Note: This document requires the installation of the fonts Georgia, Verdana and Andale Mono (code font) for proper viewing. These can be found at: http://sourceforge.net/project/showfiles.php?group_id=34153&release_id=105355

Revision 13 – (December 31, 2002)

Updated the exercises for Chapter 7. Finished rewriting Chapter 9. Added a template variation of Singleton to chapter 10. Updated the build directives. Fixed lotsa stuff. Chapters 5 and 11 still await rewrite.

Revision 12 – (December 23, 2002)

Added material on Design Patterns as Chapter 10 (Concurrency will move to Chapter 11). Added exercises for Chapter 6. Here is the status of all chapters:

100% complete: 1-4, 6, 8

Copy-edited, waiting for tech edit: 7, 10

Incomplete: 5, 9, 11

Revision 11 (December 13, 2002) –

Chapter 7 has been updated. Chapter 6 has been copy-edited and a few bugs were fixed. Chapter 4 has been tech-edited. The exercises are still out of date except for chapters 1-3.

Revision 10 (October 15, 2002) –

Chapters 1 through 3 are now 100% complete (copy-edited and tech-edited). Chapter 4 has been copy-edited. Updated Chapter 6 to fit in its new position and adding introductory material. (Chapters 5 and 7-10 are still unfinished at this point).

Revision 9 (August 29, 2002) –

Finished Chapter 4 (IOStreams). Reordered the material and added material on wide stream and locales. Removed references to strstreams. Edited the “Iostreams examples” section. Added new exercises.

Revision 8 (August 6, 2002) --

Made **ExtractCode.cpp** in Chapter 3 work for GNU C++.

Copy-edited Chapters 1 through 3.

Revision 7 (July 31, 2002) --

Fixed omissions in comments for code extraction throughout text.

Edited Chapter 3:

- Added a wide-character version of **ichar_traits**
- Replaced **SiteMapConvert.cpp** with **ExtractCode.cpp**
- Added exercises

Revision 6 (July 27, 2002) --

Finished Chapter 3 (Strings)

- Mentioned caveat about reference counting with multithreading.
- Removed first (out-of-date) HTML example
- Fixed the **ichar_traits** example
- Fixed stupid **MemCheck.cpp** error in Chapter 2

Revision 5 (July 20, 2002) --

Chapters 1 and 2 are “finished”.

- Reordered the material in Chapter 1:
    - Placed exception specifications last, and warned of their dangers with template classes
    - Added a section on Exception Safety.
    - Added material on **auto_ptr**
    - Added material illustrating **bad_exception**
    - Explained the internal implementation of exceptions and the Zero-cost Model
- Merged Chapter 3 (Debugging) into Chapter 2:
    - Added material on invariants, assertions and Design-by-contract
    - Placed the **TestSuite** in its own namespace
    - Finished the **MemCheck** system for tracking memory errors
- Removed Chapter 11 (Design Patterns)
    - Will be replaced by Chapter 10 (Concurrent Programming)

Revision 4, August 19, 2001 --

- Restructured the book; this is the first version with Chuck Allison coauthoring. Incorporated Chuck's testing framework, which will be used throughout the book and automatically included as part of the book's build process in the makefiles.
- In the code subdirectory of the unpacked distribution, you can now use make to compile for Borland, Microsoft, Gnu g++2.95 (distributed with Cygwin) and Gnu g++3.0 (tested under Linux).

- Under Windows98/NT/2000, you will get best results running under the free Cygwin environment (www.Cygwin.com), even if you're compiling for Borland or Microsoft. In particular, some linker command lines for Microsoft are too long for Win98 COMMAND.COM, but work just fine under Cygwin.

- Made many code changes to allow programs to be run inside the test framework, in particular removing the need for user input when executing programs.

- Added //{L} ../TestSuite/Test in all the programs that can be run without user input, to cause the makefile builder to generate calls to the programs as part of the build process.

"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++

2nd Edition

Volume 2: Practical Programming

Bruce Eckel, President, MindView Inc.

Chuck Allison, Utah Valley State College

dedication

To all those who have tirelessly worked
toward the development of the C++ language

What's inside...

Preface

In Volume 1 of this book, you learn the fundamentals of C and C++. In this volume, we look at more advanced features, with an eye towards developing techniques and ideas that produce robust C++ programs.

Thus, in this volume we are assuming that you are competent with the material developed in Volume 1. Comment

Goals

Our goals in this book are to:Comment

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

2. Teach "practical programming" techniques that you can use on a day-to-day basis.

3. Give you what we think is important for you to understand about the language, rather than everything we know. We believe there is an "information importance hierarchy," and there are some facts that 95% of programmers will never need to know, but that would just confuse people and add to their perception of the complexity of the language. To take an example from C, if you memorize the operator precedence table (we 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 is more important for compiler writers than for programmers.

4. 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.

5. We have endeavored not to use any particular vendor's version of C++. We have tested the code on all the implementations we could, and when one implementation absolutely refused to work because it doesn't conform to the C++ Standard, we've flagged that fact in the example (you'll see the flags in the source code) to exclude it from the build process.

6. Automate the compiling and testing of the code in the book. We have discovered that code that isn't compiled and tested is probably broken, so in this volume we've instrumented the examples with test code. In addition, the code that you can download from http://www.MindView.net has been extracted directly from the text of the book using programs that also automatically create makefiles to compile and run the tests. This way we know that the code in the book is correct.

Chapters

Here is a brief description of the chapters contained in this book:

Part 1: Building Stable Systems

1. 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.Comment

2. Defensive Programming. (Description)

Part 2: The Standard C++ Library

3. Strings in Depth. (Description)

4. 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.Comment

5. Templates in Depth. (Description)

6. STL Algorithms. (Description)

7. STL Containers & Iterators (Description)

Part 3: Special Topics

8. 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. Comment

9. 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.Comment

10. Design Patterns.

11. Concurrent Programming.

Exercises

We have 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.Comment

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).Comment

Exercise solutions

Solutions to exercises can be found in the electronic document *The C++ Annotated Solution Guide*, Volume 2, available for a small fee from www.MindView.net. [[Note this is not yet available]]Comment

Source code

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

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

```
//:! :CopyRight.txt
Copyright (c) MindView, Inc., 2002
Source code file from the book
"Thinking in C++, 2nd Edition, Volume 2."
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.MindView.net
(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.MindView.net. (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. Comment

Language standards

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

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

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. Comment

Seminars, CD-ROMs & consulting

Bruce Eckel's company, MindView, Inc., provides public hands-on training seminars based on the material in this book, and also for advanced topics. Selected material from each chapter represents a lesson, which is followed by a monitored exercise period so each student receives personal attention. We also provide on-site training, consulting, mentoring, and design & code walkthroughs. Information and sign-up forms for upcoming seminars and other contact information can be found at **http://www.MindView.net**. Comment

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 feedback system built into the electronic version of this book, which you will find at **http://www.MindView.net**. The feedback system uses unique identifiers on the paragraphs in the book, so you should click on the identifier next to the paragraph that you wish to comment on. Your help is appreciated. Comment

About the cover

The cover artwork was painted by Larry O'Brien's wife, Tina Jensen (yes, the Larry O'Brien who was the editor of Software Development Magazine for so many years, and who is the primary author of *Thinking in C#*). Not only are the pictures beautiful, but they are excellent suggestions of polymorphism. The idea for using these images came from Daniel Will-Harris, the cover designer (www.Will-Harris.com), working with Bruce Eckel.

Acknowledgements

Volume 2 of this book languished in a half-completed state for a long time while Bruce got distracted with other things, notably Java, Design Patterns and especially Python (see www.Python.org). If Chuck hadn't been willing (foolishly, he has sometimes thought) to finish the other half, this book almost certainly wouldn't have happened. There aren't that many people whom Bruce would have felt comfortable entrusting this book to. Chuck's penchant for precision, correctness and clear explanation is what has made this book as good as it is.

Jamie King acted as an intern during the completion of this book. He has been instrumental in making sure the book got finished, not only by providing feedback for Chuck, but especially because of his relentless questioning and picking of every single possible nit that he didn't completely understand. If your questions are answered by this book, it's probably because Jamie asked them first. Jamie also created many of the exercises at the end of each chapter.

The ideas and understanding in this book have come from many other sources, as well: friends like Andrea Provaglio, 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), Herb Sutter, PJ Plauger, Pete Becker, Kevlin Henney, David Abrahams, Tom Plum, Reg Charney, Tom Penello, Sam Druker, and Uwe Steinmueller; people who have spoken in the C++ track at the Software Development Conference (which Bruce created and developed, and Chuck spoke in); and very often students in seminars, who ask the questions we need to hear in order to make the material clearer. Comment

The book design, cover design, and cover photo were created by Bruce's 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, we produced the camera-ready pages ourselves, so the typesetting errors are ours. Microsoft® Word XP was used to write the book and to create camera-ready pages. The body typeface is Georgia and the headlines are in Verdana. Comment

We also wish to thank the generous professionals at the Edison Design Group and Dinkumware, Ltd., for giving us complimentary copies of their compiler and library (respectively). Without their assistance some of the examples in this book could not have been tested.

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

Evan Cofsky (Evan@TheUnixMan.com) provided all sorts of assistance on the server as well as development of programs in his now-favorite language, Python. Sharlynn Cobaugh and Paula Steuer were instrumental assistants, preventing Bruce from being washed away in a flood of projects.

Dawn McGee provided much-appreciated inspiration and enthusiasm during this project. The supporting cast of friends includes, but is not limited to: Mark Western, Gen Kiyooka, Kraig Brockschmidt, 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, The Italians (Andrea Provaglio, Laura Fallai, Marco Cantu, Corrado, Ilsa and Christina Giustozzi), Chris & Laura Strand, The Almquists, Brad Jerbic, John Kruth & Marilyn Cvitanic, Holly Payne (yes, the famous novelist!), Mark Mabry, The Robbins Families, The Moelter Families (& the McMillans), The Wilks, Dave Stoner, Laurie Adams, 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.

Part 1: Building Stable Systems

1: Exception handling

Improving 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. One reason, no doubt, is 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. Comment

The problem with C's approach to error handling could be thought of as 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. Comment

One of the major features in C++ is *exception handling*, which is a better way of thinking about and handling errors. With exception handling the following statements apply: Comment

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 dynamic scope, and so on until the error is either caught or the program terminates because there was no handler to catch that type of exception.

This chapter examines C's approach to error handling (such as it is), discusses why it did not work well for C, and explains why it won't work at all for C++. This chapter also covers **try**, **throw**, and **catch**, the C++ keywords that support exception handling. Comment

Error handling in C

In most of the examples in these volumes, we use **assert( )** as it was intended: for debugging during development with code that can be disabled with **#define NDEBUG** for the shipping product. Runtime error checking uses the **require.h** functions (**assure( )** and **require( )**) developed in Chapter 9 in Volume 1. These functions are 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 might be enough for small programs, but for complicated products you might need to write more sophisticated error-handling code. Comment

Error handling is quite straightforward in situations in which 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. Comment

The problem occurs when you *don't* have enough information in that context, and you need to pass the error information into a different context where that information does exist. In C, you can handle this situation using three approaches: Comment

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 earlier, the programmer is likely to 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 might 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 approach involves 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 might clash.

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. (In fact, if **longjmp( )** jumps past the end of a scope where destructors should be called, the behavior of the program is undefined.) 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**: Comment

```
//: C01: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;
  }
} ///:~
```

The **setjmp( )** function is odd because if you call it directly, it stores all the relevant information about the current processor state (such as the contents of the instruction pointer and runtime stack pointer) in the **jmp_buf** and returns zero. In this case it behaves like 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 return to any pre-determined location higher up in the runtime stack with **setjmp( )**/**longjmp( )** (wherever you've placed a call to **setjmp( )**). Comment

The problem in C++ is that **longjmp( )** doesn't respect objects; in particular it doesn't call destructors when it jumps out of a scope.[1] Destructor calls are essential, so this approach won't work with C++. In fact, the C++ standard states that branching into a scope with **goto** (effectively bypassing constructor calls), or branching out of a scope with **longjmp( )** where an object on the stack has a destructor, constitutes undefined behavior. Comment

Throwing an exception

If you encounter an exceptional situation in your code—that is, one in which 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 that contains that information and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like: Comment

```
//: C01:MyError.cpp
class MyError {
  const char* const data;
public:
  MyError(const char* const msg = 0) : data (msg)
{}
};

void f() {
   // Here we "throw" an exception object:
   throw MyError("something bad happened");
}

int main() {
   // As you'll see shortly,
   // we'll want a "try block" here:
  f();
} ///:~
```

[1] You might be surprised when you run the example—some C++ compilers have extended **longjmp( )** to clean up objects on the stack. This behavior is nonportable.

MyError is an ordinary class, which in this case takes a **char*** as a constructor argument. You can use any type when you throw (including built-in types), but usually you'll create special classes for throwing exceptions. Comment

The keyword **throw** causes a number of relatively magical things to happen. First, it creates a copy of the object you're throwing and, in effect, "returns" it from the function containing the throw expression, even though that object type isn't normally what the function is designed to return. A naïve way to think about exception handling is as an alternate return mechanism (although you find you can get into trouble if you take the analogy too far). You can also exit from ordinary scopes by throwing an exception. In any case, a value is returned, and the function or scope exits. Comment

Any similarity to function returns ends there because *where* you return is some place completely different from where a normal function call returns. (You end up in an appropriate part of the code—called an exception handler—that might be far removed from where the exception was thrown.) In addition, any local objects created by the time the exception occurs are destroyed. This automatic cleanup of local objects is often called "stack unwinding." Comment

In addition, you can throw as many different types of objects as you want. Typically, you'll throw a different type for each category of error. The idea is to store the information in the object and in the *name* of its class so that someone in a calling context can figure out what to do with your exception. Comment

Catching an exception

As mentioned earlier, 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. Comment

The try block

If you're inside a function and you throw an exception (or a called function throws an exception), the function exits 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 block 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**: Comment

```
try {
  // Code that may generate exceptions
}
```

If you check for errors by carefully examining the return codes from the functions you use, you need 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 and handle exceptions after the **try** block. Thus, 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. Comment

Exception handlers

Of course, the thrown exception must end up some place. This place is the *exception handler*, and you need one exception handler for every exception type you want to catch. Exception handlers immediately follow the **try** block and are denoted by the keyword **catch**: Comment

```
try {
  // Code that may generate exceptions
} catch(type1 id1) {
  // Handle exceptions of type1
} catch(type2 id2) {
  // Handle exceptions of type2
} catch(type3 id3)
  // Etc...
} catch(typeN idN)
```

```
  // Handle exceptions of typeN
}
// Normal execution resumes here...
```

The syntax of a **catch** clause resembles functions that take a single argument. The identifier (**id1**, **id2**, and so on) can be used inside the handler, just like a function argument, although you can omit the identifier if it's not needed in the handler. The exception type usually gives you enough information to deal with it. Comment

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. It then enters that **catch** clause, and the exception is considered handled. (The search for handlers stops once the **catch** clause is found.) Only the matching **catch** clause executes; control then resumes after the last handler associated with that try block. Comment

Notice that, within the **try** block, a number of different function calls might generate the same type of exception, but you need only one handler. Comment

To illustrate using **try** and **catch**, the following variation of **Nonlocal.cpp** replaces the call to **setjmp( )** with a **try** block and replaces the call to **longjmp( )** with a **throw** statement. Comment

```
//: C01:Nonlocal2.cpp
// Illustrates exceptions
#include <iostream>
using namespace std;

class Rainbow {
public:
  Rainbow() { cout << "Rainbow()" << endl; }
  ~Rainbow() { cout << "~Rainbow()" << endl; }
};

void oz() {
  Rainbow rb;
```

```
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  throw 47;
}

int main() {
  try {
    cout << "tornado, witch, munchkins...\n";
    oz();
  }
  catch (int) {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
} ///:~
```

When the **throw** statement in **oz( )** executes, program control backtracks until it finds the **catch** clause that takes an **int** parameter, at which point execution resumes with the body of that **catch** clause. The most important difference between this program and **Nonlocal.cpp** is that the destructor for the object **rb** is called when the **throw** statement causes execution to leave the function **oz( )**. Comment

There are two basic models in exception-handling theory: termination and resumption. In *termination* (which is what C++ supports), you assume the error is so critical that there's no way to automatically resume execution at the point where the exception occurred. In other words, "whoever" threw the exception decided there was no way to salvage the situation, and they don't *want* to come back. Comment

The alternative error-handling model is called *resumption*, first introduced with the PL/I language in the 1960s[2]. Using resumption semantics means that the exception handler is expected to do

[2] Visual Basic supports a limited form of resumptive exception handling with its ON ERROR facility.

something to rectify the situation, and then the faulting code is automatically retried, presuming success the second time. If you want resumption in C++, you must explicitly transfer execution back to the code where the error occurred, usually by repeating the function call that sent you there in the first place. It is not unusual, therefore, to place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory. Comment

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. 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 is 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 on which the exception can be generated from many points. Comment

Exception matching

When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they appear in the source code. When it finds a match, the exception is considered handled and no further searching occurs. Comment

Matching an exception doesn't require a perfect correlation 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"— truncated to the base type — as it is passed to the handler; this does no damage but loses all the derived-type information.) For this reason, as well as to avoid making yet another copy of the exception object, it is always better to catch an exception by *reference* instead of by value[3]. If a pointer is thrown, the usual standard pointer conversions are used to match the exception. However, no automatic type conversions are used to convert from one exception type to another in the process of matching, for example: Comment

```
//: C01:Autoexcp.cpp
// No matching conversions
#include <iostream>
using namespace std;

class Except1 {};
class Except2 {
public:
  Except2(const 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. Comment

The following example shows how a base-class handler can catch a derived-class exception: Comment

[3] In fact, you might want to always specify exception objects by const reference in exception handlers. It's very rare to modify and rethrow an exception. We are not dogmatic about this practice however.

```
//: C01: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 that is a* **Trouble** (through public inheritance[4]), 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. Comment

Notice that these examples catch exceptions by reference, although for these classes it isn't important because there are no additional

[4] Only *unambiguous, accessible base classes* can catch derived exceptions. This rule minimizes the runtime overhead needed to validate exceptions. Remember that exceptions are checked at runtime, not at compile time, and therefore the extensive information available at compile time is not available during exception handling.

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 value arguments in your handlers to avoid slicing off information. Comment

Catching any exception

Sometimes you want to create a handler that *catches* any type of exception. You do this using the ellipsis in the argument list: Comment

```
catch(...) {
  cout << "an exception was thrown" << endl;
}
```

An ellipsis catches 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. Comment

Because the ellipsis gives you no possibility to have an argument, you can't know anything about the exception or its type. It's a "catchall." Such a **catch** clause is often used to clean up some resources and then rethrow the exception. Comment

Re-throwing an exception

You usually want to re-throw an exception when you have some resource that needs to be released, such as a network connection or heap memory that needs to be deallocated. (See the section "Resource Management" later in this chapter for more detail). If an exception occurs, you don't necessarily care what error caused the exception—you just want to close the connection you opened previously. After that, you'll want to let some other context closer to the user (that is, higher up in the call chain) handle the exception. In this case the ellipsis specification is just what you want. You want to catch *any* exception, clean up your resource, and then re-throw the exception so that it can be handled elsewhere. You re-throw an exception by using **throw** with no argument inside a handler: Comment

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Deallocate your resource here, and then re-
throw…
   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 can extract any information the object may contain. Comment

Uncaught exceptions

As we explained in the beginning of this chapter, exception handling is considered better than the traditional return-an-error-code technique because exceptions can't be ignored. 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 did not catch the exception. (The location of this **try** block is not always obvious at first glance, since it's higher up in the call chain.) 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. Comment

The **terminate( )** function

If no handler at any level catches the exception, the special library function **terminate( )** (declared in the **<exception>** header) is automatically called. By default, **terminate( )** calls the Standard C library function **abort( )**, which abruptly exits the program. On Unix systems, **abort( )** also causes a core dump. When **abort( )** is called, no calls to normal program termination functions occur, which means that destructors for global and static objects do not execute. The **terminate( )** function also executes if a destructor for a local object throws an exception during stack unwinding (interrupting the exception that was in progress) or if a global or static object's constructor or destructor throws an exception. In general, do not allow a destructor to throw an exception. Comment

The **set_terminate( )** function

You can install your own **terminate( )** function using the standard **set_terminate( )** function, which returns a pointer to the **terminate( )** function you are replacing (which will be the default library version the first time you call it), 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 execute some sort of program-termination logic. If **terminate( )** is called, the problem is unrecoverable. Comment

The following example shows the use of **set_terminate( )**. Here, the return value is saved and restored so that the **terminate( )** function can be used to help isolate the section of code in which the uncaught exception is occurring: Comment

