**. Call the **ClearSpitValve( )** function for these objects. Notice that it would unpleasantly confuse the **Instrument** base class if it contained a **ClearSpitValve( )** function.

XX: Maintaining system integrity

The canonical object form

An extended canonical form

Dynamic aggregation

The examples we've seen so far are illustrative, but fairly simple. It's useful to see an example that has more complexity so you can see that the STL will work in all situations.

[[Add a factory method that takes a vector of string]]

The class that will be created as the example will be reasonably complex: it's a bicycle which can have a choice of parts. In addition, you can change the parts during the lifetime of a **Bicycle** object; this includes the ability to add new parts or to upgrade from standard-quality parts to "fancy" parts. The **BicyclePart** class is a base class with many different types, and the **Bicycle** class contains a **vector<BicyclePart*>** to hold the various combination of parts that may be attached to a **Bicycle**:

```
//: CXX:Bicycle.h
```

```
// Complex class involving dynamic aggregation
#ifndef BICYCLE_H
#define BICYCLE_H
#include <vector>
#include <string>
#include <iostream>
#include <typeinfo>

class LeakChecker {
  int count;
public:
  LeakChecker() : count(0) {}
  void print() {
    std::cout << count << std::endl;
  }
  ~LeakChecker() { print(); }
  void operator++(int) { count++; }
  void operator--(int) { count--; }
};

class BicyclePart {
  static LeakChecker lc;
public:
  BicyclePart() { lc++; }
  virtual BicyclePart* clone() = 0;
  virtual ~BicyclePart() { lc--; }
  friend std::ostream&
  operator<<(std::ostream& os, BicyclePart* bp) {
    return os << typeid(*bp).name();
  }
  friend class Bicycle;
};

enum BPart {
  Frame, Wheel, Seat, HandleBar,
  Sprocket, Derailleur,
};

template<BPart id>
class Part : public BicyclePart {
public:
  BicyclePart* clone() { return new Part<id>; }
};
```

```
class Bicycle {
public:
  typedef std::vector<BicyclePart*> VBP;
  Bicycle();
  Bicycle(const Bicycle& old);
  Bicycle& operator=(const Bicycle& old);
  // [Other operators as needed go here:]
  // [...]
  // [...]
  ~Bicycle() { purge(); }
  // So you can change parts on a bike (but be
  // careful: you must clean up any objects you
  // remove from the bicycle!)
  VBP& bikeParts() { return parts; }
  friend std::ostream&
  operator<<(std::ostream& os, Bicycle* b);
  static void print(std::vector<Bicycle*>& vb,
    std::ostream& os = std::cout);
private:
  static int counter;
  int id;
  VBP parts;
  void purge();
};

// Both the Bicycle and the generator should
// provide more variety than this. But this gives
// you the idea.
struct BicycleGenerator {
  Bicycle* operator()() {
    return new Bicycle();
  }
};
#endif // BICYCLE_H ///:~
```

The **operator<<** for **ostream** and **Bicycle** moves through and calls the **operator<<** for each **BicyclePart**, and that prints out the class name of the part so you can see what a **Bicycle** contains. The **BicyclePart::clone( )** member function is necessary in the copy-constructor of **Bicycle**, since it just has a **vector<BicyclePart*>** and wouldn't otherwise know how to copy the **BicyclePart**s correctly. The cloning process, of course, will be more involved when there are data members in a **BicyclePart**.

BicyclePart::partcount is used to keep track of the number of parts created and destroyed (so you can detect memory leaks). It is incremented every time a new **BicyclePart** is created and decremented when one is destroyed; also, when **partcount** goes to zero this is reported and if it goes below zero there will be an **assert( )** failure.

If you want to change **BicyclePart**s on a **Bicycle**, you just call **Bicycle::bikeParts( )** to get the **vector<BicyclePart*>** which you can then modify. But whenever you remove a part from a **Bicycle**, you must call **delete** for that pointer, otherwise it won't get cleaned up.

Here's the implementation:

```
//: CXX:Bicycle.cpp {O}
// Bicycle implementation
#include "Bicycle.h"
#include <map>
#include <algorithm>
#include <cassert>
using namespace std;

// Static member definitions:
LeakChecker BicyclePart::lc;
int Bicycle::counter = 0;

Bicycle::Bicycle() : id(counter++) {
  BicyclePart *bp[] = {
    new Part<Frame>,
    new Part<Wheel>, new Part<Wheel>,
    new Part<Seat>, new Part<HandleBar>,
    new Part<Sprocket>,  new Part<Derailleur>,
  };
  const int bplen = sizeof bp / sizeof *bp;
  parts = VBP(bp, bp + bplen);
}

Bicycle::Bicycle(const Bicycle& old)
  : parts(old.parts.begin(), old.parts.end()) {
  for(int i = 0; i < parts.size(); i++)
    parts[i] = parts[i]->clone();
}

Bicycle& Bicycle::operator=(const Bicycle& old) {
  purge(); // Remove old lvalues
  parts.resize(old.parts.size());
```

```
  copy(old.parts.begin(),
    old.parts.end(), parts.begin());
  for(int i = 0; i < parts.size(); i++)
    parts[i] = parts[i]->clone();
  return *this;
}

void Bicycle::purge() {
  for(VBP::iterator it = parts.begin();
    it != parts.end(); it++) {
      delete *it;
      *it = 0; // Prevent multiple deletes
  }
}

ostream& operator<<(ostream& os, Bicycle* b) {
  copy(b->parts.begin(), b->parts.end(),
    ostream_iterator<BicyclePart*>(os, "\n"));
  os << "--------" << endl;
  return os;
}

void Bicycle::print(vector<Bicycle*>& vb,
  ostream& os) {
  copy(vb.begin(), vb.end(),
    ostream_iterator<Bicycle*>(os, "\n"));
  cout << "--------" << endl;
} ///:~
```

Here's a test:

```
//: CXX:BikeTest.cpp
//{L} Bicycle
#include "Bicycle.h"
#include <algorithm>
using namespace std;

int main() {
  vector<Bicycle*> bikes;
  BicycleGenerator bg;
  generate_n(back_inserter(bikes), 12, bg);
  Bicycle::print(bikes);
} ///:~
```

Reference counting

Reference-counted class hierarchies

Exercises

1. Create a heap compactor for all dynamic memory in a particular program. This will require that you control how objects are dynamically created and used (do you overload **operator new** or does that approach work?). The typically heap-compaction scheme requires that all pointers are doubly-indirected (that is, pointers to pointers) so the "middle tier" pointer can be manipulated during compaction. Consider overloading **operator->** to accomplish this, since that operator has special behavior which will probably benefit your heap-compaction scheme. Write a program to test your heap-compaction scheme.

25: Design patterns

> "… describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander

This chapter introduces the important and yet non-traditional "patterns" approach to program design.

[[Much of the prose in this chapter still needs work, but the examples all compile. Also, more patterns and examples are forthcoming]]

Probably the most important step forward in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995).[71] That book shows 23 different solutions to particular classes of problems. In this chapter, the basic concepts of design patterns will be introduced along with examples. This should whet your appetite to read *Design Patterns* (a source of what has now become an essential, almost mandatory, vocabulary for OOP programmers).

The latter part of this chapter contains an example of the design evolution process, starting with an initial solution and moving through the logic and process of evolving the solution to more appropriate designs. The program shown (a trash recycling simulation) has evolved over time, and you can

[71] Conveniently, the examples are in C++.

look at that evolution as a prototype for the way your own design can start as an adequate solution to a particular problem and evolve into a flexible approach to a class of problems.

The pattern concept

Initially, you can think of a pattern as an especially clever and insightful way of solving a particular class of problems. That is, it looks like a lot of people have worked out all the angles of a problem and have come up with the most general, flexible solution for it. The problem could be one you have seen and solved before, but your solution probably didn't have the kind of completeness you'll see embodied in a pattern.

Although they're called "design patterns," they really aren't tied to the realm of design. A pattern seems to stand apart from the traditional way of thinking about analysis, design, and implementation. Instead, a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase. This is interesting because a pattern has a direct implementation in code and so you might not expect it to show up before low-level design or implementation (and in fact you might not realize that you need a particular pattern until you get to those phases).

The basic concept of a pattern can also be seen as the basic concept of program design: adding layers of abstraction. Whenever you abstract something you're isolating particular details, and one of the most compelling motivations behind this is to *separate things that change from things that stay the same*. Another way to put this is that once you find some part of your program that's likely to change for one reason or another, you'll want to keep those changes from propagating other modifications throughout your code. Not only does this make the code much cheaper to maintain, but it also turns out that it is usually simpler to understand (which results in lowered costs).

Often, the most difficult part of developing an elegant and cheap-to-maintain design is in discovering what I call "the vector of change." (Here, "vector" refers to the maximum gradient and not a container class.) This means finding the most important thing that changes in your system, or put another way, discovering where your greatest cost is. Once you discover the vector of change, you have the focal point around which to structure your design.

So the goal of design patterns is to isolate changes in your code. If you look at it this way, you've been seeing some design patterns already in this book. For example, inheritance could be thought of as a design

pattern (albeit one implemented by the compiler). It allows you to express differences in behavior (that's the thing that changes) in objects that all have the same interface (that's what stays the same). Composition could also be considered a pattern, since it allows you to change – dynamically or statically – the objects that implement your class, and thus the way that class works. Normally, however, features that are directly supported by a programming language are not classified as design patterns.

You've also already seen another pattern that appears in *Design Patterns*: the *iterator*. This is the fundamental tool used in the design of the STL; it hides the particular implementation of the container as you're stepping through and selecting the elements one by one. The iterator allows you to write generic code that performs an operation on all of the elements in a range without regard to the container that holds the range. Thus your generic code can be used with any container that can produce iterators.

The singleton

Possibly the simplest design pattern is the *singleton*, which is a way to provide one and only one instance of an object:

```
//: C25:SingletonPattern.cpp
#include <iostream>
using namespace std;

class Singleton {
  static Singleton s;
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& getHandle() {
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

Singleton Singleton::s(47);

int main() {
  Singleton& s = Singleton::getHandle();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::getHandle();
```

```
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

The key to creating a singleton is to prevent the client programmer from having any way to create an object except the ways you provide. To do this, you must declare all constructors as **private**, and you must create at least one constructor to prevent the compiler from synthesizing a default constructor for you.

At this point, you decide how you're going to create your object. Here, it's created statically, but you can also wait until the client programmer asks for one and create it on demand. In any case, the object should be stored privately. You provide access through public methods. Here, **getHandle( )** produces a reference to the **Singleton** object. The rest of the interface (**getValue( )** and **setValue( )**) is the regular class interface.

Note that you aren't restricted to creating only one object. This technique easily supports the creation of a limited pool of objects. In that situation, however, you can be confronted with the problem of sharing objects in the pool. If this is an issue, you can create a solution involving a check-out and check-in of the shared objects.

Variations on singleton

Any static member object inside a class is an expression of singleton: one and only one will be made. So in a sense, the language has direct support for the idea; we certainly use it on a regular basis. However, there's a problem associated with static objects (member or not), and that's the order of initialization, as described earlier in this book. If one static object depends on another, it's important that the order of initialization proceed correctly.

Fortunately, there's another language feature that allows you to control many aspects of initialization order, and that's a static object defined inside a function. This delays the initialization of the object until the first time the function is called. If the function returns a reference to the static object, it gives you the effect of a singleton while removing much of the worry of static initialization. For example, suppose you want to create a logfile upon the first call to a function which returns a reference to that logfile. This header file will do the trick:

```
//: C25:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>
```

```
inline std::ofstream& logfile() {
  static std::ofstream log("Logfile.log");
  return log;
}
#endif // LOGFILE_H ///:~
```

Since it's **inline**, the compiler and linker are responsible for guaranteeing there's only one actual instance of the function definition. The **log** object will not be created until the first time **logfile( )** is called. So if you use the function in one file:

```
//: C25:UseLog1.cpp {O}
#include "LogFile.h"
using namespace std;

void f() {
  logfile() << __FILE__ << endl;
} ///:~
```

And again in another file (for simplicity I didn't create a header file for **UseLog1.cpp**):

```
//: C25:UseLog2.cpp
//{L} UseLog1
#include "LogFile.h"
using namespace std;
void f(); // In lieu of a header file

void g() {
  logfile() << __FILE__ << endl;
}

int main() {
  f();
  g();
} ///:~
```

Then the **log** object doesn't get created until the first call to **f( )**.

You can easily combine the creation of the static object inside a member function with the singleton class. **SingletonPattern.cpp** can be modified to use this approach:

```
//: C25:SingletonPattern2.cpp
#include <iostream>
using namespace std;
```

```
class Singleton {
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& getHandle() {
    static Singleton s(47);
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::getHandle();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::getHandle();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

An especially interesting case is if two of these singletons depend on each other, like this:

```
//: C25:FunctionStaticSingleton.cpp

class Singleton1 {
  Singleton1() {}
public:
  static Singleton1& ref() {
    static Singleton1 single;
    return single;
  }
};

class Singleton2 {
  Singleton1& s1;
  Singleton2(Singleton1& s) : s1(s) {}
public:
  static Singleton2& ref() {
    static Singleton2 single(Singleton1::ref());
    return single;
  }
```

```
  Singleton1& f() { return s1; }
};

int main() {
  Singleton1& s1 = Singleton2::ref().f();
} ///:~
```

When **Singleton2::ref( )** is called, it causes its sole **Singleton2** object to be created. In the process of this creation, **Singleton1::ref( )** is called, and that causes the sole **Singleton1** object to be created. Because this technique doesn't rely on the order of linking or loading, the programmer has much better control over initialization, leading to less problems.

You'll see further examples of the singleton pattern in the rest of this chapter.

Classifying patterns

The *Design Patterns* book discusses 23 different patterns, classified under three purposes (all of which revolve around the particular aspect that can vary). The three purposes are:

1. **Creational**: how an object can be created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't have to be changed when you add a new type of object. The aforementioned *Singleton* is classified as a creational pattern, and later in this chapter you'll see examples of *Factory Method* and *Prototype*.
2. **Structural**: designing objects to satisfy particular project constraints. These work with the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections.
3. **Behavioral**: objects that handle particular types of actions within a program. These encapsulate processes that you want to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as in an iterator), or implementing an algorithm. This chapter contains examples of the *Observer* and the *Visitor* patterns.

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ but sometimes in Smalltalk. This book will not repeat all the details of the patterns shown in *Design Patterns* since that book stands on its own and should be studied separately. The catalog and examples provided here are intended to

rapidly give you a grasp of the patterns, so you can get a decent feel for what patterns are about and why they are so important.

[[Describe different form of categorization, based on what you want to accomplish rather than the way the patterns look. More categories, but should result in easier-to-understand, faster selection]]]

Features, idioms, patterns

How things have gotten confused; conflicting pattern descriptions, naïve "patterns," patterns are not trivial nor are they represented by features that are built into the language, nor are they things that you do almost all the time. Constructors and destructors, for example, could be called the "guaranteed initialization and cleanup design pattern." This is an important and essential idea, but it's built into the language.

Another example comes from various forms of aggregation. Aggregation is a completely fundamental principle in object-oriented programming: you make objects out of other objects [[make reference to basic tenets of OO]]. Yet sometimes this idea is classified as a pattern, which tends to confuse the issue. This is unfortunate because it pollutes the idea of the design pattern and suggest that anything that surprises you the first time you see it should be a design pattern.

Another misguided example is found in the Java language; the designers of the "JavaBeans" specification decided to refer to a simple naming convention as a design pattern (you say **getInfo( )** for a member function that returns an **Info** property and **setInfo( )** for one that changes the internal **Info** property; the use of the "get" and "set" strings is what they decided constituted calling it a design pattern).

Basic complexity hiding

You'll often find that messy code can be cleaned up by putting it inside a class. This is more than fastidiousness – if nothing else, it aids readability and therefore maintainability, and it can often lead to reusability.

Simple Veneer (façade, Adapter (existing system), Bridge (designed in),

Hiding types (polymorphism, iterators, proxy)

Hiding connections (mediator,)

Factories: encapsulating object creation

When you discover that you need to add new types to a system, the most sensible first step to take is to use polymorphism to create a common interface to those new types, thus separating the rest of the code in your system from the knowledge of the specific types that you are adding. This way, new types may be added without disturbing exising code … or so it seems. At first it would appear that the only place you need to change the code in such a design is the place where you inherit a new type, but this is not quite true. You must still create an object of your new type, and at the point of creation you must specify the exact constructor to use. Thus, if the code that creates objects is distributed throughout your application, you have the same problem when adding new types – you must still chase down all the points of your code where it type matters. It happens to be the *creation* of the type that matters in this case rather than the *use* of the type (which is taken care of by polymorphism), but the effect is the same: adding a new type can cause problems.

The solution is to force the creation of objects to occur through a common *factory* rather than to allow the creational code to be spread throughout your system. If all the code in your program must go through this factory whenever it needs to create one of your objects, then all you must do when you add a new object is to modify the factory.

As an example, let's revisit the **Shape** system. One approach is to make the factory a **static** method of the base class:

```
//: C25:ShapeFactory1.cpp
#include "../purge.h"
#include <iostream>
#include <string>
#include <exception>
#include <vector>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
  class BadShapeCreation : public exception {
    string reason;
```

```
  public:
    BadShapeCreation(string type) {
      reason = "Cannot create type " + type;
    }
    const char *what() const {
      return reason.c_str();
    }
  };
  static Shape* factory(string type)
    throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw\n"; }
  void erase() { cout << "Circle::erase\n"; }
  ~Circle() { cout << "Circle::~Circle\n"; }
};

class Square : public Shape {
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw\n"; }
  void erase() { cout << "Square::erase\n"; }
  ~Square() { cout << "Square::~Square\n"; }
};

Shape* Shape::factory(string type)
  throw(Shape::BadShapeCreation) {
  if(type == "Circle") return new Circle();
  if(type == "Square") return new Square();
  throw BadShapeCreation(type);
}

char* shlist[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square", "" };

int main() {
  vector<Shape*> shapes;
  try {
    for(char** cp = shlist; **cp; cp++)
```

```
      shapes.push_back(Shape::factory(*cp));
    } catch(Shape::BadShapeCreation e) {
      cout << e.what() << endl;
      return 1;
    }
    for(int i = 0; i < shapes.size(); i++) {
      shapes[i]->draw();
      shapes[i]->erase();
    }
    purge(shapes);
  } ///:~
```

The **factory( )** takes an argument that allows it to determine what type of **Shape** to create; it happens to be a **string** in this case but it could be any set of data. The **factory( )** is now the only other code in the system that needs to be changed when a new type of **Shape** is added (the initialization data for the objects will presumably come from somewhere outside the system, and not be a hard-coded array as in the above example).

To ensure that the creation can only happen in the **factory( )**, the constructors for the specific types of **Shape** are made **private**, and **Shape** is declared a **friend** so that **factory( )** has access to the constructors (you could also declare only **Shape::factory( )** to be a **friend**, but it seems reasonably harmless to declare the entire base class as a **friend**).

Polymorphic factories

The **static factory( )** method in the previous example forces all the creation operations to be focused in one spot, to that's the only place you need to change the code. This is certainly a reasonable solution, as it throws a box around the process of creating objects. However, the *Design Patterns* book emphasizes that the reason for the *Factory Method* pattern is so that different types of factories can be subclassed from the basic factory (the above design is mentioned as a special case). However, the book does not provide an example, but instead just repeats the example used for the *Abstract Factory*. Here is **ShapeFactory1.cpp** modified so the factory methods are in a separate class as virtual functions:

```
//: C25:ShapeFactory2.cpp
// Polymorphic factory methods
#include "../purge.h"
#include <iostream>
#include <string>
```

```
#include <exception>
#include <vector>
#include <map>
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class ShapeFactory {
  virtual Shape* create() = 0;
  static map<string, ShapeFactory*> factories;
public:
  virtual ~ShapeFactory() {}
  friend class ShapeFactoryInizializer;
  class BadShapeCreation : public exception {
    string reason;
  public:
    BadShapeCreation(string type) {
      reason = "Cannot create type " + type;
    }
    const char *what() const {
      return reason.c_str();
    }
  };
  static Shape*
  createShape(string id) throw(BadShapeCreation){
    if(factories.find(id) != factories.end())
      return factories[id]->create();
    else
      throw BadShapeCreation(id);
  }
};

// Define the static object:
map<string, ShapeFactory*>
  ShapeFactory::factories;

class Circle : public Shape {
  Circle() {} // Private constructor
public:
```

```
  void draw() { cout << "Circle::draw\n"; }
  void erase() { cout << "Circle::erase\n"; }
  ~Circle() { cout << "Circle::~Circle\n"; }
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
  };
};

class Square : public Shape {
  Square() {}
public:
  void draw() { cout << "Square::draw\n"; }
  void erase() { cout << "Square::erase\n"; }
  ~Square() { cout << "Square::~Square\n"; }
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Square; }
  };
};

// Singleton to initialize the ShapeFactory:
class ShapeFactoryInizializer {
  static ShapeFactoryInizializer si;
  ShapeFactoryInizializer() {
    ShapeFactory::factories["Circle"] =
      new Circle::Factory;
    ShapeFactory::factories["Square"] =
      new Square::Factory;
  }
};

// Static member definition:
ShapeFactoryInizializer
  ShapeFactoryInizializer::si;

char* shlist[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square", "" };

int main() {
```

```
  vector<Shape*> shapes;
  try {
    for(char** cp = shlist; **cp; cp++)
      shapes.push_back(
        ShapeFactory::createShape(*cp));
  } catch(ShapeFactory::BadShapeCreation e) {
    cout << e.what() << endl;
    return 1;
  }
  for(int i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

Now the factory method appears in its own class, **ShapeFactory**, as the **virtual create( )**. This is a **private** method which means it cannot be called directly, but it can be overridden. The subclasses of **Shape** must each create their own subclasses of **ShapeFactory** and override the **shape( )** method to create an object of their own type. The actual creation of shapes is performed by calling **ShapeFactory::createShape( )**, which is a static method that uses the **map** in **ShapeFactory** to find the appropriate factory object based on an identifier that you pass it. The factory is immediately used to create the shape object, but you could imagine a more complex problem where the appropriate factory object is returned and then used by the caller to create an object in a more sophisticated way. However, it seems that much of the time you don't need the intricacies of the polymorphic factory method, and a single static method in the base class (as shown in **ShapeFactory1.cpp**) will work fine.

Notice that the **ShapeFactory** must be initialized by loading its **map** with factory objects, which takes place in the singleton **ShapeFactoryInizializer**. So to add a new type to this design you must inherit the type, create a factory, and modify **ShapeFactoryInizializer** so that an instance of your factory is inserted in the map. This extra complexity again suggests the use of a **static** factory method if you don't need to create individual factory objects.

Abstract factories

The *Abstract Factory* pattern looks like the factory objects we've seen previously, with not one but several factory methods. Each of the factory methods creates a different kind of object. The idea is that at the point of

creation of the factory object, you decide how all the objects created by that factory will be used. The example given in *Design Patterns* implements portability across various graphical user interfaces (GUIs): you create a factory object appropriate to the GUI that you're working with, and from then on when you ask it for a menu, button, slider, etc. it will automatically create the appropriate version of that item for the GUI. Thus you're able to isolate, in one place, the effect of changing from one GUI to another.

As another example suppose you are creating a general-purpose gaming environment and you want to be able to support different types of games. Here's how it might look using an abstract factory:

```
//: C25:AbstractFactory.cpp
// A gaming environment
#include <iostream>
using namespace std;

class Obstacle {
public:
  virtual void action() = 0;
};

class Player {
public:
  virtual void interactWith(Obstacle*) = 0;
};

class Kitty: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "Kitty has encountered a ";
    ob->action();
  }
};

class KungFuGuy: public Player {
  virtual void interactWith(Obstacle* ob) {
    cout << "KungFuGuy now battles against a ";
    ob->action();
  }
};

class Puzzle: public Obstacle {
public:
  void action() { cout << "Puzzle\n"; }
```

```
};

class NastyWeapon: public Obstacle {
public:
  void action() { cout << "NastyWeapon\n"; }
};

// The abstract factory:
class GameElementFactory {
public:
  virtual Player* makePlayer() = 0;
  virtual Obstacle* makeObstacle() = 0;
};

// Concrete factories:
class KittiesAndPuzzles :
  public GameElementFactory {
public:
  virtual Player* makePlayer() {
    return new Kitty;
  }
  virtual Obstacle* makeObstacle() {
    return new Puzzle;
  }
};

class KillAndDismember :
  public GameElementFactory {
public:
  virtual Player* makePlayer() {
    return new KungFuGuy;
  }
  virtual Obstacle* makeObstacle() {
    return new NastyWeapon;
  }
};

class GameEnvironment {
  GameElementFactory* gef;
  Player* p;
  Obstacle* ob;
public:
  GameEnvironment(GameElementFactory* factory) :
    gef(factory), p(factory->makePlayer()),
```

```
    ob(factory->makeObstacle()) {}
  void play() {
    p->interactWith(ob);
  }
  ~GameEnvironment() {
    delete p;
    delete ob;
    delete gef;
  }
};

int main() {
  GameEnvironment
    g1(new KittiesAndPuzzles),
    g2(new KillAndDismember);
  g1.play();
  g2.play();
} ///:~
```

In this environment, **Player** objects interact with **Obstacle** objects, but there are different types of players and obstacles depending on what kind of game you're playing. You determine the kind of game by choosing a particular **GameElementFactory**, and then the **GameEnvironment** controls the setup and play of the game. In this example, the setup and play is very simple, but those activities (the *initial conditions* and the *state change*) can determine much of the game's outcome. Here, **GameEnvironment** is not designed to be inherited, although it could very possibly make sense to do that.

This also contains examples of *Double Dispatching* and the *Factory Method*, both of which will be explained later.

Virtual constructors

One of the primary goals of using a factory is so that you can organize your code so you don't have to select an exact type of constructor when creating an object. That is, you can say, "I don't know precisely what type of object you are, but here's the information: Create yourself."

In addition, during a constructor call the virtual mechanism does not operate (early binding occurs). Sometimes this is awkward. For example, in the **Shape** program it seems logical that inside the constructor for a **Shape** object, you would want to set everything up and then **draw( )** the **Shape**. **draw( )** should be a virtual function, a message to the **Shape** that it should draw itself appropriately, depending on whether it is a circle,

square, line, and so on. However, this doesn't work inside the constructor, for the reasons given in Chapter XX: Virtual functions resolve to the "local" function bodies when called in constructors.

If you want to be able to call a virtual function inside the constructor and have it do the right thing, you must use a technique to *simulate* a virtual constructor (which is a variation of the *Factory Method*). This is a conundrum. Remember the idea of a virtual function is that you send a message to an object and let the object figure out the right thing to do. But a constructor builds an object. So a virtual constructor would be like saying, "I don't know exactly what type of object you are, but build yourself anyway." In an ordinary constructor, the compiler must know which VTABLE address to bind to the VPTR, and if it existed, a virtual constructor couldn't do this because it doesn't know all the type information at compile-time. It makes sense that a constructor can't be virtual because it is the one function that absolutely must know everything about the type of the object.

And yet there are times when you want something approximating the behavior of a virtual constructor.

In the **Shape** example, it would be nice to hand the **Shape** constructor some specific information in the argument list and let the constructor create a specific type of **Shape** (a **Circle**, **Square**) with no further intervention. Ordinarily, you'd have to make an explicit call to the **Circle**, **Square** constructor yourself.

Coplien[72] calls his solution to this problem "envelope and letter classes." The "envelope" class is the base class, a shell that contains a pointer to an object of the base class. The constructor for the "envelope" determines (at runtime, when the constructor is called, not at compile-time, when the type checking is normally done) what specific type to make, then creates an object of that specific type (on the heap) and assigns the object to its pointer. All the function calls are then handled by the base class through its pointer. So the base class is acting as a proxy for the derived class:

```
//: C25:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <exception>
#include <vector>
using namespace std;
```

[72] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

```
class Shape {
  Shape* s;
  // Prevent copy-construction & operator=
  Shape(Shape&);
  Shape operator=(Shape&);
protected:
  Shape() { s = 0; };
public:
  virtual void draw() { s->draw(); }
  virtual void erase() { s->erase(); }
  virtual void test() { s->test(); };
  virtual ~Shape() {
    cout << "~Shape\n";
    if(s) {
      cout << "Making virtual call: ";
      s->erase(); // Virtual call
    }
    cout << "delete s: ";
    delete s; // The polymorphic deletion
  }
  class BadShapeCreation : public exception {
    string reason;
  public:
    BadShapeCreation(string type) {
      reason = "Cannot create type " + type;
    }
    const char *what() const {
      return reason.c_str();
    }
  };
  Shape(string type) throw(BadShapeCreation);
};

class Circle : public Shape {
  Circle(Circle&);
  Circle operator=(Circle&);
  Circle() {} // Private constructor
  friend class Shape;
public:
  void draw() { cout << "Circle::draw\n"; }
  void erase() { cout << "Circle::erase\n"; }
  void test() { draw(); }
  ~Circle() { cout << "Circle::~Circle\n"; }
};
```

```
class Square : public Shape {
  Square(Square&);
  Square operator=(Square&);
  Square() {}
  friend class Shape;
public:
  void draw() { cout << "Square::draw\n"; }
  void erase() { cout << "Square::erase\n"; }
  void test() { draw(); }
  ~Square() { cout << "Square::~Square\n"; }
};

Shape::Shape(string type)
  throw(Shape::BadShapeCreation) {
  if(type == "Circle")
    s = new Circle();
  else if(type == "Square")
    s = new Square();
  else throw BadShapeCreation(type);
  draw();  // Virtual call in the constructor
}

char* shlist[] = { "Circle", "Square", "Square",
  "Circle", "Circle", "Circle", "Square", "" };

int main() {
  vector<Shape*> shapes;
  cout << "virtual constructor calls:" << endl;
  try {
    for(char** cp = shlist; **cp; cp++)
      shapes.push_back(new Shape(*cp));
  } catch(Shape::BadShapeCreation e) {
    cout << e.what() << endl;
    return 1;
  }
  for(int i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    cout << "test\n";
    shapes[i]->test();
    cout << "end test\n";
    shapes[i]->erase();
  }
  Shape c("Circle"); // Create on the stack
```

```
  cout << "destructor calls:" << endl;
  for(int j = 0; j < shapes.size(); j++) {
    delete shapes[j];
    cout << "\n------------\n";
  }
} ///:~
```

The base class **Shape** contains a pointer to an object of type **Shape** as its only data member. When you build a "virtual constructor" scheme, you must exercise special care to ensure this pointer is always initialized to a live object.

Each time you derive a new subtype from **Shape**, you must go back and add the creation for that type in one place, inside the "virtual constructor" in the **Shape** base class. This is not too onerous a task, but the disadvantage is you now have a dependency between the **Shape** class and all classes derived from it (a reasonable trade-off, it seems). Also, because it is a proxy, the base-class interface is truly the only thing the user sees.

In this example, the information you must hand the virtual constructor about what type to create is very explicit: It's a **string** that names the type. However, your scheme may use other information – for example, in a parser the output of the scanner may be handed to the virtual constructor, which then uses that information to determine which token to create.

The virtual constructor **Shape(type)** can only be declared inside the class; it cannot be defined until after all the derived classes have been declared. However, the default constructor can be defined inside **class Shape**, but it should be made **protected** so temporary **Shape** objects cannot be created. This default constructor is only called by the constructors of derived-class objects. You are forced to explicitly create a default constructor because the compiler will create one for you automatically only if there are *no* constructors defined. Because you must define **Shape(type)**, you must also define **Shape( )**.

The default constructor in this scheme has at least one very important chore – it must set the value of the **s** pointer to zero. This sounds strange at first, but remember that the default constructor will be called as part of the construction of the *actual object* – in Coplien's terms, the "letter," not the "envelope." However, the "letter" is derived from the "envelope," so it also inherits the data member **s**. In the "envelope," **s** is important because it points to the actual object, but in the "letter," **s** is simply excess baggage. Even excess baggage should be initialized, however, and

if **s** is not set to zero by the default constructor called for the "letter," bad things happen (as you'll see later).

The virtual constructor takes as its argument information that completely determines the type of the object. Notice, though, that this type information isn't read and acted upon until runtime, whereas normally the compiler must know the exact type at compile-time (one other reason this system effectively imitates virtual constructors).

Inside the virtual constructor there's a **switch** statement that uses the argument to construct the actual ("letter") object, which is then assigned to the pointer inside the "envelope." At that point, the construction of the "letter" has been completed, so any virtual calls will be properly directed.

As an example, consider the call to **draw( )** inside the virtual constructor. If you trace this call (either by hand or with a debugger), you can see that it starts in the **draw( )** function in the base class, **Shape**. This function calls **draw( )** for the "envelope" **s** pointer to its "letter." All types derived from **Shape** share the same interface, so this virtual call is properly executed, even though it seems to be in the constructor. (Actually, the constructor for the "letter" has already completed.) As long as all virtual calls in the base class simply make calls to identical virtual function through the pointer to the "letter," the system operates properly.

To understand how it works, consider the code in **main( )**. To fill the **vector shapes**, "virtual constructor" calls are made to **Shape**. Ordinarily in a situation like this, you would call the constructor for the actual type, and the VPTR for that type would be installed in the object. Here, however, the VPTR used in each case is the one for **Shape**, not the one for the specific **Circle**, **Square**, or **Triangle**.

In the **for** loop where the **draw( )** and **erase( )** functions are called for each **Shape**, the virtual function call resolves, through the VPTR, to the corresponding type. However, this is **Shape** in each case. In fact, you might wonder why **draw( )** and **erase( )** were made **virtual** at all. The reason shows up in the next step: The base-class version of **draw( )** makes a call, through the "letter" pointer **s**, to the **virtual** function **draw( )** for the "letter." This time the call resolves to the actual type of the object, not just the base class **Shape**. Thus the runtime cost of using virtual constructors is one more virtual call every time you make a virtual function call.

In order to create any function that is overridden, such as **draw( )**, **erase( )** or **test( )**, you must proxy all calls to the **s** pointer in the base class implementation, as shown above. This is because, when the call is made, the call to the envelope's member function will resolve as being to **Shape**, and not to a derived type of **Shape**. Only when you make the

proxy call to **s** will the virtual behavior take place. In **main( )**, you can see that everything works correctly, even when calls are made inside constructors and destructors.

Destructor operation

The activities of destruction in this scheme are also tricky. To understand, let's verbally walk through what happens when you call **delete** for a pointer to a **Shape** object – specifically, a **Square** – created on the heap. (This is more complicated than an object created on the stack.) This will be a **delete** through the polymorphic interface, as in the statement **delete shapes[i]** in **main( )**.

The type of the pointer **shapes[i]** is of the base class **Shape**, so the compiler makes the call through **Shape**. Normally, you might say that it's a virtual call, so **Square**'s destructor will be called. But with the virtual constructor scheme, the compiler is creating actual **Shape** objects, even though the constructor initializes the letter pointer to a specific type of **Shape**. The virtual mechanism *is* used, but the VPTR inside the **Shape** object is **Shape**'s VPTR, not **Square**'s. This resolves to **Shape**'s destructor, which calls **delete** for the letter pointer **s**, which actually points to a **Square** object. This is again a virtual call, but this time it resolves to **Square**'s destructor.

With a destructor, however, C++ guarantees, via the compiler, that all destructors in the hierarchy are called. **Square**'s destructor is called first, followed by any intermediate destructors, in order, until finally the base-class destructor is called. This base-class destructor has code that says **delete s**. When this destructor was called originally, it was for the "envelope" **s**, but now it's for the "letter" **s**, which is there because the "letter" was inherited from the "envelope," and not because it contains anything. So *this* call to **delete** should do nothing.

The solution to the problem is to make the "letter" **s** pointer zero. Then when the "letter" base-class destructor is called, you get **delete 0**, which by definition does nothing. Because the default constructor is protected, it will be called *only* during the construction of a "letter," so that's the only situation where **s** is set to zero.

Your most common tool for hiding construction will probably be ordinary factory methods rather than the more complex approaches. The idea of adding new types with minimal effect on the rest of the system will be further explored later in this chapter.

Callbacks

Decoupling code behavior

Functor/Command

Strategy

Observer

Like the other forms of callback, this contains a hook point where you can change code. The difference is in the observer's completely dynamic nature. It is often used for the specific case of changes based on other object's change of state, but is also the basis of event management. Anytime you want to decouple the source of the call from the called code in a completely dynamic way.

The observer pattern solves a fairly common problem: What if a group of objects needs to update themselves when some other object changes state? This can be seen in the "model-view" aspect of Smalltalk's MVC (model-view-controller), or the almost-equivalent "Document-View Architecture." Suppose that you have some data (the "document") and more than one view, say a plot and a textual view. When you change the data, the two views must know to update themselves, and that's what the observer facilitates.

There are two types of objects used to implement the observer pattern in the following code. The **Observable** class keeps track of everybody who wants to be informed when a change happens, whether the "state" has changed or not. When someone says "OK, everybody should check and potentially update themselves," the **Observable** class performs this task by calling the **notifyObservers( )** member function for each observer on the list. The **notifyObservers( )** member function is part of the base class **Observable**.

There are actually two "things that change" in the observer pattern: the quantity of observing objects and the way an update occurs. That is, the observer pattern allows you to modify both of these without affecting the surrounding code.

There are a number of ways to implement the observer pattern, but the code shown here will create a framework from which you can build your

own observer code, following the example. First, this interface describes what an observer looks like:

```
//: C25:Observer.h
// The Observer interface
#ifndef OBSERVER_H
#define OBSERVER_H

class Observable;
class Argument {};

class Observer {
public:
  // Called by the observed object, whenever
  // the observed object is changed:
  virtual void
  update(Observable* o, Argument * arg) = 0;
};
#endif // OBSERVER_H ///:~
```

Since **Observer** interacts with **Observable** in this approach, **Observable** must be declared first. In addition, the **Argument** class is empty and only acts as a base class for any type of argument you wish to pass during an update. If you want, you can simply pass the extra argument as a **void***; you'll have to downcast in either case but some folks find **void*** objectionable.

Observer is an "interface" class that only has one member function, **update( )**. This function is called by the object that's being observed, when that object decides its time to update all it's observers. The arguments are optional; you could have an **update( )** with no arguments and that would still fit the observer pattern; however this is more general – it allows the observed object to pass the object that caused the update (since an **Observer** may be registered with more than one observed object) and any extra information if that's helpful, rather than forcing the **Observer** object to hunt around to see who is updating and to fetch any other information it needs.

The "observed object" that decides when and how to do the updating will be called the **Observable**:

```
//: C25:Observable.h
// The Observable class
#ifndef OBSERVABLE_H
#define OBSERVABLE_H
#include "Observer.h"
```

```
#include <set>

class Observable {
  bool changed;
  std::set<Observer*> observers;
protected:
  virtual void setChanged() { changed = true; }
  virtual void clearChanged(){ changed = false; }
public:
  virtual void addObserver(Observer& o) {
    observers.insert(&o);
  }
  virtual void deleteObserver(Observer& o) {
    observers.erase(&o);
  }
  virtual void deleteObservers() {
    observers.clear();
  }
  virtual int countObservers() {
    return observers.size();
  }
  virtual bool hasChanged() { return changed; }
  // If this object has changed, notify all
  // of its observers:
  virtual void notifyObservers(Argument* arg=0) {
    if(!hasChanged()) return;
    clearChanged(); // Not "changed" anymore
    std::set<Observer*>::iterator it;
    for(it = observers.begin();
      it != observers.end(); it++)
      (*it)->update(this, arg);
  }
};
#endif // OBSERVABLE_H ///:~
```

Again, the design here is more elaborate than is necessary; as long as there's a way to register an **Observer** with an **Observable** and for the **Observable** to update its **Observer**s, the set of member functions doesn't matter. However, this design is intended to be reusable (it was lifted from the design used in the Java standard library). As mentioned elsewhere in the book, there is no support for multithreading in the Standard C++ libraries, so this design would need to be modified in a multithreaded environment.

Observable has a flag to indicate whether it's been changed. In a simpler design, there would be no flag; if something happened, everyone would be notified. The flag allows you to wait, and only notify the **Observer**s when you decide the time is right. Notice, however, that the control of the flag's state is **protected**, so that only an inheritor can decide what constitutes a change, and not the end user of the resulting derived **Observer** class.

The collection of **Observer** objects is kept in a **set<Observer*>** to prevent duplicates; the **set insert( )**, **erase( )**, **clear( )** and **size( )** functions are exposed to allow **Observer**s to be added and removed at any time, thus providing runtime flexibility.

Most of the work is done in **notifyObservers( )**. If the **changed** flag has not been set, this does nothing. Otherwise, it first clears the **changed** flag so repeated calls to **notifyObservers( )** won't waste time. This is done before notifying the observers in case the calls to **update( )** do anything that causes a change back to this **Observable** object. Then it moves through the **set** and calls back to the **update( )** member function of each **Observer**.

At first it may appear that you can use an ordinary **Observable** object to manage the updates. But this doesn't work; to get an effect, you *must* inherit from **Observable** and somewhere in your derived-class code call **setChanged( )**. This is the member function that sets the "changed" flag, which means that when you call **notifyObservers( )** all of the observers will, in fact, get notified. *Where* you call **setChanged( )** depends on the logic of your program.

Now we encounter a dilemma. An object that should notify its observers about things that happen to it – events or changes in state – might have more than one such item of interest. For example, if you're dealing with a graphical user interface (GUI) item – a button, say – the items of interest might be the mouse clicked the button, the mouse moved over the button, and (for some reason) the button changed its color. So we'd like to be able to report all of these events to different observers, each of which is interested in a different type of event.

The problem is that we would normally reach for multiple inheritance in such a situation: "I'll inherit from **Observable** to deal with mouse clicks, and I'll … er … inherit from **Observable** to deal with mouse-overs, and, well, … hmm, that doesn't work."

The “interface” idiom

The “inner class” idiom

Here’s a situation where we do actually need to (in effect) upcast to more than one type, but in this case we need to provide several *different* implementations of the same base type. The solution is something I’ve lifted from Java, which takes C++’s nested class one step further. Java has a built-in feature called *inner classes*, which look like C++’s nested classes, but they do two other things:

1. A Java inner class automatically has access to the private elements of the class it is nested within.
2. An object of a Java inner class automatically grabs the “this” to the outer class object it was created within. In Java, the “outer this” is implicitly dereferenced whenever you name an element of the outer class.

[[Insert the definition of a closure]]. So to implement the inner class idiom in C++, we must do these things by hand. Here’s an example:

```
//: C25:InnerClassIdiom.cpp
// Example of the "inner class" idiom
#include <iostream>
#include <string>
using namespace std;

class Poingable {
public:
  virtual void poing() = 0;
};

void callPoing(Poingable& p) {
  p.poing();
}

class Bingable {
public:
  virtual void bing() = 0;
};

void callBing(Bingable& b) {
  b.bing();
}
```

```
class Outer {
  string name;
  // Define one inner class:
  class Inner1;
  friend class Outer::Inner1;
  class Inner1 : public Poingable {
    Outer* parent;
  public:
    Inner1(Outer* p) : parent(p) {}
    void poing() {
      cout << "poing called for "
        << parent->name << endl;
      // Accesses data in the outer class object
    }
  } inner1;
  // Define a second inner class:
  class Inner2;
  friend class Outer::Inner2;
  class Inner2 : public Bingable {
    Outer* parent;
  public:
    Inner2(Outer* p) : parent(p) {}
    void bing() {
      cout << "bing called for "
        << parent->name << endl;
    }
  } inner2;
public:
  Outer(const string& nm) : name(nm),
    inner1(this), inner2(this) {}
  // Return reference to interfaces
  //  implemented by the inner classes:
  operator Poingable&() { return inner1; }
  operator Bingable&() { return inner2; }
};

int main() {
  Outer x("Ping Pong");
  // Like upcasting to multiple base types!:
  callPoing(x);
  callBing(x);
} ///:~
```

The example begins with the **Poingable** and **Bingable** interfaces, each of which contain a single member function. The services provided by **callPoing( )** and **callBing( )** require that the object they receive implement the **Poingable** and **Bingable** interfaces, respectively, but they put no other requirements on that object so as to maximize the flexibility of using **callPoing( )** and **callBing( )**. Note the lack of **virtual** destructors in either interface – the intent is that you never perform object destruction via the interface.

Outer contains some private data (**name**) and it wishes to provide both a **Poingable** interface and a **Bingable** interface so it can be used with **callPoing( )** and **callBing( )**. Of course, in this situation we *could* simply use multiple inheritance. This example is just intended to show the simplest syntax for the idiom; we'll see a real use shortly. To provide a **Poingable** object without inheriting **Outer** from **Poingable**, the inner class idiom is used. First, the declaration **class Inner** says that, somewhere, there is a nested class of this name. This allows the **friend** declaration for the class, which follows. Finally, now that the nested class has been granted access to all the private elements of **Outer**, the class can be defined. Notice that it keeps a pointer to the **Outer** which created it, and this pointer must be initialized in the constructor. Finally, the **poing( )** function from **Poingable** is implemented. The same process occurs for the second inner class which implements **Bingable**. Each inner class has a single **private** instance created, which is initialized in the **Outer** constructor. By creating the member objects and returning references to them, issues of object lifetime are eliminated.

Notice that both inner class definitions are **private**, and in fact the client programmer doesn't have any access to details of the implementation, since the two access methods **operator Poingable&( )** and **operator Bingable&( )** only return a reference to the upcast interface, not to the object that implements it. In fact, since the two inner classes are **private**, the client programmer cannot even downcast to the implementation classes, thus providing complete isolation between interface and implementation.

Just to push a point, I've taken the extra liberty here of defining the automatic type conversion operators **operator Poingable&( )** and **operator Bingable&( )**. In **main( )**, you can see that these actually allow a syntax that looks like **Outer** is multiply inherited from **Poingable** and **Bingable**. The difference is that the casts in this case are one way. You can get the effect of an upcast to **Poingable** or **Bingable**, but you cannot downcast back to an **Outer**. In the following example of observer, you'll see the more typical approach: you provide access to the inner class

objects using ordinary member functions, not automatic type conversion operations.

The observer example

Armed with the **Observer** and **Observable** header files and the inner class idiom, we can look at an example of the observer pattern:

```
//: C25:ObservedFlower.cpp
// Demonstration of "observer" pattern
#include "Observable.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

class Flower {
  bool isOpen;
public:
  Flower() : isOpen(false),
    openNotifier(this), closeNotifier(this) {}
  void open() { // Opens its petals
    isOpen = true;
    openNotifier.notifyObservers();
    closeNotifier.open();
  }
  void close() { // Closes its petals
    isOpen = false;
    closeNotifier.notifyObservers();
    openNotifier.close();
  }
  // Using the "inner class" idiom:
  class OpenNotifier;
  friend class Flower::OpenNotifier;
  class OpenNotifier : public Observable {
    Flower* parent;
    bool alreadyOpen;
  public:
    OpenNotifier(Flower* f) : parent(f),
      alreadyOpen(false) {}
    void notifyObservers(Argument* arg=0) {
      if(parent->isOpen && !alreadyOpen) {
        setChanged();
        Observable::notifyObservers();
```

```
      alreadyOpen = true;
    }
  }
  void close() { alreadyOpen = false; }
 } openNotifier;
 class CloseNotifier;
 friend class Flower::CloseNotifier;
 class CloseNotifier : public Observable {
  Flower* parent;
  bool alreadyClosed;
 public:
  CloseNotifier(Flower* f) : parent(f),
    alreadyClosed(false) {}
  void notifyObservers(Argument* arg=0) {
    if(!parent->isOpen && !alreadyClosed) {
      setChanged();
      Observable::notifyObservers();
      alreadyClosed = true;
    }
  }
  void open() { alreadyClosed = false; }
 } closeNotifier;
};

class Bee {
 string name;
 // An "inner class" for observing openings:
 class OpenObserver;
 friend class Bee::OpenObserver;
 class OpenObserver : public Observer {
  Bee* parent;
 public:
  OpenObserver(Bee* b) : parent(b) {}
  void update(Observable*, Argument *) {
    cout << "Bee " << parent->name
      << "'s breakfast time!\n";
  }
 } openObsrv;
 // Another "inner class" for closings:
 class CloseObserver;
 friend class Bee::CloseObserver;
 class CloseObserver : public Observer {
  Bee* parent;
 public:
```

```
    CloseObserver(Bee* b) : parent(b) {}
    void update(Observable*, Argument *) {
      cout << "Bee " << parent->name
        << "'s bed time!\n";
    }
  } closeObsrv;
public:
  Bee(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};

class Hummingbird {
  string name;
  class OpenObserver;
  friend class Hummingbird::OpenObserver;
  class OpenObserver : public Observer {
    Hummingbird* parent;
  public:
    OpenObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s breakfast time!\n";
    }
  } openObsrv;
  class CloseObserver;
  friend class Hummingbird::CloseObserver;
  class CloseObserver : public Observer {
    Hummingbird* parent;
  public:
    CloseObserver(Hummingbird* h) : parent(h) {}
    void update(Observable*, Argument *) {
      cout << "Hummingbird " << parent->name
        << "'s bed time!\n";
    }
  } closeObsrv;
public:
  Hummingbird(string nm) : name(nm),
    openObsrv(this), closeObsrv(this) {}
  Observer& openObserver() { return openObsrv; }
  Observer& closeObserver() { return closeObsrv;}
};
```

```
int main() {
  Flower f;
  Bee ba("A"), bb("B");
  Hummingbird ha("A"), hb("B");
  f.openNotifier.addObserver(ha.openObserver());
  f.openNotifier.addObserver(hb.openObserver());
  f.openNotifier.addObserver(ba.openObserver());
  f.openNotifier.addObserver(bb.openObserver());
  f.closeNotifier.addObserver(ha.closeObserver());
  f.closeNotifier.addObserver(hb.closeObserver());
  f.closeNotifier.addObserver(ba.closeObserver());
  f.closeNotifier.addObserver(bb.closeObserver());
  // Hummingbird B decides to sleep in:
  f.openNotifier.deleteObserver(hb.openObserver());
  // Something changes that interests observers:
  f.open();
  f.open(); // It's already open, no change.
  // Bee A doesn't want to go to bed:
  f.closeNotifier.deleteObserver(
    ba.closeObserver());
  f.close();
  f.close(); // It's already closed; no change
  f.openNotifier.deleteObservers();
  f.open();
  f.close();
} ///:~
```

The events of interest are that a **Flower** can open or close. Because of the use of the inner class idiom, both these events can be separately-observable phenomena. **OpenNotifier** and **CloseNotifier** both inherit **Observable**, so they have access to **setChanged( )** and can be handed to anything that needs an **Observable**. You'll notice that, contrary to **InnerClassIdiom.cpp**, the **Observable** descendants are **public**. This is because some of their member functions must be available to the client programmer. There's nothing that says that an inner class must be **private**; in **InnerClassIdiom.cpp** I was simply following the design guideline "make things as private as possible." You could make the classes **private** and expose the appropriate methods by proxy in **Flower**, but it wouldn't gain much.

The inner class idiom also comes in handy to define more than one kind of **Observer**, in **Bee** and **Hummingbird**, since both those classes may want to independently observe **Flower** openings and closings. Notice how the inner class idiom provides something that has most of the benefits of

inheritance (the ability to access the private data in the outer class, for example) without the same restrictions.

In **main( )**, you can see one of the prime benefits of the observer pattern: the ability to change behavior at runtime by dynamically registering and un-registering **Observer**s with **Observable**s.

If you study the code above you'll see that **OpenNotifier** and **CloseNotifier** use the basic **Observable** interface. This means that you could inherit other completely different **Observer** classes; the only connection the **Observer**s have with **Flower**s is the **Observer** interface.

Multiple dispatching

When dealing with multiple types which are interacting, a program can get particularly messy. For example, consider a system that parses and executes mathematical expressions. You want to be able to say **Number + Number**, **Number * Number**, etc., where **Number** is the base class for a family of numerical objects. But when you say **a + b**, and you don't know the exact type of either **a** or **b**, so how can you get them to interact properly?

The answer starts with something you probably don't think about: C++ performs only single dispatching. That is, if you are performing an operation on more than one object whose type is unknown, C++ can invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *multiple dispatching*. Remember that polymorphism can occur only via member function calls, so if you want double dispatching to occur, there must be two member function calls: the first to determine the first unknown type, and the second to determine the second unknown type. With multiple dispatching, you must have a virtual call to determine each of the types. Generally, you'll set up a configuration such that a single member function call produces more than one dynamic member function call and thus determines more than one type in the process. To get this effect, you need to work with more than one virtual function: you'll need a virtual function call for each dispatch. The virtual functions in the following example are called **compete( )** and **eval( )**, and are both members of the same type. (In this case there will be only two dispatches, which is referred to as *double dispatching*). If you are working with two different type hierarchies that are interacting, then you'll have to have a virtual call in each hierarchy.

Here's an example of multiple dispatching:

```
//: C25:PaperScissorsRock.cpp
// Demonstration of multiple dispatching
#include "../purge.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

class Paper;
class Scissors;
class Rock;

enum Outcome { win, lose, draw };

ostream&
operator<<(ostream& os, const Outcome out) {
  switch(out) {
    default:
    case win: return os << "win";
    case lose: return os << "lose";
    case draw: return os << "draw";
  }
}

class Item {
public:
  virtual Outcome compete(const Item*) = 0;
  virtual Outcome eval(const Paper*) const = 0;
  virtual Outcome eval(const Scissors*) const= 0;
  virtual Outcome eval(const Rock*) const = 0;
  virtual ostream& print(ostream& os) const = 0;
  virtual ~Item() {}
  friend ostream&
  operator<<(ostream& os, const Item* it) {
    return it->print(os);
  }
};

class Paper : public Item {
public:
```

```
  Outcome compete(const Item* it) {
    return it->eval(this);
  }
  Outcome eval(const Paper*) const {
    return draw;
  }
  Outcome eval(const Scissors*) const {
    return win;
  }
  Outcome eval(const Rock*) const {
    return lose;
  }
  ostream& print(ostream& os) const {
    return os << "Paper    ";
  }
};

class Scissors : public Item {
public:
  Outcome compete(const Item* it) {
    return it->eval(this);
  }
  Outcome eval(const Paper*) const {
    return lose;
  }
  Outcome eval(const Scissors*) const {
    return draw;
  }
  Outcome eval(const Rock*) const {
    return win;
  }
  ostream& print(ostream& os) const {
    return os << "Scissors";
  }
};

class Rock : public Item {
public:
  Outcome compete(const Item* it) {
    return it->eval(this);
  }
  Outcome eval(const Paper*) const {
    return win;
  }
```

```
  Outcome eval(const Scissors*) const {
    return lose;
  }
  Outcome eval(const Rock*) const {
    return draw;
  }
  ostream& print(ostream& os) const {
    return os << "Rock    ";
  }
};

struct ItemGen {
  ItemGen() { srand(time(0)); }
  Item* operator()() {
    switch(rand() % 3) {
      default:
      case 0:
        return new Scissors();
      case 1:
        return new Paper();
      case 2:
        return new Rock();
    }
  }
};

struct Compete {
  Outcome operator()(Item* a, Item* b) {
    cout << a << "\t" << b << "\t";
    return a->compete(b);
  }
};

int main() {
  const int sz = 20;
  vector<Item*> v(sz*2);
  generate(v.begin(), v.end(), ItemGen());
  transform(v.begin(), v.begin() + sz,
    v.begin() + sz,
    ostream_iterator<Outcome>(cout, "\n"),
    Compete());
  purge(v);
} ///:~
```

Visitor, a type of multiple dispatching

The assumption is that you have a primary class hierarchy that is fixed; perhaps it's from another vendor and you can't make changes to that hierarchy. However, you'd like to add new polymorphic methods to that hierarchy, which means that normally you'd have to add something to the base class interface. So the dilemma is that you need to add methods to the base class, but you can't touch the base class. How do you get around this?

The design pattern that solves this kind of problem is called a "visitor" (the final one in the *Design Patterns* book), and it builds on the double dispatching scheme shown in the last section.

The visitor pattern allows you to extend the interface of the primary type by creating a separate class hierarchy of type **Visitor** to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound member function.

```
//: C25:BeeAndFlowers.cpp
// Demonstration of "visitor" pattern
#include "../purge.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;

class Gladiolus;
class Renuculus;
class Chrysanthemum;

class Visitor {
public:
  virtual void visit(Gladiolus* f) = 0;
  virtual void visit(Renuculus* f) = 0;
  virtual void visit(Chrysanthemum* f) = 0;
  virtual ~Visitor() {}
```

```
};

class Flower {
public:
  virtual void accept(Visitor&) = 0;
  virtual ~Flower() {}
};

class Gladiolus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Renuculus : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

class Chrysanthemum : public Flower {
public:
  virtual void accept(Visitor& v) {
    v.visit(this);
  }
};

// Add the ability to produce a string:
class StringVal : public Visitor {
  string s;
public:
  operator const string&() { return s; }
  virtual void visit(Gladiolus*) {
    s = "Gladiolus";
  }
  virtual void visit(Renuculus*) {
    s = "Renuculus";
  }
  virtual void visit(Chrysanthemum*) {
    s = "Chrysanthemum";
  }
};
```

```
// Add the ability to do "Bee" activities:
class Bee : public Visitor {
public:
  virtual void visit(Gladiolus*) {
    cout << "Bee and Gladiolus\n";
  }
  virtual void visit(Renuculus*) {
    cout << "Bee and Renuculus\n";
  }
  virtual void visit(Chrysanthemum*) {
    cout << "Bee and Chrysanthemum\n";
  }
};

struct FlowerGen {
  FlowerGen() { srand(time(0)); }
  Flower* operator()() {
    switch(rand() % 3) {
      default:
      case 0: return new Gladiolus();
      case 1: return new Renuculus();
      case 2: return new Chrysanthemum();
    }
  }
};

int main() {
  vector<Flower*> v(10);
  generate(v.begin(), v.end(), FlowerGen());
  vector<Flower*>::iterator it;
  // It's almost as if I added a virtual function
  // to produce a Flower string representation:
  StringVal sval;
  for(it = v.begin(); it != v.end(); it++) {
    (*it)->accept(sval);
    cout << string(sval) << endl;
  }
  // Perform "Bee" operation on all Flowers:
  Bee bee;
  for(it = v.begin(); it != v.end(); it++)
    (*it)->accept(bee);
  purge(v);
} ///:~
```

Efficiency

Flyweight

The composite

Evolving a design: the trash recycler

The nature of this problem (modeling a trash recycling system) is that the trash is thrown unclassified into a single bin, so the specific type information is lost. But later, the specific type information must be recovered to properly sort the trash. In the initial solution, RTTI (described in Chapter XX) is used.

This is not a trivial design because it has an added constraint. That's what makes it interesting – it's more like the messy problems you're likely to encounter in your work. The extra constraint is that the trash arrives at the trash recycling plant all mixed together. The program must model the sorting of that trash. This is where RTTI comes in: you have a bunch of anonymous pieces of trash, and the program figures out exactly what type they are.

One of the objectives of this program is to sum up the weight and value of the different types of trash. The trash will be kept in (potentially different types of) containers, so it makes sense to templatize the "summation" function on the container holding it (assuming that container exhibits basic STL-like behavior), so the function will be maximally flexible:

```
//: C25:sumValue.h
// Sums the value of Trash in any type of STL
// container of any specific type of Trash:
#ifndef SUMVALUE_H
#define SUMVALUE_H
#include <typeinfo>
#include <vector>
```

```
template<typename Cont>
void sumValue(Cont& bin) {
  double val = 0.0f;
  typename Cont::iterator tally = bin.begin();
  while(tally != bin.end()) {
    val +=(*tally)->weight() * (*tally)->value();
    out << "weight of "
        << typeid(*(*tally)).name()
        << " = " << (*tally)->weight()
        << endl;
    tally++;
  }
  out << "Total value = " << val << endl;
}
#endif // SUMVALUE_H ///:~
```

When you look at a piece of code like this, it can be initially disturbing because you might wonder "how can the compiler know that the member functions I'm calling here are valid?" But of course, all the template says is "generate this code on demand," and so only when you call the function will type checking come into play. This enforces that ***tally** produces an object that has member functions **weight( )** and **value( )**, and that **out** is a global **ostream**.

The **sumValue( )** function is templatized on the type of container that's holding the **Trash** pointers. Notice there's nothing in the template signature that says "this container must behave like an STL container and must hold **Trash***"; that is all implied in the code that's generated which uses the container.

The first version of the example takes the straightforward approach: creating a **vector<Trash*>**, filling it with **Trash** objects, then using RTTI to sort them out:

```
//: C25:Recycle1.cpp
// Recycling with RTTI
#include "sumValue.h"
#include "../purge.h"
#include <fstream>
#include <vector>
#include <typeinfo>
#include <cstdlib>
#include <ctime>
using namespace std;
ofstream out("Recycle1.out");
```

```
class Trash {
  double _weight;
  static int _count; // # created
  static int _dcount; // # destroyed
  // disallow automatic creation of
  // assignment & copy-constructor:
  void operator=(const Trash&);
  Trash(const Trash&);
public:
  Trash(double wt) : _weight(wt) {
    _count++;
  }
  virtual double value() const = 0;
  double weight() const { return _weight; }
  static int count() { return _count; }
  static int dcount() { return _dcount;}
  virtual ~Trash() { _dcount++; }
};

int Trash::_count = 0;
int Trash::_dcount = 0;

class Aluminum : public Trash {
  static double val;
public:
  Aluminum(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
  }
  ~Aluminum() { out << "~Aluminum\n"; }
};

double Aluminum::val = 1.67F;

class Paper : public Trash {
  static double val;
public:
  Paper(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
  }
```

```
  ~Paper() { out << "~Paper\n"; }
};

double Paper::val = 0.10F;

class Glass : public Trash {
  static double val;
public:
  Glass(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
  }
  ~Glass() { out << "~Glass\n"; }
};

double Glass::val = 0.23F;

class TrashGen {
public:
  TrashGen() { srand(time(0)); }
  static double frand(int mod) {
    return static_cast<double>(rand() % mod);
  }
  Trash* operator()() {
    for(int i = 0; i < 30; i++)
      switch(rand() % 3) {
        case 0 :
          return new Aluminum(frand(100));
        case 1 :
          return new Paper(frand(100));
        case 2 :
          return new Glass(frand(100));
      }
    return new Aluminum(0);
    // Or throw exeception...
  }
};

int main() {
  vector<Trash*> bin;
  // Fill up the Trash bin:
  generate_n(back_inserter(bin), 30, TrashGen());
  vector<Aluminum*> alBin;
```

```
  vector<Paper*> paperBin;
  vector<Glass*> glassBin;
  vector<Trash*>::iterator sorter = bin.begin();
  // Sort the Trash:
  while(sorter != bin.end()) {
    Aluminum* ap =
      dynamic_cast<Aluminum*>(*sorter);
    Paper* pp = dynamic_cast<Paper*>(*sorter);
    Glass* gp = dynamic_cast<Glass*>(*sorter);
    if(ap) alBin.push_back(ap);
    if(pp) paperBin.push_back(pp);
    if(gp) glassBin.push_back(gp);
    sorter++;
  }
  sumValue(alBin);
  sumValue(paperBin);
  sumValue(glassBin);
  sumValue(bin);
  out << "total created = "
      << Trash::count() << endl;
  purge(bin);
  out << "total destroyed = "
      << Trash::dcount() << endl;
} ///:~
```

This uses the classic structure of virtual functions in the base class that are redefined in the derived class. In addition, there are two **static** data members in the base class: **_count** to indicate the number of **Trash** objects that are created, and **_dcount** to keep track of the number that are destroyed. This verifies that proper memory management occurs. To support this, the **operator=** and copy-constructor are disallowed by declaring them **private** (no definitions are necessary; this simply prevents the compiler from synthesizing them). Those operations would cause problems with the count, and if they were allowed you'd have to define them properly.

The **Trash** objects are created, for the sake of this example, by the generator **TrashGen**, which uses the random number generator to choose the type of **Trash**, and also to provide it with a "weight" argument. The return value of the generator's **operator( )** is upcast to **Trash***, so all the specific type information is lost. In **main( )**, a **vector<Trash*>** called **bin** is created and then filled using the STL algorithm **generate_n( )**. To perform the sorting, three **vector**s are created, each of which holds a different type of **Trash***. An iterator moves through **bin** and RTTI is used to determine which specific type of **Trash** the iterator is currently

selecting, placing each into the appropriate typed bin. Finally, **sumValue( )** is applied to each of the containers, and the **Trash** objects are cleaned up using **purge( )** (defined in Chapter XX). The creation and destruction counts ensure that things are properly cleaned up.

Of course, it seems silly to upcast the types of **Trash** into a container holding base type pointers, and then to turn around and downcast. Why not just put the trash into the appropriate receptacle in the first place? (indeed, this is the whole enigma of recycling). In this program it might be easy to repair, but sometimes a system's structure and flexibility can benefit greatly from downcasting.

The program satisfies the design requirements: it works. This may be fine as long as it's a one-shot solution. However, a good program will evolve over time, so you must ask: what if the situation changes? For example, cardboard is now a valuable recyclable commodity, so how will that be integrated into the system (especially if the program is large and complicated). Since the above type-check coding in the **switch** statement and in the RTTI statements could be scattered throughout the program, you'd have to go find all that code every time a new type was added, and if you miss one the compiler won't help you.

The key to the misuse of RTTI here is that *every type is tested*. If you're only looking for a subset of types because that subset needs special treatment, that's probably fine. But if you're hunting for every type inside a **switch** statement, then you're probably missing an important point, and definitely making your code less maintainable. In the next section we'll look at how this program evolved over several stages to become much more flexible. This should prove a valuable example in program design.

Improving the design

The solutions in *Design Patterns* are organized around the question "What will change as this program evolves?" This is usually the most important question that you can ask about any design. If you can build your system around the answer, the results will be two-pronged: not only will your system allow easy (and inexpensive) maintenance, but you might also produce components that are reusable, so that other systems can be built more cheaply. This is the promise of object-oriented programming, but it doesn't happen automatically; it requires thought and insight on your part. In this section we'll see how this process can happen during the refinement of a system.

The answer to the question "What will change?" for the recycling system is a common one: more types will be added to the system. The goal of the

design, then, is to make this addition of types as painless as possible. In the recycling program, we'd like to encapsulate all places where specific type information is mentioned, so (if for no other reason) any changes can be localized inside those encapsulations. It turns out that this process also cleans up the rest of the code considerably.

"Make more objects"

This brings up a general object-oriented design principle that I first heard spoken by Grady Booch: "If the design is too complicated, make more objects." This is simultaneously counterintuitive and ludicrously simple, and yet it's the most useful guideline I've found. (You might observe that "make more objects" is often equivalent to "add another level of indirection.") In general, if you find a place with messy code, consider what sort of class would clean things up. Often the side effect of cleaning up the code will be a system that has better structure and is more flexible.

Consider first the place where **Trash** objects are created. In the above example, we're conveniently using a generator to create the objects. The generator nicely encapsulates the creation of the objects, but the neatness is an illusion because in general we'll want to create the objects based on something more than a random number generator. Some information will be available which will determine what kind of **Trash** object this should be. Because you generally need to make your objects by examining some kind of information, if you're not paying close attention you may end up with **switch** statements (as in **TrashGen**) or cascaded **if** statements scattered throughout your code. This is definitely messy, and also a place where you must change code whenever a new type is added. If new types are commonly added, a better solution is a single member function that takes all of the necessary information and produces an object of the correct type, already upcast to a **Trash** pointer. In *Design Patterns* this is broadly referred to as a *creational pattern* (of which there are several). The specific pattern that will be applied here is a variant of the *Factory Method* ("method" being a more OOPish way to refer to a member function). Here, the factory method will be a **static** member of **Trash**, but more commonly it is a member function that is overridden in the derived class.

The idea of the factory method is that you pass it the essential information it needs to know to create your object, then stand back and wait for the pointer (already upcast to the base type) to pop out as the return value. From then on, you treat the object polymorphically. Thus, you never even need to know the exact type of object that's created. In fact, the factory method hides it from you to prevent accidental misuse. If you want to use

the object without polymorphism, you must explicitly use RTTI and casting.

But there's a little problem, especially when you use the more complicated approach (not shown here) of making the factory method in the base class and overriding it in the derived classes. What if the information required in the derived class requires more or different arguments? "Creating more objects" solves this problem. To implement the factory method, the **Trash** class gets a new member function called **factory( )**. To hide the creational data, there's a new class called **Info** that contains all of the necessary information for the **factory( )** method to create the appropriate **Trash** object. Here's a simple implementation of **Info**:

```
class Info {
  int type;
  // Must change this to add another type:
  static const int maxnum = 3;
  double data;
public:
  Info(int typeNum, double dat)
    : type(typeNum % maxnum), data(dat) {}
};
```

An **Info** object's only job is to hold information for the **factory( )** method. Now, if there's a situation in which **factory( )** needs more or different information to create a new type of **Trash** object, the **factory( )** interface doesn't need to be changed. The **Info** class can be changed by adding new data and new constructors, or in the more typical object-oriented fashion of subclassing.

Here's the second version of the program with the factory method added. The object-counting code has been removed; we'll assume proper cleanup will take place in all the rest of the examples.