```
//: C01:Terminator.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;
  exit(0);
}

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 might be familiar with seeing a semicolon right after a pointer-to-function declaration, here it's just another kind of variable and can be initialized when it is defined. Comment

The class **Botch** not only throws an exception inside **f( )**, but also in its destructor. As we explained earlier, this situation 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. 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 usually a sign of poor design or sloppy coding. Comment

Cleaning up

Part of the magic of exception handling is that you can pop from normal program flow into the appropriate exception handler. Doing so wouldn't be 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. Comment

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: Comment

```
//: C01:Cleanup.cpp
// Exceptions clean up complete objects only
#include <iostream>
using namespace std;

class Trace {
  static int counter;
  int objid;
public:
  Trace() {
    objid = counter++;
    cout << "constructing Trace #" << objid <<
endl;
    if(objid == 3) throw 3;
  }
  ~Trace() {
    cout << "destructing Trace #" << objid <<
endl;
  }
};

int Trace::counter = 0;

int main() {
  try {
    Trace n1;
    // Throws exception:
    Trace array[5];
    Trace n2;  // won't get here
  } catch(int i) {
    cout << "caught " << i << endl;
  }
```

```
} ///:~
```

The class **Trace** keeps track of objects so that you can trace program progress. It keeps a count of the number of objects created with a **static** data member **counter** and tracks the number of the particular object with **objid** Comment

The main program creates a single object, **n1** (**objid** 0), and then attempts to create an array of five **Trace** objects, but an exception is thrown before the third object is fully created. The object **n2** is never created. You can see the results in the output of the program: Comment

```
constructing Trace #0
constructing Trace #1
constructing Trace #2
constructing Trace #3
destructing Trace #2
destructing Trace #1
destructing Trace #0
caught 3
```

Three array elements are successfully created, but in the middle of the constructor for the fourth element, an exception is thrown. Because the fourth construction in **main( )** (for **array[3]**) never completes, only the destructors for objects 1 and 2 are called. Finally, object **n1** is destroyed, but not object **n2**, because it was never created. Comment

Resource management

When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will my resources be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a particular problem: if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object. Thus, you must be especially diligent while writing your constructor. Comment

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: Comment

```
//: C01:Rawp.cpp
// Naked pointers
#include <iostream>
using namespace std;

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
};

class Dog {
public:
  void* operator new(size_t sz) {
    cout << "allocating a Dog" << endl;
    throw 47;
  }
  void operator delete(void* p) {
    cout << "deallocating a Dog" << endl;
    ::operator delete(p);
  }
};

class UseResources {
  Cat* bp;
  Dog* op;
public:
  UseResources(int count = 1) {
    cout << "UseResources()" << endl;
    bp = new Cat[count];
    op = new Dog;
  }
  ~UseResources() {
    cout << "~UseResources()" << endl;
    delete [] bp; // Array delete
    delete op;
  }
```

```
};

int main() {
  try {
    UseResources ur(3);
  } catch(int) {
    cout << "inside handler" << endl;
  }
} ///:~
```

The output is the following: Comment

```
UseResources()
Cat()
Cat()
Cat()
allocating a Dog
inside handler
```

The **UseResources** constructor is entered, and the **Cat** constructor is successfully completed for the three array objects. However, inside **Dog::operator new( )**, an exception is thrown (to simulate 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 also means the **Cat** objects that were successfully created on the heap were never destroyed. Comment

Making everything an object

To prevent such resource leaks, you must guard against these "raw" resource allocations in one of two ways:

- You can catch exceptions inside the constructor and then release the resource.
- You can place the allocations inside an object's constructor, and you can place the deallocations inside an object's destructor.

Using the latter approach, each allocation becomes atomic, by virtue of being part of the lifetime of a local object, and if it fails, the other resource allocation objects are properly cleaned up during stack unwinding. This technique is called Resource Acquisition Is Initialization (RAII for short), because it equates resource control with object lifetime. Using templates is an excellent way to modify the previous example to achieve this: Comment

```
//: C01:Wrapped.cpp
// Safe, atomic pointers
#include <iostream>
using namespace std;

// 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];
    cout << "PWrap constructor" << endl;
  }
  ~PWrap() {
    delete [] ptr;
    cout << "PWrap destructor" << endl;
  }
  T& operator[](int i) throw(RangeError) {
    if(i >= 0 && i < SZ) return ptr[i];
    throw RangeError();
  }
};

class Cat {
public:
  Cat() { cout << "Cat()" << endl; }
  ~Cat() { cout << "~Cat()" << endl; }
  void g() {}
};

class Dog {
public:
```

```
  void* operator new[](size_t) {
    cout << "Allocating a Dog" << endl;
    throw 47;
  }
  void operator delete[](void* p) {
    cout << "Deallocating a Dog" << endl;
    ::operator delete(p);
  }
};

class UseResources {
  PWrap<Cat, 3> cats;
  PWrap<Dog> dog;
public:
  UseResources() {
    cout << "UseResources()" << endl;
  }
  ~UseResources() {
    cout << "~UseResources()" << endl;
  }
  void f() { cats[1].g(); }
};

int main() {
  try {
    UseResources ur;
  } catch(int) {
    cout << "inside handler" << endl;
  } catch(...) {
    cout << "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 during stack unwinding. Comment

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 simple and assumes all the necessary information is in the class name, but you might also want to add a member that contains the value of the index, if that is useful. Comment

Now the output is: Comment

```
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. Comment

auto_ptr

Since dynamic memory is the most frequent resource used in a typical C++ program, the standard provides an RAII wrapper for pointers to heap memory that automatically frees the memory. The **auto_ptr** class template, defined in the **<memory>** header, has a constructor that takes a pointer to its generic type (whatever you use in your code). The **auto_ptr** class template also overloads the pointer operators ***** and **->** to forward these operations to the original pointer the **auto_ptr** object is holding. You can, therefore, use the **auto_ptr** object as if it were a raw pointer. Here's how it works: Comment

```
//: C01:Auto_ptr.cpp
// Illustrates the RAII nature of auto_ptr
#include <memory>
#include <iostream>
using namespace std;

class TraceHeap {
  int i;
public:
  static void* operator new(size_t siz) {
    void* p = ::operator new(siz);
    cout << "Allocating TraceHeap object on the
heap "
         << "at address " << p << endl;
    return p;
  }
  static void operator delete(void* p) {
    cout << "Deleting TraceHeap object at address
"
         << p << endl;
    ::operator delete(p);
  }
  TraceHeap(int i) : i(i) {}
  int getVal() const {
    return i;
  }
};

int main() {
  auto_ptr<TraceHeap> pMyObject(new TraceHeap(5));
  cout << pMyObject->getVal() << endl;  // prints
5
} ///:~
```

The **TraceHeap** class overloads the **operator new** and **operator delete** so you can see exactly what's happening. Notice that, like any other class template, you specify the type you're going to use in a template parameter. You don't say **TraceHeap***, however; **auto_ptr** already knows that it will be storing a pointer to your type. You must provide the original pointer when the **auto_ptr** object is initialized; you can't assign it later because **auto_ptr** doesn't provide such an assignment operator. The second line of **main( )** verifies that **auto_ptr**'s **operator->( )** function applies the indirection to the original, underlying pointer. Most important, even though we didn't explicitly delete the original pointer (in fact we can't here, since we didn't save its address in a variable anywhere), **pMyObject**'s destructor deletes the original pointer during stack unwinding, as the following output verifies: Comment

```
Allocating TraceHeap object on the heap at address
8930040
5
Deleting TraceHeap object at address 8930040
```

The auto_ptr class template is also handy for pointer data members. Since class objects contained by value are always destructed, **auto_ptr** members always delete the raw pointer they wrap when the containing object is destructed[5]. Comment

Function-level try blocks

Since constructors can routinely throw exceptions, you might want to handle exceptions that occur when an object's member or base subobjects are initialized. To do this, you can place the initialization of such subobjects in a *function-level try block*. In a departure from the usual syntax, the **try** block for constructor initializers is the constructor body, and the associated **catch** block follows the body of the constructor, as in the following example. Comment

```
//: C01:InitExcept.cpp
// Handles exceptions from subobjects
//{-bor}
#include <iostream>
using namespace std;
```

[5] For more detail on **auto_ptr**, see Herb Sutter's article entitled, "Using auto_ptr Effectively" in the September 1999 issue of the *C/C++ Users Journal*, pp. 63–67.

```
class Base {
  int i;
public:
  class BaseExcept {};

  Base(int i) : i(i) {
    throw BaseExcept();
  }
};

class Derived : public Base {
public:
  class DerivedExcept {
    const char* msg;
  public:
    DerivedExcept(const char* msg) : msg(msg) {}
    const char* what() const {
      return msg;
    }
  };
  Derived(int j)
  try
    : Base(j) {
    // Constructor body
    cout << "This won't print\n";
  }
  catch (BaseExcept&) {
    throw DerivedExcept("Base subobject threw");;
  }
};

int main() {
  try {
    Derived d(3);
  }
  catch (Derived::DerivedExcept& d) {
    cout << d.what() << endl;  // "Base subobject
threw"
  }
} ///:~
```

Notice that the initializer list in the constructor for **Derived** goes after the **try** keyword but before the constructor body. If an exception does indeed occur, the contained object is not constructed, so it makes no sense to return to the code that created it. For this reason, the only sensible thing to do is to throw an exception in the function-level **catch** clause. Comment

Although it is not terribly useful, C++ also allows function-level **try** blocks for *any* function, as the following example illustrates:

```
//: C01:FunctionTryBlock.cpp
// Function-level try blocks
//{-bor}
#include <iostream>
using namespace std;

int main() try {
  throw "main";
} catch(const char* msg) {
  cout << msg << endl;
  return 1;
} ///:~
```

In this case, the **catch** block can return in the same manner that the function body normally returns. Using this type of function-level **try** block isn't much different from inserting a **try-catch** around the code inside of the function body. Comment

Standard exceptions

The set of exceptions used with the Standard C++ library is also available for your 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 exactly what you need, you can derive from it. Comment

All standard exception classes derive ultimately from the class **exception**, defined in the header **<exception>**. The two main derived classes are **logic_error** and **runtime_error**, which are found in **<stdexcept>** (which itself includes **<exception>**). The class **logic_error** represents errors in programming logic, such as passing an invalid argument. Runtime errors are those that occur as the result of unforeseen forces such as hardware failure or memory exhaustion. Both **runtime_error** and **logic_error** provide a constructor that takes a **std::string** argument so that you can store a message in the exception object and extract it later with **exception::what( )**, as the following program illustrates. Comment

```
//: C01:StdExcept.cpp
// Derives an exception class from
std::runtime_error
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
  MyError(const string& msg = "") :
runtime_error(msg) {}
};

int main() {
  try {
    throw MyError("my message");
  }
  catch (MyError& x) {
    cout << x.what() << endl;
  }
} ///:~
```

Although the **runtime_error** constructor passes the message up to its **std::exception** subobject to hold, **std::exception** does not provide a constructor that takes a **std::string** argument. Therefore, you usually want to derive your exception classes from either **runtime_error** or **logic_error** (or one of their derivatives), and not from **std::exception**. Comment

The following tables describe the standard exception classes.

exception	The base class for all the exceptions thrown by the C++ standard library. You can ask **what( )** and retrieve the optional string with which the exception was initialized.
logic_error	Derived from **exception**. Reports program logic errors, which could presumably be detected by inspection.
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. Comment

You can use the classes in both of the following tables as they are, or you can use them as base classes from which to derive your own more specific types of exceptions. Comment

Exception classes derived from **logic_error**	
domain_error	Reports violations of a precondition.
invalid_argument	Indicates an invalid argument to the function from which it's thrown.
length_error	Indicates an attempt to produce an object whose

Exception classes derived from **logic_error**	
	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 runtime type identification (see Chapter 8).
bad_typeid	Reports a null pointer **p** in an expression **typeid(*p)**. (Again, a runtime type identification feature in Chapter 8).

Comment

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.

Exception specifications

You're not required to inform the people using your function what exceptions you might throw. Failure to do so can be considered uncivilized, however, because it means that users cannot be sure what code to write to catch all potential exceptions. Of course, if they have your source code, they can hunt through and look for **throw** statements, but often a library doesn't come with sources. Good documentation can help alleviate this problem, but how many software projects are well documented? C++ provides syntax that allows you to tell the user what exceptions this function throws, so the user can handle them. This is the optional *exception specification*, which adorns a function's declaration, appearing after the argument list. Comment

The exception specification reuses the keyword **throw**, followed by a parenthesized list of all the types of potential exceptions that the function can throw. Your function declaration might look like this: Comment

```
void f() throw(toobig, toosmall, divzero);
```

As far as exceptions are concerned, the traditional function declaration

```
void f();
```

means that *any* type of exception can be thrown from the function. If you say

```
void f() throw();
```

no exceptions whatsoever will be thrown from the function (so you'd better be sure that no functions farther down in the call chain let any exceptions propagate up!).Comment

For good coding policy, good documentation, and ease-of-use for the function caller, always consider using exception specifications when you write functions that throw exceptions. (Exceptions to this guideline are discussed later in this chapter.)Comment

The **unexpected( )** function

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. Should this unfortunate situation occur, the default implementation of **unexpected** calls the **terminate( )** function mentioned earlier in this chapter. Comment

The **set_unexpected( )** function

Like **terminate( )**, the **unexpected( )** mechanism allows you to install your own function to respond to unexpected exceptions. You do so with a function called **set_unexpected( ),** which, like **set_terminate( )**, takes the address of a function with no arguments and **void** return value. Also, because it returns the previous value of the **unexpected( )** pointer, 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 the features discussed so far in this section: Comment

```
//: C01:Unexpected.cpp
// Exception specifications & unexpected()
//{-msc} Doesn't terminate properly
#include <exception>
#include <iostream>
#include <cstdlib>
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

void my_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(0);
}

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 small, but they can certainly hold additional information so that the handlers can query for it. Comment

The **f( )** function 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 if someone changes **g( )** so that it throws a different type of exception (like the second version in this example, which throws an **int**), the exception specification for **f( )** is violated. Comment

The **my_unexpected( )** function has no arguments or return value, following the proper form for a custom **unexpected( )** function. It simply displays a message so that you can see that it has been called and then exits the program (**exit(0)** is used here so that the book's **make** process is not aborted). Your new **unexpected( )** function should not have a **return** statement. Comment

In **main( )**, the **try** block is within a **for** loop, so all the possibilities are exercised. In this way, you can 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. Comment

Only the **Up** and **Fit** exceptions are caught because those are the only exceptions that the programmer of **f( )** said would be thrown. Version two of **g( )** causes **my_unexpected( )** to be called because **f( )** then throws an **int**. Comment

In the call to **set_unexpected( )**, the return value is ignored, but it can also be saved in a pointer to function and be restored later, as we did in the **set_terminate( )** example earlier in this chapter. Comment

A typical **unexpected** handler logs the error and terminates the program by calling **exit( )**. It can, however, throw another exception (or re-throw the same exception) or call **abort( )**. If it throws an exception of a type allowed by the function whose specification was originally violated, the search resumes at the *call* of the function with this exception specification. (This behavior is unique to **unexpected( )**.)

If the exception thrown from your **unexpected** handler is not allowed by the original function's specification, one of the following occurs:

1. If **std::bad_exception** (defined in **<exception>**) was in the function's exception specification, the exception thrown from the unexpected handler is replaced with a **std::bad_exception** object, and the search resumes from the function as before.

2. If the original function's specification did not include **std::bad_exception**, **terminate( )** is called.

The following program illustrates this behavior. Comment

```
//: C01:BadException.cpp
//{-bor}
#include <exception>    // for std::bad_exception
#include <iostream>
#include <cstdio>
using namespace std;

// Exception classes:
class A {};
class B {};

// terminate() handler
void my_thandler() {
  cout << "terminate called\n";
  exit(0);
}

// unexpected() handlers
void my_uhandler1() {
  throw A();
}
void my_uhandler2() {
  throw;
}

// If we embed this throw statement in f or g,
// the compiler detects the violation and reports
// an error, so we put it in its own function.
void t() {
  throw B();
}

void f() throw(A) {
  t();
}
void g() throw(A, bad_exception) {
  t();
}

int main() {
  set_terminate(my_thandler);
  set_unexpected(my_uhandler1);
```

```
  try {
    f();
  }
  catch (A&) {
    cout << "caught an A from f\n";
  }
  set_unexpected(my_uhandler2);
  try {
    g();
  }
  catch (bad_exception&) {
    cout << "caught a bad_exception from g\n";
  }
  try {
    f();
  }
  catch (...) {
    cout << "This will never print\n";
  }
} ///:~
```

The **my_uhandler1( )** handler throws an acceptable exception (**A**), so execution resumes at the first catch, which succeeds. The **my_uhandler2( )** handler does not throw a valid exception (**B**), but since **g** specifies **bad_exception**, the **B** exception is replaced by a **bad_exception** object, and the second catch also succeeds. Since **f** does not include **bad_exception** in its specification, **my_thandler( )** is called as a terminate handler. Thus, the output from this program is as follows: Comment

```
caught an A from f
caught a bad_exception from g
terminate called
```

Better exception specifications?

You may feel that 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 might think he or she *should* have to say Comment

```
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 might be linked into new, exception-throwing versions). Hence, the uninformative situation whereby Comment

```
void f();
```

means, "Maybe I'll throw an exception, maybe I won't." This ambiguity is necessary to avoid hindering code evolution. If you want to specify that **f** throws no exceptions, you must use the empty list, as in: Comment

```
void f() throw();
```

Exception specifications and inheritance

Each public function in a class essentially forms a contract with the user; if you pass it certain arguments, it will perform certain operations and/or return a result. The same contract must hold true in derived classes; otherwise the expected "is-a" relationship between derived and base classes is violated. Since exception specifications are logically part of a function's declaration, they too must remain consistent across an inheritance hierarchy. For example, if a member function in a base class says it will only throw an exception of type **A**, an override of that function in a derived class must not add any other exception types to the specification list, because that would result in unexpected exceptions for the user, breaking any programs that adhere to the base class interface.

You can, however, specify *fewer* exceptions or *none at all*, since that doesn't require the user to do anything differently. You can also specify anything that "is-a" **A** in place of **A** in the derived function's specification. Here's an example. Comment

```
// C01:Covariance.cpp
// Compile Only!
//{-msc}
#include <iostream>
using namespace std;

class Base {
public:
  class BaseException {};
  class DerivedException : public BaseException
{};
  virtual void f() throw (DerivedException) {
    throw DerivedException();
  }
  virtual void g() throw (BaseException) {
    throw BaseException();
  }
};

class Derived : public Base {
public:
  void f() throw (BaseException) {
    throw BaseException();
  }
  virtual void g() throw (DerivedException) {
    throw DerivedException();
  }
};
```

A compiler should flag the override of **Derived::f( )** with an error (or at least a warning) since it changes its exception specification in a way that violates the specification of **Base::f( )**. The specification for **Derived::g( )** is acceptable because **DerivedException** "is-a" **BaseException** (not the other way around). You can think of **Base/Derived** and **BaseException/DerivedException** as parallel class hierarchies; when you are in **Derived**, you can replace references to **BaseException** in exception specifications and return values with **DerivedException**. This behavior is called *covariance* (since both sets of classes vary down their respective hierarchies together). (Reminder from Volume 1: parameter types are *not* covariant—you are not allowed to change the signature of an overridden virtual function.) Comment

When *not* to use exception specifications

If you peruse the function declarations throughout the Standard C++ library, you'll find that not a single exception specification occurs anywhere! Although this might seem strange, there is a good reason for this seeming incongruity: the library consists mainly of templates, and you never know what a generic might do. For example, suppose you are developing a generic stack template and attempt to affix an exception specification to your pop function, like this:

```
T pop() throw(logic_error);
```

Since the only error you anticipate is a stack underflow, you might think it's safe to specify a **logic_error** or some other appropriate exception type. But since you don't know much about the type **T**, what if its copy constructor could possibly throw an exception (it's not unreasonable, after all)? Then **unexpected( )** would be called, and your program would terminate. The point is that you shouldn't make guarantees that you can't stand behind. If you don't know what exceptions might occur, don't use exception specifications. That's why template classes, which constitute 90 percent of the Standard C++ library, do not use exception specifications—they specify the exceptions they know about in *documentation* and leave the rest to you. Exception specifications are mainly for non-template classes. Comment

Exception safety

Speaking of popping a stack, in Chapter 7 we'll take an in-depth look at the containers in the Standard C++ library, including the stack container. One thing you'll notice is that the declaration of the **pop( )** member function looks like this:

```
void pop();
```

You might think it strange that **pop( )** doesn't return a value. Instead, it just removes the element at the top of the stack. To retrieve the top value, you must call **top( )** before you call **pop( )**. There is an important reason for this behavior, and it has to do with *exception safety*, a crucial consideration in library design. Comment

Suppose you are implementing a stack with a dynamic array (we'll call it **data** and the counter integer **count**), and you try to write **pop( )** so that it returns a value. The code for such a **pop( )** might look something like this:

```
template<class T>
T stack<T>::pop() {
  if (count == 0)
    throw logic_error("stack underflow");
  else
    return data[--count];
}
```

What happens if the copy constructor that is called for the return value in the last line throws an exception when the value is returned? The popped element is not returned because of the exception, and yet **count** has already been decremented, so the top element you wanted has been lost forever! The problem is that this function attempts to do two things at once: (1) return a value, and (2) change the state of the stack. It is better to separate these two actions into two separate member functions, which is exactly what the standard **stack** class does. (In other words, follow the time-worn design practice of *cohesion*—every function should do *one thing well*.) Exception-safe code leaves objects in a consistent state and does not leak resources. Comment

You also need to be careful writing custom assignment operators. In Chapter 12 of Volume 1, you saw that **operator=** should adhere to the following pattern:

1. Make sure you're not assigning to self. If you are, go to step 6. (This is strictly an optimization.)

2. Allocate new memory required by pointer data members.

3. Copy data from the old memory to the new.

4. Delete the old memory.

5. Update the object's state by assigning the new heap pointers to the pointer data members.

6. Return ***this**.

It's important to not change the state of your object until all the new pieces have been safely allocated and initialized. A good technique is to move all of steps 2 and 3 into a separate function, often called **clone( )**. The following example does this for a class that has two pointer members, **theString** and **theInts**. Comment

```
//: C01:SafeAssign.cpp
//  Shows an Exception-safe operator=
#include <iostream>
#include <new>       // For std::bad_alloc
#include <cstring>
using namespace std;

// A class that has two pointer members using the
heap
class HasPointers {
  // A Handle class to hold the data
  struct MyData {
    const char* theString;
    const int* theInts;
    size_t numInts;
    MyData(const char* pString, const int* pInts,
           size_t nInts)
    : theString(pString), theInts(pInts),
    numInts(nInts) {}
  } *theData;  // The handle
```

```
  // clone and cleanup functions
  static MyData* clone(const char* otherString,
                       const int* otherInts,
size_t nInts){
    char* newChars = new
char[strlen(otherString)+1];
    int* newInts;
    try {
      newInts = new int[nInts];
    } catch (bad_alloc&) {
      delete [] newChars;
      throw;
    }
    try {
      // This example uses built-in types, so it
won't
      // throw, but for class types it could
throw, so we
      // use a try block for illustration. (This
is the
      // point of the example!)
      strcpy(newChars, otherString);
      for (size_t i = 0; i < nInts; ++i)
        newInts[i] = otherInts[i];
    } catch (...) {
      delete [] newInts;
      delete [] newChars;
      throw;
    }
    return new MyData(newChars, newInts, nInts);
  }
  static MyData* clone(const MyData* otherData) {
    return clone(otherData->theString,
                 otherData->theInts,
                 otherData->numInts);
  }
  static void cleanup(const MyData* theData) {
    delete [] theData->theString;
    delete [] theData->theInts;
    delete theData;
  }
public:
```

```
  HasPointers(const char* someString, const int*
someInts,
              size_t numInts) {
    theData = clone(someString, someInts,
numInts);
  }
  HasPointers(const HasPointers& source) {
    theData = clone(source.theData);
  }
  HasPointers& operator=(const HasPointers& rhs) {
    if (this != &rhs) {
      MyData* newData =
      clone(rhs.theData->theString,
            rhs.theData->theInts,
            rhs.theData->numInts);
      cleanup(theData);
      theData = newData;
    }
    return *this;
  }
  ~HasPointers() {
    cleanup(theData);
  }
  friend ostream& operator<<(ostream& os,
                             const HasPointers&
obj) {
    os << obj.theData->theString << ": ";
    for (size_t i = 0; i < obj.theData->numInts;
++i)
      os << obj.theData->theInts[i] << ' ';
    return os;
  }
};

int main() {
  int someNums[] = {1, 2, 3, 4};
  size_t someCount = sizeof someNums / sizeof
someNums[0];
  int someMoreNums[] = {5, 6, 7};
  size_t someMoreCount =
  sizeof someMoreNums / sizeof someMoreNums[0];
  HasPointers h1("Hello", someNums, someCount);
```

```
  HasPointers h2("Goodbye", someMoreNums,
someMoreCount);
  cout << h1 << endl;  // Hello: 1 2 3 4
  h1 = h2;
  cout << h1 << endl;  // Goodbye: 5 6 7
} ///:~
```

For convenience, **HasPointers** uses the **MyData** class as a handle to the two pointers. Whenever it's time to allocate more memory, whether during construction or assignment, the first **clone** function is ultimately called to do the job. If memory fails for the first call to the **new** operator, a **bad_alloc** exception is thrown automatically. If it happens on the second allocation (for **theInts**), we have to clean up the memory for **theString**—hence the first **try** block that catches a **bad_alloc** exception. The second **try** block isn't crucial here because we're just copying **int**s and pointers (so no exceptions will occur), but whenever you copy objects, their assignment operators can possibly cause an exception, in which case everything needs to be cleaned up. In both exception handlers, notice that we *rethrow* the exception. That's because we're just managing resources here; the user still needs to know that something went wrong, so we let the exception propagate up the dynamic chain. Software libraries that don't silently swallow exceptions are called *exception neutral*. You should always strive to write libraries that are both exception safe and exception neutral.[6]
Comment

If you inspect the previous code closely, you'll notice that none of the **delete** operations will throw an exception. This code actually depends on that fact. Recall that when you call **delete** on an object, the object's destructor is called. It turns out to be practically impossible, therefore, to design exception-safe code without assuming that destructors don't throw exceptions. Don't let

[6] If you're interested in a more in-depth analysis of exception safety issues, the definitive reference is Herb Sutter's *Exceptional C++*, Addison-Wesley, 2000.

destructors throw exceptions! (We're going to remind you about this once more before this chapter is done).[7] Comment

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. Comment

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 in which exceptions are *not* warranted. Probably the best advice for deciding when to use exceptions is to throw exceptions only when a function fails to meet its specification. Comment

Not for asynchronous events

The Standard C **signal( )** system and any similar system handle asynchronous events: events that happen outside the flow of a program, and thus events the program cannot anticipate. You cannot use C++ exceptions to handle asynchronous events because the exception and its handler are on the same call stack. That is, exceptions rely on the dynamic chain of function calls on the program's runtime stack (dynamic scope, if you will), 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). Don't throw exceptions from interrupt handlers. Comment

[7] The library function **uncaught_exception( )** returns **true** in the middle of stack unwinding, so technically you can test **uncaught_exeption( )** for **false** and let an exception escape from a destructor. We've never seen a situation in which this constituted good design, however, so we only mention it in this footnote.

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. The typical way to handle this situation is to set a flag in the interrupt handler, and check it synchronously in the mainline code. Comment

Not for benign error conditions

If you have enough information to handle an error, it's not an exception. Take care of it in the current context rather than throwing an exception to a larger context. Comment

Also, C++ exceptions are not thrown for machine-level events such as divide-by-zero[8]. It's assumed that some other mechanism, such as the operating system or hardware, deals with these events. In this way, C++ exceptions can be reasonably efficient, and their use is isolated to program-level exceptional conditions. Comment

Not for flow-of-control

An exception looks somewhat like an alternate return mechanism and somewhat like a **switch** statement, so you might be tempted to use an exception instead of these ordinary language mechanisms. 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. Comment

You're not forced to use exceptions

Some programs are quite simple (small utilities, for example). You might only need to take input and perform some processing. In these programs, you might attempt to allocate memory and fail, try to open a file and fail, and so on. It is acceptable in these programs to display a message and exit the program, allowing the system to

[8] Some compilers do throw exceptions in these cases, but they usually provide a compiler option to disable this (unusual) behavior.

clean up the mess, rather than to work 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 use them. Comment

New exceptions, old code

Another situation that arises is the modification of an existing program that doesn't use exceptions. You might 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 starightforward thing to do is surround the largest block that uses the new library (this might 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. It's even better, of course, to isolate your exception-generating code in a **try** block and write handlers to convert the exceptions into your existing error-handling scheme. Comment

It's truly important to think about exceptions when you're creating a library for someone else to use, especially in situations in which you can't know how they need to respond to critical error conditions (recall the earlier discussions on exception safety and why there are no exception specifications in the Standard C++ Library). Comment

Typical uses of exceptions

Do use exceptions to do the following:

- Fix the problem and call the function which caused the exception again.
- Patch things up and continue without retrying the function.
- Do whatever you can in the current context and rethrow the *same* exception to a higher context.
- Do whatever you can in the current context and throw a *different* exception to a higher context.

- Terminate the program.
- Wrap functions (especially C library functions) that use ordinary error schemes so they produce exceptions instead.
- Simplify. If your exception scheme makes things more complicated, it is painful and annoying to use.
- Make your library and program safer. This is a short-term investment (for debugging) and a long-term investment (for application robustness). Comment

When to 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 might come out of this function so that it can detect violations at runtime. Comment

Of course, you can't always look at the code and anticipate which 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 get a call to **unexpected( )**. Any time you use exception specifications or call functions that do, consider creating your own **unexpected( )** function that logs a message and then either throws an exception or aborts the program. Comment

As we explained earlier, you should avoid using exception specifications in template classes, since you can't anticipate what types of exceptions the template parameter classes might throw. Comment

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. Comment

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 if your users can always write their code to expect the **what( )** function defined in the **exception( )** class interface. Comment

Nest your own exceptions

If you create exceptions for your particular class, it's a good idea to nest the exception classes either inside your class or inside a namespace containing 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 global namespace. Comment

You can nest your exceptions even if you're deriving them from C++ standard exceptions. Comment

Use exception hierarchies

Using exception hierarchies is a valuable way to classify the types of critical errors that might 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. Comment

Of course, the Standard C++ exceptions are a good example of an exception hierarchy and one on which you can build. Comment

Multiple inheritance (MI)

As you'll read in Chapter 9, the only *essential* place for MI is if you need to upcast an object pointer to two different base classes—that is, if you need polymorphic behavior with both of those base classes. It turns out that exception hierarchies are useful places for multiple inheritance because a base-class handler from any of the roots of the multiply inherited exception class can handle the exception. Comment

Catch by reference, not by value

We explained in the section "Exception matching" earlier that you should catch exceptions by reference for two reasons:

- To avoid making a needless copy of the exception object when it is passed to the handler,
- To avoid object slicing when catching a derived exception as a base class object

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 might have occurred from heap exhaustion. If you throw exception objects, the exception-handling system takes care of all storage. Comment

Throw exceptions in constructors

Because a constructor has no return value, you've previously had two ways to report an error during construction: Comment

- Set a nonlocal flag and hope the user checks it.
- Return an incompletely created object and hope the user checks it.

This problem is serious because C programmers have come to rely on an implied guarantee that object creation is always successful, which is not unreasonable in C in which 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. Comment

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, a new exception can be thrown *before* the catch-clause for an existing exception is reached, which will cause a call to **terminate( )**. Comment

If you call any functions inside a destructor that can 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. Comment

Avoid naked pointers

See **Wrapped.cpp** earlier in this chapter. 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 aren't released if an exception is thrown in the constructor. Use **auto_ptr** for pointers that reference heap memory. Comment

Overhead

When an exception is thrown, there's considerable runtime overhead (but it's *good overhead*, since objects are cleaned up automatically!). For this 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 this is actually true depends on the particular compiler implementation you're using. (See the description of the "zero-cost model" later in this section.) Comment

You can think of a **throw** expression as a call to a special system function that takes the exception object as an argument and backtracks up the chain of execution. For this to work, extra information needs to be put on the stack by the compiler, to aid in stack unwinding. To understand this, you need to know about the runtime stack. Whenever a function is called, information about that function is pushed onto the runtime stack in an *activation record instance* (ARI), also called a *stack frame*. A typical stack frame contains the address of the calling function (so execution can return to it), a pointer to the ARI of the function's static parent (the scope that lexically contains the called function, so variables global to the function can be accessed), and a pointer to the function that called it (its *dynamic parent*). The path that logically results from repetitively following the dynamic parent links is the *dynamic chain*, or *call chain*, that we've mentioned previously in this chapter. This is how execution can backtrack when an exception is thrown, and it is the mechanism that makes it possible for components developed without knowledge of one another to communicate errors at runtime. Comment

To enable stack unwinding for exception handling, extra exception-related information about each function needs to be available for each stack frame. This information describes which destructors need to be called (so that local objects can be cleaned up), indicates whether the current function has a **try** block, and lists which exceptions the associated catch clauses can handle. Naturally there is space penalty for this extra information, so programs that support exception handling can be somewhat larger than those that don't[9]. Even the compile-time size of programs using exception handling is greater, since the logic of how to generate the expanded stack frames during runtime must be generated by the compiler. Comment

To illustrate this, we compiled the following program both with and without exception-handling support in Borland C++ Builder and Microsoft Visual C++.[10]

```
struct HasDestructor {
  ~HasDestructor(){}
};

void g();    // for all we know, g may throw
```

[9] This depends, of course, on how much checking of return codes you would have to insert if you weren't using exceptions.

[10] Borland enables exceptions by default; to disable exceptions use the **-x-** compiler option. Microsoft disables support by default; to turn it on, use the **-GX** option. With both compilers use the **-c** option to compile only.

```
void f() {
   HasDestructor h;
   g();
}
```

If exception handling is enabled, the compiler must keep information about **~HasDestructor( )** available at runtime in the ARI for **f( )** (so it can destroy **h** properly should **g( )** throw an exception). The following table summarizes the result of the compilations in terms of the size of the compiled (.obj) files (in bytes). Comment

Compiler\Mode	**With Exception Support**	**Without Exception Support**
Borland	616	234
Microsoft	1162	680

Don't take the percentage differences between the two modes too seriously. Remember that exceptions (should) typically constitute a small part of a program, so the space overhead tends to be much smaller (usually between 5 and 15 percent).Comment

You might think that this extra housekeeping would slow down execution, and you'd be correct. A clever compiler implementation can avoid that cost, however. Since information about exception-handling code and the offsets of local objects can be computed once at compile time, such information can be kept in a single place associated with each function, but not in each ARI. You essentially remove exception overhead from each ARI and, therefore, avoid the extra time to push them onto the stack. This approach is called the *zero-cost* model[11] of exception handling, and the optimized storage mentioned earlier is known as the *shadow stack*.[12] Comment

Summary

Error recovery is a fundamental concern for every program you write, and it's especially important in C++, in which one of the goals is to create program components for others to use. To create a robust system, each component must be robust. Comment

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. Comment

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. Comment

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 displaying its description string.

[11] The GNU C++ compiler uses the zero-cost model by default. Metrowerks Code Warrior for C++ also has an option to use the zero-cost model.

[12] Thanks to Scott Meyers and Josee Lajoie for their insights on the zero-cost model. You can find more information on how exceptions work in Josee's excellent article, "Exception Handling: Behind the Scenes," *C++ Gems*, SIGS, 1996.

2. Rewrite the Stash class from Chapter 13 of Volume 1 so that 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 ten objects, and on the eleventh object "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. Write a function with an exception specification that can throw four exception types: a **char**, an **int**, a **bool**, and your own exception class. Catch each in **main( )** and verify the catch. Derive your exception class from a standard exception. Write the function in such a way that the system recovers and tries to execute it again.
9. Modify your solution to the exercise 8 to throw a double from the function, violating the exception specification. Catch the violation with your own unexpected handler that displays a message and exits the program gracefully (meaning **abort( )** is not called).
10. Write a **Garage** class that has a **Car** that is having troubles with its **Motor**. Use a function-level **try** block in the **Garage** class constructor to catch an exception (thrown from the **Motor** class) when its **Car** object is

initialized. Throw a different exception from the body of the **Garage** constructor's handler and catch it in **main( )**.

2: Defensive programming

Writing “perfect software” may be an elusive Holy Grail for developers, but a few defensive techniques, routinely applied, can go a long way toward narrowing the gap between code and ideal.

Although the complexity of typical production software guarantees that testers will always have a job, chances are you still yearn to produce defect-free software. (At least we hope you do!) Object-oriented design techniques do much to corral the difficulty of large projects, to be sure. Eventually, however, you have to get down to writing loops and functions. These details of “programming in the small” become the building blocks of the implementation of larger components called for by your design efforts. If your loops are off by one or your functions calculate the correct values only “most” of the time, you’re in deep trouble no matter how fancy your overall methodology. In this chapter, we’re interested in coding practices that keep you on track toward a working solution regardless of the size of your project. Comment

Your code is, among other things, an expression of your attempt to solve a problem. It should be clear to the reader (including yourself) exactly what you were thinking when you designed that loop. At certain points in your program, you should be able to make bold statements that some condition or other holds. (If you can’t, you really haven’t yet solved the problem.) Such statements are called *invariants*, since they should invariably be true at the point where they appear in the code; if not, either your design is faulty, or your code does not accurately reflect your design. (In other words, you’ve got bugs!) Comment

To illustrate, consider how to write a program that plays the guessing game of Hi-lo. You play this game by having one person think of a number between 1 and 100, and having the other person guess the number. (We’ll let the computer do the guessing.) The person who holds the number tells the guesser whether their guess is high, low or correct. The best strategy for the guesser is of course *binary search*, which chooses the midpoint of the range of numbers where the sought-after number resides. The high-low response tells the guesser which half of the list holds the number, and the process repeats, halving the size of the active search range on each iteration. So how do you write a loop to drive the repetition properly? It's not sufficient to just say Comment

```
bool guessed = false;
while (!guessed) {
  …
}
```

because a malicious user might respond deceitfully, and you could spend all day guessing. What assumption, however simple, are you making each time you guess? In other words, what condition should hold *by design* on each loop iteration? Comment

The simple assumption we're after is, of course, that the secret number is within the current active range of unguessed numbers, beginning with the range [1, 100]. Suppose we label the endpoints of the range with the variables ***low*** and ***high***. Each time you pass through the loop you need to make sure that if the number was in the range [**low**, **high**] at the beginning of the loop, you calculate the new range so that it still contains the number at the end of the current loop iteration. Comment

The goal is to express the loop invariant in code so that a violation can be detected at runtime. Unfortunately, since the computer doesn't know the secret number, you can't express this condition directly in code, but you can at least make a comment to that effect:

```
while (!guessed) {
  // INVARIANT: the number is in the range [low,
high]
  …
}
```

If we were to stop this thread of discussion right here, we would have accomplished a great deal if it helps clarify how you design loops. Fortunately, we can do better than that. What happens when the user says that a guess is too high when it isn't or that it's too low when it in fact is not? The deception will in effect exclude the secret number from the new subrange. Because one lie always leads to another, eventually your range will diminish to nothing (since you shrink it by half each time and the secret number isn't in there). We can easily express this condition concretely, as the following program illustrates. Comment

```
//: C02:HiLo.cpp
// Plays the game of Hi-lo to illustrate a loop
invariant
#include <cstdlib>
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "Think of a number between 1 and 100\n";
  cout << "I will make a guess; ";
  cout << "tell me if I'm (H)igh or (L)ow\n";
  int low = 1, high = 100;
  bool guessed = false;
  while (!guessed) {
    // Invariant: the number is in the range [low,
high]
    if (low > high) {  // Invariant violation
      cout << "You cheated! I quit\n";
      return EXIT_FAILURE;
    }
    int guess = (low + high) / 2;
    cout << "My guess is " << guess << ". ";
    cout << "(H)igh, (L)ow, or (E)qual? ";
    string response;
    cin >> response;
    switch(toupper(response[0])) {
      case 'H':
        high = guess - 1;
        break;
```

```
      case 'L':
        low = guess + 1;
        break;
      case 'E':
        guessed = true;
        break;
      default:
        cout << "Invalid response\n";
        continue;
    }
  }
  cout << "I got it!\n";
  return EXIT_SUCCESS;
} ///:~
```

The violation of the invariant is easily detected with the condition **if (low > high)**, because if the user always tells the truth, we will always find the secret number before we run out of numbers to guess from. (See the last paragraph of the text that follows the program **extractCode.cpp** at the end of Chapter 3 for an explanation of the macros **EXIT_FAILURE** and **EXIT_SUCCESS**). Comment

Assertions

The condition in the Hi-lo program depends on user input, so you're powerless to always prevent a violation of the invariant. Most often, however, invariants depend only on the code you write, so they will always hold, if you've implemented your design correctly. In this case, it is clearer to make an *assertion*, which is a positive statement that reveals your design decisions. Comment