```
//: C25:Recycle2.cpp
// Adding a factory method
#include "sumValue.h"
#include "../purge.h"
#include <fstream>
#include <vector>
#include <typeinfo>
#include <cstdlib>
#include <ctime>
using namespace std;
ofstream out("Recycle2.out");
```

```
class Trash {
  double _weight;
  void operator=(const Trash&);
  Trash(const Trash&);
public:
  Trash(double wt) : _weight(wt) { }
  virtual double value() const = 0;
  double weight() const { return _weight; }
  virtual ~Trash() {}
  // Nested class because it's tightly coupled
  // to Trash:
  class Info {
    int type;
    // Must change this to add another type:
    static const int maxnum = 3;
    double data;
    friend class Trash;
  public:
    Info(int typeNum, double dat)
      : type(typeNum % maxnum), data(dat) {}
  };
  static Trash* factory(const Info& info);
};

class Aluminum : public Trash {
  static double val;
public:
  Aluminum(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
  }
  ~Aluminum() { out << "~Aluminum\n"; }
};

double Aluminum::val = 1.67F;

class Paper : public Trash {
  static double val;
public:
  Paper(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
```

```
  }
  ~Paper() { out << "~Paper\n"; }
};

double Paper::val = 0.10F;

class Glass : public Trash {
  static double val;
public:
  Glass(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newval) {
    val = newval;
  }
  ~Glass() { out << "~Glass\n"; }
};

double Glass::val = 0.23F;

// Definition of the factory method. It must know
// all the types, so is defined after all the
// subtypes are defined:
Trash* Trash::factory(const Info& info) {
  switch(info.type) {
    default: // In case of overrun
    case 0:
      return new Aluminum(info.data);
    case 1:
      return new Paper(info.data);
    case 2:
      return new Glass(info.data);
  }
}

// Generator for Info objects:
class InfoGen {
  int typeQuantity;
  int maxWeight;
public:
  InfoGen(int typeQuant, int maxWt)
    : typeQuantity(typeQuant), maxWeight(maxWt) {
    srand(time(0));
  }
  Trash::Info operator()() {
```

```
    return Trash::Info(rand() % typeQuantity,
      static_cast<double>(rand() % maxWeight));
  }
};

int main() {
  vector<Trash*> bin;
  // Fill up the Trash bin:
  InfoGen infoGen(3, 100);
  for(int i = 0; i < 30; i++)
    bin.push_back(Trash::factory(infoGen()));
  vector<Aluminum*> alBin;
  vector<Paper*> paperBin;
  vector<Glass*> glassBin;
  vector<Trash*>::iterator sorter = bin.begin();
  // Sort the Trash:
  while(sorter != bin.end()) {
    Aluminum* ap =
      dynamic_cast<Aluminum*>(*sorter);
    Paper* pp = dynamic_cast<Paper*>(*sorter);
    Glass* gp = dynamic_cast<Glass*>(*sorter);
    if(ap) alBin.push_back(ap);
    if(pp) paperBin.push_back(pp);
    if(gp) glassBin.push_back(gp);
    sorter++;
  }
  sumValue(alBin);
  sumValue(paperBin);
  sumValue(glassBin);
  sumValue(bin);
  purge(bin); // Cleanup
} ///:~
```

In the factory method **Trash::factory( )**, the determination of the exact type of object is simple, but you can imagine a more complicated system in which **factory( )** uses an elaborate algorithm. The point is that it's now hidden away in one place, and you know to come to this place to make changes when you add new types.

The creation of new objects is now more general in **main( )**, and depends on "real" data (albeit created by another generator, driven by random numbers). The generator object is created, telling it the maximum type number and the largest "data" value to produce. Each call to the generator creates an **Info** object which is passed into **Trash::factory( )**,

which in turn produces some kind of **Trash** object and returns the pointer that's added to the **vector<Trash*> bin**.

The constructor for the **Info** object is very specific and restrictive in this example. However, you could also imagine a **vector** of arguments into the **Info** constructor (or directly into a **factory( )** call, for that matter). This requires that the arguments be parsed and checked at runtime, but it does provide the greatest flexibility.

You can see from this code what "vector of change" problem the factory is responsible for solving: if you add new types to the system (the change), the only code that must be modified is within the factory, so the factory isolates the effect of that change.

A pattern for prototyping creation

A problem with the above design is that it still requires a central location where all the types of the objects must be known: inside the **factory( )** method. If new types are regularly being added to the system, the **factory( )** method must be changed for each new type. When you discover something like this, it is useful to try to go one step further and move *all* of the activities involving that specific type – including its creation – into the class representing that type. This way, the only thing you need to do to add a new type to the system is to inherit a single class.

To move the information concerning type creation into each specific type of **Trash**, the "prototype" pattern will be used. The general idea is that you have a master container of objects, one of each type you're interested in making. The "prototype objects" in this container are used *only* for making new objects. In this case, we'll name the object-creation member function **clone( )**. When you're ready to make a new object, presumably you have some sort of information that establishes the type of object you want to create. The **factory( )** method (it's not required that you use factory with prototype, but they comingle nicely) moves through the master container comparing your information with whatever appropriate information is in the prototype objects in the master container. When a match is found, **factory( )** returns a clone of that object.

In this scheme there is no hard-coded information for creation. Each object knows how to expose appropriate information to allow matching, and how to clone itself. Thus, the **factory( )** method doesn't need to be changed when a new type is added to the system.

The prototypes will be contained in a **static vector<Trash*>** called **prototypes**. This is a **private** member of the base class **Trash**. The **friend** class **TrashPrototypeInit** is responsible for putting the **Trash*** prototypes into the prototype list.

You'll also note that the **Info** class has changed. It now uses a **string** to act as type identification information. As you shall see, this will allow us to read object information from a file when creating **Trash** objects.

```
//: C25:Trash.h
// Base class for Trash recycling examples
#ifndef TRASH_H
#define TRASH_H
#include <iostream>
#include <exception>
#include <vector>
#include <string>

class TypedBin; // For a later example
class Visitor; // For a later example

class Trash {
  double _weight;
  void operator=(const Trash&);
  Trash(const Trash&);
public:
  Trash(double wt) : _weight(wt) {}
  virtual double value() const = 0;
  double weight() const { return _weight; }
  virtual ~Trash() {}
  class Info {
    std::string _id;
    double _data;
  public:
    Info(std::string ident, double dat)
      : _id(ident), _data(dat) {}
    double data() const { return _data; }
    std::string id() const { return _id; }
    friend std::ostream& operator<<(
      std::ostream& os, const Info& info) {
      return os << info._id << ':' << info._data;
    }
  };
protected:
  // Remainder of class provides support for
```

```
  // prototyping:
  static std::vector<Trash*> prototypes;
  friend class TrashPrototypeInit;
  Trash() : _weight(0) {}
public:
  static Trash* factory(const Info& info);
  virtual std::string id() = 0;  // type ident
  virtual Trash* clone(const Info&) = 0;
  // Stubs, inserted for later use:
  virtual bool
  addToBin(std::vector<TypedBin*>&) {
    return false;
  }
  virtual void accept(Visitor&) {};
};
#endif // TRASH_H ///:~
```

The basic part of the **Trash** class remains as before. The rest of the class supports the prototyping pattern. The **id( )** member function returns a **string** that can be compared with the **id( )** of an **Info** object to determine whether this is the prototype that should be cloned (of course, the evaluation can be much more sophisticated than that if you need it). Both **id( )** and **clone( )** are pure **virtual** functions so they must be overridden in derived classes.

The last two member functions, **addToBin( )** and **accept( )**, are "stubs" which will be used in later versions of the trash sorting problem. It's necessary to have these virtual functions in the base class, but in the early examples there's no need for them, so they are not pure virtuals so as not to intrude.

The **factory( )** member function has the same declaration, but the definition is what handles the prototyping. Here is the implementation file:

```
//: C25:Trash.cpp {O}
#include "Trash.h"
using namespace std;

Trash* Trash::factory(const Info& info) {
  vector<Trash*>::iterator it;
  for(it = prototypes.begin();
    it != prototypes.end(); it++) {
    // Somehow determine the new type
    // to create, and clone one:
    if (info.id() == (*it)->id())
      return (*it)->clone(info);
```

```
  }
  cerr << "Prototype not found for "
    << info << endl;
  // "Default" to first one in the vector:
  return (*prototypes.begin())->clone(info);
} ///:~
```

The **string** inside the **Info** object contains the type name of the **Trash** to be created; this **string** is compared to the **id( )** values of the objects in **prototypes**. If there's a match, then that's the object to create.

Of course, the appropriate prototype object might not be in the **prototypes** list. In this case, the **return** in the inner loop is never executed and you'll drop out at the end, where a default value is created. It might be more appropriate to throw an exception here.

As you can see from the code, there's nothing that knows about specific types of **Trash**. The beauty of this design is that this code doesn't need to be changed, regardless of the different situations it will be used in.

Trash subclasses

To fit into the prototyping scheme, each new subclass of **Trash** must follow some rules. First, it must create a **protected** default constructor, so that no one but **TrashPrototypeInit** may use it. **TrashPrototypeInit** is a singleton, creating one and only one prototype object for each subtype. This guarantees that the **Trash** subtype will be properly represented in the **prototypes** container.

After defining the "ordinary" member functions and data that the **Trash** object will actually use, the class must also override the **id( )** member (which in this case returns a **string** for comparison) and the **clone( )** function, which must know how to pull the appropriate information out of the **Info** object in order to create the object correctly.

Here are the different types of **Trash**, each in their own file.

```
//: C25:Aluminum.h
// The Aluminum class with prototyping
#ifndef ALUMINUM_H
#define ALUMINUM_H
#include "Trash.h"

class Aluminum : public Trash {
  static double val;
protected:
  Aluminum() {}
```

```
  friend class TrashPrototypeInit;
public:
  Aluminum(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newVal) {
    val = newVal;
  }
  std::string id() { return "Aluminum"; }
  Trash* clone(const Info& info) {
    return new Aluminum(info.data());
  }
};
#endif // ALUMINUM_H ///:~
```

```
//: C25:Paper.h
// The Paper class with prototyping
#ifndef PAPER_H
#define PAPER_H
#include "Trash.h"

class Paper : public Trash {
  static double val;
protected:
  Paper() {}
  friend class TrashPrototypeInit;
public:
  Paper(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newVal) {
    val = newVal;
  }
  std::string id() { return "Paper"; }
  Trash* clone(const Info& info) {
    return new Paper(info.data());
  }
};
#endif // PAPER_H ///:~
```

```
//: C25:Glass.h
// The Glass class with prototyping
#ifndef GLASS_H
#define GLASS_H
#include "Trash.h"

class Glass : public Trash {
```

```
  static double val;
protected:
  Glass() {}
  friend class TrashPrototypeInit;
public:
  Glass(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newVal) {
    val = newVal;
  }
  std::string id() { return "Glass"; }
  Trash* clone(const Info& info) {
    return new Glass(info.data());
  }
};
#endif // GLASS_H ///:~
```

And here's a new type of **Trash**:

```
//: C25:Cardboard.h
// The Cardboard class with prototyping
#ifndef CARDBOARD_H
#define CARDBOARD_H
#include "Trash.h"

class Cardboard : public Trash {
  static double val;
protected:
  Cardboard() {}
  friend class TrashPrototypeInit;
public:
  Cardboard(double wt) : Trash(wt) {}
  double value() const { return val; }
  static void value(double newVal) {
    val = newVal;
  }
  std::string id() { return "Cardboard"; }
  Trash* clone(const Info& info) {
    return new Cardboard(info.data());
  }
};
#endif // CARDBOARD_H ///:~
```

The static **val** data members must be defined and initialized in a separate code file:

```
//: C25:TrashStatics.cpp {O}
// Contains the static definitions for
// the Trash type's "val" data members
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"

double Aluminum::val = 1.67;
double Paper::val = 0.10;
double Glass::val = 0.23;
double Cardboard::val = 0.14;
///:~
```

There's one other issue: initialization of the static data members. **TrashPrototypeInit** must create the prototype objects and add them to the **static Trash::prototypes** vector. So it's *very* important that you control the order of initialization of the **static** objects, so the **prototypes** vector is created before any of the prototype objects, which depend on the prior existence of **prototypes**. The most straightforward way to do this is to put all the definitions in a single file, in the order in which you want them initialized.

TrashPrototypeInit must be defined separately because it inserts the actual prototypes into the **vector**, and throughout the chapter we'll be inheriting new types of **Trash** from the existing types. By making this one class in a separate file, a different version can be created and linked in for the new situations, leaving the rest of the code in the system alone.

```
//: C25:TrashPrototypeInit.cpp {O}
// Performs initialization of all the prototypes.
// Create a different version of this file to
// make different kinds of Trash.
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"

// Allocate the static member object:
std::vector<Trash*> Trash::prototypes;

class TrashPrototypeInit {
  Aluminum a;
```

```
  Paper p;
  Glass g;
  Cardboard c;
  TrashPrototypeInit() {
    Trash::prototypes.push_back(&a);
    Trash::prototypes.push_back(&p);
    Trash::prototypes.push_back(&g);
    Trash::prototypes.push_back(&c);
  }
  static TrashPrototypeInit singleton;
};

TrashPrototypeInit
  TrashPrototypeInit::singleton; ///:~
```

This is taken a step further by making **TrashPrototypeInit** a singleton (the constructor is **private**), even though the class definition is not available in a header file so it would seem safe enough to assume that no one could accidentally make a second instance.

Unfortunately, this is one more separate piece of code you must maintain whenever you add a new type to the system. However, it's not too bad since the linker should give you an error message if you forget (since **prototypes** is defined in this file as well). The really difficult problems come when you *don't* get any warnings or errors if you do something wrong.

Parsing **Trash** from an external file

The information about **Trash** objects will be read from an outside file. The file has all of the necessary information about each piece of trash in a single entry in the form **Trash:weight**. There are multiple entries on a line, separated by commas:

```
//:! C25:Trash.dat
Glass:54, Paper:22, Paper:11, Glass:17,
Aluminum:89, Paper:88, Aluminum:76, Cardboard:96,
Aluminum:25, Aluminum:34, Glass:11, Glass:68,
Glass:43, Aluminum:27, Cardboard:44, Aluminum:18,
Paper:91, Glass:63, Glass:50, Glass:80,
Aluminum:81, Cardboard:12, Glass:12, Glass:54,
Aluminum:36, Aluminum:93, Glass:93, Paper:80,
Glass:36, Glass:12, Glass:60, Paper:66,
Aluminum:36, Cardboard:22,
///:~
```

To parse this, the line is read and the **string** member function **find( )** produces the index of the '**:**'. This is first used with the **string** member function **substr( )** to extract the name of the trash type, and next to get the weight that is turned into a **double** with the **atof( )** function (from **<cstdlib>**).

The **Trash** file parser is placed in a separate file since it will be reused throughout this chapter. To facilitate this reuse, the function **fillBin( )** which does the work takes as its first argument the name of the file to open and read, and as its second argument a reference to an object of type **Fillable**. This uses what I've named the "interface" idiom at the beginning of the chapter, and the only attribute for this particular interface is that "it can be filled," via a member function **addTrash( )**. Here's the header file for **Fillable**:

```
//: C25:Fillable.h
// Any object that can be filled with Trash
#ifndef FILLABLE_H
#define FILLABLE_H

class Fillable {
public:
  virtual void addTrash(Trash* t) = 0;
};
#endif // FILLABLE_H ///:~
```

Notice that it follows the interface idiom of having no non-static data members, and all pure **virtual** member functions.

This way, any class which implements this interface (typically using multiple inheritance) can be filled using **fillBin( )**. Here's the header file:

```
//: C25:fillBin.h
// Open a file and parse its contents into
// Trash objects, placing each into a vector
#ifndef FILLBIN_H
#define FILLBIN_H
#include "Fillablevector.h"
#include <vector>
#include <string>

void
fillBin(std::string filename, Fillable& bin);

// Special case to handle vector:
inline void fillBin(std::string filename,
```

```
  std::vector<Trash*>& bin) {
  Fillablevector fv(bin);
  fillBin(filename, fv);
}
#endif // FILLBIN_H ///:~
```

The overloaded version will be discussed shortly. First, here is the implementation:

```
//: C25:fillBin.cpp {O}
// Implementation of fillBin()
#include "fillBin.h"
#include "Fillable.h"
#include "../C17/trim.h"
#include "../require.h"
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;

void fillBin(string filename, Fillable& bin) {
  ifstream in(filename.c_str());
  assure(in, filename.c_str());
  string s;
  while(getline(in, s)) {
    int comma = s.find(',');
    // Parse each line into entries:
    while(comma != string::npos) {
      string e = trim(s.substr(0,comma));
      // Parse each entry:
      int colon = e.find(':');
      string type = e.substr(0, colon);
      double weight =
        atof(e.substr(colon + 1).c_str());
      bin.addTrash(
        Trash::factory(
          Trash::Info(type, weight)));
      // Move to next part of line:
      s = s.substr(comma + 1);
      comma = s.find(',');
    }
  }
} ///:~
```

After the file is opened, each line is read and parsed into entries by looking for the separating comma, then each entry is parsed into its type and weight by looking for the separating colon. Note the convenience of using the **trim( )** function from chapter 17 to remove the white space from both ends of a **string**. Once the type and weight are discovered, an **Info** object is created from that data and passed to the **factory( )**. The result of this call is a **Trash*** which is passed to the **addTrash( )** function of the **bin** (which is the only function, remember, that a **Fillable** guarantees).

Anything that supports the **Fillable** interface can be used with **fillBin( )**. Of course, **vector** doesn't implement **Fillable**, so it won't work. Since **vector** is used in most of the examples, it makes sense to add the second overloaded **fillBin( )** function that takes a **vector**, as seen previously in **fillBin.h**. But how to make a **vector<Trash*>** adapt to the **Fillable** interface, which says it must have an **addTrash( )** member function? The key is in the word "adapt"; we use the adapter pattern to create a class that has a **vector** and is also **Fillable**.

By saying "is also **Fillable**," the hint is strong (is-a) to inherit from **Fillable**. But what about the **vector<Trash*>**? Should this new class inherit from that? We don't actually want to be making a new kind of **vector**, which would force everyone to only use our **vector** in this situation. Instead, we want someone to be able to have their own **vector** and say "please fill this." So the new class should just keep a reference to that **vector**:

```
//: C25:Fillablevector.h
// Adapter that makes a vector<Trash*> Fillable
#ifndef FILLABLEVECTOR_H
#define FILLABLEVECTOR_H
#include "Trash.h"
#include "Fillable.h"
#include <vector>

class Fillablevector : public Fillable {
  std::vector<Trash*>& v;
public:
  Fillablevector(std::vector<Trash*>& vv)
    : v(vv) {}
  void addTrash(Trash* t) { v.push_back(t); }
};
#endif // FILLABLEVECTOR_H ///:~
```

You can see that the only job of this class is to connect **Fillable**'s **addTrash( )** member function to **vector**'s **push_back( )** (that's the

"adapter" motivation). With this class in hand, the overloaded **fillBin( )** member function can be used with a **vector** in **fillBin.h**:

```
inline void fillBin(std::string filename,
  std::vector<Trash*>& bin) {
  Fillablevector fv(bin);
  fillBin(filename, fv);
}
```

Notice that the adapter object **fv** only exists for the duration of the function call, and it wraps **bin** in an interface that works with the other **fillBin( )** function.

This approach works for any container class that's used frequently. Alternatively, the container can multiply inherit from **Fillable**. (You'll see this later, in **DynaTrash.cpp**.)

Recycling with prototyping

Now you can see the new version of the recycling solution using the prototyping technique:

```
//: C25:Recycle3.cpp
//{L} TrashPrototypeInit
//{L} fillBin Trash TrashStatics
// Recycling with RTTI and Prototypes
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "fillBin.h"
#include "sumValue.h"
#include "../purge.h"
#include <fstream>
#include <vector>
using namespace std;
ofstream out("Recycle3.out");

int main() {
  vector<Trash*> bin;
  // Fill up the Trash bin:
  fillBin("Trash.dat", bin);
  vector<Aluminum*> alBin;
  vector<Paper*> paperBin;
  vector<Glass*> glassBin;
  vector<Trash*>::iterator it = bin.begin();
```

```
  while(it != bin.end()) {
    // Sort the Trash:
    Aluminum* ap =
      dynamic_cast<Aluminum*>(*it);
    Paper* pp = dynamic_cast<Paper*>(*it);
    Glass* gp = dynamic_cast<Glass*>(*it);
    if(ap) alBin.push_back(ap);
    if(pp) paperBin.push_back(pp);
    if(gp) glassBin.push_back(gp);
    it++;
  }
  sumValue(alBin);
  sumValue(paperBin);
  sumValue(glassBin);
  sumValue(bin);
  purge(bin);
} ///:~
```

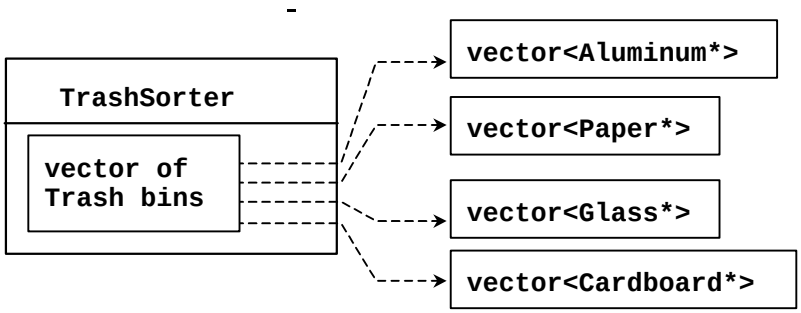
The process of opening the data file containing **Trash** descriptions and the parsing of that file have been wrapped into **fillBin( )**, so now it's no longer a part of our design focus. You will see that throughout the rest of the chapter, no matter what new classes are added, **fillBin( )** will continue to work without change, which indicates a good design.

In terms of object creation, this design does indeed severely localize the changes you need to make to add a new type to the system. However, there's a significant problem in the use of RTTI that shows up clearly here. The program seems to run fine, and yet it never detects any cardboard, even though there is cardboard in the list of trash data! This happens *because* of the use of RTTI, which looks for only the types that you tell it to look for. The clue that RTTI is being misused is that *every type in the system* is being tested, rather than a single type or subset of types. But if you forget to test for your new type, the compiler has nothing to say about it.

As you will see later, there are ways to use polymorphism instead when you're testing for every type. But if you use RTTI a lot in this fashion, and you add a new type to your system, you can easily forget to make the necessary changes in your program and produce a difficult-to-find bug. So it's worth trying to eliminate RTTI in this case, not just for aesthetic reasons – it produces more maintainable code.

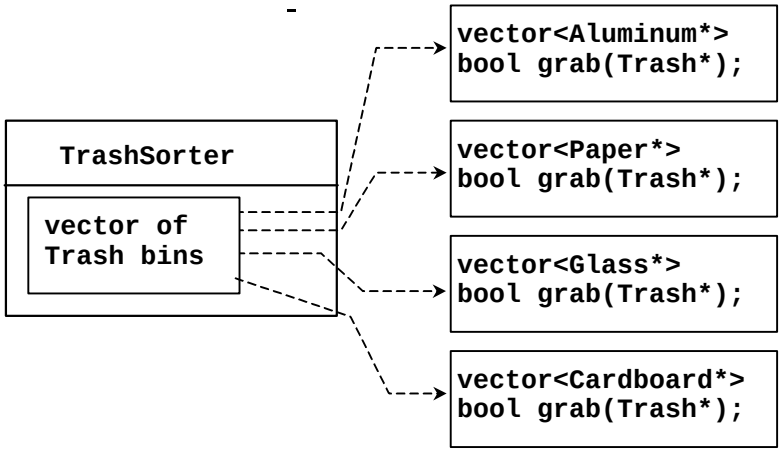
Abstracting usage

With creation out of the way, it's time to tackle the remainder of the design: where the classes are used. Since it's the act of sorting into bins that's particularly ugly and exposed, why not take that process and hide it inside a class? This is simple "complexity hiding," the principle of "If you must do something ugly, at least localize the ugliness." In an OOP language, the best place to hide complexity is inside a class. Here's a first cut:



A **TrashSorter** object holds a **vector** that somehow connects to **vector**s holding specific types of **Trash**. The most convenient solution would be a **vector<vector<Trash*>>**, but it's too early to tell if that would work out best.

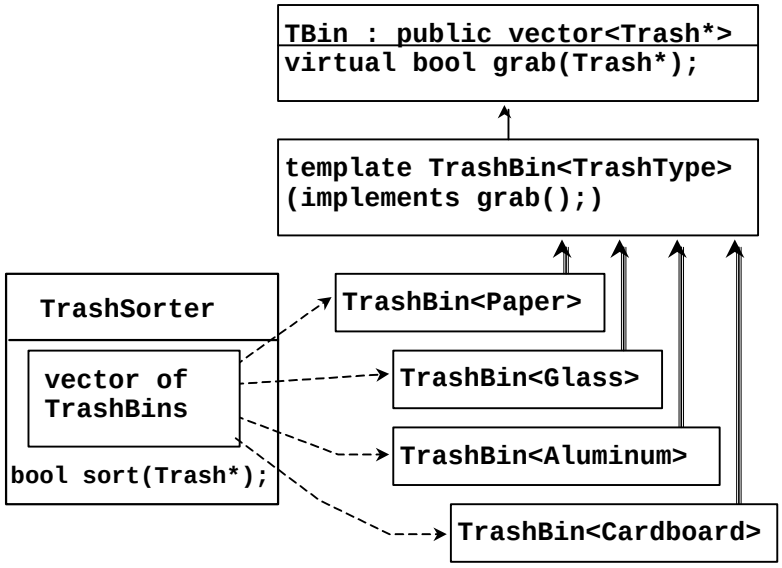
In addition, we'd like to have a **sort( )** function as part of the **TrashSorter** class. But, keeping in mind that the goal is easy addition of new types of **Trash**, how would the statically-coded **sort( )** function deal with the fact that a new type has been added? To solve this, the type information must be removed from **sort( )** so all it needs to do is call a generic function which takes care of the details of type. This, of course, is another way to describe a **virtual** function. So **sort( )** will simply move through the **vector** of **Trash** bins and call a virtual function for each. I'll call the function **grab(Trash*)**, so the structure now looks like this:



However, **TrashSorter** needs to call **grab( )** polymorphically, through a common base class for all the **vector**s. This base class is very simple, since it only needs to establish the interface for the **grab( )** function.

Now there's a choice. Following the above diagram, you could put a **vector** of **trash** pointers as a member object of each subclassed **Tbin**. However, you will want to treat each **Tbin** as a **vector**, and perform all the **vector** operations on it. You could create a new interface and forward all those operations, but that produces work and potential bugs. The type we're creating is really a **Tbin** *and* a **vector**, which suggests multiple inheritance. However, it turns out that's not quite necessary, for the following reason.

Each time a new type is added to the system the programmer will have to go in and derive a new class for the **vector** that holds the new type of **Trash**, along with its **grab( )** function. The code the programmer writes will actually be *identical code except for the type it's working with*. That last phrase is the key to introduce a template, which will do all the work of adding a new type. Now the diagram looks more complicated, although the process of adding a new type to the system will be simple. Here, **TrashBin** can inherit from **TBin**, which inherits from **vector<Trash*>** like this (the multiple-lined arrows indicated template instantiation):



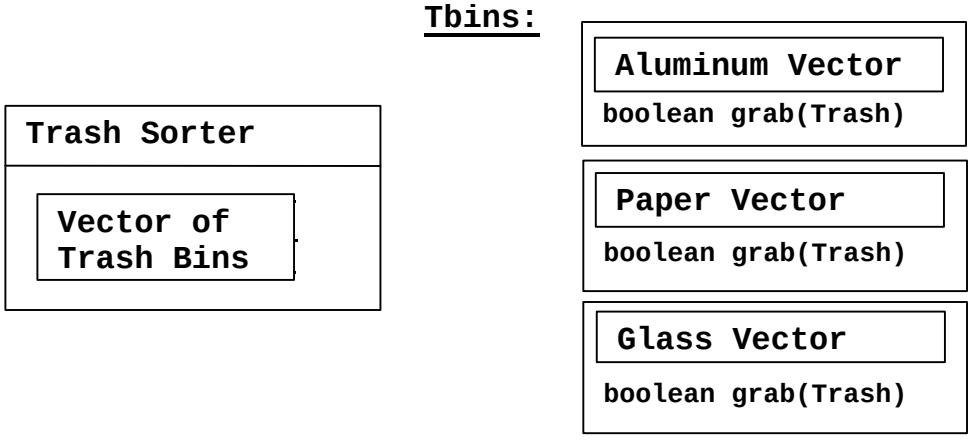
The reason **TrashBin** must be a template is so it can automatically generate the **grab( )** function. A further templatization will allow the **vector**s to hold specific types.

That said, we can look at the whole program to see how all this is implemented.

```
//: C25:Recycle4.cpp
//{L} TrashPrototypeInit
//{L} fillBin Trash TrashStatics
// Adding TrashBins and TrashSorters
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"
#include "fillBin.h"
#include "sumValue.h"
#include "../purge.h"
#include <fstream>
#include <vector>
using namespace std;
ofstream out("Recycle4.out");

class TBin : public vector<Trash*> {
public:
```

```
  virtual bool grab(Trash*) = 0;
};

template<class TrashType>
class TrashBin : public TBin {
public:
  bool grab(Trash* t) {
    TrashType* tp = dynamic_cast<TrashType*>(t);
    if(!tp) return false; // Not grabbed
    push_back(tp);
    return true; // Object grabbed
  }
};

class TrashSorter : public vector<TBin*> {
public:
  bool sort(Trash* t) {
    for(iterator it = begin(); it != end(); it++)
      if((*it)->grab(t))
        return true;
    return false;
  }
  void sortBin(vector<Trash*>& bin) {
    vector<Trash*>::iterator it;
    for(it = bin.begin(); it != bin.end(); it++)
      if(!sort(*it))
        cerr << "bin not found" << endl;
  }
  ~TrashSorter() { purge(*this); }
};

int main() {
  vector<Trash*> bin;
  // Fill up the Trash bin:
  fillBin("Trash.dat", bin);
  TrashSorter tbins;
  tbins.push_back(new TrashBin<Aluminum>());
  tbins.push_back(new TrashBin<Paper>());
  tbins.push_back(new TrashBin<Glass>());
  tbins.push_back(new TrashBin<Cardboard>());
  tbins.sortBin(bin);
  for(TrashSorter::iterator it = tbins.begin();
    it != tbins.end(); it++)
    sumValue(**it);
```

```
  sumValue(bin);
  purge(bin);
} ///:~
```



TrashSorter needs to call each **grab( )** member function and get a different result depending on what type of **Trash** the current **vector** is holding. That is, each **vector** must be aware of the type it holds. This "awareness" is accomplished with a **virtual** function, the **grab( )** function, which thus eliminates at least the outward appearance of the use of RTTI. The implementation of **grab( )** *does* use RTTI, but it's templatized so as long as you put a new **TrashBin** in the **TrashSorter** when you add a type, everything else is taken care of.

Memory is managed by denoting **bin** as the "master container," the one responsible for cleanup. With this rule in place, calling **purge( )** for **bin** cleans up all the **Trash** objects. In addition, **TrashSorter** assumes that it "owns" the pointers it holds, and cleans up all the **TrashBin** objects during destruction.

A basic OOP design principle is "Use data members for variation in state, use polymorphism for variation in behavior." Your first thought might be that the **grab( )** member function certainly behaves differently for a **vector** that holds **Paper** than for one that holds **Glass**. But what it does is strictly dependent on the type, and nothing else.

1. **TbinList** holds a set of **Tbin** pointers, so that **sort( )** can iterate through the **Tbin**s when it's looking for a match for the **Trash** object you've handed it.
2. **sortBin( )** allows you to pass an entire **Tbin** in, and it moves through the **Tbin**, picks out each piece of **Trash**, and sorts it into the appropriate specific **Tbin**. Notice the genericity of this code: it doesn't

change at all if new types are added. If the bulk of your code doesn't need changing when a new type is added (or some other change occurs) then you have an easily-extensible system.

3. Now you can see how easy it is to add a new type. Few lines must be changed to support the addition. If it's really important, you can squeeze out even more by further manipulating the design.
4. One member function call causes the contents of **bin** to be sorted into the respective specifically-typed bins.

Applying double dispatching

The above design is certainly satisfactory. Adding new types to the system consists of adding or modifying distinct classes without causing code changes to be propagated throughout the system. In addition, RTTI is not as "misused" as it was in **Recycle1.cpp**. However, it's possible to go one step further and eliminate RTTI altogether from the operation of sorting the trash into bins.

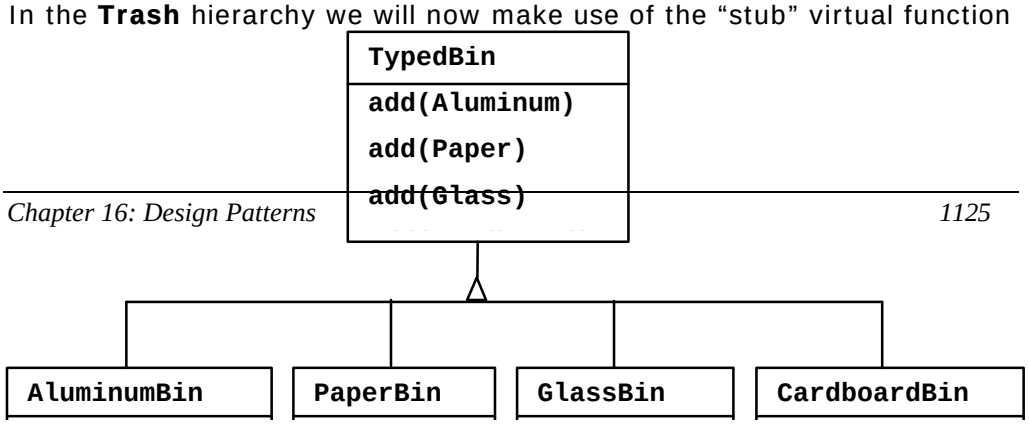
To accomplish this, you must first take the perspective that all type-dependent activities – such as detecting the type of a piece of trash and putting it into the appropriate bin – should be controlled through polymorphism and dynamic binding.

The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. But whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically-bound member function calls) is to handle type-specific information for you. So why are you hunting for types?

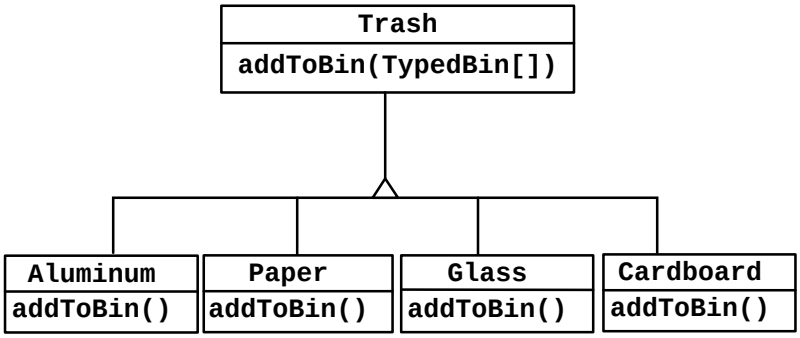
The multiple-dispatch pattern demonstrated at the beginning of this chapter uses **virtual** functions to determine all type information, thus eliminating RTTI.

Implementing the double dispatch

In the **Trash** hierarchy we will now make use of the "stub" virtual function



addToBin( ) that was added to the base class **Trash** but unused up until now. This takes an argument of a container of **TypedBin**. A **Trash** object uses **addToBin( )** with this container to step through and try to add itself



to the appropriate bin, and this is where you'll see the double dispatch.

The new hierarchy is **TypedBin**, and it contains its own member function called **add( )** that is also used polymorphically. But here's an additional twist: **add( )** is *overloaded* to take arguments of the different types of **Trash**. So an essential part of the double dispatching scheme also involves overloading (or at least having a group of virtual functions to call; overloading happens to be particularly convenient here).