For example, suppose you are implementing a vector of integers, which, as you know, is an expandable array that grows on demand. The function that adds an element to the vector must first verify that there is an open slot in the underlying array that holds the elements; otherwise, it needs to request more heap space and copy the existing elements to the new space before adding the new element (and of course deleting the old array). Such a function might look like the following: Comment

```
void MyVector::push_back(int x) {
   if (nextSlot == capacity)
      grow();
   assert(nextSlot < capacity);
   data[nextSlot++] = x;
}
```

In this example, **data** is a dynamic array of **int**s with **capacity** slots and **nextSlot** slots in use. The purpose of **grow( )** is to expand the size of **data** so that the new value of **capacity** is strictly greater than **nextSlot**. Proper behavior of **MyVector** depends on this design decision, and it will never fail if the rest of the supporting code is correct, so we *assert* the condition with the **assert( )** macro (defined in the header **<cassert>**). Comment

The Standard C library **assert( )** macro is brief, to the point, and portable. If the condition in its parameter evaluates to non-zero, execution continues uninterrupted; if it doesn't, a message containing the text of the offending expression along with its source file name and line number is printed to the standard error channel and the program aborts. Is that too drastic? In practice, it is much more drastic to let execution continue when a basic design assumption has failed. Your program needs to be fixed. Comment

If all goes well, you will have thoroughly tested your code with all assertions intact by the time the final product is deployed. (We'll say more about testing later.) Depending on the nature of your application, the machine cycles needed to test all assertions at runtime might be too much of a performance hit in the field. If that's the case, you can remove all the assertion code automatically by defining the macro **NDEBUG** and rebuilding the application. Comment

To see how this works, note that a typical implementation of **assert( )** looks something like this:

```
#ifdef NDEBUG
```

```
  #define assert(cond) ((void)0)
#else
  void assertImpl(const char*, const char*, long);
  #define assert(cond) \
    ((cond) ? (void)0 : assertImpl(???))
#endif
```

When the macro **NDEBUG** is defined, the code decays to the expression **(void) 0**, so all that's left in the compilation stream is an essentially empty statement as a result of the semicolon you appended to each **assert( )** invocation. If **NDEBUG** is not defined, **assert(cond)** expands to a conditional statement that, when **cond** is zero, calls a compiler-dependent function (which we named **assertImpl( )**) with a string argument representing the text of **cond**, along with the file name and line number where the assertion appeared. (We used "???" as a place holder in the example, but the string mentioned is actually computed there, along with the file name and the line number where the macro occurs in that file. How these values are obtained is immaterial to our discussion.) If you want to turn assertions on and off at different points in your program, you not only have to **#define** or **#undef NDEBUG**, but you have to re-include **<cassert>**. Macros are evaluated as the preprocessor encounters them and therefore use whatever **NDEBUG** state applies at that point in time. The most common way to define **NDEBUG** once for an entire program is as a compiler option, whether through project settings in your visual environment or via the command line, as in

```
mycc –DNDEBUG myfile.cpp
```

Most compilers use the **–D** flag to define macro names. (Substitute the name of your compiler's executable for **mycc** above.) The advantage of this approach is that you can leave your assertions in the source code as an invaluable bit of documentation, and yet there is no runtime penalty. Because the code in an assertion disappears when **NDEBUG** is defined, it is important that you *never do work in an assertion*. Only test conditions that do not change the state of your program. Comment

Whether using **NDEBUG** for released code is a good idea remains a subject of debate. Tony Hoare, one of the most influential computer scientists of all time,[1] has suggested that turning off runtime checks such as assertions is similar to a sailing enthusiast who wears a life jacket while training on land and then discards it when he actually goes to sea.[2] If an assertion fails in production, you have a problem much worse than degradation in performance, so choose wisely. Comment

Not all conditions should be enforced by assertions, of course. User errors and runtime resource failures should be signaled by throwing exceptions, as we explained in detail in Chapter 1. It is tempting to use assertions for most error conditions while roughing out code, with the intent to replace many of them later with robust exception handling. Like any other temptation, use caution, since you might forget to make all the necessary changes later. Remember: assertions are intended to verify design decisions that will only fail because of faulty programmer logic. The ideal is to solve all assertion violations during development. You shouldn't use assertions for conditions that aren't totally in your control (for example, conditions that depend on user input). In particular, you wouldn't want to use assertions to validate function arguments; throw a **logic_error** instead. Comment

The use of assertions as a tool to ensure program correctness was formalized by Bertrand Meyer in his *Design by Contract* methodology.[3] Every function has an implicit contract with clients that, given certain *pre-conditions*, guarantees certain *post-conditions*. In other words, the pre-conditions are the requirements for using the function, such as supplying arguments within certain ranges, and the post-conditions are the results delivered by the function, either by return value or by side-effect. Comment

---

[1] Among other things he invented Quicksort.

[2] As quoted in *Programming Language Pragmatics*, by Michael L. Scott, Morgan-Kaufmann, 2000.

[3] See his book, *Object-Oriented Software Construction*, Prentice-Hall, 1994.

What should you do when clients fail to give you valid input? They have broken the contract, and you need to let them know. As we mentioned earlier, this is not the best time to abort the program (although you're justified in doing so since the contract was violated), but an exception is certainly in order. This is why the Standard C++ library throws exceptions derived from **logic_error**, such as **out_of_range.**[4] If there are functions that only you call, however, such as private functions in a class of your own design, the **assert( )** macro is appropriate, since you have total control over the situation and you certainly want to debug your code before shipping. Comment

Since post-conditions are totally your responsibility, you might think assertions also apply, and you would be partially right. It is appropriate to use an assertion for *any invariant at any time*, including when a function has finished its work. This especially applies to class member functions that maintain the state of an object. In the **MyVector** example earlier, for instance, a reasonable invariant for all public member functions would be

```
assert(0 <= nextSlot && nextSlot <= capacity);
```

or, if **nextSlot** is an unsigned integer, simply

```
assert(nextSlot <= capacity);
```

Such an invariant is called a *class invariant* and can reasonably be enforced by an assertion. Subclasses play the role of *subcontractor* to their base classes in that they must maintain the original contract the base class has with its clients. For this reason, the pre-conditions in derived classes must impose no extra requirements

[4] This is still an assertion *conceptually*, but since we don't want to halt execution, the **assert( )** macro is not appropriate. Java 1.4, for example, throws an exception when an assertion fails.

beyond those in the base contract, and the post-conditions must deliver at least as much.[5] Comment

Validating results returned to the client, however, is nothing more or less than *testing*, so using post-condition assertions in this case would be duplicating work. There's nothing wrong with it; it's just an exercise in redundancy. Yes, it's good documentation, but more than one developer has been fooled into using post-condition assertions as a *substitute* for unit testing. Bad idea! Comment

The simplest automated unit test framework that could possibly work

Writing software is all about meeting requirements. It doesn't take much experience, however, to figure out that coming up with requirements in the first place is no easy task, and, more important, requirements are not static. It's not unheard of to discover at a weekly project meeting that what you just spent the week doing is not exactly what the users really want. Comment

Frustrating? Yes. Reasonable? Also, yes! It is unreasonable to expect mere humans to be able to articulate software requirements in detail without sampling an evolving, working system. It's much better to specify a little, design a little, code a little, test a little. Then, after evaluating the outcome, do it all over again. The ability to develop from soup to nuts in such an iterative fashion is one of the great advances of this object-oriented era in software history. It requires nimble programmers who can craft resilient code. Change is hard. Comment

[5] There is a nice phrase to help remember this phenomenon: "Require no more; promise no less," first coined in *C++ FAQs*, by Marshall Cline and Greg Lomow (Addison-Wesley, 1994). Since pre-conditions can weaken in derived classes, we say that they are *contravariant*, and, conversely, post-conditions are *covariant* (which explains why we mentioned the covariance of exception specifications in Chapter 1).

Ironically, another impetus for change comes from you, the programmer. The craftsperson in you likely has the habit of continually improving the physical design of working code. What maintenance programmer hasn't had occasion to curse the aging, flagship company product as a convoluted patchwork of spaghetti, wholly resistant to modification? Management's knee-jerk reluctance to let you tamper with a functioning system, while not totally unfounded, robs code of the resilience it needs to endure. "If it ain't broke, don't fix it" eventually gives way to, "We can't fix it—rewrite it." Change is necessary. Comment

Fortunately, our industry has finally gotten used to the discipline of *refactoring*, the art of internally restructuring code to improve its design, without changing the functionality visible to the user.[6] Such tweaks include extracting a new function from another, or its inverse, combining member functions; replacing a member function with an object; parameterizing a member function or class; and replacing conditionals with polymorphism. Refactoring helps code embrace evolution. Comment

Whether the force for change comes from users or programmers, however, there is still the risk that changes today will break what worked yesterday. What is needed is a way to build code that withstands the winds of change and actually improves over time. Comment

Many practices purport to support such a quick-on-your-feet motif, of which Extreme Programming is only one.[7] In this section we explore what we think is the key to making flexible, incremental

[6] A good read on this subject is Martin Fowler's *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 2000). See also www.refactoring.com. Refactoring is a crucial practice of Extreme Programming (XP). The title of this section is a variation on the theme, “TheSimplestThingThatCouldPossiblyWork,” another XP staple. XP is a code-centric discipline for getting software done right, on time, within budget, while having fun along the way. Visit www.xprogramming.com for more detail.

[7] Lightweight methodologies such as XP have “joined forces” in the Agile Alliance (see http://www.agilealliance.org/home).

development succeed: a ridiculously easy-to-use automated unit test framework. (Please note that we in no way mean to de-emphasize the role of *testers*, software professionals who test others' code for a living. They are indispensable. We are merely describing a way to help developers write better code.) Comment

Developers write *unit tests* to gain the confidence to say the two most important things that any developer can say:

1. I understand the requirements.

2. My code meets those requirements to the best of my knowledge.

There is no better way ensure that you know what the code you're about to write should do than to write the unit tests first. This simple exercise helps focus the mind on the task ahead and will likely lead to working code faster than just jumping into coding. Or, to express it in XP terms: Testing + Programming is *faster* than just Programming. Writing tests first also puts you on guard up front against boundary conditions that might cause your code to break, so your code is more robust right out of the chute. Comment

Once your code passes all your tests, you have the peace of mind that if the system you contribute to isn't working, it's not your fault. The statement "All my tests pass" is a powerful trump card in the workplace that cuts through any amount of politics and hand waving. Comment

Automated testing

So what does a unit test look like? Too often developers just use some well-behaved input to produce some expected output, which they inspect visually. Two dangers exist in this approach. First, programs don't always receive only well-behaved input. We all know that we should test the boundaries of program input, but it's hard to think about this when you're trying to just get things working. If you write the test for a function first before you start coding, you can wear your "tester hat" and ask yourself, "What could possibly make this break?" Code a test that will prove the function you'll write isn't broken, and then put on your developer hat and make it happen. You'll write better code than if you hadn't written the test first. Comment

The second danger is that inspecting output visually is tedious and error prone. Most any such thing a human can do a computer can do, but without human error. It's better to formulate tests as collections of *Boolean expressions* and have a test program report any failures. Comment

For example, suppose you need to build a **Date** class that has the following properties:

- A date can be initialized with a string (YYYYMMDD), three integers (Y, M, D), or nothing (giving today's date).
- A date object can yield its year, month, and day or a string of the form "YYYYMMDD".
- All relational comparisons are available, as well as computing the duration between two dates (in years, months, and days).
- Dates to be compared need to be able to span an arbitrary number of centuries (for example, 1600–2200).

Your class can store three integers representing the year, month, and day. (Just be sure the year is at least 16 bits in size to satisfy the last bulleted item.) The interface for your **Date** class might look like this: Comment

```
// A first pass at Date.h
#ifndef DATE_H
#define DATE_H
#include <string>

class Date {
public:
```

```
  // A struct to hold elapsed time:
  struct Duration {
    int years;
    int months;
    int days;
    Duration(int y, int m, int d)
      : years(y), months(m), days(d) {}
  };
  Date();
  Date(int year, int month, int day);
  Date(const std::string&);
  int getYear() const;
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
  friend bool operator<(const Date&, const Date&);
  friend bool operator>(const Date&, const Date&);
  friend bool operator<=(const Date&, const Date&);
  friend bool operator>=(const Date&, const Date&);
  friend bool operator==(const Date&, const Date&);
  friend bool operator!=(const Date&, const Date&);
  friend Duration duration(const Date&, const
Date&);
};
#endif
```

Before you even think about implementation, you can solidify your grasp of the requirements for this class by writing the beginnings of a test program. You might come up with something like the following:

```
//: C02:SimpleDateTest.cpp
//{L} Date
// You'll need the full Date.h from the Appendix:
#include "Date.h"
#include <iostream>
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) {
  if(t) nPass++; else nFail++;
```

```
}

int main() {
  Date mybday(1951, 10, 1);
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
}
/* Expected output:
Passed: 3, Failed: 0
*/ ///:~
```

In this trivial case, the function **test( )** maintains the global variables **nPass** and **nFail**. The only visual inspection you do is to read the final score. If a test failed, a more sophisticated **test( )** displays an appropriate message. The framework described later in this chapter has such a test function, among other things. Comment

You can now implement enough of the **Date** class to get these tests to pass, and then you can proceed iteratively in like fashion until all the requirements are met. By writing tests first, you are more likely to think of corner cases that might break your upcoming implementation, and you're more likely to write the code correctly the first time. Such an exercise might produce the following "final" version of a test for the **Date** class: Comment

```
//: C02:SimpleDateTest2.cpp
//{L} Date
#include <iostream>
#include "Date.h"
using namespace std;

// Test machinery
int nPass = 0, nFail = 0;
void test(bool t) {
  if(t) nPass++; else nFail++;
}

int main() {
```

```
  Date mybday(1951, 10, 1);
  Date today;
  Date myevebday("19510930");

  // Test the operators
  test(mybday < today);
  test(mybday <= today);
  test(mybday != today);
  test(mybday == mybday);
  test(mybday >= mybday);
  test(mybday <= mybday);
  test(myevebday < mybday);
  test(mybday > myevebday);
  test(mybday >= myevebday);
  test(mybday != myevebday);

  // Test the functions
  test(mybday.getYear() == 1951);
  test(mybday.getMonth() == 10);
  test(mybday.getDay() == 1);
  test(myevebday.getYear() == 1951);
  test(myevebday.getMonth() == 9);
  test(myevebday.getDay() == 30);
  test(mybday.toString() == "19511001");
  test(myevebday.toString() == "19510930");

  // Test duration
  Date d2(2002, 7, 4);
  Date::Duration dur = duration(mybday, d2);
  test(dur.years == 50);
  test(dur.months == 9);
  test(dur.days == 3);

  // Report results:
  cout << "Passed: " << nPass << ", Failed: "
       << nFail << endl;
} ///:~
```

The word “final” above was quoted because this test can of course be more fully developed. For example we haven’t tested that long durations are handled correctly. To save space on the printed page we’ll stop here, but you get the idea. The full implementation for the

Date class is available in the files **Date.h** and **Date.cpp** in the appendix and on the Mindview website[8]. Comment

The TestSuite Framework

Some automated C++ unit test tools are available on the World Wide Web for download, such as **CppUnit.**[9] These are well designed and implemented, but our purpose here is not only to present a test mechanism that is easy to use, but also easy to understand internally and even tweak if necessary. So, in the spirit of “TheSimplestThingThatCouldPossiblyWork,” we have developed the *TestSuite Framework*, a namespace named **TestSuite** that contains two key classes: **Test** and **Suite**. Comment

The **Test** class is an abstract class you derive from to define a test object. It keeps track of the number of passes and failures for you and displays the text of any test condition that fails. Your main task in defining a test is simply to override the **run( )** member function, which should in turn call the **test_( )** macro for each Boolean test condition you define. Comment

To define a test for the **Date** class using the framework, you can inherit from **Test** as shown in the following program:

```
//: C02:DateTest.h
#ifndef DATE_TEST_H
#define DATE_TEST_H
#include "Date.h"
#include "../TestSuite/Test.h"

class DateTest : public TestSuite::Test {
  Date mybday;
  Date today;
  Date myevebday;
public:
```

---

[8] Our Date class is also “internationalized”, in that it supports wide character sets. This is introduced at the end of the next chapter.

[9] See http://sourceforge.net/projects/cppunit for more information.

```
  DateTest() : mybday(1951, 10, 1),
myevebday("19510930") {
  }
  void run() {
    testOps();
    testFunctions();
    testDuration();
  }
  void testOps() {
    test_(mybday < today);
    test_(mybday <= today);
    test_(mybday != today);
    test_(mybday == mybday);
    test_(mybday >= mybday);
    test_(mybday <= mybday);
    test_(myevebday < mybday);
    test_(mybday > myevebday);
    test_(mybday >= myevebday);
    test_(mybday != myevebday);
  }
  void testFunctions() {
    test_(mybday.getYear() == 1951);
    test_(mybday.getMonth() == 10);
    test_(mybday.getDay() == 1);
    test_(myevebday.getYear() == 1951);
    test_(myevebday.getMonth() == 9);
    test_(myevebday.getDay() == 30);
    test_(mybday.toString() == "19511001");
    test_(myevebday.toString() == "19510930");
  }
  void testDuration() {
    Date d2(2002, 7, 4);
    Date::Duration dur = duration(mybday, d2);
    test_(dur.years == 50);
    test_(dur.months == 9);
    test_(dur.days == 3);
  }
};
#endif ///:~
```

Running the test is a simple matter of instantiating a **DateTest** object and calling its **run( )** member function. Comment

```
//: C02:DateTest.cpp
// Automated Testing (with a Framework)
//{L} Date ../TestSuite/Test
#include <iostream>
#include "DateTest.h"
using namespace std;

int main() {
  DateTest test;
  test.run();
  return test.report();
}
/* Output:
Test "DateTest":
        Passed: 21,      Failed: 0
*/ ///:~
```

The **Test::report( )** function displays the previous output and returns the number of failures, so it is suitable to use as a return value from **main( )**. Comment

The **Test** class uses RTTI[10] to get the name of your class (for example, **DateTest**) for the report. There is also a **setStream( )** member function if you want the test results sent to a file instead of to the standard output (the default). You'll see the **Test** class implementation later in this chapter. Comment

The **test_ ( )** macro can extract the text of the Boolean condition that fails, along with its file name and line number.[11] To see what happens when a failure occurs, you can introduce an intentional error in the code, say by reversing the condition in the first call to **test_( )** in **DateTest::testOps( )** in the previous example code.

---

[10] "Runtime Type Identification", discussed in chapter 9. Specifically, we use the **name( )** member function of the **typeinfo** class. By the way, if you're using Microsoft Visual C++, you need to specify the compile option **/GR**. If you don't, you'll get an access violation at runtime.

[11] In particular, we use *stringizing* (via the **#** operator) and the predefined macros **__FILE__** and **__LINE__**. See the code later in the chapter.

The output indicates exactly what test was in error and where it happened: Comment

```
DateTest failure: (mybday > today) , DateTest.h
(line 31)
Test "DateTest":
        Passed: 20      Failed: 1
```

In addition to **test_( )**, the framework includes the functions **succeed_( )** and **fail_( )**, for cases in which a Boolean test won't do. These functions apply when the class you're testing might throw exceptions. During testing, you want to arrange an input set that will cause the exception to occur to make sure it's doing its job. If it doesn't, it's an error, in which case you call **fail_( )** explicitly to display a message and update the failure count. If it does throw the exception as expected, you call **succeed_ ( )** to update the success count. Comment

To illustrate, suppose we update the specification of the two non-default **Date** constructors to throw a **DateError** exception (a type nested inside **Date** and derived from **std::logic_error**) if the input parameters do not represent a valid date: Comment

```
Date(const string& s) throw(DateError);
Date(int year, int month, int day)
throw(DateError);
```

The **DateTest::run( )** member function can now call the following function to test the exception handling:

```
  void testExceptions() {
    try {
      Date d(0,0,0);  // Invalid
      fail_("Invalid date undetected in Date int
ctor");
    }
    catch (Date::DateError&) {
      succeed_();
    }
    try {
      Date d("");  // Invalid
```

```
      fail_("Invalid date undetected in Date
string ctor");
    }
    catch (Date::DateError&) {
      succeed_();
    }
  }
```

In both cases, if an exception is not thrown, it is an error. Notice that you have to manually pass a message to **fail_( )**, since no Boolean expression is being evaluated. Comment

Test suites

Real projects usually contain many classes, so you need a way to group tests so that you can just push a single button to test the entire project[12]. The **Suite** class allows you to collect tests into a functional unit. You derive **Test** objects to a **Suite** with the **addTest( )** member function, or you can swallow an entire existing suite with **addSuite( )**. We have a number of date-related classes to illustrate how to use a test suite. Here's an actual test run: Comment

```
// Illustrates a suite of related tests
#include <iostream>
#include "suite.h"          // includes test.h
#include "JulianDateTest.h"
#include "JulianTimeTest.h"
#include "MonthInfoTest.h"
#include "DateTest.h"
#include "TimeTest.h"
using namespace std;

int main() {
    Suite s("Date and Time Tests");
    s.addTest(new MonthInfoTest);
    s.addTest(new JulianDateTest);
    s.addTest(new JulianTimeTest);
```

[12] Batch files and shell scripts work well for this, of course. The **Suite** class is a C++-based way of organizing related tests.

```
    s.addTest(new DateTest);
    s.addTest(new TimeTest);
    s.run();
    long nFail = s.report();
    s.free();
    return nFail;
}
/* Output:
Suite "Date and Time Tests"
===========================
Test "MonthInfoTest":
   Passed: 18 Failed: 0
Test "JulianDateTest":
   Passed: 36 Failed: 0
Test "JulianTimeTest":
   Passed: 29 Failed: 0
Test "DateTest":
   Passed: 57 Failed: 0
Test "TimeTest":
   Passed: 84 Failed: 0
===========================
*/
```

Each of the five test files included as headers tests a unique date component. You must give the suite a name when you create it. The **Suite::run( )** member function calls **Test::run( )** for each of its contained tests. Much the same thing happens for **Suite::report( )**, except that it is possible to send the individual test reports to a destination stream that is different from that of the suite report. If the test passed to **addSuite( )** has a stream pointer assigned already, it keeps it. Otherwise, it gets its stream from the **Suite** object. (As with **Test**, there is a second argument to the suite constructor that defaults to **std::cout**.) The destructor for **Suite** does not automatically delete the contained **Test** pointers because they don't have to reside on the heap; that's the job of **Suite::free( )**. Comment

The test framework code

The test framework code library is in a subdirectory called **TestSuite** in the code distribution available on the Mindview website. To use it, therefore, the **TestSuite** subdirectory in your header must include the search path, you must link the object files, and thus you must also include the **TestSuite** subdirectory in the library search path. Comment

Here is the header for **Test.h**:

```
//: TestSuite:Test.h
#ifndef TEST_H
#define TEST_H
#include <string>
#include <iostream>
#include <cassert>
using std::string;
using std::ostream;
using std::cout;

// The following have underscores because
// they are macros. For consistency,
// succeed_() also has an underscore.

#define test_(cond) \
  do_test(cond, #cond, __FILE__, __LINE__)
#define fail_(str) \
  do_fail(str, __FILE__, __LINE__)

namespace TestSuite {

class Test {
public:
  Test(ostream* osptr = &cout);
  virtual ~Test(){}
  virtual void run() = 0;
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const;
  void setStream(ostream* osptr);
  void succeed_();
  long report() const;
  virtual void reset();
protected:
  void do_test(bool cond, const string& lbl,
```

```
    const char* fname, long lineno);
  void do_fail(const string& lbl,
    const char* fname, long lineno);
private:
  ostream* osptr;
  long nPass;
  long nFail;
  // Disallowed:
  Test(const Test&);
  Test& operator=(const Test&);
};

inline Test::Test(ostream* osptr) {
  this->osptr = osptr;
  nPass = nFail = 0;
}

inline long Test::getNumPassed() const {
  return nPass;
}

inline long Test::getNumFailed() const {
  return nFail;
}

inline const ostream* Test::getStream() const {
  return osptr;
}

inline void Test::setStream(ostream* osptr) {
  this->osptr = osptr;
}

inline void Test::succeed_() {
  ++nPass;
}

inline void Test::reset() {
  nPass = nFail = 0;
}

} // namespace TestSuite
```

```
#endif // TEST_H ///:~
```

There are three virtual functions in the **Test** class:

- A virtual destructor
- The function **reset( )**
- The pure virtual function **run( )**

As explained in Volume 1, it is an error to delete a derived heap object through a base pointer unless the base class has a virtual destructor. Any class intended to be a base class (usually evidenced by the presence of at least one other virtual function) should have a virtual destructor. The default implementation of the **Test::reset( )** resets the success and failure counters to zero. You might want to override this function to reset the state of the data in your derived test object; just be sure to call **Test::reset( )** explicitly in your override so that the counters are reset. The **Test::run( )** member function is pure virtual, of course, since you are required to override it in your derived class. Comment

The **test_( )** and **fail_( )** macros can include file name and line number information available from the preprocessor. We originally omitted the trailing underscores in the names, but the original **fail( )** macro collided with **ios::fail( )**, causing all kinds of compiler errors. Comment

Here is the implementation of **Test**:

```
//: TestSuite:Test.cpp {O}
#include "Test.h"
#include <iostream>
#include <typeinfo>  // Note: Visual C++ requires
/GR""
using namespace std;
using namespace TestSuite;

void Test::do_test(bool cond,
  const std::string& lbl, const char* fname,
```

```
  long lineno) {
  if (!cond)
    do_fail(lbl, fname, lineno);
  else
    succeed_();
}

void Test::do_fail(const std::string& lbl,
  const char* fname, long lineno) {
  ++nFail;
  if (osptr) {
    *osptr << typeid(*this).name()
           << "failure: (" << lbl << ") , "
           << fname
           << " (line " << lineno << ")\n";
  }
}

long Test::report() const {
  if (osptr) {
    *osptr << "Test \"" << typeid(*this).name()
           << "\":\n\tPassed: " << nPass
           << "\tFailed: " << nFail
           << endl;
  }
  return nFail;
} ///:~
```

No rocket science here. The **Test** class just keeps track of the number of successes and failures as well as the stream where you want **Test::report( )** to display the results. The **test_( )** and **fail_( )** macros extract the current file name and line number information from the preprocessor and pass the file name to **do_test( )** and the line number to **do_fail( )**, which do the actual work of displaying a message and updating the appropriate counter. We can't think of a good reason to allow copy and assignment of test objects, so we have disallowed these operations by making their prototypes private and omitting their respective function bodies.
Comment

Here is the header file for **Suite**:Comment

```
//: TestSuite:Suite.h
#ifndef SUITE_H
#define SUITE_H
#include "../TestSuite/Test.h"
#include <vector>
#include <stdexcept>
using std::vector;
using std::logic_error;

namespace TestSuite {

class TestSuiteError : public logic_error {
public:
  TestSuiteError(const string& s = "")
    : logic_error(s) {}
};

class Suite {
public:
  Suite(const string& name, ostream* osptr =
&cout);
  string getName() const;
  long getNumPassed() const;
  long getNumFailed() const;
  const ostream* getStream() const;
  void setStream(ostream* osptr);
  void addTest(Test* t) throw (TestSuiteError);
  void addSuite(const Suite&);
  void run();  // Calls Test::run() repeatedly
  long report() const;
  void free();  // Deletes tests
private:
  string name;
  ostream* osptr;
  vector<Test*> tests;
  void reset();
  // Disallowed ops:
  Suite(const Suite&);
  Suite& operator=(const Suite&);
};

inline
```

```
Suite::Suite(const string& name, ostream* osptr)
   : name(name) {
  this->osptr = osptr;
}

inline string Suite::getName() const {
  return name;
}

inline const ostream* Suite::getStream() const {
  return osptr;
}

inline void Suite::setStream(ostream* osptr) {
  this->osptr = osptr;
}

} // namespace TestSuite
#endif // SUITE_H ///:~
```

The **Suite** class holds pointers to its **Test** objects in a vector. Notice the exception specification on the **addTest( )** member function. When you add a test to a suite, **Suite::addTest( )** verifies that the pointer you pass is not null; if it is null, it throws a **TestSuiteError** exception. Since this makes it impossible to add a null pointer to a suite, **addSuite( )** asserts this condition on each of its tests, as do the other functions that traverse the vector of tests (see the following implementation). Copy and assignment are disallowed as they are in the **Test** class. Comment

```
//: TestSuite:Suite.cpp {O}
#include "Suite.h"
#include <iostream>
#include <cassert>
using namespace std;
using namespace TestSuite;

void Suite::addTest(Test* t) throw(TestSuiteError)
{
  // Verify test is valid and has a stream:
  if (t == 0)
```

```
    throw TestSuiteError(
      "Null test in Suite::addTest");
  else if (osptr && !t->getStream())
    t->setStream(osptr);
  tests.push_back(t);
  t->reset();
}

void Suite::addSuite(const Suite& s) {
  for (size_t i = 0; i < s.tests.size(); ++i) {
    assert(tests[i]);
    addTest(s.tests[i]);
  }
}

void Suite::free() {
  for (size_t i = 0; i < tests.size(); ++i) {
    delete tests[i];
    tests[i] = 0;
  }
}

void Suite::run() {
  reset();
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->run();
  }
}

long Suite::report() const {
  if (osptr) {
    long totFail = 0;
    *osptr << "Suite \"" << name
             << "\"\n=======";
    size_t i;
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=\n";
    for (i = 0; i < tests.size(); ++i) {
      assert(tests[i]);
      totFail += tests[i]->report();
```

```
    }
    *osptr << "=======";
    for (i = 0; i < name.size(); ++i)
      *osptr << '=';
    *osptr << "=\n";
    return totFail;
  }
  else
    return getNumFailed();
}

long Suite::getNumPassed() const {
  long totPass = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totPass += tests[i]->getNumPassed();
  }
  return totPass;
}

long Suite::getNumFailed() const {
  long totFail = 0;
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    totFail += tests[i]->getNumFailed();
  }
  return totFail;
}

void Suite::reset() {
  for (size_t i = 0; i < tests.size(); ++i) {
    assert(tests[i]);
    tests[i]->reset();
  }
} ///:~
```

We will be using the **TestSuite** framework wherever it applies throughout the rest of this book. Comment

Debugging techniques

The best debugging habit to get into is to use assertions as explained in the beginning of this chapter; by doing so you'll be more likely to find logic errors before they cause real trouble. This section contains some other tips and techniques that might help during debugging. Comment

Trace macros

Sometimes it's helpful to print the code of each statement as it is executed, either to **cout** or to a trace file. Here's a preprocessor macro to accomplish this: Comment

```
#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: Comment

```
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 exactly what you want. Thus, you must use this technique carefully. Comment

The following is a variation on the **TRACE( )** macro:

```
#define D(a) cout << #a "=[" << a << "]" << '\n';
```

If you want to display an expression, you simply put it inside a call to **D( ).** The expression is displayed, 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 this macro any time you want to test an intermediate value to make sure things are okay. Comment

Of course, these two macros are actually just the two most fundamental things you do with a debugger: trace through the code execution and display values. A good debugger is an excellent productivity tool, but sometimes debuggers are not available, or it's not convenient to use them. These techniques always work, regardless of the situation. Comment

Trace file

DISCLAIMER: This section and the next contain code which is officially unsanctioned by the C++ standard. In particular, we redefine **cout** and **new** via macros, which can cause surprising results if you're not careful. Our examples work on all of the compilers we use, however, and provide useful information. This is the only place in this book where we will depart from the sanctity of standard-compliant coding practice. Use at your own risk!

The following 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): Comment

```
//: C03: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 previous file:

```
//: C03: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(); // Dumps file contents to
file
} ///:~
```

Finding memory leaks

The following straightforward debugging techniques are explained Volume 1.

1. For array bounds checking, use the **Array** template in **C16:Array3.cpp** of Volume 1 for all arrays. You can turn off the checking and increase efficiency when you're ready to ship. (This doesn't deal with the case of taking a pointer to an array, though—perhaps that could be made into a template somehow as well). Comment

2. Check for nonvirtual destructors in base classes. Comment

Tracking new/delete and malloc/free

Common problems with memory allocation include mistakenly calling **delete** for memory not on the free store, deleting the free store more than once, and, most often, forgetting to delete such a pointer at all. This section discusses a system that can help you track down these kinds of problems.

As an *additional disclaimer* beyond that of the preceding section: because of the way we overload **new**, the following technique may not work on all platforms, and will only work for programs that do not call the *function* **operator new( )** explicitly. We have been quite careful in this book to only present code that fully conforms to the C++ standard, but in this one instance we're making an exception for the following reasons:

1. Even though it's technically illegal, it works on many compilers[13].
2. We illustrate some useful thinking along the way.

To use the memory checking system, you simply include the header file **MemCheck.h**, link the **MemCheck.obj** file into your application, so that all the calls to **new** and **delete** are intercepted, and call the macro **MEM_ON( )** (explained later in this section) to initiate memory tracing. A trace of all allocations and deallocations is printed to the standard output (via **stdout**). When you use this system, all calls to **new** store information about the file and line where they were called. This is accomplished by using the *placement syntax* for **operator new.**[14] Although you typically use the placement syntax when you need to place objects at a specific point in memory, it also allows you to create an **operator new( )** with any number of arguments. This is used to advantage in the

---

[13] Our key technical reviewer, Pete Becker of Dinkumware. Ltd., brought to our attention that it is illegal to use macros to replace C++ keywords. His take on this technique was as follows: “"This is a dirty trick. Dirty tricks are sometimes necessary to figure out why code isn't working, so you may want to keep this in your toolbox, but don't ship any code with it." Caveat programmor :-).

[14] Thanks to Reg Charney of the C++ Standards Committee for suggesting this trick.

following example to store the results of the **__FILE__** and **__LINE__** macros whenever **new** is called: Comment

```
//: C02:MemCheck.h
#ifndef MEMCHECK_H
#define MEMCHECK_H
#include <cstddef>  // for size_t

// Hijack the new operator (both scalar and array
versions)
void* operator new(std::size_t, const char*,
long);
void* operator new[](std::size_t, const char*,
long);
#define new new (__FILE__, __LINE__)

extern bool traceFlag;
#define TRACE_ON() traceFlag = true
#define TRACE_OFF() traceFlag = false

extern bool activeFlag;
#define MEM_ON() activeFlag = true
#define MEM_OFF() activeFlag = false

#endif
///:~
```

It is important that you include this file in any source file in which you want to track free store activity, but you must include it *last* (after your other **#include** directives). Most headers in the standard library are templates, and since most compilers use the *inclusion model* of template compilation (meaning all source code is in the headers), the macro that replaces **new** in **MemCheck.h** would usurp all instances of the **new** operator in the library source code (and would likely result in compile errors). Besides, you are only interested in tracking your own memory errors, not the library's. Comment

In the following file, which contains the memory tracking implementation, everything is done with C standard I/O rather than with C++ iostreams. It shouldn't make a difference, really,

since we're not interfering with iostreams' use of the free store, but it's safer to not take a chance. (Besides, we tried it, Some compilers complained, but all compilers were happy with the **stdio** version.)
Comment

```
//: C02:MemCheck.cpp {O}
#include <cstdio>
#include <cstdlib>
#include <cassert>
using namespace std;
#undef new

// Global flags set by macros in MemCheck.h
bool traceFlag = true;
bool activeFlag = false;

namespace {

// Memory map entry type
struct Info {
  void* ptr;
  const char* file;
  long line;
};

// Memory map data
const size_t MAXPTRS = 10000u;
Info memMap[MAXPTRS];
size_t nptrs = 0;

// Searches the map for an address
int findPtr(void* p) {
  for (int i = 0; i < nptrs; ++i)
    if (memMap[i].ptr == p)
      return i;
  return -1;
}

void delPtr(void* p) {
  int pos = findPtr(p);
  assert(p >= 0);
  // Remove pointer from map
```

```
  for (size_t i = pos; i < nptrs-1; ++i)
    memMap[i] = memMap[i+1];
  --nptrs;
}

// Dummy type for static destructor
struct Sentinel {
  ~Sentinel() {
    if (nptrs > 0) {
      printf("Leaked memory at:\n");
      for (size_t i = 0; i < nptrs; ++i)
        printf("\t%p (file: %s, line %ld)\n",
          memMap[i].ptr, memMap[i].file,
memMap[i].line);
    }
    else
      printf("No user memory leaks!\n");
  }
};

// Static dummy object
Sentinel s;

} // End anonymous namespace

// Overload scalar new
void* operator new(size_t siz, const char* file,
  long line) {
  void* p = malloc(siz);
  if (activeFlag) {
    if (nptrs == MAXPTRS) {
      printf("memory map too small (increase
MAXPTRS)\n");
      exit(1);
    }
    memMap[nptrs].ptr = p;
    memMap[nptrs].file = file;
    memMap[nptrs].line = line;
    ++nptrs;
  }
  if (traceFlag) {
```

```
    printf("Allocated %u bytes at address %p ",
siz, p);
    printf("(file: %s, line: %ld)\n", file, line);
  }
  return p;
}

// Overload array new
void* operator new[](size_t siz, const char* file,
  long line) {
  return operator new(siz, file, line);
}

// Override scalar delete
void operator delete(void* p) {
  if (findPtr(p) >= 0) {
    free(p);
    assert(nptrs > 0);
    delPtr(p);
    if (traceFlag)
      printf("Deleted memory at address %p\n", p);
  }
  else if (!p && activeFlag)
    printf("Attempt to delete unknown pointer:
%p\n", p);
}

// Override array delete
void operator delete[](void* p) {
  operator delete(p);
} ///:~
```

The Boolean flags **traceFlag** and **activeFlag** are global, so they can be modified in your code by the macros **TRACE_ON( )**, **TRACE_OFF( )**, **MEM_ON( )**, and **MEM_OFF( )**. In general, enclose all the code in your **main( )** within a **MEM_ON( )**-**MEM_OFF( )** pair so that memory is always tracked. Tracing, which echoes the activity of the replacement functions for **operator new( )** and **operator delete( )**, is on by default, but you can turn it off with **TRACE_OFF( )**. In any case, the final results are always printed (see the test runs later in this chapter).

The **MemCheck** facility tracks memory by keeping all addresses allocated by **operator new( )** in an array of **Info** structures, which also holds the file name and line number where the call to **new** occurred. As much information as possible is kept inside the anonymous namespace so as not to collide with any names you might have placed in the global namespace. The **Sentinel** class exists solely to have a static object's destructor called as the program shuts down. This destructor inspects **memMap** to see if any pointers are waiting to be deleted (in which case you have a memory leak). Comment

Our **operator new( )** uses **malloc( )** to get memory, and then adds the pointer and its associated file information to **memMap**. The **operator delete( )** function undoes all that work by calling **free( )** and decrementing **nptrs**, but first it checks to see if the pointer in question is in the map in the first place. If it isn't, either you're trying to delete an address that isn't on the free store, or you're trying to delete one that's already been deleted and therefore previously removed from the map. The **activeFlag** variable is important here because we don't want to process any deallocations from any system shutdown activity. By calling **MEM_OFF( )** at the end of your code, **activeFlag** will be set to **false**, and such subsequent calls to **delete** will be ignored. (Of course, that's bad in a real program, but as we said earlier, our purpose here is to find *your* leaks; we're not debugging the library.) For simplicity, we forward all work for array **new** and **delete** to their scalar counterparts. Comment

The following is a simple test using the **MemCheck** facility.