```
//: C25:TypedBin.h
#ifndef TYPEDBIN_H
#define TYPEDBIN_H
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"
#include <vector>

// Template to generate double-dispatching
// trash types by inheriting from originals:
template<class TrashType>
class DD : public TrashType {
protected:
  DD() : TrashType(0) {}
  friend class TrashPrototypeInit;
public:
  DD(double wt) : TrashType(wt) {}
  bool addToBin(std::vector<TypedBin*>& tb) {
    for(int i = 0; i < tb.size(); i++)
```

```
      if(tb[i]->add(this))
        return true;
    return false;
  }
  // Override clone() to create this new type:
  Trash* clone(const Trash::Info& info) {
    return new DD(info.data());
  }
};

// vector<Trash*> that knows how to
// grab the right type
class TypedBin : public std::vector<Trash*> {
protected:
  bool addIt(Trash* t) {
    push_back(t);
    return true;
  }
public:
  virtual bool add(DD<Aluminum>*) {
    return false;
  }
  virtual bool add(DD<Paper>*) {
    return false;
  }
  virtual bool add(DD<Glass>*) {
    return false;
  }
  virtual bool add(DD<Cardboard>*) {
    return false;
  }
};

// Template to generate specific TypedBins:
template<class TrashType>
class BinOf : public TypedBin {
public:
  // Only overrides add() for this specific type:
  bool add(TrashType* t) { return addIt(t); }
};
#endif // TYPEDBIN_H ///:~
```

In each particular subtype of **Aluminum**, **Paper**, **Glass,** and **Cardboard**, the **addToBin( )** member function is implemented, but it *looks* like the

code is exactly the same in each case. The code in each **addToBin( )** calls **add( )** for each **TypedBin** object in the array. But notice the argument: **this**. The type of **this** is different for each subclass of **Trash**, so the code is different. So this is the first part of the double dispatch, because once you're inside this member function you know you're **Aluminum**, or **Paper**, etc. During the call to **add( )**, this information is passed via the type of **this**. The compiler resolves the call to the proper overloaded version of **add( )**. But since **tb[i]** produces a pointer to the base type **TypedBin**, this call will end up calling a different member function depending on the type of **TypedBin** that's currently selected. That is the second dispatch.

You can see that the overloaded **add( )** methods all return **false**. If the member function is not overloaded in a derived class, it will continue to return **false**, and the caller (**addToBin( )**, in this case) will assume that the current **Trash** object has not been added successfully to a container, and continue searching for the right container.

In each of the subclasses of **TypedBin**, only one overloaded member function is overridden, according to the type of bin that's being created. For example, **CardboardBin** overrides **add(DD<Cardboard>)**. The overridden member function adds the **Trash** pointer to its container and returns **true**, while all the rest of the **add( )** methods in **CardboardBin** continue to return **false**, since they haven't been overridden. With C++ **template**s, you don't have to explicitly write the subclasses or place the **addToBin( )** member function in **Trash**.

To set up for prototyping the new types of trash, there must be a different initializer file:

```
//: C25:DDTrashPrototypeInit.cpp {O}
#include "TypedBin.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"

std::vector<Trash*> Trash::prototypes;

class TrashPrototypeInit {
  DD<Aluminum> a;
  DD<Paper> p;
  DD<Glass> g;
  DD<Cardboard> c;
  TrashPrototypeInit() {
    Trash::prototypes.push_back(&a);
```

```
    Trash::prototypes.push_back(&p);
    Trash::prototypes.push_back(&g);
    Trash::prototypes.push_back(&c);
  }
  static TrashPrototypeInit singleton;
};

TrashPrototypeInit
  TrashPrototypeInit::singleton; ///:~
```

Here's the rest of the program:

```
//: C25:DoubleDispatch.cpp
//{L} DDTrashPrototypeInit
//{L} fillBin Trash TrashStatics
// Using multiple dispatching to handle more than
// one unknown type during a member function call
#include "TypedBin.h"
#include "fillBin.h"
#include "sumValue.h"
#include "../purge.h"
#include <iostream>
#include <fstream>
using namespace std;
ofstream out("DoubleDispatch.out");

class TrashBinSet : public vector<TypedBin*> {
public:
  TrashBinSet() {
    push_back(new BinOf<DD<Aluminum> >());
    push_back(new BinOf<DD<Paper> >());
    push_back(new BinOf<DD<Glass> >());
    push_back(new BinOf<DD<Cardboard> >());
  };
  void sortIntoBins(vector<Trash*>& bin) {
    vector<Trash*>::iterator it;
    for(it = bin.begin(); it != bin.end(); it++)
      // Perform the double dispatch:
      if(!(*it)->addToBin(*this))
        cerr << "Couldn't add " << *it << endl;
  }
  ~TrashBinSet() { purge(*this); }
};

int main() {
```

```
  vector<Trash*> bin;
  TrashBinSet bins;
  // fillBin() still works, without changes, but
  // different objects are cloned:
  fillBin("Trash.dat", bin);
  // Sort from the master bin into the
  // individually-typed bins:
  bins.sortIntoBins(bin);
  TrashBinSet::iterator it;
  for(it = bins.begin(); it != bins.end(); it++)
    sumValue(**it);
  // ... and for the master bin
  sumValue(bin);
  purge(bin);
} ///:~
```

TrashBinSet encapsulates all of the different types of **TypedBin**s, along with the **sortIntoBins( )** member function, which is where all the double dispatching takes place. You can see that once the structure is set up, sorting into the various **TypedBin**s is remarkably easy. In addition, the efficiency of two virtual calls and the double dispatch is probably better than any other way you could sort.

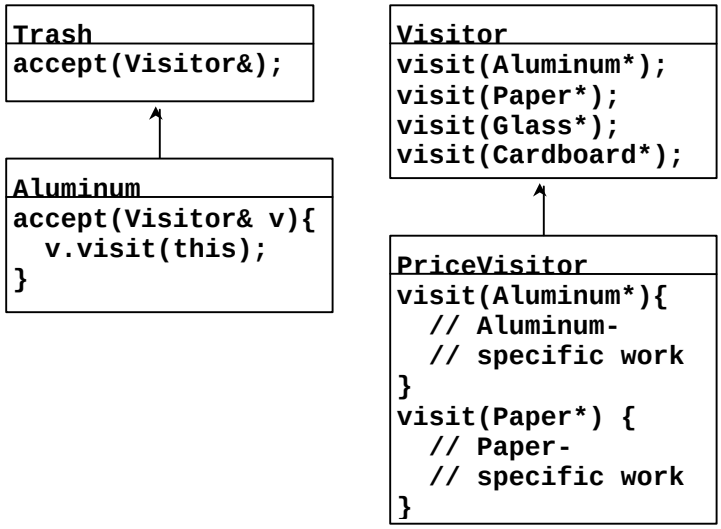
Notice the ease of use of this system in **main( )**, as well as the complete independence of any specific type information within **main( )**. All other methods that talk only to the **Trash** base-class interface will be equally invulnerable to changes in **Trash** types.

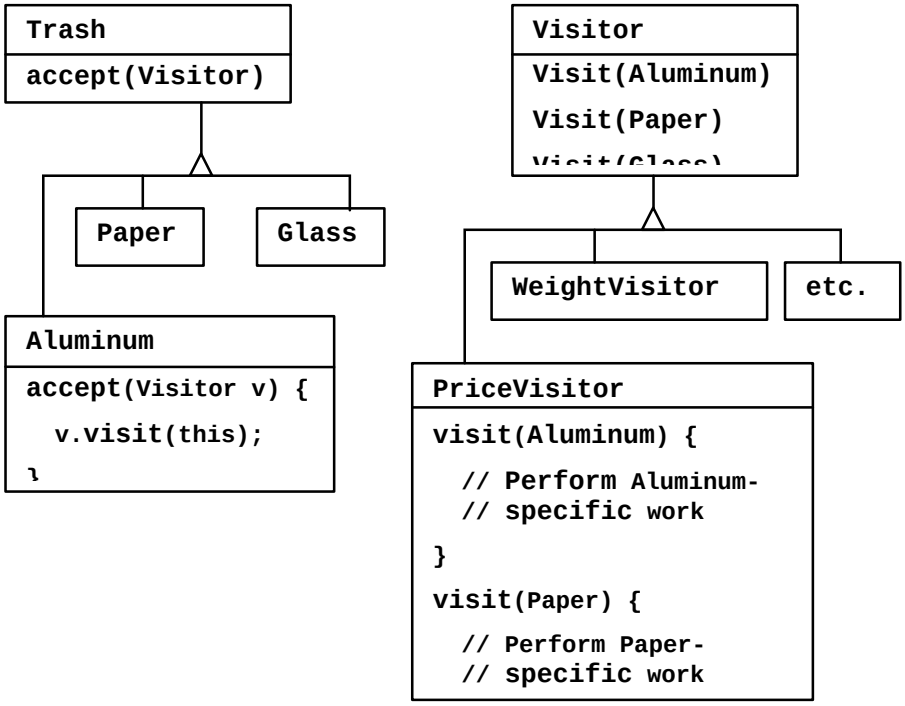
The changes necessary to add a new type are relatively isolated: you inherit the new type of **Trash** with its **addToBin( )** member function, then make a small modification to **TypedBin**, and finally you add a new type into the vector in **TrashBinSet** and modify **DDTrashPrototypeInit.cpp**.

Applying the visitor pattern

Now consider applying a design pattern with an entirely different goal to the trash-sorting problem. As demonstrated earlier in this chapter, the visitor pattern's goal is to allow the addition of new polymorphic operations to a frozen inheritance hierarchy.

For this pattern, we are no longer concerned with optimizing the addition of new types of **Trash** to the system. Indeed, this pattern makes adding a new type of **Trash** *more* complicated. It looks like this:





Now, if **t** is a **Trash** pointer to an **Aluminum** object, the code:

```
PriceVisitor pv;
t->accept(pv);
```

causes two polymorphic member function calls: the first one to select **Aluminum**'s version of **accept( )**, and the second one within **accept( )** when the specific version of **visit( )** is called dynamically using the base-class **Visitor** pointer **v**.

This configuration means that new functionality can be added to the system in the form of new subclasses of **Visitor**. The **Trash** hierarchy doesn't need to be touched. This is the prime benefit of the visitor pattern: you can add new polymorphic functionality to a class hierarchy without touching that hierarchy (once the **accept( )** methods have been installed). Note that the benefit is helpful here but not exactly what we started out to accomplish, so at first blush you might decide that this isn't the desired solution.

But look at one thing that's been accomplished: the visitor solution avoids sorting from the master **Trash** sequence into individual typed sequences. Thus, you can leave everything in the single master sequence and simply pass through that sequence using the appropriate visitor to accomplish

the goal. Although this behavior seems to be a side effect of visitor, it does give us what we want (avoiding RTTI).

The double dispatching in the visitor pattern takes care of determining both the type of **Trash** and the type of **Visitor**. In the following example, there are two implementations of **Visitor**: **PriceVisitor** to both determine and sum the price, and **WeightVisitor** to keep track of the weights.

You can see all of this implemented in the new, improved version of the recycling program. As with **DoubleDispatch.cpp**, the **Trash** class has had an extra member function stub (**accept( )**) inserted in it to allow for this example.

Since there's nothing concrete in the **Visitor** base class, it can be created as an **interface**:

```
//: C25:Visitor.h
// The base interface for visitors
// and template for visitable Trash types
#ifndef VISITOR_H
#define VISITOR_H
#include "Trash.h"
#include "Aluminum.h"
#include "Paper.h"
#include "Glass.h"
#include "Cardboard.h"

class Visitor {
public:
  virtual void visit(Aluminum* a) = 0;
  virtual void visit(Paper* p) = 0;
  virtual void visit(Glass* g) = 0;
  virtual void visit(Cardboard* c) = 0;
};

// Template to generate visitable
// trash types by inheriting from originals:
template<class TrashType>
class Visitable : public TrashType {
protected:
  Visitable () : TrashType(0) {}
  friend class TrashPrototypeInit;
public:
  Visitable(double wt) : TrashType(wt) {}
  // Remember "this" is pointer to current type:
  void accept(Visitor& v) { v.visit(this); }
```

```
  // Override clone() to create this new type:
  Trash* clone(const Trash::Info& info) {
    return new Visitable(info.data());
  }
};
#endif // VISITOR_H ///:~
```

As before, a different version of the initialization file is necessary:

```
//: C25:VisitorTrashPrototypeInit.cpp {O}
#include "Visitor.h"

std::vector<Trash*> Trash::prototypes;

class TrashPrototypeInit {
  Visitable<Aluminum> a;
  Visitable<Paper> p;
  Visitable<Glass> g;
  Visitable<Cardboard> c;
  TrashPrototypeInit() {
    Trash::prototypes.push_back(&a);
    Trash::prototypes.push_back(&p);
    Trash::prototypes.push_back(&g);
    Trash::prototypes.push_back(&c);
  }
  static TrashPrototypeInit singleton;
};

TrashPrototypeInit
  TrashPrototypeInit::singleton; ///:~
```

The rest of the program creates specific **Visitor** types and sends them through a single list of **Trash** objects:

```
//: C25:TrashVisitor.cpp
//{L} VisitorTrashPrototypeInit
//{L} fillBin Trash TrashStatics
// The "visitor" pattern
#include "Visitor.h"
#include "fillBin.h"
#include "../purge.h"
#include <iostream>
#include <fstream>
using namespace std;
ofstream out("TrashVisitor.out");
```

```
// Specific group of algorithms packaged
// in each implementation of Visitor:
class PriceVisitor : public Visitor {
  double alSum; // Aluminum
  double pSum; // Paper
  double gSum; // Glass
  double cSum; // Cardboard
public:
  void visit(Aluminum* al) {
    double v = al->weight() * al->value();
    out << "value of Aluminum= " << v << endl;
    alSum += v;
  }
  void visit(Paper* p) {
    double v = p->weight() * p->value();
    out <<
      "value of Paper= " << v << endl;
    pSum += v;
  }
  void visit(Glass* g) {
    double v = g->weight() * g->value();
    out <<
      "value of Glass= " << v << endl;
    gSum += v;
  }
  void visit(Cardboard* c) {
    double v = c->weight() * c->value();
    out <<
      "value of Cardboard = " << v << endl;
    cSum += v;
  }
  void total(ostream& os) {
    os <<
      "Total Aluminum: $" << alSum << "\n" <<
      "Total Paper: $" << pSum << "\n" <<
      "Total Glass: $" << gSum << "\n" <<
      "Total Cardboard: $" << cSum << endl;
  }
};

class WeightVisitor : public Visitor {
  double alSum; // Aluminum
  double pSum; // Paper
```

```
  double gSum; // Glass
  double cSum; // Cardboard
public:
  void visit(Aluminum* al) {
    alSum += al->weight();
    out << "weight of Aluminum = "
        << al->weight() << endl;
  }
  void visit(Paper* p) {
    pSum += p->weight();
    out << "weight of Paper = "
      << p->weight() << endl;
  }
  void visit(Glass* g) {
    gSum += g->weight();
    out << "weight of Glass = "
        << g->weight() << endl;
  }
  void visit(Cardboard* c) {
    cSum += c->weight();
    out << "weight of Cardboard = "
        << c->weight() << endl;
  }
  void total(ostream& os) {
    os << "Total weight Aluminum:"
       << alSum << endl;
    os << "Total weight Paper:"
       << pSum << endl;
    os << "Total weight Glass:"
       << gSum << endl;
    os << "Total weight Cardboard:"
       << cSum << endl;
  }
};

int main() {
  vector<Trash*> bin;
  // fillBin() still works, without changes, but
  // different objects are prototyped:
  fillBin("Trash.dat", bin);
  // You could even iterate through
  // a list of visitors!
  PriceVisitor pv;
  WeightVisitor wv;
```

```
  vector<Trash*>::iterator it = bin.begin();
  while(it != bin.end()) {
    (*it)->accept(pv);
    (*it)->accept(wv);
    it++;
  }
  pv.total(out);
  wv.total(out);
  purge(bin);
} ///:~
```

Note that the shape of **main( )** has changed again. Now there's only a single **Trash** bin. The two **Visitor** objects are accepted into every element in the sequence, and they perform their operations. The visitors keep their own internal data to tally the total weights and prices.

Finally, there's no run-time type identification other than the inevitable cast to **Trash** when pulling things out of the sequence.

One way you can distinguish this solution from the double dispatching solution described previously is to note that, in the double dispatching solution, only one of the overloaded methods, **add( )**, was overridden when each subclass was created, while here *each* one of the overloaded **visit( )** methods is overridden in every subclass of **Visitor**.

More coupling?

There's a lot more code here, and there's definite coupling between the **Trash** hierarchy and the **Visitor** hierarchy. However, there's also high cohesion within the respective sets of classes: they each do only one thing (**Trash** describes trash, while **Visitor** describes actions performed on **Trash**), which is an indicator of a good design. Of course, in this case it works well only if you're adding new **Visitor**s, but it gets in the way when you add new types of **Trash**.

Low coupling between classes and high cohesion within a class is definitely an important design goal. Applied mindlessly, though, it can prevent you from achieving a more elegant design. It seems that some classes inevitably have a certain intimacy with each other. These often occur in pairs that could perhaps be called *couplets*, for example, containers and iterators. The **Trash-Visitor** pair above appears to be another such couplet.

RTTI considered harmful?

Various designs in this chapter attempt to remove RTTI, which might give you the impression that it's "considered harmful" (the condemnation used for poor **goto**). This isn't true; it is the *misuse* of RTTI that is the problem. The reason our designs removed RTTI is because the misapplication of that feature prevented extensibility, which contravened the stated goal of adding a new type to the system with as little impact on surrounding code as possible. Since RTTI is often misused by having it look for every single type in your system, it causes code to be non-extensible: when you add a new type, you have to go hunting for all the code in which RTTI is used, and if you miss any you won't get help from the compiler.

However, RTTI doesn't automatically create non-extensible code. Let's revisit the trash recycler once more. This time, a new tool will be introduced, which I call a **TypeMap**. It inherits from a **map** that holds a variant of **type_info** object as the key, and **vector<Trash*>** as the value. The interface is simple: you call **addTrash( )** to add a new **Trash** pointer, and the **map** class provides the rest of the interface. The keys represent the types contained in the associated **vector**. The beauty of this design (suggested by Larry O'Brien) is that the **TypeMap** dynamically adds a new key-value pair whenever it encounters a new type, so whenever you add a new type to the system (even if you add the new type at runtime), it adapts.

The example will again build on the structure of the **Trash** types, and will use **fillBin( )** to parse and insert the values into the **TypeMap**. However, **TypeMap** is not a **vector<Trash*>**, and so it must be adapted to work with **fillBin( )** by multiply inheriting from **Fillable**. In addition, the Standard C++ **type_info** class is too restrictive to be used as a key, so a kind of wrapper class **TypeInfo** is created, which simply extracts and stores the **type_info char*** representation of the type (making the assumption that, within the realm of a single compiler, this representation will be unique for each type).

```
//: C25:DynaTrash.cpp
//{L} TrashPrototypeInit
//{L} fillBin Trash TrashStatics
// Using a map of vectors and RTTI
// to automatically sort Trash into
// vectors. This solution, despite the
// use of RTTI, is extensible.
```

```
#include "Trash.h"
#include "fillBin.h"
#include "sumValue.h"
#include "../purge.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <typeinfo>
using namespace std;
ofstream out("DynaTrash.out");

// Must adapt from type_info in Standard C++,
// since type_info is too restrictive:
template<class T> // T should be a base class
class TypeInfo {
  string id;
public:
  TypeInfo(T* t) : id(typeid(*t).name()) {}
  const string& name() { return id; }
  friend bool operator<(const TypeInfo& lv,
    const TypeInfo& rv){
    return lv.id < rv.id;
  }
};

class TypeMap :
  public map<TypeInfo<Trash>, vector<Trash*> >,
  public Fillable {
public:
  // Satisfies the Fillable interface:
  void addTrash(Trash* t) {
    (*this)[TypeInfo<Trash>(t)].push_back(t);
  }
  ~TypeMap() {
    for(iterator it = begin(); it != end(); it++)
      purge((*it).second);
  }
};

int main() {
  TypeMap bin;
  fillBin("Trash.dat", bin); // Sorting happens
  TypeMap::iterator it;
```

```
  for(it = bin.begin(); it != bin.end(); it++)
    sumValue((*it).second);
} ///:~
```

TypeInfo is templatized because **typeid( )** does not allow the use of **void***, which would be the most general way to solve the problem. So you are required to work with some specific class, but this class should be the most base of all the classes in your hierarchy. **TypeInfo** must define an **operator<** because a **map** needs it to order its keys.

Although powerful, the definition for **TypeMap** is simple; the **addTrash( )** member function does most of the work. When you add a new **Trash** pointer, the a **TypeInfo<Trash>** object for that type is generated. This is used as a key to determine whether a **vector** holding objects of that type is already present in the **map**. If so, the **Trash** pointer is added to that **vector**. If not, the **TypeInfo** object and a new **vector** are added as a key-value pair.

An iterator to the map, when dereferenced, produces a **pair** object where the key (**TypeInfo**) is the **first** member, and the value (**Vector<Trash*>**) is the **second** member. And that's all there is to it.

The **TypeMap** takes advantage of the design of **fillBin( )**, which doesn't just try to fill a **vector** but instead anything that implements the **Fillable** interface with its **addTrash( )** member function. Since **TypeMap** is multiply inherited from **Fillable**, it can be used as an argument to **fillBin( )** like this:

```
fillBin("Trash.dat", bin);
```

An interesting thing about this design is that even though it wasn't created to handle the sorting, **fillBin( )** is performing a sort every time it inserts a **Trash** pointer into **bin**. When the **Trash** is thrown into **bin** it's immediately sorted by **TypeMap**'s internal sorting mechanism. Stepping through the **TypeMap** and operating on each individual **vector** becomes a simple matter, and uses ordinary STL syntax.

As you can see, adding a new type to the system won't affect this code at all, nor the code in **TypeMap**. This is certainly the smallest solution to the problem, and arguably the most elegant as well. It does rely heavily on RTTI, but notice that each key-value pair in the **map** is looking for only one type. In addition, there's no way you can "forget" to add the proper code to this system when you add a new type, since there isn't any code you need to add, other than that which supports the prototyping process (and you'll find out right away if you forget that).

Summary

Coming up with a design such as **TrashVisitor.cpp** that contains a larger amount of code than the earlier designs can seem at first to be counterproductive. It pays to notice what you're trying to accomplish with various designs. Design patterns in general strive to *separate the things that change from the things that stay the same*. The "things that change" can refer to many different kinds of changes. Perhaps the change occurs because the program is placed into a new environment or because something in the current environment changes (this could be: "The user wants to add a new shape to the diagram currently on the screen"). Or, as in this case, the change could be the evolution of the code body. While previous versions of the trash-sorting example emphasized the addition of new *types* of **Trash** to the system, **TrashVisitor.cpp** allows you to easily add new *functionality* without disturbing the **Trash** hierarchy. There's more code in **TrashVisitor.cpp**, but adding new functionality to **Visitor** is cheap. If this is something that happens a lot, then it's worth the extra effort and code to make it happen more easily.

The discovery of the vector of change is no trivial matter; it's not something that an analyst can usually detect before the program sees its initial design. The necessary information will probably not appear until later phases in the project: sometimes only at the design or implementation phases do you discover a deeper or more subtle need in your system. In the case of adding new types (which was the focus of most of the "recycle" examples) you might realize that you need a particular inheritance hierarchy only when you are in the maintenance phase and you begin extending the system!

One of the most important things that you'll learn by studying design patterns seems to be an about-face from what has been promoted so far in this book. That is: "OOP is all about polymorphism." This statement can produce the "two-year-old with a hammer" syndrome (everything looks like a nail). Put another way, it's hard enough to "get" polymorphism, and once you do, you try to cast all your designs into that one particular mold.

What design patterns say is that OOP isn't just about polymorphism. It's about "separating the things that change from the things that stay the same." Polymorphism is an especially important way to do this, and it turns out to be helpful if the programming language directly supports polymorphism (so you don't have to wire it in yourself, which would tend to make it prohibitively expensive). But design patterns in general show *other* ways to accomplish the basic goal, and once your eyes have been opened to this you will begin to search for more creative designs.

Since the *Design Patterns* book came out and made such an impact, people have been searching for other patterns. You can expect to see more of these appear as time goes on. Here are some sites recommended by Jim Coplien, of C++ fame (*http://www.bell-labs.com/~cope*), who is one of the main proponents of the patterns movement:

http://st-www.cs.uiuc.edu/users/patterns
http://c2.com/cgi/wiki
http://c2.com/ppr
http://www.bell-labs.com/people/cope/Patterns/Process/index.html
http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns
http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic
http://www.cs.wustl.edu/~schmidt/patterns.html
http://www.espinc.com/patterns/overview.html

> Also note there has been a yearly conference on design patterns, called PLOP, that produces a published proceedings. The third one of these proceedings came out in late 1997 (all published by Addison-Wesley).

Exercises

1. Using **SingletonPattern.cpp** as a starting point, create a class that manages a fixed number of its own objects.
2. Create a minimal Observer-Observable design in two classes, without base classes and without the extra arguments in **Observer.h** and the member functions in **Observable.h**. Just create the bare minimum in the two classes, then demonstrate your design by creating one **Observable** and many **Observer**s, and cause the **Observable** to update the **Observer**s.
3. Change **InnerClassIdiom.cpp** so that **Outer** uses multiple inheritance instead of the inner class idiom.
4. Add a class **Plastic** to TrashVisitor.cpp.
5. Add a class **Plastic** to DynaTrash.cpp.
6. Explain how **AbstractFactory.cpp** demonstrates *Double Dispatching* and the *Factory Method*.
7. Create a business-modeling environment with three types of **Inhabitant**: **Dwarf** (for engineers), **Elf** (for marketers) and **Troll** (for managers). Now create a class called **Project** that creates the different inhabitants and causes them to **interact( )** with each other using multiple dispatching.

8. Modify the above example to make the interactions more detailed. Each **Inhabitant** can randomly produce a **Weapon** using **getWeapon( )**: a **Dwarf** uses **Jargon** or **Play**, an **Elf** uses **InventFeature** or **SellImaginaryProduct**, and a **Troll** uses **Edict** and **Schedule**. You must decide which weapons "win" and "lose" in each interaction (as in **PaperScissorsRock.cpp**). Add a **battle( )** member function to **Project** that takes two **Inhabitant**s and matches them against each other. Now create a **meeting( )** member function for **Project** that creates groups of **Dwarf**, **Elf** and **Manager** and battles the groups against each other until only members of one group are left standing. These are the "winners."

26: Tools & topics

Tools created & used during the development of this book and various other handy things

The code extractor

The code for this book is automatically extracted directly from the ASCII text version of this book. The book is normally maintained in a word processor capable of producing camera-ready copy, automatically creating the table of contents and index, etc. To generate the code files, the book is saved into a plain ASCII text file, and the program in this section automatically extracts all the code files, places them in appropriate subdirectories, and generates all the makefiles. The entire contents of the book can then be built, for each compiler, by invoking a single **make** command. This way, the code listings in the book can be regularly tested and verified, and in addition various compilers can be tested for some degree of compliance with Standard C++ (the degree to which all the examples in the book can exercise a particular compiler, which is not too bad).

The code in this book is designed to be as generic as possible, but it is only tested under two operating systems: 32-bit Windows and Linux (using the Gnu C++ compiler **g++**, which means it should compile under other versions of Unix without too much trouble). You can easily get the latest sources for the book onto your machine by going to the web site **www.BruceEckel.com** and downloading the zipped archive containing all the code files and makefiles. If you unzip this you'll have the book's directory tree available. However, it may not be configured for your particular compiler or operating system. In this case, you can generate your own using the ASCII text file for the book (available at **www.BruceEckel.com**) and the **ExtractCode.cpp** program in this

section. Using a text editor, you find the **CompileDB.txt** file inside the ASCII text file for the book, edit it (leaving it the book's text file) to adapt it to your compiler and operating system, and then hand it to the ExtractCode program to generate your own source tree and makefiles.

You've seen that each file to be extracted contains a starting marker (which includes the file name and path) and an ending marker. Files can be of any type, and if the colon after the comment is directly followed by a '!' then the starting and ending marker lines are not reproduced in the generated file. In addition, you've seen the other markers **{O}**, **{L}**, and **{T}** that have been placed inside comments; these are used to generate the makefile for each subdirectory.

If there's a mistake in the input file, then the program must report the error, which is the **error( )** function at the beginning of the program. In addition, directory manipulation is not supported by the standard libraries, so this is hidden away in the class **OSDirControl**. If you discover that this class will not compile on your system, you must replace the non-portable function calls in **OSDirControl** with eqivalent calls from your library.

Although this program is very useful for distributing the code in the book, you'll see that it's also a useful example in its own right, since it partitions everything into sensible objects and also makes heavy use of the STL and the standard **string** class. You may note that one or two pieces of code might be duplicated from other parts of the book, and you might observe that some of the tools created within the program might have been broken out into their own reusable header files and **cpp** files. However, for easy unpacking of the book's source code it made more sense to keep everything lumped together in a single file.