```
//: C02:MemTest.cpp
//{L} MemCheck
// Test of MemCheck system
#include <iostream>
#include <vector>
#include <cstring>
#include "MemCheck.h"   // Must appear last!
using namespace std;
```

```
class Foo {
  char* s;
public:
  Foo(const char*s ) {
    this->s = new char[strlen(s) + 1];
    strcpy(this->s, s);
  }
  ~Foo() {
    delete [] s;
  }
};

int main() {
  MEM_ON();
  cout << "hello\n";
  int* p = new int;
  delete p;
  int* q = new int[3];
  delete [] q;
  int* r;
  delete r;
  vector<int> v;
  v.push_back(1);
  Foo s("goodbye");
  MEM_OFF();
} ///:~
```

This example verifies that you can use **MemCheck** in the presence of streams, standard containers, and classes that allocate memory in constructors. The pointers **p** and **q** are allocated and deallocated without any problem, but **r** is not a valid heap pointer, so the output indicates the error as an attempt to delete an unknown pointer.
Comment

```
hello
Allocated 4 bytes at address 0xa010778 (file:
memtest.cpp, line: 25)
Deleted memory at address 0xa010778
Allocated 12 bytes at address 0xa010778 (file:
memtest.cpp, line: 27)
Deleted memory at address 0xa010778
Attempt to delete unknown pointer: 0x1
```

```
Allocated 8 bytes at address 0xa0108c0 (file:
memtest.cpp, line: 14)
Deleted memory at address 0xa0108c0
No user memory leaks!
```

Because of the call to **MEM_OFF( )**, no subsequent calls to **operator delete( )** by **vector** or **ostream** are processed. You still might get some calls to **delete** from reallocations performed by the containers. Comment

If you call **TRACE_OFF( )** at the beginning of the program, the output is as follows:

```
hello
Attempt to delete unknown pointer: 0x1
No user memory leaks!
```

Comment

Summary

Much of the headache of software engineering can be avoided by being very deliberate about what you're doing. You've probably been using mental assertions as you've crafted your loops and functions anyway, even if you haven't routinely used the **assert( )** macro. If you'll use **assert( )**, you'll find logic errors sooner and end up with more readable code as well. Remember to only use assertions for invariants, though, and not for runtime error handling.

Nothing will give you more peace of mind than thoroughly tested code. If it's been a hassle for you in the past, use an automated framework, such as the one we've presented here, to integrate routine testing into your daily work. You (and your users!) will be glad you did.

Exercises

1. Write a test program using the **TestSuite** Framework for the standard vector class that thoroughly tests the

following member functions with a vector of integers: **push_back( )** (appends an element to the end of the vector), **front( )** (returns the first element in the vector), **back( )** (returns the last element in the vector), **pop_back( )** (removes the last element without returning it), **at( )** (returns the element in a specified index position), and **size( )** (returns the number of elements). Be sure to verify that **vector::at( )** throws a **std::out_of_range** exception if the supplied index is out of range.

2. Suppose you are asked to develop a class named **Rational** that supports rational numbers (fractions). The fraction in a **Rational** object should always be stored in lowest terms, and a denominator of zero is an error. Here is a sample interface for such a **Rational** class:

```
class Rational {
public:
   Rational(int numerator = 0, int
denominator = 1);
   Rational operator-() const;
   friend Rational operator+(const Rational&,
                             const
Rational&);
   friend Rational operator-(const Rational&,
                             const
Rational&);
   friend Rational operator*(const Rational&,
                             const
Rational&);
   friend Rational operator/(const Rational&,
                             const
Rational&);
   friend ostream& operator<<(ostream&,
                              const
Rational&);
   friend istream& operator>>(istream&,
Rational&);
   Rational& operator+=(const Rational&);
```

```
    Rational& operator-=(const Rational&);
    Rational& operator*=(const Rational&);
    Rational& operator/=(const Rational&);
    friend bool operator<(const Rational&,
                          const Rational&);
    friend bool operator>(const Rational&,
                          const Rational&);
    friend bool operator<=(const Rational&,
                           const Rational&);
    friend bool operator>=(const Rational&,
                           const Rational&);
    friend bool operator==(const Rational&,
                           const Rational&);
    friend bool operator!=(const Rational&,
                           const Rational&);
};
```

Write a complete specification for this class, including pre-conditions, post-conditions, and exception specifications.

3. Write a test using the **TestSuite** framework that thoroughly tests all the specifications from the previous exercise, including testing exceptions.
4. Implement the **Rational** class so that all the tests from the previous exercise pass. Use assertions only for invariants.
5. 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 typical heap-compaction scheme requires that all pointers are doubly indirected (that is, pointers to pointers) so that the "middle tier" pointer can be manipulated during compaction. Consider overloading **operator->** to accomplish this, since that operator has special behavior that will probably benefit your heap-compaction scheme. Write a program to test your heap-compaction scheme. (Advanced)

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 part of the book gives you an in-depth introduction to key portions of the Standard C++ library. Comment

The most complete and also the most obscure reference to the full libraries is the Standard itself. Bjarne Stroustrup's *The C++ Programming Language*, *Third Edition* (Addison-Wesley, 2000) remains a reliable reference for both the language and the library. The most celebrated library-only reference is *The C++ Standard Library: A Tutorial and Reference*, by Nicolai Josuttis (Addison-Wesley, 1999). The goal of the chapters in this part of the book is to provide you with an encyclopedia of descriptions and examples so that you'll have a good starting point for solving any problem that requires the use of the Standard libraries. However, some techniques and topics are rarely used and are not covered here. If you can't find it in these chapters, reach for the other two books; this book is not intended to replace those books but rather to complement them. In particular, we hope that after going through the material in the following chapters you'll have a much easier time understanding those books. Comment

You will notice that these chapters do not contain exhaustive documentation describing every function and class in the Standard C++ library. We've left the full descriptions to others; in particular to P.J. Plauger's *Dinkumware C/C++ Library Reference* at http://www.dinkumware.com. This is an excellent online source 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 up something. . You can view this online 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). Electronic documentation is effective not only because you can always have it with you, but also because you can do an electronic search for what you want. Comment

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 A lists additional references. Comment

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. The **string** class might 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. Comment

Chapter 4 covers the **iostreams** library, which contains classes for processing input and output with files, string targets, and the system console. Comment

Although Chapter 5, "Templates in Depth," is not explicitly a library chapter, it is necessary preparation for the two that follow. In Chapter 6 we examine the generic algorithms offered by the Standard C++ library. Because they are implemented with templates, these algorithms can be applied to any *sequence* of objects. Chapter 7 covers the standard containers and their associated iterators. We cover algorithms first because they can be fully explored by using only arrays and the vector container (which we have been using since early in Volume 1). It is also natural to use the standard algorithms in connection with containers, so it's a good idea to be familiar with the algorithm before studying the containers.

3: Strings in depth

One of the biggest time-wasters in C is using character arrays for string processing: 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.

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. Comment

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 you'll see how C++ **string**s accommodate variation in character sets and string data conversion.[1] Comment

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. No matter which programming idiom you choose, there are really only about three things you want to do with a **string**:

[1] Some of the material in this chapter was originally created by Nancy Nicolaisen.

- Create or modify the sequence of characters stored in the **string**.
- Detect the presence or absence of elements within the **string**.
- Translate between various schemes for representing **string** characters. Comment

You'll see how each of these jobs is accomplished using C++ **string** objects. Comment

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 significant differences between C++ **string**s and their C progenitors. First, and most important, C++ strings hide the physical representation of the sequence of characters they contain. You don't have to be concerned at all about array dimensions or null terminators. 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. C++ strings therefore 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. Comment

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: Comment

```
//: C03:StringStorage.cpp
//{L} ../TestSuite/Test
#include <string>
#include <iostream>
#include "../TestSuite/Test.h"
using namespace std;

class StringStorageTest : public TestSuite::Test {
public:
  void run() {
    string s1("12345");
    // This may copy the first to the second or
    // use reference counting to simulate a copy
    string s2 = s1;
    test_(s1 == s2);
    // Either way, this statement must ONLY modify
s1
    s1[0] = '6';
    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    test_(s1 != s2);
  }
};

int main() {
  StringStorageTest t;
  t.run();
  return t.report();
} ///:~
```

An implementation that only makes unique copies when a string is modified is said to use a *copy-on-write* strategy. This approach saves time and space when strings are used only as value parameters or in other read-only situations.

Whether a library implementation uses reference counting or not should be transparent to users of the **string** class. Unfortunately, this is not always the case. In multithreaded programs, it is practically impossible to use a reference-counting implementation safely.[2] Comment

Creating and initializing C++ strings

Creating and initializing strings is a straightforward proposition and fairly flexible. In the **SmallString.cpp** example in this section, 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 zero length and absence of data elements through the use of class member functions. Comment

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 do the following: Comment

- Create an empty **string** and defer initializing it with character data.

[2] It's difficult to make reference-counting implementations thread safe. (See Herb Sutter, *More Exceptional C++*, pp. 104–14). See Chapter 10 for more on programming with multiple threads.

- Initialize a **string** by passing a literal, quoted character array as an argument to the constructor.
- Initialize a **string** using the equal sign (**=**).
- Use one **string** to initialize another. Comment

```
//: C03: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 variations offer more flexibility and control. You can do the following:

- 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. Comment

Here's a program that illustrates these features.

```
//: C03:SmallString2.cpp
#include <string>
#include <iostream>
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);
  cout << s4 << endl;
  // Copy 6 chars from the middle of the source
  string s5(s2, 15, 6);
  cout << s5 << endl;
  // Copy from middle to end
  string s6(s3, 6, 15);
  cout << s6 << endl;
  // 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);
  cout << quoteMe << endl;
} ///:~
```

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 arguments have default values. 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**. Comment

Here's the output from the program:

```
What is
doing
Elvis in a UFO
What is that one clam doing with Elvis in a UFO?
```

Notice the final line of the example. C++ allows **string** initialization techniques to be mixed in a single statement, a flexible and convenient feature. Also notice that the last initializer copies *just one character* from the source **string**. Comment

Another slightly more subtle initialization technique involves the use of the **string** iterators **string::begin( )** and **string::end( ).** This technique treats a **string** like a *container* object (which you've seen primarily in the form of **vector** so far—you'll see many more containers in Chapter 7), which uses *iterators* to indicate the start and end of a sequence of characters. In this way you can hand a **string** constructor two iterators, and it copies from one to the other into the new **string**: Comment

```
//: C03:StringIterators.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

int main() {
  string source("xxx");
  string s(source.begin(), source.end());
  assert(s == source);
} ///:~
```

The iterators are not restricted to **begin( )** and **end( )**; you can increment, decrement, and add integer offsets to them, allowing you to extract a subset of characters from the source **string**. Comment

C++ strings may *not* be initialized with single characters or with ASCII or other integer values. You can initialize a string with a number of copies of a single character, however. Comment

```
//: C03:UhOh.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  // Error: no single char inits
  //! string nothingDoing1('a');
  // Error: no integer inits
  //! string nothingDoing2(0x37);
  // The following is legal:
  string okay(5, 'a');
```

```
  assert(okay == string("aaaaa"));
} ///:~
```

Operating on strings

If you've programmed in C, you are accustomed to the convenience of a large family of functions for writing, searching, modifying, and copying **char** arrays. However, there are two unfortunate aspects of the Standard C library functions for handling **char** arrays. First, there are two loosely organized families of them: the "plain" group, and the ones that require you to supply a count of the number of characters to be considered in the operation at hand. The roster of functions in the C **char** array handling library shocks the unsuspecting user with a long list of cryptic, mostly unpronounceable names. Although the kinds and number of arguments to the functions are somewhat consistent, to use them properly you must be attentive to details of function naming and parameter passing. Comment

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 little to keep the C **char** array handling functions from manipulating the memory beyond the limits of the allocated space, sometimes with disastrous results. Comment

C++ provides a vast improvement in the convenience and safety of **string** objects. For purposes of actual string handling operations, there are about the same number of distinct member function names in the **string** class as there are functions in the C library, but because of overloading there is much more functionality. Coupled with sensible naming practices and the judicious use of default arguments, these features combine to make the **string** class much easier to use than the C library. Comment

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 change size and you don't know how 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. Comment

```
//: C03: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
  // before 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 from one particular compiler: Comment

```
I saw Elvis in a UFO.
Size = 22
Capacity = 31
I thought I saw Elvis in a UFO.
Size = 32
Capacity = 47
I thought I saw Elvis in a UFO. I've been
working too hard.
Size = 59
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. Notice the ease with which we changed the size of the storage allocated to the string. The **size( )** function, of course, returns the number of characters currently stored in the string and is identical to the **length( )** member function. The **capacity( )** function returns the size of the current underlying allocation, meaning the number of characters the string can hold without requesting more storage. The **reserve( )** function is an optimization mechanism that allows you to indicate your intention to specify a certain amount of storage for future use; **capacity( )** always returns a value at least as large as the most recent call to **reserve( )**. A **resize( )** function appends spaces if the new size is greater than the current string size or truncates the string otherwise. (An overload of **resize( )** allows you to specify a different character to append.) Comment

The exact fashion in which the **string** member functions allocate space for your data depends on the implementation of the library. When we tested one implementation with the previous example, it appeared that reallocations occurred on even word (that is, full-integer) 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. Comment

Replacing string characters

The **insert( )** function 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 you want the size of the string to remain unchanged, use the **replace( )** function to overwrite characters. There are quite a number of overloaded versions of **replace( )**, but the simplest one takes three arguments: an integer indicating where to start in the string, an integer indicating 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 simple example: Comment

```
//: C03:StringReplace.cpp
// Simple find-and-replace in strings
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  assert(s == "A piece $tag$ of text");
```

```
  int start = s.find(tag);
  assert(start == 8);
  assert(tag.size() == 5);
  s.replace(start, tag.size(), "hello there");
  assert(s == "A piece hello there of text");
} ///:~
```

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**), and then it is found and replaced. Comment

You should actually check to see if you've found anything before you perform a **replace( )**. The previous example replaces with a **char***, but there's an overloaded version that replaces with a **string**. Here's a more complete demonstration **replace( ):**Comment

```
//: C03:Replace.cpp
#include <cassert>
#include <cstddef>  // for size_t
#include <string>
using namespace std;

void replaceChars(string& modifyMe,
  const string& findMe, const string& newChars) {
  // Look in modifyMe for the "find string"
  // starting at position 0
  size_t 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);
```

```
  assert(bigNews == "I thought I saw Elvis in a "
         "wig. I have been working too hard.");
} ///:~
```

If **replace** doesn't find the search string, it returns **string::npos**. The **npos** data member is a static constant member of the **string** class that represents a nonexistent character position.[3] Comment

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 needed, for example, when you make a "replacement" that would expand the original string beyond the end of the current allocation. Here's an example: Comment

```
//: C03:ReplaceAndGrow.cpp
#include <cassert>
#include <string>
using namespace std;

int main() {
  string bigNews("I have been working the
grave.");
  string replacement("yard shift.");
  // The first arg says "replace chars
  // beyond the end of the existing string":
  bigNews.replace(bigNews.size() - 1,
    replacement.size(), replacement);
  assert(bigNews == "I have been working the "
         "graveyard shift.");
} ///:~
```

The call to **replace( )** begins "replacing" beyond the end of the existing array, which is equivalent to an append operation. Notice that in this example **replace( )** expands the array accordingly. Comment

[3] It as an abbreviation for "no position."

You may have been hunting through this chapter trying to do something relatively simple such as replace all the instances of one character with a different character. Upon finding the previous material 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? Comment

You can easily write such a function using the **find( )** and **replace( )** member functions as follows:

```
//: C03:ReplaceAll.cpp {O}
#include <cstddef>
#include <string>
using namespace std;

string& replaceAll(string& context, const string&
from,
  const string& to) {
  size_t lookHere = 0;
  size_t foundHere;
  while ((foundHere = context.find(from, lookHere))
    != string::npos) {
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
  return context;
} ///:~
```

The version of **find( )** used here takes as a second argument the position to start looking in and returns **string::npos** if it doesn't find it. It is important to advance the position held in the variable **lookHere** past the replacement string, of course, in case **from** is a substring of **to**. The following program tests the **replaceAll** function: Comment

```
//: C03:ReplaceAllTest.cpp
//{-msc}
//{L} ReplaceAll
#include <iostream>
#include <cassert>
```

```
using namespace std;

string& replaceAll(string& context, const string&
from,
  const string& to);

int main() {
  string text = "a man, a plan, a canal, panama";
  replaceAll(text, "an", "XXX");
  assert(text == "a mXXX, a plXXX, a cXXXal,
pXXXama");
} ///:~
```

As you can see, the **string** class by itself doesn't solve all possible problems. Many solutions have been left to the algorithms in the Standard library,[4] because the **string** class can look just like an STL sequence (by virtue of the iterators discussed earlier). All the generic algorithms work on a "range" of elements within a container. Usually that range is just "from the beginning of the container to the end." A **string** object looks like a container of characters: 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 **replace( )** algorithm to replace all the instances of the single character 'X' with 'Y': Comment

```
//: C03:StringCharReplace.cpp
#include <algorithm>
#include <cassert>
#include <string>
using namespace std;

int main() {
  string s("aaaXaaaXXaaXXXaXXXXaaa");
  replace(s.begin(), s.end(), 'X', 'Y');
  assert(s == "aaaYaaaYYaaYYYaYYYYaaa");
} ///:~
```

[4] Discussed in depth in Chapter 6.

Notice that this **replace( )** is *not* called as a member function of **string**. Also, unlike the **string::replace( )** functions that only perform one replacement, the **replace( )** algorithm replaces *all instances* of one character with another. Comment

The **replace( )** algorithm only works with single objects (in this case, **char** objects) and will not replace quoted **char** arrays or **string** objects. Since a **string** behaves like an STL sequence, a number of other algorithms can be applied to it, which might solve other problems that are not directly addressed by the **string** member functions. Comment

Concatenation using nonmember 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 similar to adding numeric data. Comment

```
//: C03:AddStrings.cpp
#include <string>
#include <cassert>
using namespace std;

int main() {
  string s1("This ");
  string s2("That ");
  string s3("The other ");
  // operator+ concatenates strings
  s1 = s1 + s2;
  assert(s1 == "This That ");
  // Another way to concatenates strings
  s1 += s3;
  assert(s1 == "This That The other ");
  // You can index the string on the right
  s1 += s3 + s3[4] + "ooh lala";
  assert(s1 == "This That The other The other "
        "ooh lala");
```

```
} ///:~
```

Using the **operator+** and **operator+=** operators is a flexible and convenient way to combine **string** data. On the right side of the statement, you can use almost any type that evaluates to a group of one or more characters. Comment

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 :Comment

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** if no match is found. (**npos** is a const of –1 [cast as a **std::size_t**] and indicates that a search failed.)
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 *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 *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 a parameter 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 the occurrences of a given character or group of characters within the string. Comment

The following program uses the method of *The Sieve of Erasthones* to find prime numbers less than 50. This method starts with the number 2, marks all subsequent multiples of 2 as not prime, and repeats the process for the next prime candidate. Notice that we define the string object **sieveChars** using a constructor idiom that sets the initial size of the character array and writes the value 'P' to each of its member. Comment

```
//: C03:Sieve.cpp
//{L} ../TestSuite/Test
#include <cmath>
#include <cstddef>
#include <string>
#include "../TestSuite/Test.h"
using namespace std;

class SieveTest : public TestSuite::Test {
  string sieveChars;
public:
  // Create a 50 char string and set each
  // element to 'P' for Prime
  SieveTest() : sieveChars(50, 'P') {}
  void run() {
    findPrimes();
    testPrimes();
  }
  bool isPrime(int p) {
    if (p == 0 || p == 1) return false;
    int root = int(sqrt(double(p)));
    for (int i = 2; i <= root; ++i)
      if (p % i == 0) return false;
    return true;
  }
  void findPrimes() {
    // 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:
    size_t sieveSize = sieveChars.size();
    int root = int(sqrt(double(sieveSize)));
    for (int i = 2; i <= root; ++i)
```

```
      // Find all the multiples:
      for (size_t factor = 2; factor * i <
sieveSize;
           ++factor)
        sieveChars[factor * i] = 'N';
  }
  void testPrimes() {
    size_t i = sieveChars.find('P');
    while (i != string::npos) {
      test_(isPrime(i++));
      i = sieveChars.find('P', i);
    }
    i = sieveChars.find_first_not_of('P');
    while (i != string::npos) {
      test_(!isPrime(i++));
      i = sieveChars.find_first_not_of('P', i);
    }
  }
};

int main() {
  SieveTest t;
  t.run();
  return t.report();
} ///:~
```

The **find( )** function allows you to walk forward through a **string**, detecting multiple occurrences of a character or a group of characters, and **find_first_not_of( )** allows you to find other characters or substrings. Comment

There are no functions in the **string** class to change the case of a string, but you can easily create these functions using the Standard C library functions **toupper( )** and **tolower( )**, which change the case of one character at a time. The following example illustrates a case-insensitive search: Comment

```
//: C03:Find.cpp
//{L} ../TestSuite/Test
#include <cctype>
#include <cstddef>
#include <string>
```

```
#include "../TestSuite/Test.h"
using namespace std;

// Make an uppercase copy of s
string upperCase(const string& s) {
  string upper(s);
  for(size_t i = 0; i < s.length(); ++i)
    upper[i] = toupper(upper[i]);
  return upper;
}

// Make a lowercase copy of s
string lowerCase(const string& s) {
  string lower(s);
  for(size_t i = 0; i < s.length(); ++i)
    lower[i] = tolower(lower[i]);
  return lower;
}

class FindTest : public TestSuite::Test {
  string chooseOne;
public:
  FindTest() : chooseOne("Eenie, Meenie, Miney,
Mo") {}
  void testUpper() {
    string upper = upperCase(chooseOne);
    const string LOWER =
"abcdefghijklmnopqrstuvwxyz";
    test_(upper.find_first_of(LOWER) ==
string::npos);
  }
  void testLower() {
    string lower = lowerCase(chooseOne);
    const string UPPER =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    test_(lower.find_first_of(UPPER) ==
string::npos);
  }
  void testSearch() {
    // Case sensitive search
    size_t i = chooseOne.find("een");
    test_(i == 8);
```

```
    // Search lowercase:
    string test = lowerCase(chooseOne);
    i = test.find("een");
    test_(i == 0);
    i = test.find("een", ++i);
    test_(i == 8);
    i = test.find("een", ++i);
    test_(i == string::npos);
    // Search uppercase:
    test = upperCase(chooseOne);
    i = test.find("EEN");
    test_(i == 0);
    i = test.find("EEN", ++i);
    test_(i == 8);
    i = test.find("EEN", ++i);
    test_(i == string::npos);
  }
  void run() {
    testUpper();
    testLower();
    testSearch();
  }
};

int main() {
  FindTest t;
  t.run();
  return t.report();
} ///:~
```

Both the **upperCase( )** and **lowerCase( )** functions follow the same form: they make a copy of the argument **string** and change the case. The **NewFind.cpp** program isn't the best solution to the case-sensitivity problem, so we'll revisit it when we examine **string** comparisons. Comment

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. Comment

```
//: C03:Rparse.cpp
//{L} ../TestSuite/Test
#include <string>
#include <vector>
#include "../TestSuite/Test.h"
using namespace std;

class RparseTest : public TestSuite::Test {
  // To store the words:
  vector<string> strings;
public:
  void parseForData() {
    // The ';' characters will be delimiters
    string s("now.;sense;make;to;going;is;This");
    // 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:
      ++current;
      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 + 1;
      // 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));
  }
  void testData() {
    // Test order them in the new order:
    test_(strings[0] == "This");
    test_(strings[1] == "is");
    test_(strings[2] == "going");
```

```
    test_(strings[3] == "to");
    test_(strings[4] == "make");
    test_(strings[5] == "sense");
    test_(strings[6] == "now.");
    string sentence;
    for(int i = 0; i < strings.size() - 1; i++)
      sentence += strings[i] += " ";
    // Manually put last word in to avoid an extra
space
    sentence += strings[strings.size() - 1];
    test_(sentence == "This is going to make sense
now.");
  }
  void run() {
    parseForData();
    testData();
  }
};

int main() {
  RparseTest t;
  t.run();
  return t.report();
} ///:~
```

The string member function **rfind( )** backs through the string looking for tokens and reporting the array index of matching characters or **string::npos** if it is unsuccessful. Comment

Finding first/last of a set of characters

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 from both ends of a string. Notice that it doesn't touch the original string, but instead returns a new string: Comment

```
//: C03: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 beg = s.find_first_not_of("
\a\b\f\n\r\t\v");
  int end = s.find_last_not_of(" \a\b\f\n\r\t\v");
  if(beg == std::string::npos) // No non-spaces
    return "";
  return std::string(s, beg, end - beg + 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 builds a new **string** from the old one, giving the starting count and the length. Comment

Testing such a general-purpose tool needs to be thorough: Comment

```
//: C03:TrimTest.cpp
//{L} ../TestSuite/Test
#include <iostream>
#include "trim.h"
#include "../TestSuite/Test.h"
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 ",
  "\t \n \r \v \f",
  "" // Must also test the empty string
};

class TrimTest : public TestSuite::Test {
public:
  void testTrim() {
    test_(trim(s[0]) == "abcdefghijklmnop");
    test_(trim(s[1]) == "abcdefghijklmnop");
```

```
    test_(trim(s[2]) == "abcdefghijklmnop");
    test_(trim(s[3]) == "a");
    test_(trim(s[4]) == "ab");
    test_(trim(s[5]) == "abc");
    test_(trim(s[6]) == "a b c");
    test_(trim(s[7]) == "a b c");
    test_(trim(s[8]) == "a \t b \t c");
    test_(trim(s[9]) == "");
    test_(trim(s[10]) == "");
  }
  void run() {
    testTrim();
  }
};

int main() {
  TrimTest t;
  t.run();
  return t.report();
} ///:~
```

In the array of **strings**, 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 are not removed from the middle of a **string**. Comment

Removing characters from strings

Removing characters is easy and efficient with the **erase( )** member function, which takes two arguments: where to start removing characters (which defaults to **0**), and how many to remove (which defaults to **string::npos).** If you specify more characters than remain in the string, the remaining characters are all erased anyway (so calling **erase( )** without any arguments removes all characters from a string). Sometimes it's useful to take an HTML file and strip its tags and special characters so that you have something approximating the text that would be displayed in the Web browser, only as a plain text file. The following uses **erase( )** to do the job: Comment

```
//: C03:HTMLStripper.cpp
//{L} ReplaceAll
// Filter to remove html tags and markers
#include <cassert>
#include <cmath>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <string>
#include "../require.h"
using namespace std;

string& replaceAll(string& context, const string&
from,
  const string& to);

string& stripHTMLTags(string& s) {
  static bool inTag = false;
  bool done = false;
  while (!done) {
    if (inTag) {
      // The previous line started an HTML tag
      // but didn't finish. Must search for '>'.
      size_t rightPos = s.find('>');
      if (rightPos != string::npos) {
        inTag = false;
        s.erase(0, rightPos + 1);
      }
      else {
        done = true;
        s.erase();
      }
    }
    else {
      // Look for start of tag:
      size_t leftPos = s.find('<');
      if (leftPos != string::npos) {
        // See if tag close is in this line
        size_t rightPos = s.find('>');
        if (rightPos == string::npos) {
          inTag = done = true;
          s.erase(leftPos);
```

```
        }
        else
          s.erase(leftPos, rightPos - leftPos +
1);
      }
      else
        done = true;
    }
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  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]);
  string s;
  while(getline(in, s))
    if (!stripHTMLTags(s).empty())
      cout << s << endl;
} ///:~
```

This example will even strip HTML tags that span multiple lines[5]. This is accomplished with the static flag, **inTag**, which is **true** whenever the start of a tag is found, but the accompanying tag end is not found in the same line. All forms of **erase( )** appear in the **stripHTMLFlags( )** function.[6] The version of **getline( )** we use here is a global function declared in the **<string>** header and is

[5] To keep the exposition simple, this version does not handle nested tags, such as comments.

[6] It is tempting to use mathematics here to factor out some of these calls to **erase( )**, but since in some cases one of the operands is **string::npos** (the largest unsigned integer available), integer overflow occurs and wrecks the algorithm.

handy because it stores an arbitrarily long line in its **string** argument. You don't have to worry about the dimension of a character array as you do with **istream::getline( )**. Notice that this program uses the **replaceAll( )** function from earlier in this chapter. In the next chapter, we'll use string streams to create a more elegant solution. Comment

Comparing strings

Comparing strings is inherently different from comparing numbers. Numbers have constant, universally meaningful values. To evaluate the relationship between the magnitudes 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 32 through 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**. Comment

C++ provides several ways to compare strings, and each has advantages. The simplest to use are the nonmember, overloaded operator functions: **operator ==, operator != operator >, operator <, operator >=,** and **operator <=.** Comment

```
//: C03:CompStr.cpp
//{L} ../TestSuite/Test
#include <string>
#include "../TestSuite/Test.h"
using namespace std;
```

```
class CompStrTest : public TestSuite::Test {
public:
  void run() {
    // Strings to compare
    string s1("This");
    string s2("That");
    test_(s1 == s1);
    test_(s1 != s2);
    test_(s1 > s2);
    test_(s1 >= s2);
    test_(s1 >= s1);
    test_(s2 < s1);
    test_(s2 <= s1);
    test_(s1 <= s1);
  }
};

int main() {
  CompStrTest t;
  t.run();
  return t.report();
} ///:~
```

The overloaded comparison operators are useful for comparing both full strings and individual string character elements. Comment

Notice in the following code fragment the flexibility of argument types on both the left and right side of the comparison operators. For efficiency, the **string** class provides overloaded operators for the direct comparison of string objects, quoted literals, and pointers to C-style strings without having to create temporary **string** objects. Comment

```
// The lvalue is a quoted literal and
// the rvalue is a string
if("That" == s2)
  cout << "A match" << endl;
// The left operand below is a string and the
right is a
// pointer to a C-style null terminated string
if(s1 != s2.c_str())
```

```
  cout << "No match" << endl;
```

The **c_str( )** function returns a **const char*** that points to a C-style, null-terminated string equivalent to the contents of the **string** object. This comes in handy when you want to pass a string to a standard C function, such as **atoi( )** or any of the functions defined in the **<cstring>** header. It is an error to use the value returned by **c_str( )** as non-**const** argument to any function.
Comment

You won't find the logical not (**!**) or the logical comparison operators (**&&** and **||**) among operators for a string. (Neither will you find overloaded versions of the bitwise C operators **&**, **|**, **^**, or **~**.) The overloaded nonmember comparison operators for the string class are limited to the subset that has clear, unambiguous application to single characters or groups of characters. Comment

The **compare( )** member function offers you a great deal more sophisticated and precise comparison than the nonmember operator set. 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. The following example compares complete strings: Comment

```
//: C03:Compare.cpp
// Demonstrates compare(), swap()
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This");
  string second("That");
  assert(first.compare(first) == 0);
  assert(second.compare(second) == 0);
  // Which is lexically greater?
  assert(first.compare(second) > 0);
  assert(second.compare(first) < 0);
  first.swap(second);
  assert(first.compare(second) < 0);
```

```
  assert(second.compare(first) > 0);
} ///:~
```

The **swap( )** function in this example does what its name implies: it exchanges the contents of its object and argument. 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( )**: Comment

s1.compare(s1StartPos, s1NumberChars, s2, s2StartPos, s2NumberChars); Comment

Here's an example: Comment

```
//: C03:Compare2.cpp
// Illustrate overloaded compare()
#include <cassert>
#include <string>
using namespace std;

int main() {
  string first("This is a day that will live in
infamy");
  string second("I don't believe that this is what
"
                "I signed up for");
  // Compare "his is" in both strings:
  assert(first.compare(1, 7, second, 22, 7) == 0);
  // Compare "his is a" to "his is w":
  assert(first.compare(1, 9, second, 22, 9) < 0);
} ///:~
```

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 indexing mechanisms produce the same result in C++ if all goes well: Comment

```
//: C03:StringIndexing.cpp
#include <cassert>
```

```
#include <string>
using namespace std;
int main(){
  string s("1234");
  assert(s[1] == '2');
  assert(s.at(1) == '2');
} ///:~
```

There is one important difference, however, 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: Comment

```
//: C03:BadStringIndexing.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

int main(){
  string s("1234");
  // at() saves you by throwing an exception:
  try {
    s.at(5);
  } catch(exception& e) {
    cerr << e.what() << endl;
  }
} ///:~
```

Responsible programmers will not use errant indexes, but should you want to benefits of automatic index checking, using **at( )** in place of **[ ]** will give you a chance to gracefully recover from references to array elements that don't exist. Execution of this program on one of our test compilers gave the following output:

```
invalid string position
```

The **at( )** member throws an object of class **out_of_range**, which derives (ultimately) from **std::exception.** By catching this object in an exception handler, you can take appropriate remedial actions such as recalculating the offending subscript or growing the array. Using **string::operator[]( )** gives no such protection and is as dangerous as **char** array processing in C.[7] Comment

Strings and character traits

The program **Find.cpp** earlier in this chapter leads us 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. Comment

Consider what it means for a character to have "case." Written Hebrew, Farsi, and Kanji don't use the concept of upper- and lowercase, so for those languages this idea has no meaning. It would seem that if there were a way to designate some languages as "all uppercase" or "all lowercase," we could design a generalized solution. However, some languages that 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. Comment

Although we usually treat the C++ **string** as a class, this is really not the case. The **string** type is actually a specialization of a more general constituent, the **basic_string< >** template. Observe how **string** is declared in the standard C++ header file:[8] Comment

```
typedef basic_string<char> string;
```

---

[7] Alert: For the safety reasons mentioned, the C++ Standards Committee is considering a proposal to redefine **string::operator[]** to behave identically to **string::at()** for C++0x.

[8] Your implementation can define all three template arguments here. Because the last two template parameters have default arguments, such a declaration is equivalent to what we show here.

To really understand the nature of the string class, 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: Comment

```
template<class charT,
  class traits = char_traits<charT>,
  class allocator = allocator<charT> >
  class basic_string;
```

In Chapter 5, we examine templates in great detail (much more than in Chapter 16 of Volume 1). For now, the main thing to notice about the two previous declarations is that the **string** type is created when the **basic_string** template is instantiated with **char.** Inside the **basic_string< >** template declaration, the line Comment

```
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 in which you need string-oriented classes that manipulate types other than **char** (wide characters, 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. Comment

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 **char_traits< >** template, because that defines the behavior of the individual character comparison member functions. Comment

You can use this information to make a new type of **string** class that ignores case. First, we'll define a new case-insensitive **char_traits< >** template that inherits from the existing template. 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 earlier, we'll also have to supply a new implementation for the **char_traits** functions **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.
Comment

```
//: C03:ichar_traits.h
// Creating your own character traits
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H
#include <cassert>
#include <cctype>
#include <cmath>
#include <ostream>
#include <string>

using std::toupper;
using std::tolower;
using std::ostream;
using std::string;
using std::char_traits;
using std::allocator;
using std::basic_string;

struct ichar_traits : char_traits<char> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(char c1st, char c2nd) {
    return toupper(c1st) == toupper(c2nd);
  }
  static bool ne(char c1st, char c2nd) {
    return !eq(c1st, c2nd);
  }
  static bool lt(char c1st, char c2nd) {
    return toupper(c1st) < toupper(c2nd);
  }
  static int compare(const char* str1,
    const char* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
```

```
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(tolower(*str1) < tolower(*str2))
        return -1;
      else if(tolower(*str1) > tolower(*str2))
        return 1;
      assert(tolower(*str1) == tolower(*str2));
      str1++; str2++; // Compare the other chars
    }
    return 0;
  }
  static const char* find(const char* s1,
    size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<char, ichar_traits> istring;

inline ostream& operator<<(ostream& os, const
istring& s) {
  return os << string(s.c_str(), s.length());
}
#endif // ICHAR_TRAITS_H  ///:~
```

We provide a **typedef** named **istring** so that our class will act like an ordinary **string** in every way, except that it will make all comparisons without respect to case. For convenience, we've also provided an overloaded **operator<<( )** so that you can print **istring**s. Here's an example: Comment

```
//: C03:ICompare.cpp
#include <cassert>
#include <iostream>
#include "ichar_traits.h"
```

```
using namespace std;

int main() {
  // The same letters except for case:
  istring first = "tHis";
  istring second = "ThIS";
  cout << first << endl;
  cout << second << endl;
  assert(first.compare(second) == 0);
  assert(first.find('h') == 1);
  assert(first.find('I') == 2);
  assert(first.find('x') == string::npos);
} ///:~
```

This is just a toy example, of course. In order to make **istring** fully equivalent to **string**, we'd have to create the other functions necessary to support the new **istring** type. Comment

The **<string>** header provides a wide string class via the following **typedef**:

```
typedef basic_string<wchar_t> wstring;
```

Wide string support also reveals itself in *wide streams* (**wostream** in place of **ostream**, also defined in **<iostream>**) and in the header **<cwctype>**, a wide-character version of **<cctype>**. This along with the **wchar_t** specialization of **char_traits** in the standard library allows us to do a wide-character version of **ichar_traits**:

```
//: C03:iwchar_traits.h
//{-bor}
//{-g++}
// Creating your own wide-character traits
#ifndef IWCHAR_TRAITS_H
#define IWCHAR_TRAITS_H
#include <cassert>
#include <cwctype>
#include <cmath>
#include <ostream>
#include <string>
```

```
using std::towupper;
using std::towlower;
using std::wostream;
using std::wstring;
using std::char_traits;
using std::allocator;
using std::basic_string;

struct iwchar_traits : char_traits<wchar_t> {
  // We'll only change character-by-
  // character comparison functions
  static bool eq(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) == towupper(c2nd);
  }
  static bool ne(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) != towupper(c2nd);
  }
  static bool lt(wchar_t c1st, wchar_t c2nd) {
    return towupper(c1st) < towupper(c2nd);
  }
  static int compare(const wchar_t* str1,
    const wchar_t* str2, size_t n) {
    for(size_t i = 0; i < n; i++) {
      if(str1 == 0)
        return -1;
      else if(str2 == 0)
        return 1;
      else if(towlower(*str1) < towlower(*str2))
        return -1;
      else if(towlower(*str1) > towlower(*str2))
        return 1;
      assert(towlower(*str1) == towlower(*str2));
      str1++; str2++; // Compare the other
wchar_ts
    }
    return 0;
  }
  static const wchar_t* find(const wchar_t* s1,
    size_t n, wchar_t c) {
    while(n-- > 0)
      if(towupper(*s1) == towupper(c))
        return s1;
```

```
      else
        ++s1;
    return 0;
  }
};

typedef basic_string<wchar_t, iwchar_traits>
iwstring;

inline wostream& operator<<(wostream& os,
  const iwstring& s) {
  return os << wstring(s.c_str(), s.length());
}
#endif // IWCHAR_TRAITS_H  ///:~
```

As you can see, this is mostly an exercise in placing a 'w' in the appropriate place in the source code. The test program looks like this:

```
//: C03:IWCompare.cpp
//{-g++}
#include <cassert>
#include <iostream>
#include "iwchar_traits.h"
using namespace std;

int main() {
  // The same letters except for case:
  iwstring wfirst = L"tHis";
  iwstring wsecond = L"ThIS";
  wcout << wfirst << endl;
  wcout << wsecond << endl;
  assert(wfirst.compare(wsecond) == 0);
  assert(wfirst.find('h') == 1);
  assert(wfirst.find('I') == 2);
  assert(wfirst.find('x') == wstring::npos);
} ///:~
```

Unfortunately, some compilers still do not provide robust support for wide characters. Comment

A string application

If you've looked at the sample code in this book closely, you've noticed that certain tokens in the comments surround the code. These are used by a Python program that Bruce wrote to extract the code into files and set up makefiles for building the code. For example, a double-slash followed by a colon at the beginning of a line denotes the first line of a source file. The rest of the line contains information describing the file's name and location and whether it should be only compiled rather than fully built into an executable file. For example, the first line in the previous program above contains the string **C03:IWCompare.cpp**, indicating that the file **IWCompare.cpp** should be extracted into the directory **C03**. Comment

The last line of a source file contains the a triple-slash followed by a colon and a tilde. If the first line has an exclamation point immediately after the colon, the first and last lines of the source code are not to be output to the file (this is for data-only files). (If you're wondering why we're avoiding showing you these tokens, it's because we don't want to break the code extractor when applied to the text of the book!) Comment

Bruce's Python program does a lot more than just extract code. If the token **"{O}"** follows the file name, its makefile entry will only be set up to compile the file and not to link it into an executable. (The Test Framework in Chapter 2 is built this way.) To link such a file with another source example, the target executable's source file will contain an **"{L}"** directive, as in Comment

```
//{L} ../TestSuite/Test
```

This section will present a program to just extract all the code so that you can compile and inspect it manually. You can use this program to extract all the code in this book by saving the document file as a text file (let's call it TICV2.txt) and by executing something like the following on a shell command line: Comment

```
C:> extractCode TICV2.txt /TheCode
```

This command reads the text file **TICV2.txt** and writes all the source code files in subdirectories under the top-level directory **/TheCode**. The directory tree will look like the following:

```
TheCode/
   C0B/
   C01/
   C02/
   C03/
   C04/
   C05/
   C06/
   C07/
   C08/
   C09/
   C10/
   TestSuite/
```

The source files containing the examples from each chapter will be in the corresponding directory. Comment