```
//: C26:ExtractCode.cpp
// Automatically extracts code files from
// ASCII text of this book.
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <algorithm>
using namespace std;

string copyright =
  "// From Thinking in C++, 2nd Edition\n"
```

```
  "// Available at http://www.BruceEckel.com\n"
  "// (c) Bruce Eckel 1999\n"
  "// Copyright notice in Copyright.txt\n";

string usage =
  " Usage:ExtractCode source\n"
  "where source is the ASCII file containing \n"
  "the embedded tagged sourcefiles. The ASCII \n"
  "file must also contain an embedded compiler\n"
  "configuration file called CompileDB.txt \n"
  "See Thinking in C++, 2nd ed. for details\n";

// Tool to remove the white space from both ends:
string trim(const string& s) {
  if(s.length() == 0)
    return s;
  int b = s.find_first_not_of(" \t");
  int e = s.find_last_not_of(" \t");
  if(b == -1) // No non-spaces
    return "";
  return string(s, b, e - b + 1);
}

// Manage all the error messaging:
void error(string problem, string message) {
  static const string border(
  "----------------------------------------\n");
  class ErrReport {
    int count;
    string fname;
  public:
    ofstream errs;
    ErrReport(char* fn = "ExtractCodeErrors.txt")
      : count(0),fname(fn),errs(fname.c_str()) {}
    void operator++(int) { count++; }
    ~ErrReport() {
      errs.close();
      // Dump error messages to console
      ifstream in(fname.c_str());
      cerr << in.rdbuf() << endl;
      cerr << count << " Errors found" << endl;
      cerr << "Messages in " << fname << endl;
```

```
    }
  };
  // Created on first call to this function;
  // Destructor reports total errors:
  static ErrReport report;
  report++;
  report.errs << border << message << endl
    << "Problem spot: " << problem << endl;
}

///// OS-specific code, hidden inside a class:
#ifdef __GNUC__  // For egcs under Linux/Unix
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>
class OSDirControl {
public:
  static string getCurrentDir() {
    char path[PATH_MAX];
    getcwd(path, PATH_MAX);
    return string(path);
  }
  static void makeDir(string dir) {
    mkdir(dir.c_str(), 0777);
  }
  static void changeDir(string dir) {
    chdir(dir.c_str());
  }
};
#else // For Dos/Windows:
#include <direct.h>
class OSDirControl {
public:
  static string getCurrentDir() {
    char path[_MAX_PATH];
    getcwd(path, _MAX_PATH);
    return string(path);
  }
  static void makeDir(string dir) {
    mkdir(dir.c_str());
  }
  static void changeDir(string dir) {
```

```
    chdir(dir.c_str());
  }
};
#endif ///// End of OS-specific code

class PushDirectory {
  string oldpath;
public:
  PushDirectory(string path);
  ~PushDirectory() {
    OSDirControl::changeDir(oldpath);
  }
  void pushOneDir(string dir) {
    OSDirControl::makeDir(dir);
    OSDirControl::changeDir(dir);
  }
};

PushDirectory::PushDirectory(string path) {
  oldpath = OSDirControl::getCurrentDir();
  while(path.length() != 0) {
    int colon = path.find(':');
    if(colon != string::npos) {
      pushOneDir(path.substr(0, colon));
      path = path.substr(colon + 1);
    } else {
      pushOneDir(path);
      return;
    }
  }
}

//---------------- Manage code files -------------
// A CodeFile object knows everything about a
// particular code file, including contents, path
// information, how to compile, link, and test
// it, and which compilers it won't compile with.
enum TType {header, object, executable, none};

class CodeFile {
  TType _targetType;
  string _rawName, // Original name from input
```

```
    _path, // Where the source file lives
    _file, // Name of the source file
    _base, // Name without extension
    _tname, // Target name
    _testArgs; // Command-line arguments
  vector<string>
    lines, // Contains the file
    _compile, // Compile dependencies
    _link; // How to link the executable
  set<string>
    _noBuild; // Compilers it won't compile with
  bool writeTags; // Whether to write the markers
  // Initial makefile processing for the file:
  void target(const string& s);
  // For quoted #include headers:
  void headerLine(const string& s);
  // For special dependency tag marks:
  void dependLine(const string& s);
public:
  CodeFile(istream& in, string& s);
  const string& rawName() { return _rawName; }
  const string& path() { return _path; }
  const string& file() { return _file; }
  const string& base() { return _base; }
  const string& targetName() { return _tname; }
  TType targetType() { return _targetType; }
  const vector<string>& compile() {
    return _compile;
  }
  const vector<string>& link() {
    return _link;
  }
  const set<string>& noBuild() {
    return _noBuild;
  }
  const string& testArgs() { return _testArgs; }
  // Add a compiler it won't compile with:
  void addFailure(const string& failure) {
    _noBuild.insert(failure);
  }
  bool compilesOK(string compiler) {
    return _noBuild.count(compiler) == 0;
```

```
  }
  friend ostream&
  operator<<(ostream& os, const CodeFile& cf) {
    copy(cf.lines.begin(), cf.lines.end(),
      ostream_iterator<string>(os, ""));
    return os;
  }
  void write() {
    PushDirectory pd(_path);
    ofstream listing(_file.c_str());
    listing << *this;  // Write the file
  }
  void dumpInfo(ostream& os);
};

void CodeFile::target(const string& s) {
  // Find the base name of the file (without
  // the extension):
  int lastDot = _file.find_last_of('.');
  if(lastDot == string::npos) {
    error(s, "Missing extension");
    exit(1);
  }
  _base = _file.substr(0, lastDot);
  // Determine the type of file and target:
  if(s.find(".h") != string::npos ||
     s.find(".H") != string::npos) {
    _targetType = header;
    _tname = _file;
    return;
  }
  if(s.find(".txt") != string::npos
      || s.find(".TXT") != string::npos
      || s.find(".dat") != string::npos
      || s.find(".DAT") != string::npos) {
    // Text file, not involved in make
    _targetType = none;
    _tname = _file;
    return;
  }
  // C++ objs/exes depend on their own source:
  _compile.push_back(_file);
```

```
  if(s.find("{O}") != string::npos) {
    // Don't build an executable from this file
    _targetType = object;
    _tname = _base;
  } else {
    _targetType = executable;
    _tname = _base;
    // The exe depends on its own object file:
    _link.push_back(_base);
  }
}

void CodeFile::headerLine(const string& s) {
  int start = s.find('\"');
  int end = s.find('\"', start + 1);
  int len = end - start - 1;
  _compile.push_back(s.substr(start + 1, len));
}

void CodeFile::dependLine(const string& s) {
  const string linktag("//{L} ");
  string deps = trim(s.substr(linktag.length()));
  while(true) {
    int end = deps.find(' ');
    string dep = deps.substr(0, end);
    _link.push_back(dep);
    if(end == string::npos) // Last one
      break;
    else
      deps = trim(deps.substr(end));
  }
}

CodeFile::CodeFile(istream& in, string& s) {
  // If false, don't write begin & end tags:
  writeTags = (s[3] != '!');
  // Assume a space after the starting tag:
  _file = s.substr(s.find(' ') + 1);
  // There will always be at least one colon:
  int lastColon = _file.find_last_of(':');
  if(lastColon == string::npos) {
    error(s, "Missing path");
```

```
    lastColon = 0; // Recover from error
  }
  _rawName = trim(_file);
  _path = _file.substr(0, lastColon);
  _file = _file.substr(lastColon + 1);
  _file =_file.substr(0,_file.find_last_of(' '));
  cout << "path = [" << _path << "] "
    << "file = [" << _file << "]" << endl;
  target(s); // Determine target type
  if(writeTags){
    lines.push_back(s + '\n');
    lines.push_back(copyright);
  }
  string s2;
  while(getline(in, s2)) {
    // Look for specified link dependencies:
    if(s2.find("//{L}") == 0) // 0: Start of line
      dependLine(s2);
    // Look for command-line arguments for test:
    if(s2.find("//{T}") == 0) // 0: Start of line
      _testArgs = s2.substr(strlen("//{T}") + 1);
    // Look for quoted includes:
    if(s2.find("#include \"") != string::npos) {
      headerLine(s2); // Grab makefile info
    }
    // Look for end marker:
    if(s2.find("//" "/:~") != string::npos) {
      if(writeTags)
        lines.push_back(s2 + '\n');
      return;  // Found the end
    }
    // Make sure you don't see another start:
    if(s2.find("//" ":") != string::npos
       || s2.find("/*" ":") != string::npos) {
      error(s, "Error: new file started before"
        " previous file concluded");
      return;
    }
    // Write ordinary line:
    lines.push_back(s2 + '\n');
  }
}
```

```
void CodeFile::dumpInfo(ostream& os) {
  os << _path << ':' << _file << endl;
  os << "target: " << _tname << endl;
  os << "compile: " << endl;
  for(int i = 0; i < _compile.size(); i++)
    os << '\t' << _compile[i] << endl;
  os << "link: " << endl;
  for(int i = 0; i < _link.size(); i++)
    os << '\t' << _link[i] << endl;
  if(_noBuild.size() != 0) {
    os << "Won't build with: " << endl;
    copy(_noBuild.begin(), _noBuild.end(),
      ostream_iterator<string>(os, "\n"));
  }
}

//--------- Manage compiler information ---------
class CompilerData {
  // Information about each compiler:
  vector<string> rules; // Makefile rules
  set<string> fails; // Non-compiling files
  string objExtension; // File name extensions
  string exeExtension;
  // For OS-specific activities:
  bool _dos, _unix;
  // Store the information for all the compilers:
  static map<string, CompilerData> compilerInfo;
  static set<string> _compilerNames;
public:
  CompilerData() : _dos(false), _unix(false) {}
  // Read database of various compiler's
  // information and failure listings for
  // compiling the book files:
  static void readDB(istream& in);
  // For enumerating all the compiler names:
  static set<string>& compilerNames() {
    return _compilerNames;
  }
  // Tell this CodeFile which compilers
  // don't work with it:
  static void addFailures(CodeFile& cf);
```

```
  // Produce the proper object file name
  // extension for this compiler:
  static string obj(string compiler);
  // Produce the proper executable file name
  // extension for this compiler:
  static string exe(string compiler);
  // For inserting a particular compiler's
  // rules into a makefile:
  static void
  writeRules(string compiler, ostream& os);
  // Change forward slashes to backward
  // slashes if necessary:
  static string
  adjustPath(string compiler, string path);
  // So you can ask if it's a Unix compiler:
  static bool isUnix(string compiler) {
    return compilerInfo[compiler]._unix;
  }
  // So you can ask if it's a dos compiler:
  static bool isDos(string compiler) {
    return compilerInfo[compiler]._dos;
  }
  // Display information (for debugging):
  static void dump(ostream& os = cout);
};

// Static initialization:
map<string,CompilerData>
  CompilerData::compilerInfo;
set<string> CompilerData::_compilerNames;

void CompilerData::readDB(istream& in) {
  string compiler; // Name of current compiler
  string s;
  while(getline(in, s)) {
    if(s.find("#//" "/:~") == 0)
      return; // Found end tag
    s = trim(s);
    if(s.length() == 0) continue; // Blank line
    if(s[0] == '#') continue; // Comment
    if(s[0] == '{') { // Different compiler
      compiler = s.substr(0, s.find('}'));
```

```
      compiler = trim(compiler.substr(1));
      if(compiler.length() != 0)
        _compilerNames.insert(compiler);
      continue; // Changed compiler name
    }
    if(s[0] == '(') { // Object file extension
      string obj = s.substr(1);
      obj = trim(obj.substr(0, obj.find(')')));
      compilerInfo[compiler].objExtension =obj;
      continue;
    }
    if(s[0] == '[') { // Executable extension
      string exe = s.substr(1);
      exe = trim(exe.substr(0, exe.find(']')));
      compilerInfo[compiler].exeExtension =exe;
      continue;
    }
    if(s[0] == '&') { // Special directive
      if(s.find("dos") != string::npos)
        compilerInfo[compiler]._dos = true;
      else if(s.find("unix") != string::npos)
        compilerInfo[compiler]._unix = true;
      else
        error("Compiler Information Database",
          "unknown special directive: " + s);
      continue;
    }
    if(s[0] == '@') { // Makefile rule
      string rule(s.substr(1)); // Remove the @
      if(rule[0] == ' ') // Space means tab
        rule = '\t' + trim(rule);
      compilerInfo[compiler].rules
        .push_back(rule);
      continue;
    }
    // Otherwise, it's a failure line:
    compilerInfo[compiler].fails.insert(s);
  }
  error("CompileDB.txt","Missing end tag");
}

void CompilerData::addFailures(CodeFile& cf) {
```

```
  set<string>::iterator it =
    _compilerNames.begin();
  while(it != _compilerNames.end()) {
    if(compilerInfo[*it]
       .fails.count(cf.rawName()) != 0)
      cf.addFailure(*it);
    it++;
  }
}

string CompilerData::obj(string compiler) {
  if(compilerInfo.count(compiler) != 0) {
    string ext(
      compilerInfo[compiler].objExtension);
    if(ext.length() != 0)
      ext = '.' + ext; // Use '.' if it exists
    return ext;
  } else
    return "No such compiler information";
}

string CompilerData::exe(string compiler) {
  if(compilerInfo.count(compiler) != 0) {
    string ext(
      compilerInfo[compiler].exeExtension);
    if(ext.length() != 0)
      ext = '.' + ext; // Use '.' if it exists
    return ext;
  } else
    return "No such compiler information";
}

void CompilerData::writeRules(
  string compiler, ostream& os) {
  if(_compilerNames.count(compiler) == 0) {
    os << "No info on this compiler" << endl;
    return;
  }
  vector<string>& r =
    compilerInfo[compiler].rules;
  copy(r.begin(), r.end(),
    ostream_iterator<string>(os, "\n"));
```

```
}

string CompilerData::adjustPath(
  string compiler, string path) {
  // Use STL replace() algorithm:
  if(compilerInfo[compiler]._dos)
    replace(path.begin(), path.end(), '/', '\\');
  return path;
}

void CompilerData::dump(ostream& os) {
  ostream_iterator<string> out(os, "\n");
  *out++ = "Compiler Names:";
  copy(_compilerNames.begin(),
    _compilerNames.end(), out);
  map<string, CompilerData>::iterator compIt;
  for(compIt = compilerInfo.begin();
    compIt != compilerInfo.end(); compIt++) {
    os << "******************************\n";
    os << "Compiler: [" << (*compIt).first <<
      "]" << endl;
    CompilerData& cd = (*compIt).second;
    os << "objExtension: " << cd.objExtension
      << "\nexeExtension: " << cd.exeExtension
      << endl;
    *out++ = "Rules:";
    copy(cd.rules.begin(), cd.rules.end(), out);
    cout << "Won't compile with: " << endl;
    copy(cd.fails.begin(), cd.fails.end(), out);
  }
}

// ---------- Manage makefile creation ----------
// Create the makefile for this directory, based
// on each of the CodeFile entries:
class Makefile {
  vector<CodeFile> codeFiles;
  // All the different paths
  // (for creating the Master makefile):
  static set<string> paths;
  void
  createMakefile(string compiler, string path);
```

```
public:
  Makefile() {}
  void addEntry(CodeFile& cf) {
    paths.insert(cf.path()); // Record all paths
    // Tell it what compilers don't work with it:
    CompilerData::addFailures(cf);
    codeFiles.push_back(cf);
  }
  // Write the makefile for each compiler:
  void writeMakefiles(string path);
  // Create the master makefile:
  static void writeMaster(string flag = "");
};

// Static initialization:
set<string> Makefile::paths;

void Makefile::writeMakefiles(string path) {
  if(trim(path).length() == 0)
    return; // No makefiles in root directory
  PushDirectory pd(path);
  set<string>& compilers =
    CompilerData::compilerNames();
  set<string>::iterator it = compilers.begin();
  while(it != compilers.end())
    createMakefile(*it++, path);
}

void Makefile::createMakefile(
  string compiler, string path) {
  string // File name extensions:
    exe(CompilerData::exe(compiler)),
    obj(CompilerData::obj(compiler));
  string filename(compiler + ".makefile");
  ofstream makefile(filename.c_str());
  makefile <<
    "# From Thinking in C++, 2nd Edition\n"
    "# At http://www.BruceEckel.com\n"
    "# (c) Bruce Eckel 1999\n"
    "# Copyright notice in Copyright.txt\n"
    "# Automatically-generated MAKEFILE \n"
    "# For examples in directory "+ path + "\n"
```

```
  "# using the " + compiler + " compiler\n"
  "# Note: does not make files that will \n"
  "# not compile with this compiler\n"
  "# Invoke with: make -f "
  + compiler + ".makefile\n"
  << endl;
CompilerData::writeRules(compiler, makefile);
vector<string> makeAll, makeTest,
  makeBugs, makeDeps, linkCmd;
// Write the "all" dependencies:
makeAll.push_back("all: ");
makeTest.push_back("test: all ");
makeBugs.push_back("bugs: ");
string line;
vector<CodeFile>::iterator it;
for(it = codeFiles.begin();
  it != codeFiles.end(); it++) {
  CodeFile& cf = *it;
  if(cf.targetType() == executable) {
    line = "\\\n\t"+cf.targetName()+ exe + ' ';
    if(cf.compilesOK(compiler) == false) {
      makeBugs.push_back(
        CompilerData::adjustPath(
          compiler,line));
    } else {
      makeAll.push_back(
        CompilerData::adjustPath(
          compiler,line));
      line = "\\\n\t" + cf.targetName() + exe +
        ' ' + cf.testArgs() + ' ';
      makeTest.push_back(
        CompilerData::adjustPath(
          compiler,line));
    }
    // Create the link command:
    int linkdeps = cf.link().size();
    string linklist;
    for(int i = 0; i < linkdeps; i++)
      linklist +=
        cf.link().operator[](i) + obj + " ";
    line = cf.targetName() + exe + ": "
      + linklist + "\n\t$(CPP) $(OFLAG)"
```

```
      + cf.targetName() + exe
      + ' ' + linklist + "\n\n";
    linkCmd.push_back(
      CompilerData::adjustPath(compiler,line));
  }
  // Create dependencies
  if(cf.targetType() == executable
    || cf.targetType() == object) {
    int compiledeps = cf.compile().size();
    string objlist(cf.base() + obj + ": ");
    for(int i = 0; i < compiledeps; i++)
      objlist +=
        cf.compile().operator[](i) + " ";
    makeDeps.push_back(
      CompilerData::adjustPath(
        compiler, objlist) +"\n");
  }
}
ostream_iterator<string> mkos(makefile, "");
*mkos++ = "\n";
// The "all" target:
copy(makeAll.begin(), makeAll.end(), mkos);
*mkos++ = "\n\n";
// Remove continuation marks from makeTest:
vector<string>::iterator si = makeTest.begin();
int bsl;
for(; si != makeTest.end(); si++)
  if((bsl= (*si).find("\\\n")) != string::npos)
    (*si).erase(bsl, strlen("\\"));
// Now print the "test" target:
copy(makeTest.begin(), makeTest.end(), mkos);
*mkos++ = "\n\n";
// The "bugs" target:
copy(makeBugs.begin(), makeBugs.end(), mkos);
if(makeBugs.size() == 1)
  *mkos++ = "\n\t@echo No compiler bugs in "
    "this directory!";
*mkos++ = "\n\n";
// Link commands:
copy(linkCmd.begin(), linkCmd.end(), mkos);
*mkos++ = "\n";
// Demendencies:
```

```
  copy(makeDeps.begin(), makeDeps.end(), mkos);
  *mkos++ = "\n";
}

void Makefile::writeMaster(string flag) {
  string filename = "makefile";
  if(flag.length() != 0)
    filename += '.' + flag;
  ofstream makefile(filename.c_str());
  makefile << "# Master makefile for "
    "Thinking in C++, 2nd Ed. by Bruce Eckel\n"
    "# at http://www.BruceEckel.com\n"
    "# Compiles all the code in the book\n"
    "# Copyright notice in Copyright.txt\n\n"
    "help: \n"
    "\t@echo To compile all programs from \n"
    "\t@echo Thinking in C++, 2nd Ed., type\n"
    "\t@echo one of the following commands,\n"
    "\t@echo according to your compiler:\n";
  set<string>& n = CompilerData::compilerNames();
  set<string>::iterator nit;
  for(nit = n.begin(); nit != n.end(); nit++)
    makefile <<
      string("\t@echo make " + *nit + "\n");
  makefile << endl;
  // Make for each compiler:
  for(nit = n.begin(); nit != n.end(); nit++) {
    makefile << *nit << ":\n";
    for(set<string>::iterator it = paths.begin();
      it != paths.end(); it++) {
      // Ignore the root directory:
      if((*it).length() == 0) continue;
      makefile << "\tcd " << *it;
      // Different commands for unix vs. dos:
      if(CompilerData::isUnix(*nit))
        makefile << "; ";
      else
        makefile << "\n\t";
      makefile << "make -f " << *nit
        << ".makefile";
      if(flag.length() != 0) {
        makefile << ' ';
```

```
      if(flag == "bugs")
        makefile << "-i ";
      makefile << flag;
     }
     makefile << "\n";
     if(CompilerData::isUnix(*nit) == false)
      makefile << "\tcd ..\n";
    }
    makefile << endl;
  }
}

int main(int argc, char* argv[]) {
  if(argc < 2) {
    error("Command line error", usage);
    exit(1);
  }
  // For development & testing, leave off notice:
  if(argc == 3)
    if(string(argv[2]) == "-nocopyright")
      copyright = "";
  // Open the input file to read the compiler
  // information database:
  ifstream in(argv[1]);
  if(!in) {
    error(string("can't open ") + argv[1],usage);
    exit(1);
  }
  string s;
  while(getline(in, s)) {
    // Break up the strings to prevent a match when
    // this code is seen by this program:
    if(s.find("#:" " :CompileDB.txt")
      != string::npos) {
      // Parse the compiler information database:
      CompilerData::readDB(in);
      break; // Out of while loop
    }
  }
  if(in.eof())
    error("CompileDB.txt", "Can't find data");
  in.seekg(0, ios::beg); // Back to beginning
```

```
  map<string, Makefile> makeFiles;
  while(getline(in, s)) {
    // Look for tag at beginning of line:
    if(s.find("//" ":") == 0
       || s.find("/*" ":") == 0
       || s.find("#" ":") == 0) {
      CodeFile cf(in, s);
      cf.write();  // Tell it to write itself
      makeFiles[cf.path()].addEntry(cf);
    }
  }
  // Write all the makefiles, telling each
  // the path where it belongs:
  map<string, Makefile>::iterator mfi;
  for(mfi = makeFiles.begin();
    mfi != makeFiles.end(); mfi++)
    (*mfi).second.writeMakefiles((*mfi).first);
  // Create the master makefile:
  Makefile::writeMaster();
  // Write the makefile that tries the bug files:
  Makefile::writeMaster("bugs");
} ///:~
```

The first tool you see is **trim( )**, which was lifted from the **string**s chapter earlier in the book. It removes the whitespace from both ends of a **string** object. This is followed by the **usage** string which is printed whenever something goes wrong with the program.

The **error( )** function is global because it uses a trick with static members of functions. **error( )** is designed so that if it is never called, no error reporting occurs, but if it is called one or more times then an error file is created and the total number of errors is reported at the end of the program execution. This is accomplished by creating a nested class **ErrReport** and making a **static ErrReport** object inside **error( )**. That way, an **ErrReport** object is only created the first time **error( )** is called, so if **error( )** is never called no error reporting will occur. **ErrReport** creates an **ofstream** to write the errors to, and the **ErrReport** destructor closes the **ofstream**, then re-opens it and dumps it to **cerr**. This way, if the error report is too long and scrolls off the screen, you can use an editor to look at it. The count of the number of errors is held in **ErrReport**, and this is also reported upon program termination.

The job of a **PushDirectory** object is to capture the current directory, then created and move down each directory in the path (the path can be

arbitrarily long). Each subdirectory in the file's path description is separated by a '**:**' and the **mkdir( )** and **chdir( )** (or the equivalent on your system) are used to move into only one directory at a time, so the actual character that's used to separate directory paths is safely ignored. The destructor returns the path to the one that was captured before all the creating and moving took place.

Unfortunately, there are no functions in Standard C or Standard C++ to control directory creation and movement, so this is captured in the class **OSDirControl**. After reading the design patterns chapter, your first impulse might be to use the full "Bridge" pattern. However, there's a lot more going on here. Bridge generally works with things that are already classes, and here we are actually creating the class to encapsulating operating system directory control. In addition, this requires **#ifdef**s and **#include**s for each different operating system and compiler. However, the basic idea is that of a Bridge, since the rest of the code (**PushDirectory** is actually the only thing that uses this, and thus it acts as the Bridge abstraction) treats an **OsDirControl** object as a standard interface.

All the information about a particular source code file is encapsulated in a **CodeFile** object. This includes the type of target the file should produce, variations on the name of the file including the name of the target file it's supposed to produce. The entire contents of the file is contained in the **vector<string> lines**. In addition, the file's dependencies (the files which, if they change, should cause a recompilation of the current file) and the files on the linker command line are also **vector<string>** objects. The **CodeFile** object keeps all the compilers it won't work with in **_noBuild**, which is a **set<string>** because it's easier to look up an element in a **set**. The **writeTags** flag indicates whether the beginning and ending markers from the book listing should actually be output to the generated file.

The three private helper functions **target( )**, **headerLine( )** and **dependLine( )** are used by the **CodeFile** constructor while it is parsing the input stream. In fact, the **CodeFile** constructor does much of the work and most of the rest of the member functions simply return values that are stored in the **CodeFile** object. Exceptions to this are **addFailure( )** which stores a compiler that won't work, and **compilesOK( )** which, when given a compiler tells whether this file will compile successfully with that compiler. The **ostream operator<<** uses the STL **copy( )** algorithm and **write( )** uses **operator<<** to write the file into a particular directory and file name.

Looking at the implementation, you'll see that the helper functions **target( )**, **headerLine( )** and **dependLine( )** are just using **string** functions in order to search and manipulate the lines. The constructor is what initiates everything. The idea is that the main program opens the file and reads it until it sees the starting marker for a code file. At that point it makes a **CodeFile** object and hands the constructor the **istream** (so the constructor can read the rest of the code file) and the first line that was already read, since it contains valuable information. This first line is dissected for the file name information and the target type. The beginning of the file is written (source and copyright information is added) and the rest of the file is read, until the ending tag. The top few lines may contain information about link dependencies and command line arguments, or they may be files that are **#included** using quotes rather than angle brackets. Quotes indicate they are from local directories and should be added to the makefile dependency.

You'll notice that a number of the markers strings in this program are broken up into two adjacent character strings, relying on the preprocessor to concatenate those strings. This is to prevent them from causing the **ExtractCode** program from accidentally mistaking the strings embedded in the program with the end marker, when **ExtractCode** is extracting it's own source code.

The goal of **CompilerData** is to capture and make available all the information about particular compiler idiosyncrasies. At first glance, the **CompilerData** class appears to be a container of **static** member functions, a library of functions wrapped in a class. Actually, the class contains two **static** data members; the simpler one is a **set<string>** that holds all the compiler names, but **compilerInfo** is a **map** that maps **string** objects (the compiler name) to **CompilerData** objects. Each individual **CompilerData** object in **compilerInfo** contains a **vector<string>** which is the "rules" that are placed in the makefile (these rules are different for different compilers) and a **set<string>** which indicates the files that won't compile with this particular compiler. Also, each compiler creates different extensions for object files and executable files, and these are also stored. There are two flags which indicate if this is a "dos" or "unix" style environment (this causes differences in path information and command styles for the resulting makefiles).

The member function **readDB( )** is responsible for taking an **istream** and parsing it into a series of **CompilerData** objects which are stored in **compilerInfo**. By choosing a relatively simple format (which you can see in Appendix D) the parsing of this configuration file is fairly simple: the

first character on a line determines what information the line contains; a '**#**' sign is a comment, a '**{**' indicates that the next compiler configuration is beginning and this is the new compiler name, a '**(**' is used to establish the object file extension name, a '**&**' indicates the "dos" or "unix" directive, and '@' is a makefile rule which is placed verbatim at the beginning of the makefile. If there is no special character at the beginning of the line, the it must be a file that fails to compile.

The **addFailures( )** member function takes it's **CodeFile** argument (by reference, so it can modify the outside object) and checks each compiler to see if it works with that particular code file; if not, it adds that compiler to the **CodeFile** object's failure list.

Both **obj( )** and **exe( )** return the appropriate file extension for a particular compiler. Note that some situations don't expect extensions, and so the '**.**' is added only if there is an extension.

When the makefile is being created, one of the first things to do is add the various **make** rules, such as the prefixes and target rules (see Appendix D for examples). This is accomplished with **writeRules( )**. Note the use of the STL **copy( )** algorithm.

Although dos compilers have no trouble with forward slashes as part of the paths of **#include** files, most dos **make** programs expect backslashes as part of paths in dependency lists. To adjust for this, the **adjustPath( )** function checks to see if this is a dos compiler, and if so it uses the STL **replace( )** algorithm, treating the **path** string object as a container, to replace forward-slash characters with backward slashes.

The last class, **Makefile**, is used to create all the makefiles, including the master makefile that moves into each subdirectory and calls the other makefiles. Each **Makefile** contains a group of **CodeFile** objects, stored in a **vector**. You call **addEntry( )** to put a new **CodeFile** into the **Makefile**; this also adds the failure list to the **CodeFile**. In addition, there is a **static set<string>** which contains all the different paths where all the different makefiles will be written; this is used to build the master makefile so it can call all the makefiles in all the subdirectories. The **addEntry( )** function also updates this set of paths.

To write the makefile for a particular path (once the entire book file has been read), you call **writeMakefiles( )** and hand it the path you want it to write the makefile for. This function simply iterates through all the compilers in **compilers** and calls **createMakefile( )** for each one, passing it the compiler name and the path. The latter function is where the real work gets done. First the file name extensions are captured into

local **string** objects, then the file name is created from the name of the compiler with ".makefile" concatenated (you can use a file with a name other than "makefile" by using the **make -f** flag). After writing the header comments and the rules for that particular compiler/operating-system combination (remember, these rules come from the compiler configuration file), a **vector<string>** is created to hold all the different regions of the makefile: the master target list **makeAll**, the testing commands **makeTest**, the dependencies **makeDeps**, and the commands for linking into executables **linkCmd**. The reason it's necessary to have lists for these four regions is that each **CodeFile** object causes entries into each region, so the regions are built as the list of **CodeFile**s is traversed, and then finally each region is written in its proper order. This is the function which decides whether a file is going to be included, and also calls **adjustPath( )** to conditionally change forward slashes to backward slashes.

To write the master makefile in **writeMaster( )**, the initial comments are written. The default target is called "help," and it is used if you simply type **make**. This provides very simple help to the first time user, including the options for **make** that this makefile supports (that is, all the different compilers the makefile is set up for). Then it creates the list of commands for each compiler, which basically consists of: descending into a subdirectory, call **make** (recursively) on the appropriate makefile in that subdirectory, and then rising back up to the book's root subdirectory. Makefiles in unix and dos work very differently from each other in this situation: in unix, you **cd** to the directory, followed by a semicolon and then the command you want to execute – returning to the root directory happens automatically. While in dos, you must **cd** both down and then back up again, all on separate lines. So the **writeMaster( )** function must interrogate to see if a compiler is running under Unix and write different commands accordingly.

Because of the work done in designing the classes (and this was an iterative process; it didn't just pop out this way), **main( )** is quite straightforward to read. After opening the input file, the **getline( )** function is used to read each input line until the line containing **CompileDB.txt** is found; this indicates the beginning of the compiler database listing. Once that has been parsed, **seekg( )** is used to move the file pointer back to the beginning so all the code files can be extracted.

Each line is read and if one of the start markers is found in the line, a **CodeFile** object is created using that line (which has essential information) and the input stream. The constructor returns when it

finishes reading its file, and at that point you can turn around and call **write( )** for the code file, and it is automatically written to the correct spot (an earlier version of this program collected all the **CodeFile** objects first and put them in a container, then wrote one directory at a time, but the approach shown above has code that's easier to understand and the performance impact is not really significant for a tool like this.

For makefile management, a **map<string, Makefile>** is created, where the **string** is the path where the makefile exists. The nice thing about this approach is that the **Makefile** objects will be automatically created whenever you access a new path, as you can see in the line

```
makeFiles[cf.path()].addEntry(cf);
```

then to write all the makefiles you simply iterate through the **makeFiles map**.

Debugging

This section contains some tips and techniques which may help during debugging.

assert( )

The Standard C library **assert( )** macro is brief, to the point and portable. In addition, when you're finished debugging you can remove all the code by defining **NDEBUG**, either on the command-line or in code.

Also, **assert( )** can be used while roughing out the code. Later, the calls to **assert( )** that are actually providing information to the end user can be replaced with more civilized messages.

Trace macros

Sometimes it's very helpful to print the code of each statement before it is executed, either to **cout** or to a trace file. Here's a preprocessor macro to accomplish this:

```
#define TRACE(ARG) cout << #ARG << endl; ARG
```

Now you can go through and surround the statements you trace with this macro. Of course, it can introduce problems. For example, if you take the statement:

```
for(int i = 0; i < 100; i++)
  cout << i << endl;
```

And put both lines inside **TRACE( )** macros, you get this:

```
TRACE(for(int i = 0; i < 100; i++))
TRACE(  cout << i << endl;)
```

Which expands to this:

```
cout << "for(int i = 0; i < 100; i++)" << endl;
for(int i = 0; i < 100; i++)
  cout << "cout << i << endl;" << endl;
cout << i << endl;
```

Which isn't what you want. Thus, this technique must be used carefully.

A variation on the **TRACE( )** macro is this:

```
#define D(a) cout << #a "=[" << a << "]" << nl;
```

If there's an expression you want to display, you simply put it inside a call to **D( )** and the expression will be printed, followed by its value (assuming there's an overloaded operator **<<** for the result type). For example, you can say **D(a + b)**. Thus you can use it anytime you want to test an intermediate value to make sure things are OK.

Of course, the above two macros are actually just the two most fundamental things you do with a debugger: trace through the code execution and print values. A good debugger is an excellent productivity tool, but sometimes debuggers are not available, or it's not convenient to use them. The above techniques always work, regardless of the situation.

Trace file

This code allows you to easily create a trace file and send all the output that would normally go to **cout** into the file. All you have to do is **#define** TRACEON and include the header file (of course, it's fairly easy just to write the two key lines right into your file):

```
//: C26:Trace.h
// Creating a trace file
#ifndef TRACE_H
#define TRACE_H
#include <fstream>

#ifdef TRACEON
```

```
ofstream TRACEFILE__("TRACE.OUT");
#define cout TRACEFILE__
#endif

#endif // TRACE_H ///:~
```

Here's a simple test of the above file:

```
//: C26:Tracetst.cpp
// Test of trace.h
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

#define TRACEON
#include "Trace.h"

int main() {
  ifstream f("Tracetst.cpp");
  assure(f, "Tracetst.cpp");
  cout << f.rdbuf();
} ///:~
```

This also uses the **assure( )** function defined earlier in the book.

Abstract base class for debugging

In the Smalltalk tradition, you can create your own object-based hierarchy, and install pure virtual functions to perform debugging. Then everyone on the team must inherit from this class and redefine the debugging functions. All objects in the system will then have debugging functions available.

Tracking **new**/**delete** & **malloc**/**free**

Common problems with memory allocation include calling **delete** for things you have **malloc**ed, calling **free** for things you allocated with **new**, forgetting to release objects from the free store, and releasing them more

than once. This section provides a system to help you track these kinds of problems down.

To use the memory checking system, you simply link the **obj** file in and all the calls to **malloc( )**, **realloc( )**, **calloc( )**, **free( )**, **new** and **delete** are intercepted. However, if you also include the following file (which is optional), all the calls to **new** will store information about the file and line where they were called. This is accomplished with a use of the *placement syntax* for **operator new** (this trick was suggested by Reg Charney of the C++ Standards Committee). The placement syntax is intended for situations where you need to place objects at a specific point in memory. However, it allows you to create an **operator new** with any number of arguments. This is used to advantage here to store the results of the **__FILE__** and **__LINE__** macros whenever **new** is called:

```
//: C26:MemCheck.h
// Memory testing system
// This file is only included if you want to
// use the special placement syntax to find
// out the line number where "new" was called.
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <cstdlib>  // size_t

// Use placement syntax to pass extra arguments.
// From an idea by Reg Charney:
void* operator new(
  std::size_t sz, char* file, int line);
#define new new(__FILE__, __LINE__)

#endif // MEMCHECK_H ///:~
```

In the following file containing the function definitions, you will note that everything is done with standard IO rather than iostreams. This is because, for example, the **cout** constructor allocates memory. Standard IO ensures against cyclical conditions that can lock up the system.

```
//: C26:MemCheck.cpp {O}
// Memory allocation tester
#include <cstdlib>
#include <cstring>
#include <cstdio>
using namespace std;
// MemCheck.h must not be included here
```

```
// Output file object using cstdio
// (cout constructor calls malloc())
class OFile {
  FILE* f;
public:
  OFile(char* name) : f(fopen(name, "w")) {}
  ~OFile() { fclose(f); }
  operator FILE*() { return f; }
};
extern OFile memtrace;
// Comment out the following to send all the
// information to the trace file:
#define memtrace stdout

const unsigned long _pool_sz = 50000L;
static unsigned char _memory_pool[_pool_sz];
static unsigned char* _pool_ptr = _memory_pool;

void* getmem(size_t sz) {
  if(_memory_pool + _pool_sz - _pool_ptr < sz) {
    fprintf(stderr,
          "Out of memory. Use bigger model\n");
    exit(1);
  }
  void* p = _pool_ptr;
  _pool_ptr += sz;
  return p;
}

// Holds information about allocated pointers:
class MemBag {
public:
  enum type { Malloc, New };
private:
  char* typestr(type t) {
    switch(t) {
      case Malloc: return "malloc";
      case New: return "new";
      default: return "?unknown?";
    }
  }
```

```
  struct M {
    void* mp;  // Memory pointer
    type t;     // Allocation type
    char* file; // File name where allocated
    int line;  // Line number where allocated
    M(void* v, type tt, char* f, int l)
      : mp(v), t(tt), file(f), line(l) {}
  }* v;
  int sz, next;
  static const int increment = 50 ;
public:
  MemBag() : v(0), sz(0), next(0) {}
  void* add(void* p, type tt = Malloc,
          char* s = "library", int l = 0) {
    if(next >= sz) {
      sz += increment;
      // This memory is never freed, so it
      // doesn't "get involved" in the test:
      const int memsize = sz * sizeof(M);
      // Equivalent of realloc, no registration:
      void* p = getmem(memsize);
      if(v) memmove(p, v, memsize);
      v = (M*)p;
      memset(&v[next], 0,
           increment * sizeof(M));
    }
    v[next++] = M(p, tt, s, l);
    return p;
  }
  // Print information about allocation:
  void allocation(int i) {
    fprintf(memtrace, "pointer %p"
      " allocated with %s",
      v[i].mp, typestr(v[i].t));
    if(v[i].t == New)
      fprintf(memtrace, " at %s: %d",
        v[i].file, v[i].line);
    fprintf(memtrace, "\n");
  }
  void validate(void* p, type T = Malloc) {
    for(int i = 0; i < next; i++)
      if(v[i].mp == p) {
```

```
      if(v[i].t != T) {
        allocation(i);
        fprintf(memtrace,
        "\t was released as if it were "
        "allocated with %s \n", typestr(T));
      }
      v[i].mp = 0;  // Erase it
      return;
    }
   fprintf(memtrace,
   "pointer not in memory list: %p\n", p);
  }
  ~MemBag() {
   for(int i = 0; i < next; i++)
    if(v[i].mp != 0) {
      fprintf(memtrace,
      "pointer not released: ");
      allocation(i);
    }
  }
};
extern MemBag MEMBAG_;

void* malloc(size_t sz) {
  void* p = getmem(sz);
  return MEMBAG_.add(p, MemBag::Malloc);
}

void* calloc(size_t num_elems, size_t elem_sz) {
  void* p = getmem(num_elems * elem_sz);
  memset(p, 0, num_elems * elem_sz);
  return MEMBAG_.add(p, MemBag::Malloc);
}

void* realloc(void* block, size_t sz) {
  void* p = getmem(sz);
  if(block) memmove(p, block, sz);
  return MEMBAG_.add(p, MemBag::Malloc);
}

void free(void* v) {
  MEMBAG_.validate(v, MemBag::Malloc);
```

```
}

void* operator new(size_t sz) {
  void* p = getmem(sz);
  return MEMBAG_.add(p, MemBag::New);
}

void*
operator new(size_t sz, char* file, int line) {
  void* p = getmem(sz);
  return MEMBAG_.add(p, MemBag::New, file,line);
}

void operator delete(void* v) {
  MEMBAG_.validate(v, MemBag::New);
}

MemBag MEMBAG_;
// Placed here so the constructor is called
// AFTER that of MEMBAG_ :
#ifdef memtrace
#undef memtrace
#endif
OFile memtrace("memtrace.out");
// Causes 1 "pointer not in memory list" message
///:~
```

OFile is a simple wrapper around a **FILE***; the constructor opens the file and the destructor closes it. The **operator FILE*( )** allows you to simply use the **OFile** object anyplace you would ordinarily use a **FILE*** (in the **fprintf( )** statements in this example). The **#define** that follows simply sends everything to standard output, but if you need to put it in a trace file you simply comment out that line.

Memory is allocated from an array called **_memory_pool**. The **_pool_ptr** is moved forward every time storage is allocated. For simplicity, the storage is never reclaimed, and **realloc( )** doesn't try to resize the storage in the same place.

All the storage allocation functions call **getmem( )** which ensures there is enough space left and moves the **_pool_ptr** to allocate your storage. Then they store the pointer in a special container of class **MemBag** called **MEMBAG_**, along with pertinent information (notice the two versions of **operator new**; one which just stores the pointer and the other which

stores the file and line number). The **MemBag** class is the heart of the system.

You will see many similarities to **xbag** in **MemBag**. A distinct difference is **realloc( )** is replaced by a call to **getmem( )** and **memmove( )**, so that storage allocated for the **MemBag** is not registered. In addition, the **type enum** allows you to store the way the memory was allocated; the **typestr( )** function takes a type and produces a string for use with printing.

The nested **struct M** holds the pointer, the type, a pointer to the file name (which is assumed to be statically allocated) and the line where the allocation occurred. **v** is a pointer to an array of **M** objects – this is the array which is dynamically sized.

The **allocation( )** function prints out a different message depending on whether the storage was allocated with **new** (where it has line and file information) or **malloc( )** (where it doesn't). This function is used inside **validate( )**, which is called by **free( )** and **delete( )** to ensure everything is OK, and in the destructor, to ensure the pointer was cleaned up (note that in **validate( )** the pointer value **v[i].mp** is set to zero, to indicate it has been cleaned up).

The following is a simple test using the memcheck facility. The **MemCheck.obj** file must be linked in for it to work:

```
//: C26:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system
#include "MemCheck.h"

int main() {
  void* v = std::malloc(100);
  delete v;
  int* x = new int;
  std::free(x);
  new double;
} ///:~
```

The trace file created in **MemCheck.cpp** causes the generation of one "pointer not in memory list" message, apparently from the creation of the file pointer on the heap. [[This may not still be true – test it]]

CGI programming in C++

The World-Wide Web has become the common tongue of connectivity on planet earth. It began as simply a way to publish primitively-formatted documents in a way that everyone could read them regardless of the machine they were using. The documents are created in *hypertext markup language* (HTML) and placed on a central server machine where they are handed to anyone who asks. The documents are requested and read using a *web browser* that has been written or ported to each particular platform.

Very quickly, just reading a document was not enough and people wanted to be able to collect information from the clients, for example to take orders or allow database lookups from the server. Many different approaches to *client-side programming* have been tried such as Java applets, Javascript, and other scripting or programming languages. Unfortunately, whenever you publish something on the Internet you face the problem of a whole history of browsers, some of which may support the particular flavor of your client-side programming tool, and some which won't. The only reliable and well-established solution[73] to this problem is to use straight HTML (which has a very limited way to collect and submit information from the client) and *common gateway interface* (CGI) programs that are run on the server. The Web server takes an encoded request submitted via an HTML page and responds by invoking a CGI program and handing it the encoded data from the request. This request is classified as either a "GET" or a "POST" (the meaning of which will be explained later) and if you look at the URL window in your Web browser when you push a "submit" button on a page you'll often be able to see the encoded request and information.

CGI can seem a bit intimidating at first, but it turns out that it's just messy, and not all that difficult to write. (An innocent statement that's true of many things – *after* you understand them.) A CGI program is quite straightforward since it takes its input from environment variables and standard input, and sends its output to standard output. However, there is

[73] Actually, Java Servlets look like a much better solution than CGI, but – at least at this writing – Servlets are still an up-and-coming solution and you're unlikely to find them provided by your typical ISP.

some decoding that must be done in order to extract the data that's been sent to you from the client's web page. In this section you'll get a crash course in CGI programming, and we'll develop tools that will perform the decoding for the two different types of CGI submissions (GET and POST). These tools will allow you to easily write a CGI program to solve any problem. Since C++ exists on virtually all machines that have Web servers (and you can get GNU C++ free for virtually any platform), the solution presented here is quite portable.

Encoding data for CGI

To submit data to a CGI program, the HTML "form" tag is used. The following very simple HTML page contains a form that has one user-input field along with a "submit" button:

```
//:! C26:SimpleForm.html
<HTML><HEAD>
<TITLE>A simple HTML form</TITLE></HEAD>
Test, uses standard html GET
<Form method="GET" ACTION="/cgi-bin/CGI_GET.exe">
<P>Field1: <INPUT TYPE = "text" NAME = "Field1"
VALUE = "This is a test" size = "40"></p>
<p><input type = "submit" name = "submit" > </p>
</Form></HTML>
///:~
```

Everything between the **<Form** and the **</Form>** is part of this form (You can have multiple forms on a single page, but each one is controlled by its own method and submit button). The "method" can be either "get" or "post," and the "action" is what the server does when it receives the form data: it calls a program. Each form has a method, an action, and a submit button, and the rest of the form consists of input fields. The most commonly-used input field is shown here: a text field. However, you can also have things like check boxes, drop-down selection lists and radio buttons.

CGI_GET.exe is the name of the executable program that resides in the directory that's typically called "cgi-bin" on your Web server.[74] (If the named program is not in the cgi-bin directory, you won't see any results.)

[74] Free Web servers are relatively common and can be found by browsing the Internet; Apache, for example, is the most popular Web server on the Internet.

Many Web servers are Unix machines (mine runs Linux) that don't traditionally use the **.exe** extension for their executable programs, but you can call the program anything you want under Unix. By using the **.exe** extension the program can be tested without change under most operating systems.

If you fill out this form and press the "submit" button, in the URL address window of your browser you will see something like:

```
http://www.pooh.com/cgi-bin/CGI_GET.exe?Field1=
This+is+a+test&submit=Submit+Query
```

(Without the line break, of course.) Here you see a little bit of the way that data is encoded to send to CGI. For one thing, spaces are not allowed (since spaces typically separate command-line arguments). Spaces are replaced by '**+**' signs. In addition, each field contains the field name (which is determined by the form on the HTML page) followed by an '**=**' and the field data, and terminated by a '**&**'.

At this point, you might wonder about the '**+**', '**=,**' and '**&**'. What if those are used in the field, as in "John & Marsha Smith"? This is encoded to:

```
John+%26+Marsha+Smith
```

That is, the special character is turned into a '**%**' followed by its ASCII value in hex. Fortunately, the web browser automatically performs all encoding for you.

The CGI parser

There are many examples of CGI programs written using Standard C. One argument for doing this is that Standard C can be found virtually everywhere. However, C++ has become quite ubiquitous, especially in the form of the GNU C++ Compiler[75] (**g++**) that can be downloaded free from the Internet for virtually any platform (and often comes pre-installed with operating systems such as Linux). As you will see, this means that you can get the benefit of object-oriented programming in a CGI program.

[75] GNU stands for "Gnu's Not Unix." The project, created by the Free Software Foundation, was originally intended to replace the Unix operating system with a free version of that OS. Linux appears to have replaced this initiative, but the GNU tools have played an integral part in the development of Linux, which comes packaged with many GNU components.

Since what we're concerned with when parsing the CGI information is the field name-value pairs, one class (**CGIpair**) will be used to represent a single name-value pair and a second class (**CGImap**) will use **CGIpair** to parse each name-value pair that is submitted from the HTML form into keys and values that it will hold in a **map** of **strings** so you can easily fetch the value for each field at your leisure.

One of the reasons for using C++ here is the convenience of the STL, in particular the **map** class. Since **map** has the **operator[ ],** you have a nice syntax for extracting the data for each field. The **map** template will be used in the creation of **CGImap**, which you'll see is a fairly short definition considering how powerful it is.

The project will start with a reusable portion, which consists of **CGIpair** and **CGImap** in a header file. Normally you should avoid cramming this much code into a header file, but for these examples it's convenient and it doesn't hurt anything:

```
//: C26:CGImap.h
// Tools for extracting and decoding data from
// from CGI GETs and POSTs.
#include <string>
#include <vector>
#include <iostream>
using namespace std;

class CGIpair : public pair<string, string> {
public:
  CGIpair() {}
  CGIpair(string name, string value) {
    first = decodeURLString(name);
    second = decodeURLString(value);
  }
  // Automatic type conversion for boolean test:
  operator bool() const {
    return (first.length() != 0);
  }
private:
  static string decodeURLString(string URLstr) {
    const int len = URLstr.length();
    string result;
    for(int i = 0; i < len; i++) {
      if(URLstr[i] == '+')
        result += ' ';
```

```
      else if(URLstr[i] == '%') {
        result +=
          translateHex(URLstr[i + 1]) * 16 +
          translateHex(URLstr[i + 2]);
        i += 2; // Move past hex code
      } else // An ordinary character
        result += URLstr[i];
    }
    return result;
  }
  // Translate a single hex character; used by
  // decodeURLString():
  static char translateHex(char hex) {
    if(hex >= 'A')
      return (hex & 0xdf) - 'A' + 10;
    else
      return hex - '0';
  }
};

// Parses any CGI query and turns it into an
// STL vector of CGIpair which has an associative
// lookup operator[] like a map. A vector is used
// instead of a map because it keeps the original
// ordering of the fields in the Web page form.
class CGImap : public vector<CGIpair> {
  string gq;
  int index;
  // Prevent assignment and copy-construction:
  void operator=(CGImap&);
  CGImap(CGImap&);
public:
  CGImap(string query): index(0), gq(query){
    CGIpair p;
    while((p = nextPair()) != 0)
      push_back(p);
  }
  // Look something up, as if it were a map:
  string operator[](const string& key) {
    iterator i = begin();
    while(i != end()) {
      if((*i).first == key)
```

```
      return (*i).second;
    i++;
   }
   return string(); // Empty string == not found
  }
  void dump(ostream& o, string nl = "<br>") {
   for(iterator i = begin(); i != end(); i++) {
    o << (*i).first << " = "
      << (*i).second << nl;
   }
  }
private:
  // Produces name-value pairs from the query
  // string. Returns an empty Pair when there's
  // no more query string left:
  CGIpair nextPair() {
   if(gq.length() == 0)
    return CGIpair(); // Error, return empty
   if(gq.find('=') == -1)
    return CGIpair(); // Error, return empty
   string name = gq.substr(0, gq.find('='));
   gq = gq.substr(gq.find('=') + 1);
   string value = gq.substr(0, gq.find('&'));
   gq = gq.substr(gq.find('&') + 1);
   return CGIpair(name, value);
  }
};

// Helper class for getting POST data:
class Post : public string {
public:
  Post() {
   // For a CGI "POST," the server puts the
   // length of the content string in the
   // environment variable CONTENT_LENGTH:
   char* clen = getenv("CONTENT_LENGTH");
   if(clen == 0) {
    cout << "Zero CONTENT_LENGTH, Make sure "
      "this is a POST and not a GET" << endl;
    return;
   }
   int len = atoi(clen);
```

```
    char* s = new char[len];
    cin.read(s, len); // Get the data
    append(s, len); // Add it to this string
    delete []s;
  }
}; ///:~
```

The **CGIpair** class starts out quite simply: it inherits from the standard library **pair** template to create a **pair** of **string**s, one for the name and one for the value. The second constructor calls the member function **decodeURLString( )** which produces a **string** after stripping away all the extra characters added by the browser as it submitted the CGI request. There is no need to provide functions to select each individual element – because **pair** is inherited publicly, you can just select the **first** and **second** elements of the **CGIpair**.

The **operator bool** provides automatic type conversion to **bool**. If you have a **CGIpair** object called **p** and you use it in an expression where a Boolean result is expected, such as

```
if(p) { //...
```

then the compiler will recognize that it has a **CGIpair** and it needs a Boolean, so it will automatically call **operator bool** to perform the necessary conversion.

Because the **string** objects take care of themselves, you don't need to explicitly define the copy-constructor, **operator=** or destructor – the default versions synthesized by the compiler do the right thing.

The remainder of the **CGIpair** class consists of the two methods **decodeURLString( )** and a helper member function **translateHex( )** which is used by **decodeURLString( )**. (Note that **translateHex( )** does not guard against bad input such as "%1H.") **decodeURLString( )** moves through and replaces each '**+**' with a space, and each hex code (beginning with a '**%**') with the appropriate character. It's worth noting here and in **CGImap** the power of the **string** class – you can index into a **string** object using **operator[ ]**, and you can use methods like **find( )** and **substring( )**.

CGImap parses and holds all the name-value pairs submitted from the form as part of a CGI request. You might think that anything that has the word "map" in it's name should be inherited from the STL **map**, but **map** has it's own way of ordering the elements it stores whereas here it's useful to keep the elements in the order that they appear on the Web

page. So **CGImap** is inherited from **vector<CGIpair>**, and **operator[ ]** is overloaded so you get the associative-array lookup of a **map**.

You can also see that **CGImap** has a copy-constructor and an **operator=**, but they're both declared as **private**. This is to prevent the compiler from synthesizing the two functions (which it will do if you don't declare them yourself), but it also prevents the client programmer from passing a **CGImap** by value or from using assignment.

CGImap's job is to take the input data and parse it into name-value pairs, which it will do with the aid of **CGIpair** (effectively, **CGIpair** is only a helper class, but it also seems to make it easier to understand the code). After copying the query string (you'll see where the query string comes from later) into a local **string** object **gq**, the **nextPair( )** member function is used to parse the string into raw name-value pairs, delimited by '**=**' and '**&**' signs. Each resulting **CGIpair** object is added to the **vector** using the standard **vector::push_back( )**. When **nextPair( )** runs out of input from the query string, it returns zero.

The **CGImap::operator[ ]** takes the brute-force approach of a linear search through the elements. Since the **CGImap** is intentionally not sorted and they tend to be small, this is not too terrible. The **dump( )** function is used for testing, typically by sending information to the resulting Web page, as you might guess from the default value of **nl**, which is an HTML "break line" token.

Using GET can be fine for many applications. However, GET passes its data to the CGI program through an environment variable (called **QUERY_STRING**), and operating systems typically run out of environment space with long GET strings (you should start worrying at about 200 characters). CGI provides a solution for this: POST. With POST, the data is encoded and concatenated the same way as with GET, but POST uses standard input to pass the encoded query string to the CGI program and has no length limitation on the input. All you have to do in your CGI program is determine the length of the query string. This length is stored in the environment variable **CONTENT_LENGTH**. Once you know the length, you can allocate storage and read the precise number of bytes from standard input. Because POST is the less-fragile solution, you should probably prefer it over GET, unless you know for sure that your input will be short. In fact, one might surmise that the only reason for GET is that it is slightly easier to code a CGI program in C using GET. However, the last class in **CGImap.h** is a tool that makes handling a POST just as easy as handling a GET, which means you can always use POST.

The **class Post** inherits from a string and only has a constructor. The job of the constructor is to get the query data from the POST into itself (a **string**). It does this by reading the **CONTENT_LENGTH** environment variable using the Standard C library function **getenv( )**. This comes back as a pointer to a C character string. If this pointer is zero, the CONTENT_LENGTH environment variable has not been set, so something is wrong. Otherwise, the character string must be converted to an integer using the Standard C library function **atoi( )**. The resulting length is used with **new** to allocate enough storage to hold the query string (plus its null terminator), and then **read( )** is called for **cin**. The **read( )** function takes a pointer to the destination buffer and the number of bytes to read. The resulting buffer is inserted into the current **string** using **string::append( )**. At this point, the POST data is just a **string** object and can be easily used without further concern about where it came from.

Testing the CGI parser

Now that the basic tools are defined, they can easily be used in a CGI program like the following which simply dumps the name-value pairs that are parsed from a GET query. Remember that an iterator for a **CGImap** returns a **CGIpair** object when it is dereferenced, so you must select the **first** and **second** parts of that **CGIpair**:

```
//: C26:CGI_GET.cpp
// Tests CGImap by extracting the information
// from a CGI GET submitted by an HTML Web page.
#include "CGImap.h"

int main() {
  // You MUST print this out, otherwise the
  // server will not send the response:
  cout << "Content-type: text/plain\n" << endl;
  // For a CGI "GET," the server puts the data
  // in the environment variable QUERY_STRING:
  CGImap query(getenv("QUERY_STRING"));
  // Test: dump all names and values
  for(CGImap::iterator it = query.begin();
    it != query.end(); it++) {
    cout << (*it).first << " = "
      << (*it).second << endl;
  }
} ///:~
```

When you use the GET approach (which is controlled by the HTML page with the METHOD tag of the FORM directive), the Web server grabs everything after the '?' and puts in into the operating-system environment variable **QUERY_STRING**. So to read that information all you have to do is get the **QUERY_STRING**. You do this with the standard C library function **getenv( )**, passing it the identifier of the environment variable you wish to fetch. In **main( )**, notice how simple the act of parsing the **QUERY_STRING** is: you just hand it to the constructor for the **CGImap** object called **query** and all the work is done for you. Although an iterator is used here, you can also pull out the names and values from **query** using **CGImap::operator[ ]**.

Now it's important to understand something about CGI. A CGI program is handed its input in one of two ways: through QUERY_STRING during a GET (as in the above case) or through standard input during a POST. But a CGI program only returns its results through standard output, via **cout**. Where does this output go? Back to the Web server, which decides what to do with it. The server makes this decision based on the **content-type** header, which means that if the **content-type** header isn't the first thing it sees, it won't know what to do with the data. Thus it's essential that you start the output of all CGI programs with the **content-type** header.

In this case, we want the server to feed all the information directly back to the client program. The information should be unchanged, so the **content-type** is **text/plain**. Once the server sees this, it will echo all strings right back to the client as a simple text Web page.

To test this program, you must compile it in the cgi-bin directory of your host Web server. Then you can perform a simple test by writing an HTML page like this:

```
//:! C26:GETtest.html
<HTML><HEAD>
<TITLE>A test of standard HTML GET</TITLE>
</HEAD> Test, uses standard html GET
<Form method="GET" ACTION="/cgi-bin/CGI_GET.exe">
<P>Field1: <INPUT TYPE = "text" NAME = "Field1"
VALUE = "This is a test" size = "40"></p>
<P>Field2: <INPUT TYPE = "text" NAME = "Field2"
VALUE = "of the emergency" size = "40"></p>
<P>Field3: <INPUT TYPE = "text" NAME = "Field3"
VALUE = "broadcast system" size = "40"></p>
<P>Field4: <INPUT TYPE = "text" NAME = "Field4"
VALUE = "this is only a test" size = "40"></p>
```

```
<P>Field5: <INPUT TYPE = "text" NAME = "Field5"
VALUE = "In a real emergency" size = "40"></p>
<P>Field6: <INPUT TYPE = "text" NAME = "Field6"
VALUE = "you will be instructed" size = "40"></p>
<p><input type = "submit" name = "submit" > </p>
</Form></HTML>
///:~
```

Of course, the **CGI_GET.exe** program must be compiled on some kind of Web server and placed in the correct subdirectory (typically called "cgi-bin" in order for this web page to work. The dominant Web server is the freely-available Apache (see http://www.Apache.org), which runs on virtually all platforms. Some word-processing/spreadsheet packages even come with Web servers. It's also quite cheap and easy to get an old PC and install Linux along with an inexpensive network card. Linux automatically sets up the Apache server for you, and you can test everything on your local network as if it were live on the Internet. One way or another it's possible to install a Web server for local tests, so you don't need to have a remote Web server and permission to install CGI programs on that server.

One of the advantages of this design is that, now that **CGIpair** and **CGImap** are defined, most of the work is done for you so you can easily create your own CGI program simply by modifying **main( )**.

Using POST

The **CGIpair** and **CGImap** from **CGImap.h** can be used as is for a CGI program that handles POSTs. The only thing you need to do is get the data from a **Post** object instead of from the **QUERY_STRING** environment variable. The following listing shows how simple it is to write such a CGI program:

```
//: C26:CGI_POST.cpp
// CGImap works as easily with POST as it
// does with GET.
#include "CGImap.h"
#include <iostream>
using namespace std;

int main() {
  cout << "Content-type: text/plain\n" << endl;
  Post p; // Get the query string
  CGImap query(p);
```

```
  // Test: dump all names and values
  for(CGImap::iterator it = query.begin();
    it != query.end(); it++) {
    cout << (*it).first << " = "
      << (*it).second << endl;
  }
} ///:~
```

After creating a **Post** object, the query string is no different from a GET query string, so it is handed to the constructor for **CGImap**. The different fields in the vector are then available just as in the previous example. If you wanted to get even more terse, you could even define the **Post** as a temporary directly inside the constructor for the **CGImap** object:

```
CGImap query(Post());
```

To test this program, you can use the following Web page:

```
//:! C26:POSTtest.html
<HTML><HEAD>
<TITLE>A test of standard HTML POST</TITLE>
</HEAD>Test, uses standard html POST
<Form method="POST" ACTION="/cgi-bin/CGI_POST.exe">
<P>Field1: <INPUT TYPE = "text" NAME = "Field1"
VALUE = "This is a test" size = "40"></p>
<P>Field2: <INPUT TYPE = "text" NAME = "Field2"
VALUE = "of the emergency" size = "40"></p>
<P>Field3: <INPUT TYPE = "text" NAME = "Field3"
VALUE = "broadcast system" size = "40"></p>
<P>Field4: <INPUT TYPE = "text" NAME = "Field4"
VALUE = "this is only a test" size = "40"></p>
<P>Field5: <INPUT TYPE = "text" NAME = "Field5"
VALUE = "In a real emergency" size = "40"></p>
<P>Field6: <INPUT TYPE = "text" NAME = "Field6"
VALUE = "you will be instructed" size = "40"></p>
<p><input type = "submit" name = "submit" > </p>
</Form></HTML>
///:~
```

When you press the “submit” button, you’ll get back a simple text page containing the parsed results, so you can see that the CGI program works correctly. The server turns around and feeds the query string to the CGI program via standard input.

Handling mailing lists

Managing an email list is the kind of problem many people need to solve for their Web site. As it is turning out to be the case for everything on the Internet, the simplest approach is always the best. I learned this the hard way, first trying a variety of Java applets (which some firewalls do not allow) and even JavaScript (which isn't supported uniformly on all browsers). The result of each experiment was a steady stream of email from the folks who couldn't get it to work. When you set up a Web site, your goal should be to never get email from anyone complaining that it doesn't work, and the best way to produce this result is to use plain HTML (which, with a little work, can be made to look quite decent).

The second problem was on the server side. Ideally, you'd like all your email addresses to be added and removed from a single master file, but this presents a problem. Most operating systems allow more than one program to open a file. When a client makes a CGI request, the Web server starts up a new invocation of the CGI program, and since a Web server can handle many requests at a time, this means that you can have many instances of your CGI program running at once. If the CGI program opens a specific file, then you can have many programs running at once that open that file. This is a problem if they are each reading and writing to that file.

There may be a function for your operating system that "locks" a file, so that other invocations of your program do not access the file at the same time. However, I took a different approach, which was to make a unique file for each client. Making a file unique was quite easy, since the email name itself is a unique character string. The filename for each request is then just the email name, followed by the string ".add" or ".remove". The contents of the file is also the email address of the client. Then, to produce a list of all the names to add, you simply say something like (in Unix):

```
cat *.add > addlist
```

(or the equivalent for your system). For removals, you say:

```
cat *.remove > removelist
```

Once the names have been combined into a list you can archive or remove the files.

The HTML code to place on your Web page becomes fairly straightforward. This particular example takes an email address to be added or removed from my C++ mailing list:

```
<h1 align="center"><font color="#000000">
The C++ Mailing List</font></h1>
<div align="center"><center>

<table border="1" cellpadding="4"
cellspacing="1" width="550" bgcolor="#FFFFFF">
  <tr>
    <td width="30" bgcolor="#FF0000"> </td>
    <td align="center" width="422" bgcolor="#0">
    <form action="/cgi-bin/mlm.exe" method="GET">
    <input type="hidden" name="subject-field"
    value="cplusplus-email-list">
    <input type="hidden" name="command-field"
    value="add"><p>
    <input type="text" size="40"
    name="email-address">
    <input type="submit" name="submit"
    value="Add Address to C++ Mailing List">
    </p></form></td>
    <td width="30" bgcolor="#FF0000"> </td>
  </tr>
  <tr>
    <td width="30" bgcolor="#000000"> </td>
    <td align="center" width="422"
    bgcolor="#FF0000">
    <form action="/cgi-bin/mlm.exe" method="GET">
    <input type="hidden" name="subject-field"
    value="cplusplus-email-list">
    <input type="hidden" name="command-field"
    value="remove"><p>
    <input type="text" size="40"
    name="email-address">
    <input type="submit" name="submit"
    value="Remove Address From C++ Mailing List">
    </p></form></td>
    <td width="30" bgcolor="#000000"> </td>
  </tr>
</table>
</center></div>
```

Each form contains one data-entry field called **email-address**, as well as a couple of hidden fields which don't provide for user input but carry information back to the server nonetheless. The **subject-field** tells the

CGI program the subdirectory where the resulting file should be placed. The **command-field** tells the CGI program whether the user is requesting that they be added or removed from the list. From the **action**, you can see that a GET is used with a program called **mlm.exe** (for "mailing list manager"). Here it is:

```
//: C26:mlm.cpp
// A GGI program to maintain a mailing list
#include "CGImap.h"
#include <fstream>
using namespace std;
const string contact("Bruce@EckelObjects.com");
// Paths in this program are for Linux/Unix. You
// must use backslashes (two for each single
// slash) on Win32 servers:
const string rootpath("/home/eckel/");

int main() {
  cout << "Content-type: text/html\n"<< endl;
  CGImap query(getenv("QUERY_STRING"));
  if(query["test-field"] == "on") {
    cout << "map size: " << query.size() << "<br>";
    query.dump(cout, "<br>");
  }
  if(query["subject-field"].size() == 0) {
    cout << "<h2>Incorrect form. Contact " <<
    contact << endl;
    return 0;
  }
  string email = query["email-address"];
  if(email.size() == 0) {
    cout << "<h2>Please enter your email address"
      << endl;
    return 0;
  }
  if(email.find_first_of(" \t") != string::npos){
    cout << "<h2>You cannot use white space "
      "in your email address" << endl;
    return 0;
  }
  if(email.find('@') == string::npos) {
    cout << "<h2>You must use a proper email"
      " address including an '@' sign" << endl;
```

```
    return 0;
  }
  if(email.find('.') == string::npos) {
    cout << "<h2>You must use a proper email"
      " address including a '.'" << endl;
    return 0;
  }
  string fname = email;
  if(query["command-field"] == "add")
    fname += ".add";
  else if(query["command-field"] == "remove")
    fname += ".remove";
  else {
    cout << "error: command-field not found. Contact "
      << contact << endl;
    return 0;
  }
  string path(rootpath + query["subject-field"]
    + "/" + fname);
  ofstream out(path.c_str());
  if(!out) {
    cout << "cannot open " << path << "; Contact"
      << contact << endl;
    return 0;
  }
  out << email << endl;
  cout << "<br><H2>" << email << " has been ";
  if(query["command-field"] == "add")
    cout << "added";
  else if(query["command-field"] == "remove")
    cout << "removed";
  cout << "<br>Thank you</H2>" << endl;
} ///:~
```

Again, all the CGI work is done by the **CGImap**. From then on it's a matter of pulling the fields out and looking at them, then deciding what to do about it, which is easy because of the way you can index into a **map** and also because of the tools available for standard **string**s. Here, most of the programming has to do with checking for a valid email address. Then a file name is created with the email address as the name and ".add" or ".remove" as the extension, and the email address is placed in the file.

Maintaining your list

Once you have a list of names to add, you can just paste them to end of your list. However, you might get some duplicates so you need a program to remove those. Because your names may differ only by upper and lowercase, it's useful to create a tool that will read a list of names from a file and place them into a container of strings, forcing all the names to lowercase as it does:

```
//: C26:readLower.h
// Read a file into a container of string,
// forcing each line to lower case.
#ifndef READLOWER_H
#define READLOWER_H
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <cctype>

inline char downcase(char c) {
  using namespace std; // Compiler bug
  return tolower(c);
}

std::string lcase(std::string s) {
  std::transform(s.begin(), s.end(),
    s.begin(), downcase);
  return s;
}

template<class SContainer>
void readLower(char* filename, SContainer& c) {
  std::ifstream in(filename);
  assure(in, filename);
  const int sz = 1024;
  char buf[sz];
  while(in.getline(buf, sz))
    // Force to lowercase:
    c.push_back(string(lcase(buf)));
}
#endif // READLOWER_H ///:~
```

Since it's a **template**, it will work with any container of **string** that supports **push_back( )**. Again, you may want to change the above to the form **readln(in, s)** instead of using a fixed-sized buffer, which is more fragile.

Once the names are read into the list and forced to lowercase, removing duplicates is trivial:

```
//: C26:RemoveDuplicates.cpp
// Remove duplicate names from a mailing list
#include "readLower.h"
#include "../require.h"
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 2);
  vector<string> names;
  readLower(argv[1], names);
  long before = names.size();
  // You must sort first for unique() to work:
  sort(names.begin(), names.end());
  // Remove adjacent duplicates:
  unique(names.begin(), names.end());
  long removed = before - names.size();
  ofstream out(argv[2]);
  assure(out, argv[2]);
  copy(names.begin(), names.end(),
      ostream_iterator<string>(out,"\n"));
  cout << removed << " names removed" << endl;
} ///:~
```

A **vector** is used here instead of a **list** because sorting requires random-access which is much faster in a **vector**. (A **list** has a built-in **sort( )** so that it doesn't suffer from the performance that would result from applying the normal **sort( )** algorithm shown above).

The sort must be performed so that all duplicates are adjacent to each other. Then **unique( )** can remove all the adjacent duplicates. The program also keeps track of how many duplicate names were removed.

When you have a file of names to remove from your list, **readLower( )** comes in handy again:

```
//: C26:RemoveGroup.cpp
// Remove a group of names from a list
#include "readLower.h"
#include "../require.h"
#include <list>
using namespace std;

typedef list<string> Container;

int main(int argc, char* argv[]) {
  requireArgs(argc, 3);
  Container names, removals;
  readLower(argv[1], names);
  readLower(argv[2], removals);
  long original = names.size();
  Container::iterator rmit = removals.begin();
  while(rmit != removals.end())
    names.remove(*rmit++); // Removes all matches
  ofstream out(argv[3]);
  assure(out, argv[3]);
  copy(names.begin(), names.end(),
       ostream_iterator<string>(out,"\n"));
  long removed = original - names.size();
  cout << "On removal list: " << removals.size()
    << "\n Removed: " << removed << endl;
} ///:~
```

Here, a **list** is used instead of a **vector** (since **readLower( )** is a **template**, it adapts). Although there is a **remove( )** algorithm that can be applied to containers, the built-in **list::remove( )** seems to work better. The second command-line argument is the file containing the list of names to be removed. An iterator is used to step through that list, and the **list::remove( )** function removes every instance of each name from the master list. Here, the list doesn't need to be sorted first.

Unfortunately, that's not all there is to it. The messiest part about maintaining a mailing list is the bounced messages. Presumably, you'll just want to remove the addresses that produce bounces. If you can combine all the bounced messages into a single file, the following program has a pretty good chance of extracting the email addresses; then you can use **RemoveGroup** to delete them from your list.

```
//: C26:ExtractUndeliverable.cpp
// Find undeliverable names to remove from
```

```
// mailing list from within a mail file
// containing many messages
#include "../require.h"
#include <cstdio>
#include <string>
#include <set>
using namespace std;

char* start_str[] = {
  "following address",
  "following recipient",
  "following destination",
  "undeliverable to the following",
  "following invalid",
};

char* continue_str[] = {
  "Message-ID",
  "Please reply to",
};

// The in() function allows you to check whether
// a string in this set is part of your argument.
class StringSet {
  char** ss;
  int sz;
public:
  StringSet(char** sa, int sza):ss(sa),sz(sza) {}
  bool in(char* s) {
    for(int i = 0; i < sz; i++)
      if (strstr(s, ss[i]) != 0)
        return true;
    return false;
  }
};

// Calculate array length:
#define ALEN(A) ((sizeof A)/(sizeof *A))

StringSet
  starts(start_str, ALEN(start_str)),
  continues(continue_str, ALEN(continue_str));
```

```
int main(int argc, char* argv[]) {
  requireArgs(argc, 2,
    "Usage:ExtractUndeliverable infile outfile");
  FILE* infile = fopen(argv[1], "rb");
  FILE* outfile = fopen(argv[2], "w");
  require(infile != 0); require(outfile != 0);
  set<string> names;
  const int sz = 1024;
  char buf[sz];
  while(fgets(buf, sz, infile) != 0) {
    if(starts.in(buf)) {
      puts(buf);
      while(fgets(buf, sz, infile) != 0) {
        if(continues.in(buf)) continue;
        if(strstr(buf, "---") != 0) break;
        const char* delimiters= " \t<>():;,\n\"";
        char* name = strtok(buf, delimiters);
        while(name != 0) {
          if(strstr(name, "@") != 0)
            names.insert(string(name));
          name = strtok(0, delimiters);
        }
      }
    }
  }
  set<string>::iterator i = names.begin();
  while(i != names.end())
    fprintf(outfile, "%s\n", (*i++).c_str());
} ///:~
```

The first thing you'll notice about this program is that contains some C functions, including C I/O. This is not because of any particular design insight. It just seemed to work when I used the C elements, and it started behaving strangely with C++ I/O. So the C is just because it works, and you may be able to rewrite the program in more "pure C++" using your C++ compiler and produce correct results.

A lot of what this program does is read lines looking for string matches. To make this convenient, I created a **StringSet** class with a member function **in( )** that tells you whether any of the strings in the set are in the argument. The **StringSet** is initialized with a constant two-dimensional of

strings and the size of that array. Although the **StringSet** makes the code easier to read, it's also easy to add new strings to the arrays.

Both the input file and the output file in **main( )** are manipulated with standard I/O, since it's not a good idea to mix I/O types in a program. Each line is read using **fgets( )**, and if one of them matches with the **starts StringSet**, then what follows will contain email addresses, until you see some dashes (I figured this out empirically, by hunting through a file full of bounced email). The **continues StringSet** contains strings whose lines should be ignored. For each of the lines that potentially contains an addresses, each address is extracted using the Standard C Library function **strtok( )** and then it is added to the **set<string>** called **names**. Using a **set** eliminates duplicates (you may have duplicates based on case, but those are dealt with by **RemoveGroup.cpp**. The resulting **set** of names is then printed to the output file.

Mailing to your list

There are a number of ways to connect to your system's mailer, but the following program just takes the simple approach of calling an external command ("fastmail," which is part of Unix) using the Standard C library function **system( )**. The program spends all its time building the external command.

When people don't want to be on a list anymore they will often ignore instructions and just reply to the message. This can be a problem if the email address they're replying with is different than the one that's on your list (sometimes it has been routed to a new or aliased address). To solve the problem, this program prepends the text file with a message that informs them that they can remove themselves from the list by visiting a URL. Since many email programs will present a URL in a form that allows you to just click on it, this can produce a very simple removal process. If you look at the URL, you can see it's a call to the **mlm.exe** CGI program, including removal information that incorporates the same email address the message was sent to. That way, even if the user just replies to the message, all you have to do is click on the URL that comes back with their reply (assuming the message is automatically copied back to you).

```
//: C26:Batchmail.cpp
// Sends mail to a list using Unix fastmail
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
```

```
#include <strstream>
#include <cstdlib> // system() function
using namespace std;

string subject("New Intensive Workshops");
string from("Bruce@EckelObjects.com");
string replyto("Bruce@EckelObjects.com");
ofstream logfile("BatchMail.log");

int main(int argc, char* argv[]) {
  requireArgs(argc, 2,
    "Usage: Batchmail namelist mailfile");
  ifstream names(argv[1]);
  assure(names, argv[1]);
  string name;
  while(getline(names, name)) {
    ofstream msg("m.txt");
    assure(msg, "m.txt");
    msg << "To be removed from this list, "
      "DO NOT REPLY TO THIS MESSAGE. Instead, \n"
      "click on the following URL, or visit it "
      "using your Web browser. This \n"
      "way, the proper email address will be "
      "removed. Here's the URL:\n"
      << "http://www.mindview.net/cgi-bin/"
      "mlm.exe?subject-field=workshop-email-list"
      "&command-field=remove&email-address="
      << name << "&submit=submit\n\n"
      "----------------------------------\n\n";
    ifstream text(argv[2]);
    assure(text, argv[1]);
    msg << text.rdbuf() << endl;
    msg.close();
    string command("fastmail -F " + from +
      " -r " + replyto + " -s \"" + subject +
      "\" m.txt " + name);
    system(command.c_str());
    logfile << command << endl;
    static int mailcounter = 0;
    const int bsz = 25;
    char buf[bsz];
    // Convert mailcounter to a char string:
```

```
    ostrstream mcounter(buf, bsz);
    mcounter << mailcounter++ << ends;
    if((++mailcounter % 500) == 0) {
      string command2("fastmail -F " + from +
        " -r " + replyto + " -s \"Sent " +
        string(buf) +
        " messages \" m.txt eckel@aol.com");
      system(command2.c_str());
    }
  }
} ///:~
```

The first command-line argument is the list of email addresses, one per line. The names are read one at a time into the **string** called **name** using **getline( )**. Then a temporary file called **m.txt** is created to build the customized message for that individual; the customization is the note about how to remove themselves, along with the URL. Then the message body, which is in the file specified by the second command-line argument, is appended to **m.txt**. Finally, the command is built inside a **string**: the "-F" argument to **fastmail** is who it's from, the "-r" argument is who to reply to. The "-s" is the subject line, the next argument is the file containing the mail and the last argument is the email address to send it to.

You can start this program in the background and tell Unix not to stop the program when you sign off of the server. However, it takes a while to run for a long list (this isn't because of the program itself, but the mailing process). I like to keep track of the progress of the program by sending a status message to another email account, which is accomplished in the last few lines of the program.

A general information-extraction CGI program

One of the problems with CGI is that you must write and compile a new program every time you want to add a new facility to your Web site. However, much of the time all that your CGI program does is capture information from the user and store it on the server. If you could use hidden fields to specify what to do with the information, then it would be possible to write a single CGI program that would extract the information

from any CGI request. This information could be stored in a uniform format, in a subdirectory specified by a hidden field in the HTML form, and in a file that included the user's email address – of course, in the general case the email address doesn't guarantee uniqueness (the user may post more than one submission) so the date and time of the submission can be mangled in with the file name to make it unique. If you can do this, then you can create a new data-collection page just by defining the HTML and creating a new subdirectory on your server. For example, every time I come up with a new class or workshop, all I have to do is create the HTML form for signups – no CGI programming is required.

The following HTML page shows the format for this scheme. Since a CGI POST is more general and doesn't have any limit on the amount of information it can send, it will always be used instead of a GET for the **ExtractInfo.cpp** program that will implement this system. Although this form is simple, yours can be as complicated as you need it.

```
//:! C26:INFOtest.html
<html><head><title>
Extracting information from an HTML POST</title>
</head>
<body bgcolor="#FFFFFF" link="#0000FF"
vlink="#800080"> <hr>
<p>Extracting information from an HTML POST</p>
<form action="/cgi-bin/ExtractInfo.exe"
  method="POST">
<input type="hidden" name="subject-field"
value="test-extract-info">
<input type="hidden" name="reminder"
value="Remember your lunch!">
<input type="hidden" name="test-field"
value="on">
<input type="hidden" name="mail-copy"
value="Bruce@EckelObjects.com;eckel@aol.com">
<input type="hidden" name="confirmation"
value="confirmation1">
<p>Email address (Required): <input
type="text" size="45" name="email-address" >
</p>Comment:<br>
<textarea name="Comment" rows="6" cols="55">
</textarea>
<p><input type="submit" name="submit">
<input type="reset" name="reset"</p>
```

```
</form><hr></body></html>
///:~
```

Right after the form's **action** statement, you see

```
<input type="hidden"
```

This means that particular field will not appear on the form that the user sees, but the information will still be submitted as part of the data for the CGI program.

The value of this field named "subject-field" is used by **ExtractInfo.cpp** to determine the subdirectory in which to place the resulting file (in this case, the subdirectory will be "test-extract-info"). Because of this technique and the generality of the program, the only thing you'll usually need to do to start a new database of data is to create the subdirectory on the server and then create an HTML page like the one above. The **ExtractInfo.cpp** program will do the rest for you by creating a unique file for each submission. Of course, you can always change the program if you want it to do something more unusual, but the system as shown will work most of the time.

The contents of the "reminder" field will be displayed on the form that is sent back to the user when their data is accepted. The "test-field" indicates whether to dump test information to the resulting Web page. If "mail-copy" exists and contains anything other than "no" the value string will be parsed for mailing addresses separated by ';' and each of these addresses will get a mail message with the data in it. The "email-address" field is required in each case and the email address will be checked to ensure that it conforms to some basic standards.

The "confirmation" field causes a second program to be executed when the form is posted. This program parses the information that was stored from the form into a file, turns it into human-readable form and sends an email message back to the client to confirm that their information was received (this is useful because the user may not have entered their email address correctly; if they don't get a confirmation message they'll know something is wrong). The design of the "confirmation" field allows the person creating the HTML page to select more than one type of confirmation. Your first solution to this may be to simply call the program directly rather than indirectly as was done here, but you don't want to allow someone else to choose – by modifying the web page that's downloaded to them – what programs they can run on your machine.

Here is the program that will extract the information from the CGI request:

```
//: C26:ExtractInfo.cpp
// Extracts all the information from a CGI POST
// submission, generates a file and stores the
// information on the server. By generating a
// unique file name, there are no clashes like
// you get when storing to a single file.
#include "CGImap.h"
#include <iostream>
#include <fstream>
#include <cstdio>
#include <ctime>
using namespace std;

const string contact("Bruce@EckelObjects.com");
// Paths in this program are for Linux/Unix. You
// must use backslashes (two for each single
// slash) on Win32 servers:
const string rootpath("/home/eckel/");

void show(CGImap& m, ostream& o);
// The definition for the following is the only
// thing you must change to customize the program
void
store(CGImap& m, ostream& o, string nl = "\n");

int main() {
  cout << "Content-type: text/html\n"<< endl;
  Post p; // Collect the POST data
  CGImap query(p);
  // "test-field" set to "on" will dump contents
  if(query["test-field"] == "on") {
    cout << "map size: " << query.size() << "<br>";
    query.dump(cout);
  }
  if(query["subject-field"].size() == 0) {
    cout << "<h2>Incorrect form. Contact " <<
    contact << endl;
    return 0;
  }
  string email = query["email-address"];
  if(email.size() == 0) {
    cout << "<h2>Please enter your email address"
```

```
    << endl;
  return 0;
}
if(email.find_first_of(" \t") != string::npos){
  cout << "<h2>You cannot include white space "
    "in your email address" << endl;
  return 0;
}
if(email.find('@') == string::npos) {
  cout << "<h2>You must include a proper email"
    " address including an '@' sign" << endl;
  return 0;
}
if(email.find('.') == string::npos) {
  cout << "<h2>You must include a proper email"
    " address including a '.'" << endl;
  return 0;
}
// Create a unique file name with the user's
// email address and the current time in hex
const int bsz = 1024;
char fname[bsz];
time_t now;
time(&now); // Encoded date & time
sprintf(fname, "%s%X.txt", email.c_str(), now);
string path(rootpath + query["subject-field"] +
   "/" + fname);
ofstream out(path.c_str());
if(!out) {
  cout << "cannot open " << path << "; Contact"
    << contact << endl;
  return 0;
}
// Store the file and path information:
out << "///{" << path << endl;
// Display optional reminder:
if(query["reminder"].size() != 0)
  cout <<"<H1>" << query["reminder"] <<"</H1>";
show(query, cout); // For results page
store(query, out); // Stash data in file
cout << "<br><H2>Your submission has been "
  "posted as<br>" << fname << endl
```

```
    << "<br>Thank you</H2>" << endl;
  out.close();
  // Optionally send generated file as email
  // to recipients specified in the field:
  if(query["mail-copy"].length() != 0 &&
     query["mail-copy"] != "no") {
    string to = query["mail-copy"];
    // Parse out the recipient names, separated
    // by ';', into a vector.
    vector<string> recipients;
    int ii = to.find(';');
    while(ii != string::npos) {
      recipients.push_back(to.substr(0, ii));
      to = to.substr(ii + 1);
      ii = to.find(';');
    }
    recipients.push_back(to); // Last one
    // "fastmail" only available on Linux/Unix:
    for(int i = 0; i < recipients.size(); i++) {
      string cmd("fastmail -s"" \"" +
        query["subject-field"] + "\" " +
        path + " " + recipients[i]);
      system(cmd.c_str());
    }
  }
  // Execute a confirmation program on the file.
  // Typically, this is so you can email a
  // processed data file to the client along with
  // a confirmation message:
  if(query["confirmation"].length() != 0) {
    string conftype = query["confirmation"];
    if(conftype == "confirmation1") {
      string command("./ProcessApplication.exe "+
        path + " &");
      // The data file is the argument, and the
      // ampersand runs it as a separate process:
      system(command.c_str());
      string logfile("Extract.log");
      ofstream log(logfile.c_str());
    }
  }
}
```

```
// For displaying the information on the html
// results page:
void show(CGImap& m, ostream& o) {
  string nl("<br>");
  o << "<h2>The data you entered was:"
    << "</h2><br>"
    << "From[" << m["email-address"] << ']' <<nl;
  for(CGImap::iterator it = m.begin();
    it != m.end(); it++) {
    string name = (*it).first,
      value = (*it).second;
    if(name != "email-address" &&
       name != "confirmation" &&
       name != "submit" &&
       name != "mail-copy" &&
       name != "test-field" &&
       name != "reminder")
      o << "<h3>" << name << ": </h3>"
        << "<pre>" << value << "</pre>";
  }
}

// Change this to customize the program:
void store(CGImap& m, ostream& o, string nl) {
  o << "From[" << m["email-address"] << ']' <<nl;
  for(CGImap::iterator it = m.begin();
    it != m.end(); it++) {
    string name = (*it).first,
      value = (*it).second;
    if(name != "email-address" &&
       name != "confirmation" &&
       name != "submit" &&
       name != "mail-copy" &&
       name != "test-field" &&
       name != "reminder")
      o << nl << "[{[" << name << "]}]" << nl
        << "[([" << nl << value << nl << "])]"
        << nl;
    // Delimiters were added to aid parsing of
    // the resulting text file.
  }
```

```
} ///:~
```

The program is designed to be as generic as possible, but if you want to change something it is most likely the way that the data is stored in a file (for example, you may want to store it in a comma-separated ASCII format so that you can easily read it into a spreadsheet). You can make changes to the storage format by modifying **store( )**, and to the way the data is displayed by modifying **show( )**.

main( ) begins using the same three lines you'll start with for any POST program. The rest of the program is similar to **mlm.cpp** because it looks at the "test-field" and **"**email-address" (checking it for correctness). The file name combines the user's email address and the current date and time in hex – notice that **sprintf( )** is used because it has a convenient way to convert a value to a hex representation. The entire file and path information is stored in the file, along with all the data from the form, which is tagged as it is stored so that it's easy to parse (you'll see a program to parse the files a bit later). All the information is also sent back to the user as a simply-formatted HTML page, along with the reminder, if there is one. If "mail-copy" exists and is not "no," then the names in the "mail-copy" value are parsed and an email is sent to each one containing the tagged data. Finally, if there is a "confirmation" field, the value selects the type of confirmation (there's only one type implemented here, but you can easily add others) and the command is built that passes the generated data file to the program (called **ProcessApplication.exe**). That program will be created in the next section.

Parsing the data files

You now have a lot of data files accumulating on your Web site, as people sign up for whatever you're offering. Here's what one of them might look like:

```
//:! C23:TestData.txt
///{/home/eckel/super-cplusplus-workshop-
registration/Bruce@EckelObjects.com35B589A0.txt
From[Bruce@EckelObjects.com]

[{[subject-field]}]
[([
super-cplusplus-workshop-registration
])]
```

```
[{[Date-of-event]}]
[([
Sept 2-4
])]

[{[name]}]
[([
Bruce Eckel
])]

[{[street]}]
[([
20 Sunnyside Ave, Suite A129
])]

[{[city]}]
[([
Mill Valley
])]

[{[state]}]
[([
CA
])]

[{[country]}]
[([
USA
])]

[{[zip]}]
[([
94941
])]

[{[busphone]}]
[([
415-555-1212
])]
///:~
```

This is a brief example, but there are as many fields as you have on your HTML form. Now, if your event is compelling you'll have a whole lot of

these files and what you'd like to do is automatically extract the information from them and put that data in any format you'd like. For example, the **ProcessApplication.exe** program mentioned above will use the data in an email confirmation message. You'll also probably want to put the data in a form that can be easily brought into a spreadsheet. So it makes sense to start by creating a general-purpose tool that will automatically parse any file that is created by **ExtractInfo.cpp**:

```
//: C26:FormData.h
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

class DataPair : public pair<string, string> {
public:
  DataPair() {}
  DataPair(istream& in) { get(in); }
  DataPair& get(istream& in);
  operator bool() {
    return first.length() != 0;
  }
};

class FormData : public vector<DataPair> {
public:
  string filePath, email;
  // Parse the data from a file:
  FormData(char* fileName);
  void dump(ostream& os = cout);
  string operator[](const string& key);
}; ///:~
```

The **DataPair** class looks a bit like the **CGIpair** class, but it's simpler. When you create a **DataPair**, the constructor calls **get( )** to extract the next pair from the input stream. The **operator bool** indicates an empty **DataPair**, which usually signals the end of an input stream.

FormData contains the path where the original file was placed (this path information is stored within the file), the email address of the user, and a **vector<DataPair>** to hold the information. The **operator[ ]** allows you to perform a map-like lookup, just as in **CGImap**.

Here are the definitions:

```
//: C26:FormData.cpp {O}
#include "FormData.h"
#include "../require.h"

DataPair& DataPair::get(istream& in) {
  first.erase(); second.erase();
  string ln;
  getline(in,ln);
  while(ln.find("[{[") == string::npos)
    if(!getline(in, ln)) return *this; // End
  first = ln.substr(3, ln.find("]}]") - 3);
  getline(in, ln); // Throw away [([
  while(getline(in, ln))
    if(ln.find("])]") == string::npos)
      second += ln + string(" ");
    else
      return *this;
}

FormData::FormData(char* fileName) {
  ifstream in(fileName);
  assure(in, fileName);
  require(getline(in, filePath) != 0);
  // Should be start of first line:
  require(filePath.find("///{") == 0);
  filePath = filePath.substr(strlen("///{"));
  require(getline(in, email) != 0);
   // Should be start of 2nd line:
  require(email.find("From[") == 0);
  int begin = strlen("From[");
  int end = email.find("]");
  int length = end - begin;
  email = email.substr(begin, length);
  // Get the rest of the data:
  DataPair dp(in);
  while(dp) {
    push_back(dp);
    dp.get(in);
  }
}
```

```
string FormData::operator[](const string& key) {
  iterator i = begin();
  while(i != end()) {
    if((*i).first == key)
      return (*i).second;
    i++;
  }
  return string(); // Empty string == not found
}

void FormData::dump(ostream& os) {
  os << "filePath = " << filePath << endl;
  os << "email = " << email << endl;
  for(iterator i = begin(); i != end(); i++)
    os << (*i).first << " = "
       << (*i).second << endl;
} ///:~
```

The **DataPair::get( )** function assumes you are using the same **DataPair** over and over (which is the case, in **FormData::FormData( )**) so it first calls **erase( )** for its **first** and **second string**s. Then it begins parsing the lines for the key (which is on a single line and is denoted by the "**[{[**" and "**]}]**") and the value (which may be on multiple lines and is denoted by a begin-marker of "**[([**" and an end-marker of "**])]**") which it places in the **first** and **second** members, respectively.

The **FormData** constructor is given a file name to open and read. The **FormData** object always expects there to be a file path and an email address, so it reads those itself before getting the rest of the data as **DataPair**s.

With these tools in hand, extracting the data becomes quite easy:

```
//: C26:FormDump.cpp
//{L} FormData
#include "FormData.h"
#include "../require.h"

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  FormData fd(argv[1]);
  fd.dump();
} ///:~
```

The only reason that **ProcessApplication.cpp** is busier is that it is building the email reply. Other than that, it just relies on **FormData**:

```
//: C26:ProcessApplication.cpp
//{L} FormData
#include "FormData.h"
#include "../require.h"
using namespace std;

const string from("Bruce@EckelObjects.com");
const string replyto("Bruce@EckelObjects.com");
const string basepath("/home/eckel");

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  FormData fd(argv[1]);
  char tfname[L_tmpnam];
  tmpnam(tfname); // Create a temporary file name
  string tempfile(basepath + tfname + fd.email);
  ofstream reply(tempfile.c_str());
  assure(reply, tempfile.c_str());
  reply << "This message is to verify that you "
    "have been added to the list for the "
    << fd["subject-field"] << ". Your signup "
    "form included the following data; please "
    "ensure it is correct. You will receive "
    "further updates via email. Thanks for your "
    "interest in the class!" << endl;
  FormData::iterator i;
  for(i = fd.begin(); i != fd.end(); i++)
    reply << (*i).first << " = "
       << (*i).second << endl;
  reply.close();
  // "fastmail" only available on Linux/Unix:
  string command("fastmail -F " + from +
    " -r " + replyto + " -s \"" +
    fd["subject-field"] + "\" " +
    tempfile + " " + fd.email);
  system(command.c_str()); // Wait to finish
  remove(tempfile.c_str()); // Erase the file
} ///:~
```

This program first creates a temporary file to build the email message in. Although it uses the Standard C library function **tmpnam( )** to create a temporary file name, this program takes the paranoid step of assuming that, since there can be many instances of this program running at once, it's possible that a temporary name in one instance of the program could collide with the temporary name in another instance. So to be extra careful, the email address is appended onto the end of the temporary file name.

The message is built, the **DataPair**s are added to the end of the message, and once again the Linux/Unix **fastmail** command is built to send the information. An interesting note: if, in Linux/Unix, you add an ampersand (**&**) to the end of the command before giving it to **system( )**, then this command will be spawned as a background process and **system( )** will immediately return (the same effect can be achieved in Win32 with **start**). Here, no ampersand is used, so **system( )** does not return until the command is finished – which is a good thing, since the next operation is to delete the temporary file which is used in the command.

The final operation in this project is to extract the data into an easily-usable form. A spreadsheet is a useful way to handle this kind of information, so this program will put the data into a form that's easily readable by a spreadsheet program:

```
//: C26:DataToSpreadsheet.cpp
//{L} FormData
#include "FormData.h"
#include "../require.h"
#include <string>
using namespace std;

string delimiter("\t");

int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++) {
    FormData fd(argv[i]);
    cout << fd.email << delimiter;
    FormData::iterator i;
    for(i = fd.begin(); i != fd.end(); i++)
      if((*i).first != "workshop-suggestions")
        cout << (*i).second << delimiter;
    cout << endl;
  }
} ///:~
```

Common data interchange formats use various delimiters to separate fields of information. Here, a tab is used but you can easily change it to something else. Also note that I have checked for the "workshop-suggestions" field and specifically excluded that, because it tends to be too long for the information I want in a spreadsheet. You can make another version of this program that only extracts the "workshop-suggestions" field.

This program assumes that all the file names are expanded on the command line. Using it under Linux/Unix is easy since file-name global expansion ("globbing") is handled for you. So you say:

```
DataToSpreadsheet *.txt >> spread.out
```

In Win32 (at a DOS prompt) it's a bit more involved, since you must do the "globbing" yourself:

```
For %f in (*.txt) do DataToSpreadsheet %f >> spread.out
```

This technique is generally useful for writing Win32/DOS command lines.

Summary

Exercises

1. In **ExtractInfo.cpp**, change **store( )** so it stores the data in comma-separated ASCII format
2. (This exercise may require a little research and ingenuity, but you'll have a good idea of how server-side programming works when you're done.) Gain access to a Web server somehow, even if you do so by installing a Web server that runs on your local machine (the Apache server is freely available from http://www.Apache.org and runs on most platforms). Install and test **ExtractInfo.cpp** as a CGI program, using **INFOtest.html**.
3. Create a program called **ExtractSuggestions.cpp** that is a modification of **DataToSpreadsheet.cpp** which will only extract the suggestions along with the name and email address of the person that made them.

A: Coding style

This appendix is not about indenting and placement of parentheses and curly braces, although that will be mentioned. This is about the general guidelines used in this book for organizing the code listings.

Although many of these issues have been introduced throughout the book, this appendix appears at the end so it can be assumed that every topic is fair game, and if you don't understand something you can look it up in the appropriate section.

All the decisions about coding style in this book have been deliberately made and considered, sometimes over a period of years. Of course, everyone has their reasons for organizing code the way they do, and I'm just trying to tell you how I arrived at mine and the constraints and environmental factors that brought me to those decisions.

File names

In C, it has been traditional to name header files (containing declarations) with an extension of **.h** and implementation files (that cause storage to be allocated and code to be generated) with an extension of **.c**. C++ went through an evolution. It was first developed on Unix, where the operating system was aware of upper and lower case in file names. The original file names were simply capitalized versions of the C extensions: **.H** and **.C**. This of course didn't work for operating systems that didn't distinguish upper and lower case, like DOS. DOS C++ vendors used extensions of **hxx** and **cxx** for header files and implementation files, respectively, or **hpp** and **cpp**. Later, someone figured out that the only reason you needed a different extension for a file was so the compiler could determine whether to compile it as a C or C++ file. Because the compiler never compiled header files directly, only the implementation file extension needed to be changed. The custom, virtually across all systems, has now become to use **cpp** for implementation files and **h** for header files.

Begin and end comment tags

A very important issue with this book is that all code that you see in the book must be automatically extractable and compilable, so it can be verified to be correct (with at least one compiler). To facilitate this, all code listings that are meant to be compiled (as opposed to code fragments, of which there are few) have comment tags at the beginning and end. These tags are used by the code-extraction tool **ExtractCode.cpp** in chapter 23 to pull each code listing out of the plain-ASCII text version of this book (which you can find on the Web site http://www.BruceEckel.com).

The end-listing tag simply tells **ExtractCode.cpp** that it's the end of the listing, but the begin-listing tag is followed by information about what subdirectory the file belongs in (generally organized by chapters, so a file that belongs in Chapter 8 would have a tag of **C08**), followed by a colon and the name of the listing file.

Because **ExtractCode.cpp** also creates a **makefile** for each subdirectory, information about how a program is made and the command-line used to test it is also incorporated into the listings. If a program is stand alone (it doesn't need to be linked with anything else) it has no extra information. This is also true for header files. However, if it doesn't contain a **main( )** and is meant to be linked with something else, then it has an **{O}** after the file name. If this listing is meant to be the main program but needs to be linked with other components, there's a separate line that begins with **//{L}** and continues with all the files that need to be linked (without extensions, since those can vary from platform to platform).

Here's an example of a stand-alone program:

Here's a more complicated example that involves two header files, two implementation files and a main program that requires a link line:

Here's the **makefile** that **ExtractCode.cpp** generated for this appendix:

If a file should be extracted but the begin- and end-tags should not be included in the extracted file (for example, if it's a file of test data) then the begin-tag is immediately followed by a '**!**', like this:

Parens, braces and indentation

You may notice the formatting style in this book is different from many traditional C styles. Of course, everyone feels their own style is the most rational. However, the style used here has a simple logic behind it, which will be presented here mixed in with ideas on why some of the other styles developed.

The formatting style is motivated by one thing: presentation, both in print and in live seminars. You may feel your needs are different because you don't make a lot of presentations, but working code is read much more than it is written, so it should be easy for the reader to perceive. My two most important criteria are "scannability" (how easy it is for the reader to grasp the meaning of a single line) and the number of lines that can fit on a page. This latter may sound funny, but when you are giving a live presentation, it's very distracting to shuffle back and forth between slides, and a few wasted lines can cause this.

Everyone seems to agree that code inside braces should be indented. What people don't agree on, and the place where there's the most inconsistency within formatting styles is this: where does the opening brace go? This one question, I feel, is what causes such inconsistencies among coding styles (For an enumeration of coding styles, see C++ Programming Guidelines, by Tom Plum & Dan Saks, Plum Hall 1991). I'll try to convince you that many of today's coding styles come from pre-Standard C constraints (before function prototypes) and are thus inappropriate now.

First, my answer to the question: the opening brace should always go on the same line as the "precursor" (by which I mean "whatever the body is about: a class, function, object definition, if statement, etc."). This is a single, consistent rule I apply to all the code I write, and it makes formatting much simpler. It makes the "scannability" easier – when you look at this line:

```
void func(int a);
```

you know, by the semicolon at the end of the line, that this is a declaration and it goes no further, but when you see the line:

```
void func(int a) {
```

you immediately know it's a definition because the line finishes with an opening brace, and not a semicolon. Similarly, for a class:

```
class Thing;
```

is a class name declaration, and

```
class Thing {
```

is a class declaration. You can tell by looking at the single line in all cases whether it's a declaration or definition. And of course, putting the opening brace on the same line, instead of a line by itself, allows you to fit more lines on a page.

So why do we have so many other styles? In particular, you'll notice that most people create classes following the above style (which Stroustrup uses in all editions of his book *The C++ Programming Language* from Addison-Wesley) but create function definitions by putting the opening brace on a single line by itself (which also engenders many different indentation styles). Stroustrup does this except for short inline functions. With the approach I describe here, everything is consistent – you name whatever it is (**class**, function, **enum**, etc) and on that same line you put the opening brace to indicate that the body for this thing is about to follow. Also, the opening brace is the same for short inlines and ordinary function definitions.

I assert that the style of function definition used by many folks comes from pre-function-prototyping C, where you had to say:

```
void bar()
 int x;
 float y;
{
 /* body here */
}
```

Here, it would be quite ungainly to put the opening brace on the same line, so no one did it. However, they did make various decisions about whether the braces should be indented with the body of the code, or whether they should be at the level of the "precursor." Thus we got many different formatting styles.

There are other arguments for placing the brace on the line immediately following the declaration (of a class, struct, function, etc.). The following came from a reader, and are presented here so you know what the issues are:

> Experienced ‘vi’ (vim) users know that typing the ‘]’ key twice will take the user to the next occurance of ‘{‘ (or ^L) in column 0. This feature is extremely useful in navigating code (jumping to the next function or class definition). [My comment: when I was initially working under Unix, Gnu emacs was just appearing and I became enmeshed in that. As a result, ‘vi’ has never made sense to me, and thus I do not think in terms of “column 0 locations.” However, there is a fair contingent of ‘vi’ users out there, and they are affected by this issue]
>
> Placing the ‘{‘ on the next line eliminates some confusing code in complex conditionals, aiding in the scannability. Example:
>
> ```
> if(cond1
>   && cond2
>   && cond3) {
>   statement;
> }
> ```
>
> The above [asserts the reader] has poor scannability. However,
>
> ```
> if (cond1
> && cond2
> && cond3
> {
> statement;
> }
> ```
>
> breaks up the ‘if’ from the body, resulting in better readability.
>
> Finally, it’s much easier to visually align braces when they are aligned in the same column. They visually "stick out" much better. [End of reader comment]

The issue of where to put the opening curly brace is probably the most discordant issue. I’ve learned to scan both forms, and in the end it comes down to what you’ve grown comfortable with. However, I note that the official Java coding standard (found on Sun’s Java Web site) is effectively the same as the one I present here – since more folks are programming in both languages, the consistency between coding styles may be helpful.

The approach I use removes all the exceptions and special cases, and logically produces a single style of indentation, as well. Even within a function body, the consistency holds, as in:

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
```

```
  cout << x * i << endl;
}
```

The style is very easy to teach and remember – you use a single, consistent rule for all your formatting, not one for classes, one for functions and possibly others for for loops, if statements, etc. The consistency alone, I feel, makes it worthy of consideration. Above all, C++ is a new language and (although we must make many concessions to C) we shouldn't be carrying too many artifacts with us that cause problems in the future. Small problems multiplied by many lines of code become big problems. (For a thorough examination of the subject, albeit in C, see David Straker: C Style: Standards and Guidelines, Prentice-Hall 1992).

The other constraint I must work under is the line width, since the book has a limitation of 50 characters. What happens when something is too long to fit on one line? Well, again I strive to have a consistent policy for the way lines are broken up, so they can be easily viewed. As long as something is all part of a single definition, it should …

Order of header inclusion

Headers are included from "the most specific to the most general." That is, any header files in the local directory are included first, then any of my own "tool" headers such as **require.h** or **purge.h**, then any third-party library headers, then the standard C++ library headers, and finally the C library headers.

The justification for this comes from John Lakos in *Large-Scale C++ Software Design* (Addison-Wesley, 1996):

> *Latent usage errors can be avoided by ensuring that the .h file of a component parses by itself -- without externally-provided declarations or definitions... Including the .h file as the very first line of the .c file ensures that no critical piece of information intrinsic to the physical interface of the component is missing from the .h file (or, if there is, that you will find out about it as soon as you try to compile the .c file).*

If the order of header inclusion goes "from most specific to most general," then it's more likely that if your header doesn't parse by itself, you'll find out about it sooner and prevent annoyances down the road.

Include guards on header files

Include guards are always used in headers files [more detail here].

Use of namespaces

[More detail will be given here]. In header files, any “pollution” of the namespace in which the header is included must be scrupulously avoided, so no **using** declarations of any kind are allowed outside of function definitions.

In **cpp** files, any global **using** definitions will only affect that file, and so they are generally used for ease of reading and writing code, especially in small programs.

Use of **require( )** and **assure( )**

The **require( )** and **assure( )** functions in **require.h** are used consistently throughout most of the book, so that they may properly report problems. [more detail here]

B: Programming guidelines

This appendix[76] is a collection of suggestions for C++ programming. They've been collected over the course of my teaching and programming experience and

also from the insights of friends including Dan Saks (co-author with Tom Plum of *C++ Programming Guidelines*, Plum Hall, 1991), Scott Meyers (author of *Effective C++*, Addison-Wesley, 1992), and Rob Murray (author of *C++ Strategies & Tactics*, Addison-Wesley, 1993). Many of these tips are summarized from the pages of this book.

4. Don't automatically rewrite all your existing C code in C++ unless you need to significantly change its functionality (that is, don't fix it if it isn't broken). *Recompiling* in C++ is a very valuable activity because it may reveal hidden bugs. However, taking C code that works fine and rewriting it in C++ may not be the most valuable use of your time, unless the C++ version will provide a lot of opportunities for reuse as a class.

5. Separate the class creator from the class user (*client programmer*). The class user is the "customer" and doesn't

[76] This appendix was suggested by Andrew Binstock, editor of Unix Review, as an article for that magazine.

need or want to know what's going on behind the scenes of the class. The class creator must be the expert in class design and write the class so it can be used by the most novice programmer possible, yet still work robustly in the application. Library use will be easy only if it's transparent.

6. When you create a class, make your names as clear as possible. Your goal should be to make the user's interface conceptually simple. To this end, use function overloading and default arguments to create a clear, easy-to-use interface.

7. Data hiding allows you (the class creator) to change as much as possible in the future without damaging client code in which the class is used. In this light, keep everything as **private** as possible, and make only the class interface **public**, always using functions rather than data. Make data **public** only when forced. If class users don't need to access a function, make it **private**. If a part of your class must be exposed to inheritors as **protected**, provide a function interface rather than expose the actual data. In this way, implementation changes will have minimal impact on derived classes.

8. Don't fall into analysis paralysis. Some things you don't learn until you start coding and get some kind of system working. C++ has built-in firewalls; let them work for you. Your mistakes in a class or set of classes won't destroy the integrity of the whole system.

9. Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes. If your method produces more than that, ask yourself if all the elements have value over the lifetime of the program. If they do not, maintaining them will cost you. Members of development teams tend not to maintain anything that does not contribute to their productivity; this is a fact of life that many design methods don't account for.

10. Remember the fundamental rule of software engineering: *All problems can be simplified by introducing an extra level*

of conceptual indirection.[77] This one idea is the basis of abstraction, the primary feature of object-oriented programming.

11. Make classes as atomic as possible; that is, give each class a single, clear purpose. If your classes or your system design grows too complicated, break complex classes into simpler ones.

12. From a design standpoint, look for and separate things that change from things that stay the same. That is, search for the elements in a system that you might want to change without forcing a redesign, then encapsulate those elements in classes.

13. Watch out for *variance*. Two semantically different objects may have identical actions, or responsibilities, and there is a natural temptation to try to make one a subclass of the other just to benefit from inheritance. This is called variance, but there's no real justification to force a superclass/subclass relationship where it doesn't exist. A better solution is to create a general base class that produces an interface for both as derived classes – it requires a bit more space, but you still benefit from inheritance and will probably make an important discovery about the natural language solution.

14. Watch out for *limitation* during inheritance. The clearest designs add new capabilities to inherited ones. A suspicious design removes old capabilities during inheritance without adding new ones. But rules are made to be broken, and if you are working from an old class library, it may be more efficient to restrict an existing class in its subclass than it would be to restructure the hierarchy so your new class fits in where it should, above the old class.

15. Don't extend fundamental functionality by subclassing. If an interface element is essential to a class it should be in the base class, not added during derivation. If you're adding

[77] Explained to me by Andrew Koenig.

member functions by inheriting, perhaps you should rethink the design.

16. Start with a minimal interface to a class, as small and simple as you need. As the class is used, you'll discover ways you must expand the interface. However, once a class is in use you cannot shrink the interface without disturbing client code. If you need to add more functions, that's fine; it won't disturb code, other than forcing recompiles. But even if new member functions replace the functionality of old ones, leave the existing interface alone (you can combine the functionality in the underlying implementation if you want). If you need to expand the interface of an existing function by adding more arguments, leave the existing arguments in their current order, and put default values on all the new arguments; this way you won't disturb any existing calls to that function.

17. Read your classes aloud to make sure they're logical, referring to base classes as "is-a" and member objects as "has-a.."

18. When deciding between inheritance and composition, ask if you need to upcast to the base type. If not, prefer composition (member objects) to inheritance. This can eliminate the perceived need for multiple inheritance. If you inherit, users will think they are supposed to upcast.

19. Sometimes you need to inherit in order to access **protected** members of the base class. This can lead to a perceived need for multiple inheritance. If you don't need to upcast, first derive a new class to perform the protected access. Then make that new class a member object inside any class that needs to use it, rather than inheriting.

20. Typically, a base class will only be an interface to classes derived from it. When you create a base class, default to making the member functions pure virtual. The destructor can also be pure virtual (to force inheritors to explicitly redefine it), but remember to give the destructor a function body, because all destructors in a hierarchy are always called.

21. When you put a **virtual** function in a class, make all functions in that class **virtual**, and put in a **virtual** destructor. Start removing the **virtual** keyword when you're tuning for efficiency. This approach prevents surprises in the behavior of the interface.

22. Use data members for variation in value and **virtual** functions for variation in behavior. That is, if you find a class with state variables and member functions that switch behavior on those variables, you should probably redesign it to express the differences in behavior within subclasses and **virtual** functions.

23. If you must do something nonportable, make an abstraction for that service and localize it within a class. This extra level of indirection prevents the nonportability from being distributed throughout your program.

24. Avoid multiple inheritance. It's for getting you out of bad situations, especially repairing class interfaces where you don't have control of the broken class (see Chapter XX). You should be an experienced programmer before designing multiple inheritance into your system.

25. Don't use private inheritance. Although it's in the language and seems to have occasional functionality, it introduces significant ambiguities when combined with run-time type identification. Create a private member object instead of using private inheritance.

26. If two classes are associated with each other in some functional way (such as containers and iterators) try to make one a **public** nested **friend** class of the other, as the STL does with iterators inside containers. This not only emphasizes the association between the classes, but it allows the class name to be reused by nesting it within another class. Again, the STL does this by placing **iterator** inside each container class, thereby providing them with a common interface.
    The other reason you'll want to nest a class is as part of the **private** implementation. Here, nesting is beneficial for implementation hiding rather than class association and the prevention of namespace pollution as above.

27. Operator overloading is only “syntactic sugar”: a different way to make a function call. If overloading an operator doesn’t make the class interface clearer and easier to use, don’t do it. Create only one automatic type conversion operator for a class. In general, follow the guidelines and format given in Chapter XX when overloading operators.

28. First make a program work, then optimize it. In particular, don’t worry about writing **inline** functions, making some functions non**virtual**, or tweaking code to be efficient when you are first constructing the system. Your primary goal should be to prove the design, unless the design requires a certain efficiency.

29. Don’t let the compiler create the constructors, destructors, or the **operator=** for you. Those are training wheels. Class designers should always say exactly what the class should do and keep the class entirely under control. If you don’t want a copy-constructor or **operator=**, declare them private. Remember that if you create any constructor, it prevents the default constructor from being synthesized.

30. If your class contains pointers, you must create the copy-constructor, **operator=**, and destructor for the class to work properly.

31. When you write a copy-constructor for a derived class, remember to call the base-class copy-constructor explicitly. If you don’t, the default constructor will be called for the base class and that probably isn’t what you want. To call the base-class copy-constructor, pass it the derived object you’re copying from:
Derived(const Derived& d) : base(d) { // ...

32. To minimize recompiles during development of a large project, use the handle class/Cheshire cat technique demonstrated in Chapter XX, and remove it only if runtime efficiency is a problem.

33. Avoid the preprocessor. Always use **const** for value substitution and **inline**s for macros.

34. Keep scopes as small as possible so the visibility and lifetime of your objects are as small as possible. This reduces the chance of using an object in the wrong context and hiding a difficult-to-find bug. For example, suppose you have a container and a piece of code that iterates through it. If you copy that code to use with a new container, you may accidentally end up using the size of the old container as the upper bound of the new one. If, however, the old container is out of scope, the error will be caught at compile time.

35. Avoid global variables. Always strive to put data inside classes. Global functions are more likely to occur naturally than global variables, although you may later discover that a global function may fit better as a **static** member of a class.

36. If you need to declare a class or function from a library, always do so by including a header file. For example, if you want to create a function to write to an **ostream**, never declare **ostream** yourself using an incomplete type specification like this,
class ostream;
This approach leaves your code vulnerable to changes in representation. (For example, **ostream** could actually be a **typedef**.) Instead, always use the header file:
#include <iostream>
When creating your own classes, if a library is big, provide your users an abbreviated form of the header file with incomplete type specifications (that is, class name declarations) for cases where they only need to use pointers. (It can speed compilations.)

37. When choosing the return type of an overloaded operator, think about chaining expressions together. When defining **operator=**, remember x=x. Return a copy or reference to the lvalue (**return *this**) so it can be used in a chained expression (**A = B = C**).

38. When writing a function, pass arguments by **const** reference as your first choice. As long as you don't need to modify the object being passed in, this practice is best because it has the simplicity of pass-by-value syntax but

doesn't require expensive constructions and destructions to create a local object, which occurs when passing by value. Normally you don't want to be worrying too much about efficiency issues when designing and building your system, but this habit is a sure win.

39. Be aware of temporaries**.** When tuning for performance, watch out for temporary creation, especially with operator overloading. If your constructors and destructors are complicated, the cost of creating and destroying temporaries can be high. When returning a value from a function, always try to build the object "in place" with a constructor call in the return statement:
return MyType(i, j);
rather than
MyType x(i, j);
return x;
The former return statement eliminates a copy-constructor call and destructor call.

40. When creating constructors, consider exceptions. In the best case, the constructor won't do anything that throws an exception. In the next-best scenario, the class will be composed and inherited from robust classes only, so they will automatically clean themselves up if an exception is thrown. If you must have naked pointers, you are responsible for catching your own exceptions and then deallocating any resources pointed to before you throw an exception in your constructor. If a constructor must fail, the appropriate action is to throw an exception.

41. Do only what is minimally necessary in your constructors. Not only does this produce a lower overhead for constructor calls (many of which may not be under your control) but your constructors are then less likely to throw exceptions or cause problems.

42. The responsibility of the destructor is to release resources allocated during the lifetime of the object, not just during construction.

43. Use exception hierarchies, preferably derived from the Standard C++ exception hierarchy and nested as public

classes within the class that throws the exceptions. The person catching the exceptions can then catch the specific types of exceptions, followed by the base type. If you add new derived exceptions, client code will still catch the exception through the base type.

44. Throw exceptions by value and catch exceptions by reference. Let the exception-handling mechanism handle memory management. If you throw pointers to exceptions created on the heap, the catcher must know to destroy the exception, which is bad coupling. If you catch exceptions by value, you cause extra constructions and destructions; worse, the derived portions of your exception objects may be sliced during upcasting by value.

45. Don't write your own class templates unless you must. Look first in the Standard Template Library, then to vendors who create special-purpose tools. Become proficient with their use and you'll greatly increase your productivity.

46. When creating templates, watch for code that does not depend on type and put that code in a nontemplate base class to prevent needless code bloat. Using inheritance or composition, you can create templates in which the bulk of the code they contain is type-dependent and therefore essential.

47. Don't use the **stdio.h** functions such as **printf( )**. Learn to use iostreams instead; they are type-safe and type-extensible, and significantly more powerful. Your investment will be rewarded regularly (see Chapter XX). In general, always use C++ libraries in preference to C libraries.

48. Avoid C's built-in types. They are supported in C++ for backward compatibility, but they are much less robust than C++ classes, so your bug-hunting time will increase.

49. Whenever you use built-in types as globals or automatics, don't define them until you can also initialize them. Define variables one per line along with their initialization. When defining pointers, put the '*****' next to the type name. You can safely do this if you define one variable per line. This style tends to be less confusing for the reader.

50. Guarantee that initialization occurs in all aspects of your code. Perform all member initialization in the constructor initializer list, even built-in types (using pseudo-constructor calls). Use any bookkeeping technique you can to guarantee no uninitialized objects are running around in your system. Using the constructor initializer list is often more efficient when initializing subobjects; otherwise the default constructor is called, and you end up calling other member functions – probably **operator=** – on top of that in order to get the initialization you want.

51. Don't use the form **MyType a = b;** to define an object. This one feature is a major source of confusion because it calls a constructor instead of the **operator=**. For clarity, always be specific and use the form **MyType a(b);** instead. The results are identical, but other programmers won't be confused.

52. Use the explicit casts in C++. A cast overrides the normal typing system and is a potential error spot. Since the explicit casts divide C's one-cast-does-all into classes of well-marked casts, anyone debugging and maintaining the code can easily find all the places where logical errors are most likely to happen.

53. For a program to be robust, each component must be robust. Use all the tools provided by C++: implementation hiding, exceptions, const-correctness, type checking, and so on in each class you create. That way you can safely move to the next level of abstraction when building your system.

54. Build in **const**-correctness. This allows the compiler to point out bugs that would otherwise be subtle and difficult to find. This practice takes a little discipline and must be used consistently throughout your classes, but it pays off.

55. Use compiler error checking to your advantage. Perform all compiles with full warnings, and fix your code to remove all warnings. Write code that utilizes the compiler errors and warnings rather than that which causes runtime errors (for example, don't use variadic argument lists, which disable all type checking). Use **assert( )** for debugging, but use exceptions to work with runtime errors.

56. Prefer compile-time errors to runtime errors. Try to handle an error as close to the point of its occurrence as possible. Prefer dealing with the error at that point to throwing an exception. Catch any exceptions in the nearest handler that has enough information to deal with them. Do what you can with the exception at the current level; if that doesn't solve the problem, rethrow the exception.

57. If you're using exception specifications, install your own **unexpected( )** function using **set_unexpected( )**. Your **unexpected( )** should log the error and rethrow the current exception. That way, if an existing function gets redefined and starts throwing exceptions, you will have a record of the culprit and can modify your calling code to handle the exception.

58. Create a user-defined **terminate( )** (indicating a programmer error) to log the error that caused the exception, then release system resources, and exit the program.

59. If a destructor calls any functions, those functions may throw exceptions. A destructor cannot throw an exception (this can result in a call to **terminate( )**, which indicates a programming error), so any destructor that calls functions must catch and manage its own exceptions.

60. Don't create your own "decorated" private data member names, unless you have a lot of pre-existing global values; otherwise, let classes and namespaces do that for you.

61. If you're going to use a loop variable after the end of a **for** loop, define the variable *before* the **for** control expression. This way, you won't have any surprises when implementations change to limit the lifetime of variables defined within **for** control-expressions to the controlled expression.

62. Watch for overloading. A function should not conditionally execute code based on the value of an argument, default or not. In this case, you should create two or more overloaded functions instead.

63. Hide your pointers inside container classes. Bring them out only when you are going to immediately perform operations on them. Pointers have always been a major source of bugs. When you use **new**, try to drop the resulting pointer into a container. Prefer that a container "own" its pointers so it's responsible for cleanup. If you must have a free-standing pointer, always initialize it, preferably to an object address, but to zero if necessary. Set it to zero when you delete it to prevent accidental multiple deletions.

64. Don't overload global **new** and **delete**; always do it on a class-by-class basis. Overloading the global versions affects the entire client programmer project, something only the creators of a project should control. When overloading **new** and **delete** for classes, don't assume you know the size of the object; someone may be inheriting from you. Use the provided argument. If you do anything special, consider the effect it could have on inheritors.

65. Don't repeat yourself. If a piece of code is recurring in many functions in derived classes, put that code into a single function in the base class and call it from the derived class functions. Not only do you save code space, you provide for easy propagation of changes. This is possible even for pure virtual functions (see Chapter XX). You can use an inline function for efficiency. Sometimes the discovery of this common code will add valuable functionality to your interface.

66. Prevent object slicing. It virtually never makes sense to upcast an object by value. To prevent this, put pure virtual functions in your base class.

67. Sometimes simple aggregation does the job. A "passenger comfort system" on an airline consists of disconnected elements: seat, air conditioning, video, etc., and yet you need to create many of these in a plane. Do you make private members and build a whole new interface? No – in this case, the components themselves are also part of the public interface, so you should create public member objects. Those objects have their own private implementations, which are still safe.

C: Recommended reading

C

Thinking in C: Foundations for Java & C++, by Chuck Allison (a MindView, Inc. Seminar on CD ROM, 1999, available at http://www.MindView.net). A course including lectures and slides in the foundations of the C Language to prepare you to learn Java or C++. This is not an exhaustive course in C; only the necessities for moving on to the other languages are included. An extra section covering features for the C++ programmer is included. Prerequisite: experience with a high-level programming language, such as Pascal, BASIC, Fortran, or LISP.

General C++

The C++ Programming Language, 3rd edition, by Bjarne Stroustrup (Addison-Wesley 1997). To some degree, the goal of the book that you're currently holding is to allow you to use Bjarne's book as a reference. Since his book contains the description of the language by the author of that language, it's typically the place where you'll go to resolve any uncertainties about what C++ is or isn't supposed to do. When you get the knack of the language and are ready to get serious, you'll need it.

C++ Primer, 3rd Edition, by Stanley Lippman and Josee Lajoie (Addison-Wesley 1998). Not that much of a primer anymore; it's evolved into a thick book filled with lots of detail, and the one that I reach for along with

Stroustrup's when trying to resolve an issue. *Thinking in C++* should provide a basis for understanding the *C++ Primer* as well as Stroustrup's book.

C & C++ Code Capsules, by Chuck Allison (Prentice-Hall, 1998). Assumes that you already know C and C++, and covers some of the issues that you may be rusty on, or that you may not have gotten right the first time. This book fills in C gaps as well as C++ gaps.

The C++ ANSI/ISO Standard. This is *not* free, unfortunately (I certainly didn't get paid for my time and effort on the Standards Committee – in fact, it cost me a lot of money). But at least you can buy the electronic form in PDF for only $18 at http://www.cssinfo.com.

Large Scale C++ (?) by John Lakos.

C++ Gems, Stan Lippman, editor. SIGS publications.

The Design & Evolution of C++, by Bjarne Stroustrup

My own list of books

Not all of these are currently available.

Computer Interfacing with Pascal & C (Self-published via the Eisys imprint; only available via the Web site)

Using C++

C++ Inside & Out

Thinking in C++, 1st edition

Black Belt C++, the Master's Collection (edited by Bruce Eckel) (out of print).

Thinking in Java

Depth & dark corners

Books that go more deeply into topics of the language, and help you avoid the typical pitfalls inherent in developing C++ programs.

Effective C++ and More Effective C++, by Scott Meyers.

Ruminations on C++ by Koenig & Moo.

Analysis & Design

***Object-Oriented Design with Applications* 2nd edition** (??) by Grady Booch, Benjamin/Cummings, 1993 (??).The Booch method is one of the original, most basic, and most widely referenced. Because it was developed early, it was meant to be applied to a variety of programming problems. It focuses on the unique features of OOP: classes, methods, and inheritance. This is still considered one of the best introductory books.

Jacobsen

Object Analysis and Design: Description of Methods, edited by Andrew T.F. Hutt of the Object Management Group (OMG), John Wiley & Sons, 1994. A summary of Object-Oriented Analysis and Design techniques; very nice as an overview or if you want to synthesize your own method by choosing elements of others.

Before you choose any method, it's helpful to gain perspective from those who are not trying to sell one. It's easy to adopt a method without really understanding what you want out of it or what it will do for you. Others are using it, which seems a compelling reason. However, humans have a strange little psychological quirk: If they want to believe something will solve their problems, they'll try it. (This is experimentation, which is good.) But if it doesn't solve their problems, they may redouble their efforts and begin to announce loudly what a great thing they've discovered. (This is denial, which is not good.) The assumption here may be that if you can get other people in the same boat, you won't be lonely, even if it's going nowhere.

This is not to suggest that all methodologies go nowhere, but that you should be armed to the teeth with mental tools that help you stay in experimentation mode ("It's not working; let's try something else") and out of denial mode ("No, that's not really a problem. Everything's wonderful, we don't need to change"). I think the following books, read *before* you choose a method, will provide you with these tools.

Software Creativity, by Robert Glass (Prentice-Hall, 1995). This is the best book I've seen that discusses *perspective* on the whole methodology issue. It's a collection of short essays and papers that Glass has written and sometimes acquired (P.J. Plauger is one contributor), reflecting his many years of thinking and study on the subject. They're entertaining and only long enough to say what's necessary; he doesn't ramble and lose your interest. He's not just blowing smoke, either; there are hundreds of

references to other papers and studies. All programmers and managers should read this book before wading into the methodology mire.

Object Lessons by Tom Love (SIGS Books, 1993). Another good "perspective" book.

Peopleware, by Tom Demarco and Timothy Lister (Dorset House, 2nd edition 1999). Although they have backgrounds in software development, this book is about projects and teams in general. But the focus is on the *people* and their needs rather than the technology and its needs. They talk about creating an environment where people will be happy and productive, rather than deciding what rules those people should follow to be adequate components of a machine. This latter attitude, I think, is the biggest contributor to programmers smiling and nodding when XYZ method is adopted and then quietly doing whatever they've always done.

Complexity, by M. Mitchell Waldrop (Simon & Schuster, 1992). This chronicles the coming together of a group of scientists from different disciplines in Santa Fe, New Mexico, to discuss real problems that the individual disciplines couldn't solve (the stock market in economics, the initial formation of life in biology, why people do what they do in sociology, etc.). By crossing physics, economics, chemistry, math, computer science, sociology, and others, a multidisciplinary approach to these problems is developing. But more importantly, a different way of *thinking* about these ultra-complex problems is emerging: Away from mathematical determinism and the illusion that you can write an equation that predicts all behavior and toward first *observing* and looking for a pattern and trying to emulate that pattern by any means possible. (The book chronicles, for example, the emergence of genetic algorithms.) This kind of thinking, I believe, is useful as we observe ways to manage more and more complex software projects.

The STL Design Patterns

D:Compiler specifics

This appendix contains the compiler information database which is used by the automatic code extraction program **ExtractCode.cpp** to build makefiles that will work properly with the various compilers.

The explanation for this database format is found in the description of **ExtractCode.cpp** in Chapter XX. The information about files that won't compile, along with the data for commercial compilers, is removed from the published version, and only the online version at http://www.BruceEckel.com (along with the source code file at that location) will contain that data (since it will be subject to change as new compilers are added and as bugs are removed).

```
#: :CompileDB.txt
# Compiler information listings for Thinking in
# C++ 2nd Edition By Bruce Eckel. See copyright
# notice in Copyright.txt.
# This is used by ExtractCode.cpp to generate the
# makefiles for the book, including the command-
# line flags for each vendor's compiler and
# linker. Following that are the code listings
# from the book that will not compile for each
# compiler. The listings are, to the best of my
# knowledge, correct Standard C++ (According to
# the Final Draft International Standard). Please
# note that the tests were performed with the
# most recent compiler that I had at the time,
# and may have changed since this file was
```

```
# created.
# After ExtractCode.cpp creates the makefiles
# for each chapter subdirectory, you can say
# "make egcs", for example, and all the programs
# that will successfully compile with egcs will
# be built.
##########################################
######
# Compiling all files, for a (theoretical) fully-
# conformant compiler. This assumes a typical
# compiler under dos:
{ all }
# Object file name extension in parentheses:
(obj)
# Executable file extension in square brackets:
[exe]
# The leading '&' is for special directives. The
# dos directive means to replace '/'
# with '\' in all directory paths:
&dos
# The following lines will be inserted directly
# into the makefile (sans the leading '@' sign)
# If your environment variables are set to
# establish these you won't need to use arguments
# on the make command line to set them:
# CPP: the name of your C++ compiler
# CPPFLAGS: Compilation flags for your compiler
# OFLAG: flag to give the final executable name
#@CPP = yourcompiler
#@CPPFLAGS =
#@OFLAG = -e
@.SUFFIXES : .obj .cpp .c
@.cpp.obj :
@   $(CPP) $(CPPFLAGS) -c $<
@.c.obj :
@   $(CPP) $(CPPFLAGS) -c $<
# Assumes all files will compile
# See later for an example of Unix configuration
##########################################
######
# Borland C++ Builder 4 –- With Upgrade!!!
# Target name used in makefile:
```

```
{ Borland }
# Object file name extension in parentheses:
(obj)
# Executable file extension in square brackets:
[exe]
# The leading '&' is for special directives. The
# dos directive means to replace '/'
# with '\' in all directory paths:
&dos
# Inserted directly into the makefile (without
# the leading '@' sign):
@# Note: this requires the upgrade from
@# www.Borland.com for successful compilation!
@CPP = Bcc32
@CPPFLAGS = -w-inl -w-csu -wnak
@OFLAG = -e
@.SUFFIXES : .obj .cpp .c
@.cpp.obj :
@   $(CPP) $(CPPFLAGS) -c $<
@.c.obj :
@   $(CPP) $(CPPFLAGS) -c $<
# Doesn't support static const
# array initialization:
C10:StaticArray.cpp
# Problem with string constructors at run-time:
C17:ICompare.cpp
# Template bug:
C19:ArraySize.cpp
# Not sure:
C20:AssocInserter.cpp
# Bitset is Broken in this compiler's library:
C20:BitSet.cpp
# Standard Library problem:
C21:SearchReplace.cpp
# Function-level try blocks not implemented:
C23:FunctionTryBlock.cpp
# Uses the SGI STL extensions, so it actually
# isn't supposed to compile with this
# compiler:
C20:MapVsHashMap.cpp
C21:MemFun4.cpp
C21:Compose2.cpp
```

```
##########################################################
######
# Visual C++ 6.0 –- With Service Pack 3!!!
# Target name used in makefile:
{ Microsoft }
# Object file name extension in parentheses:
(obj)
# Executable file extension in square brackets:
[exe]
# The leading '&' is for special directives. The
# dos directive means to replace '/'
# with '\' in all directory paths:
&dos
# Inserted directly into the makefile (without
# the leading '@' sign):
@# Note: this requires the service Pack 3 from
@# www.Microsoft.com for successful compilation!
@CPP = cl
@CPPFLAGS = -GX -GR
@OFLAG = -o
@.SUFFIXES : .obj .cpp .c
@.cpp.obj :
@   $(CPP) $(CPPFLAGS) -c $<
@.c.obj :
@   $(CPP) $(CPPFLAGS) -c $<
C02:Incident.cpp
# It can't even handle multiple "for(int i =...:"
# statements in the same scope (a really old
# language feature!):
C02:Intvector.cpp
C03:Assert.cpp
C07:MemTest.cpp
C09:Cpptime.cpp
C12:Comma.cpp
C13:GlobalNew.cpp
# Common problem with namespaces and C libraries:
C17:ICompare.cpp
C18:FileClassTest.cpp
C18:Datagen.cpp
C18:Datascan.cpp
# Can't do template type induction properly:
C19:ArraySize.cpp
```

```
# Doesn't know about template-templates:
C19:TemplateTemplate.cpp
C19:applyGromit2.cpp
# Can't do template specializations:
C19:Sorted.cpp
# Can't do explicit template instantiation:
C19:ExplicitInstantiation.cpp
# Missing part of iostreams:
C20:StreamIt.cpp
# Problem with STL:
C20:BasicSequenceOperations.cpp
C20:VectorCoreDump.cpp
C20:DequeConversion.cpp
C20:Stack2.cpp
# Problem with static class initializer:
C20:BankTeller.cpp
# Missing STL functionality:
C20:VectorOfBool.cpp
# STL problem:
C20:AssocInserter.cpp
# Various problems:
C20:WildLifeMonitor.cpp
C20:MultiSet1.cpp
C20:Thesaurus.cpp
# These use the SGI STL extensions, so they
# actually aren't supposed to compile
# with this compiler:
C20:MapVsHashMap.cpp
C21:Compose2.cpp
# Namespace problem again, and other issues:
C21:FunctionObjects.cpp
C21:Binder1.cpp
C21:Binder3.cpp
C21:Binder4.cpp
C21:RandGenTest.cpp
C21:MemFun1.cpp
C21:MemFun2.cpp
C21:FindBlanks.cpp
C21:MemFun3.cpp
C21:MemFun4.cpp
C21:FillGenerateTest.cpp
C21:Counting.cpp
```

```
C21:Manipulations.cpp
C21:SearchReplace.cpp
C21:Comparison.cpp
C21:Removing.cpp
C21:SortTest.cpp
C21:SortedSearchTest.cpp
C21:MergeTest.cpp
C21:SetOperations.cpp
C21:ForEach.cpp
C21:Transform.cpp
C21:CalcInventory.cpp
C21:TransformNames.cpp
C21:SpecialList.cpp
C21:NumericTest.cpp
# Most compilers don't support this yet:
C23:FunctionTryBlock.cpp
# Lack of support for 'static const' again:
C25:Recycle2.cpp
C26:ExtractCode.cpp
C26:MemTest.cpp
# All these do not compile only becase of the
# lack of support for 'static const'. To make
# them compile, you must substitute the
# 'enum hack' shown in chapter 8:
C08:StringStack.cpp
C08:Quoter.cpp
C08:Volatile.cpp
C10:StaticArray.cpp
C11:HowMany2.cpp
C11:Autocc.cpp
C11:Pmem2.cpp
C12:Smartp.cpp
C12:Iosop.cpp
C12:Copymem.cpp
C12:Refcount.cpp
C12:RefcountTrace.cpp
C13:MallocClass.cpp
C13:Framis.cpp
C13:ArrayNew.cpp
C14:FName1.cpp
C14:FName2.cpp
C16:IStack.cpp
```

```
C16:Stemp.cpp
C16:Stemp2.cpp
C16:Stackt.cpp
C23:Cleanup.cpp
C24:Selfrtti.cpp
C24:Reinterp.cpp
##########################################
######
# The egcs (Experimental g++ compiler) snapshot
# under Linux, dated July 18, 1998
{ egcs }
(o)
[]
# The unix directive controls the way some of the
# makefile lines are generated:
&unix
@CPP = g++
@OFLAG = -o
@.SUFFIXES : .o .cpp .c
@.cpp.o :
@   $(CPP) $(CPPFLAGS) -c $<
@.c.o :
@   $(CPP) $(CPPFLAGS) -c $<
# Files that won't compile
# Error in streambuf.h:
C18:Cppcheck.cpp
# Not sure:
C19:applyGromit2.cpp
# Missing the standard library 'at()':
C19:Sorted.cpp
C19:ExplicitInstantiation.cpp
# Problem with the egcs iterator header
C20:StreambufIterator.cpp
C20:RawStorageIterator.cpp
# egcs is missing istreambuf_iterator
C20:WordList2.cpp
C20:TokenizeTest.cpp
C20:TokenIteratorTest.cpp
C20:WordCount.cpp
C20:MultiSetWordCount.cpp
# egcs is missing std::iterator:
C20:Ring.cpp
```

```
# egcs is missing char_traits
C17:ICompare.cpp
# egcs vector and deque (at least) are missing
# the "at()" functions:
C20:IndexingVsAt.cpp
# There's a problem with the egcs string class:
C17:Compare2.cpp
# Broken in this compiler's library:
C20:BitSet.cpp
# These are because <sstream> isn't implemented
C18:NumberPhotos.cpp
C19:stringConvTest.cpp
C20:StringVector.cpp
C20:FEditTest.cpp
C20:StringDeque.cpp
C20:VectorOfBool.cpp
C20:WildLifeMonitor.cpp
C21:SortTest.cpp
C21:SortedSearchTest.cpp
C21:Binder4.cpp
C21:ForEach.cpp
# Problem in parsing PrintSequence.h:
C21:Counting.cpp
C21:Manipulations.cpp
C21:SearchReplace.cpp
C21:Comparison.cpp
C21:Removing.cpp
C21:CalcInventory.cpp
C21:TransformNames.cpp
C21:SpecialList.cpp
C21:NumericTest.cpp
# The end tag is required:
#///:~
```

Index