Here's the program:

```
//: C03:ExtractCode.cpp
// Extracts code from text
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
// Legacy non-standard C header for mkdir()
#ifdef __GNUC__
#include <sys/stat.h>
#elif defined(__BORLANDC__) || defined(_MSC_VER)
#include <direct.h>
#else
#error Compiler not supported
#endif
```

```
// Check to see if directory exists
// by attempting to open a new file
// for output within it.
bool exists(string fname) {
  size_t len = fname.length();
  if(fname[len-1] != '/' && fname[len-1] != '\\')
    fname.append("/");
  fname.append("000.tmp");
  ofstream outf(fname.c_str());
  bool existFlag = outf;
  if (outf) {
    outf.close();
    remove(fname.c_str());
  }
  return existFlag;
}

int main(int argc, char* argv[]) {
  // See if input file name provided
  if(argc == 1) {
    cerr << "usage: extractCode file [dir]\n";
    exit(EXIT_FAILURE);
  }
  // See if input file exists
  ifstream inf(argv[1]);
  if(!inf) {
    cerr << "error opening file: " << argv[1] <<
endl;
    exit(EXIT_FAILURE);
  }
  // Check for optional output directory
  string root("./");  // current is default
  if(argc == 3) {
    // See if output directory exists
    root = argv[2];
    if(!exists(root)) {
      cerr << "no such directory: " << root <<
endl;
      exit(EXIT_FAILURE);
    }
    size_t rootLen = root.length();
```

```
    if(root[rootLen-1] != '/' && root[rootLen-1]
!= '\\')
      root.append("/");
  }
  // Read input file line by line
  // checking for code delimiters
  string line;
  bool inCode = false;
  bool printDelims = true;
  ofstream outf;
  while (getline(inf, line)) {
    size_t findDelim = line.find("//" "/:~");
    if(findDelim != string::npos) {
      // Output last line and close file
      if (!inCode) {
        cerr << "Lines out of order\n";
        exit(EXIT_FAILURE);
      }
      assert(outf);
      if (printDelims)
        outf << line << endl;
      outf.close();
      inCode = false;
      printDelims = true;
    } else {
      findDelim = line.find("//" ":");
      if(findDelim == 0) {
        // Check for '!' directive
        if(line[3] == '!') {
          printDelims = false;
          ++findDelim;  // To skip '!' for next
search
        }
        // Extract subdirectory name, if any
        size_t startOfSubdir =
          line.find_first_not_of(" \t",
findDelim+3);
        findDelim = line.find(':', startOfSubdir);
        if (findDelim == string::npos) {
          cerr << "missing filename information\n"
<< endl;
          exit(EXIT_FAILURE);
```

```
        }
        string subdir;
        if(findDelim > startOfSubdir)
          subdir = line.substr(startOfSubdir,
                               findDelim -
startOfSubdir);
        // Extract file name (better be one!)
        size_t startOfFile = findDelim + 1;
        size_t endOfFile =
          line.find_first_of(" \t", startOfFile);
        if(endOfFile == startOfFile) {
          cerr << "missing filename\n";
          exit(EXIT_FAILURE);
        }
        // We have all the pieces; build fullPath
name
        string fullPath(root);
        if(subdir.length() > 0)
          fullPath.append(subdir).append("/");
        assert(fullPath[fullPath.length()-1] ==
'/');
        if (!exists(fullPath))
#ifdef __GNUC__
          mkdir(fullPath.c_str(), 0);  // Create
subdir
#else
          mkdir(fullPath.c_str());  // Create
subdir
#endif
        fullPath.append(line.substr(startOfFile,
                        endOfFile - startOfFile));
        outf.open(fullPath.c_str());
        if(!outf) {
          cerr << "error opening " << fullPath
               << " for output\n";
          exit(EXIT_FAILURE);
        }
        inCode = true;
        cout << "Processing " << fullPath << endl;
        if(printDelims)
          outf << line << endl;
      }
```

```
      else if(inCode) {
        assert(outf);
        outf << line << endl;  // output middle
code line
      }
    }
  }
  exit(EXIT_SUCCESS);
} ///:~
```

First, you'll notice some conditional compilation directives. The **mkdir( )** function, which creates a directory in the file system, is defined by the POSIX[9] standard in the header **<sys/stat.h>**. Unfortunately, many compilers still use a different header (**<direct.h>**). The respective signatures for **mkdir( )** also differ: POSIX specifies two arguments, the older versions just one. For this reason, there is more conditional compilation later in the program to choose the right call to **mkdir( )**. We normally don't use conditional compilation in the examples in this book, but this particular program is too useful not to put a little extra work into, since you can use it to extract all the code with it. Comment

The **exists( )** function in **ExtractCode.cpp** tests whether a directory exists by opening a temporary file in it. If the open fails, the directory doesn't exist. You remove a file by sending its name as a **char*** to **std::remove( )**. Comment

The main program validates the command-line arguments and then reads the input file a line at a time, looking for the special source code delimiters. The Boolean flag **inCode** indicates that the program is in the middle of a source file, so lines should be output. The **printDelims** flag will be true if the opening token is not followed by an exclamation point; otherwise the first and last lines are not written. It is important to check for the closing delimiter first, because the start token is a subset of it, and searching for the start token first would return a successful find for both cases. If we

[9] POSIX, an IEEE standard, stands for "Portable Operating System Interface" and is a generalization of many of the low-level system calls found in UNIX systems.

encounter the closing token, we verify that we are in the middle of processing a source file; otherwise, something is wrong with the way the delimiters are laid out in the text file. If **inCode** is true, all is well, and we (optionally) write the last line and close the file. When the opening token is found, we parse the directory and file name components and open the file. The following **string**-related functions were used in this example: **length( )**, **append( )**, **getline( )**, **find( )** (two versions), **find_first_not_of( )**, **substr( )**, **find_first_of( )**, **c_str( )**, and, of course, **operator<<( )**. Comment

We also use a standard C technique for reporting program status to the calling context by returning different values from **main( )**. It is portable to use the statement **return 0;** to indicate success, but there is no portable value to indicate failure. For this reason we use the macro declared for this very purpose in **<cstdlib>**: **EXIT_FAILURE**. As a matter of consistency, whenever we use **EXIT_FAILURE** we also use **EXIT_SUCCESS**, even though the latter is always defined as zero. Comment

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. Comment

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 that overrides the character traits used to create the **basic_string** object. Comment

Exercises

1. Write a program that reverses the order of the characters in a string.
2. 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 prints whether the string was a palindrome or not.
3. Make your program from exercise 2 return **true** even if symmetric letters differ in case. For example, "Civic" would still return **true** although the first letter is capitalized.
4. Make your program from exercise 3 report true even if the string contains punctuation and spaces. For example "Able was I, ere I saw Elba." would report **true**.
5. Using the following strings and only **char**s (no string literals or magic numbers):

```
string one("I walked down the canyon with the
moving mountain bikers.");
string two("The bikers passed by me too close
for comfort.");
string three("I went hiking instead.")
```

produce the following sentence:

"I moved down the canyon with the mountain bikers. The mountain bikers passed by me too close for comfort. So I went hiking instead."

6. Write a program named **replace** that takes three command-line arguments representing an input text file, a string to replace (call it **from**), and a replacement string (call it **to**). The program should write a new file to standard output with all occurrences of **from** replaced by **to**.
7. Repeat the previous exercise but replace all instances of **from** regardless of case.

4: Iostreams

You can do much more 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 that you need to remember only one interface? That's the idea behind iostreams. They're much easier, safer, and sometimes even more efficient than the assorted functions from the Standard C **stdio** library. Comment

The iostreams classes are usually the first part of the C++ library that new C++ programmers learn to use. This chapter discusses how iostreams are an improvement over C's **stdio** facilities and explores the behavior of file and string streams in addition to the standard console streams. Comment

Why iostreams?

You might wonder what's wrong with the good old C library. Why not "wrap" the C library in a class and be done with it? Indeed, this is the perfect thing to do in some situations. For example, suppose you want to make sure the file represented by a **stdio FILE** pointer is always safely opened and properly closed, without having to rely on the user to remember to call the **close( )** function. The following program is such an attempt. Comment

```
//: C04:FileClass.h
// stdio files wrapped
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class FileClass {
  std::FILE* f;
public:
  struct FileClassError : std::runtime_error {
  public:
    FileClassError(const char* msg)
      : std::runtime_error(msg) {}
  };
  FileClass(const char* fname, const char* mode =
"r");
  ~FileClass();
  std::FILE* fp();
};
#endif // FILECLASS_H ///:~
```

When you perform file I/O in C, 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.” Comment

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: Comment

```
//: C04:FileClass.cpp {O}
// FileClassImplementation
#include "FileClass.h"
#include <cstdlib>
#include <cstdio>
```

```
using namespace std;

FileClass::FileClass(const char* fname, const
char* mode) {
  if((f = fopen(fname, mode)) == 0)
    throw FileClassError("Error opening file");
}

FileClass::~FileClass() { fclose(f); }

FILE* FileClass::fp() { return f; } ///:~
```

The constructor calls **fopen( )**, as you would normally do, but it also ensures that the result isn't zero, which indicates a failure upon opening the file. If the file does not open as expected, an exception is thrown. Comment

The destructor closes the file, and the access function **fp( )** returns **f**. Here's a simple example using **class FileClass**: Comment

```
//: C04:FileClassTest.cpp
// Tests FileClass
//{L} FileClass
#include <cstdlib>
#include <iostream>
#include "FileClass.h"
using namespace std;

int main() {
  try {
    FileClass f("FileClassTest.cpp");
    const int BSIZE = 100;
    char buf[BSIZE];
    while(fgets(buf, BSIZE, f.fp()))
      fputs(buf, stdout);
  }
  catch(FileClass::FileClassError& e) {
    cout << e.what() << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
} // 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; the file is closed by the destructor at the end of its scope. Comment

Even though the **FILE** pointer is private, it isn't particularly safe because **fp( )** retrieves it. Since the only effect seems to be guaranteed initialization and cleanup, 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 or even close it, so the safety is in guaranteeing a valid **FILE** pointer rather than proper contents of the structure. Comment

If you want complete safety, you must prevent the user from directly accessing the **FILE** pointer. Some version of all the normal file I/O functions must show up as class members so that everything you can do with the C approach is available in the C++ class: Comment

```
//: C04: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.) Comment

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 in which the initialization doesn't happen in the constructor, but some time after the enclosing object is created. Comment

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 **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 underlying **FILE** structure in this class.)[1] Comment

This approach 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). Comment

The big stumbling block is the runtime interpreter used for the variable argument list functions. This is the code that parses your format string at runtime and grabs and interprets arguments from the variable argument list. It's a problem for four reasons. Comment

1. Even if you use only a fraction of the functionality of the interpreter, the whole thing gets loaded into your executable. So if you say `printf("%c", 'x');`, you'll get the whole package, including the parts that print floating-point numbers and strings. There's no standard option for reducing the amount of space used by the program. Comment

2. Because the interpretation happens at runtime, you can't get rid of a performance overhead. 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 direct 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). Comment

3. A worse problem is that the format string is not evaluated until runtime: there can be no compile-time error checking. You're probably 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

[1] The implementation and test files for FULLWRAP are available in the freely distributed source code for this book. See the preface for details.

make your life easier. It seems a shame to throw type safety away for an I/O library, especially because I/O is used a lot. Comment

4. For C++, the most crucial problem is that the **printf( )** family of functions is not particularly extensible. They're really designed to handle only the four basic data types in C (**char**, **int**, **float**, **double**, **wchar_t**, **char***, **wchar_t***, and **void*)** and their variations. You might think that every time you add a new class, you could add overloaded **printf( )** and **scanf( )** functions (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 such as C++, whose goal is to be able to easily add new data types, this is an ungainly restriction. Comment

Iostreams to the rescue

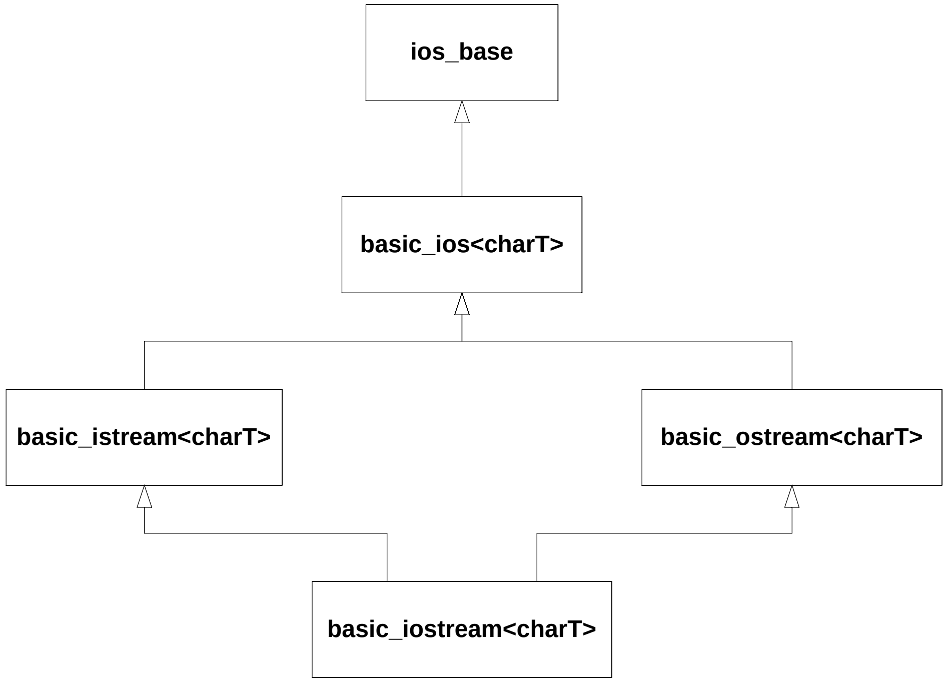
All these issues make it clear that one of the first priorities for the standard class libraries for C++ should handle I/O. Because "hello, world" is the first program just about everyone writes in a new language, and because I/O is part of virtually every program, the I/O library in C++ must be particularly easy to use. It also has the much greater challenge that it must accommodate any new class. Thus, its constraints require that this foundation class library be a truly inspired design. In addition to gaining a great deal of leverage and clarity in your dealings with I/O and formatting, you'll also see in this chapter how a really powerful C++ library can work. Comment

Inserters and extractors

A *stream* is an object that transports and formats characters of a fixed width. You can have an input stream (via descendants of the **istream** class), an output stream (with **ostream** objects), or a stream that does both simultaneously (with objects derived from **iostream**). The iostreams library provides different types of such classes: **ifstream**, **ofstream**, and **fstream** for files, and **istringstream**, **ostringstream**, and **stringstream** for interfacing with the Standard C++ **string** class. All these stream classes have nearly identical interfaces, so you can use streams in a uniform manner, whether you're working with a file, standard I/O, a region of memory, or a **string** object. The single interface you learn also works for extensions added to support new classes. Some functions implement your formatting commands, and some functions read and write characters without formatting. Comment

The stream classes mentioned earlier are actually template specializations,[2] much like the standard **string** class is a specialization of the **basic_string** template. The basic classes in the iostreams inheritance hierarchy are shown in the following figure. Comment

[2] Explained in depth in Chapter 5.



The **ios_base** class declares everything that is common to all streams, independent of the type of character the stream handles. These declarations are mostly constants and functions to manage them, some of which you'll see throughout this chapter. The rest of the classes are templates that have the underlying character type as a parameter. The **istream** class, for example, is defined as follows:
Comment

```
typedef basic_istream<char> istream;
```

All the classes mentioned earlier are defined via similar type definitions. There are also type definitions for all stream classes using **wchar_t** (the wide character type discussed in Chapter 3) instead of **char**. We'll look at these at the end of this chapter. The **basic_ios** template defines functions common to both input and output, but that depends on the underlying character type (we won't use these much). The template **basic_istream** defines generic functions for input, and **basic_ostream** does the same for output. The classes for file and string streams introduced later add functionality for their specific stream types. Comment

In the iostreams library, two operators are overloaded to simplify the use of iostreams. The operator **<<** is often referred to as an *inserter* for iostreams, and the operator **>>** is often referred to as an *extractor*. Comment

Extractors parse the information that's expected by the destination object according to its type. 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 predefined whenever you include the **<iostream>** header. Comment

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

There's an overloaded **operator >>** for every built-in data type. You can also overload your own, as you'll see later. Comment

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 **<<**:
Comment

```
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( )**, despite improved type checking. Fortunately, the overloaded inserters and extractors are designed to be chained together into a more complicated expression that is much easier to write (and read): Comment

```
  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;
```

Defining inserters and extractors for your own classes is just a matter of overloading the associated operators to do the right things, namely:

- Make the first parameter a non-**const** reference to the stream (**istream** for input, **ostream** for output)
- Perform the operation by insert/extracting data to/from the stream (by processing the components of the object, of course)
- Return a reference to the stream

The stream should be non-**const** because processing stream data changes the state of the stream. By returning the stream, you allow for chaining stream operations in a single statement, as shown earlier. Comment

As an example, consider how to output the representation of a **Date** object in MM-DD-YYYY format. The following inserter does the job:

```
ostream& operator<<(ostream& os, const Date& d) {
```

```
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << '-'
     << setw(2) << d.getDay() << '-'
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}
```

This function cannot be a member of the **Date** class, of course, because the left operand of the **<<** operator must be the output stream. The **fill( )** member function of **ostream** changes the padding character used when the width of an output field, determined by the *manipulator* **setw( )**, is greater than needed for the data. We use a '0' character so that months before October will display with a leading zero, such as "09" for September. The **fill( )** function also returns the previous fill character (which defaults to a single space) so that we can restore it later with the manipulator **setfill( )**. We discuss manipulators in depth later in this chapter. Comment

Extractors require a little more care because things sometimes go wrong with input data. The way to signal a stream error is to set the stream's *fail bit*, as follows:

```
istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
}
```

When an error bit is set in a stream, all further streams operations are ignored until the stream is restored to a good state (explained shortly). That's why the code above continues extracting even if **ios::failbit** gets set. This implementation is somewhat forgiving in that it allows white space between the numbers and dashes in a date string (because the **>>** operator skips white space by default when reading built-in types). The following are valid date strings for this extractor: Comment

```
"08-10-2002"
"8-10-2002"
"08 - 10 - 2002"
```

but these are not:

```
"A-10-2002"  // No alpha characters allowed
"08%10/2002" // Only dashes allowed as a delimiter
```

We'll discuss stream state in more depth in the section "Handling stream errors" later in this chapter. Comment

Common usage

As the **Date** extractor illustrated, you must be on guard for erroneous input. If the input produces an unexpected value, the process is skewed, and it's difficult to recover. In addition, formatted input defaults to white space delimiters. Consider what happens when we collect the code fragments from earlier in this chapter into a single program: Comment

```
//: C04: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: Comment

```
12 1.4 c this is a test
```

We expect the same output as if we gave it:

```
12
1.4
c
this is a test
```

but 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**, we overrun the buffer. Comment

In practice, you'll usually want to get input from interactive programs a line at a time as a sequence of characters, scan them, and then 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. Comment

Another thing to consider is the whole concept of a command-line interface. This made sense in the past when the console was little more than a glass typewriter, but the world is rapidly changing to one in which 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 simple examples or tests, and take the following approaches: Comment

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. Comment

2. Read the input without attempting to convert it, as we just suggested. When the input is some place where it can't foul things up during conversion, you can safely scan it. Comment

3. Output is different. If you're using a GUI, **cout** doesn't necessarily 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. Comment

Another common practice saves compile time on large projects. Consider, for example, how you would declare the Date stream operators introduced earlier in the chapter in a header file. You only need to include the prototypes for the functions, so it's not really necessary to include the entire **<iostream>** header in **Date.h**. The standard practice is to only declare classes, something like this: Comment

```
class ostream;
```

This is an age-old technique for separating interface from implementation and is often called a **forward declaration** (and **ostream** at this point would be considered an *incomplete type*, since the class definition has not yet been seen by the compiler). Comment

This will not work as is, however, for two reasons:

1. The stream classes are defined in the **std** namespace.

2. They are templates.

The proper declaration would be:

```
namespace std {
  template<class charT, class traits =
char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char> ostream;
}
```

(As you can see, like the **string** class, the streams classes use the character traits classes mentioned in Chapter 3). Since it would be terribly tedious to type all that for every stream class you want to reference, the standard provides a header that does it for you: **<iosfwd>**. The **Date** header would then look something like this: Comment

```
// Date.h
#include <iosfwd>
class Date {
  friend std::ostream& operator<<(std::ostream&,
                                  const Date&);
  friend std::istream& operator>>(std::istream&,
Date&);
  // etc.
```

Comment

Line-oriented input

To grab input a line at a time, you have three choices:

The member function **get( )**

The member function **getline( )**

The global function **getline( )** defined in the **<string>** header

The first two functions take three arguments:

A pointer to a character buffer in which to store the result

The size of that buffer (so it's not overrun)

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. Comment

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.) The **getline( )** function, on the other hand, extracts the delimiter from the input stream, but still doesn't store it in the result buffer. Comment

The **getline( )** function defined in **<string>** is convenient. It is not a member function, but rather a stand-alone function declared in the namespace **std**. It takes only two non-default arguments, the input stream and the **string** object to populate. Like its namesake, it reads characters until it encounters the first occurrence of the delimiter (**'\n'** by default) and consumes and discards the delimiter. The advantage of this function is that it reads into a **string** object, so you don't have to worry about buffer size. Comment

Generally, when you're processing a text file that you read a line at a time, you'll want to use one of the **getline( )** functions. Comment

Overloaded versions of get( )

The **get( )** function 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*; and one that stores directly into the underlying buffer structure of another iostream object. The latter is explored later in the chapter. Comment

Reading raw bytes

If you know exactly what you're dealing with and want to move the bytes directly into a variable, an array, or a structure in memory, you can use the unformatted I/O function **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 (using the same compiler, of course). You'll see examples of all these functions later. Comment

Handling stream errors

The **Date** extractor shown earlier sets a stream's fail bit under certain conditions. How does the user know when such a failure occurs? You can detect stream errors by either calling certain stream member functions to see if an error state has occurred, or if you don't care what the particular error was, you can just evaluate the stream in a Boolean context. Both techniques derive from the state of a stream's error bits. Comment

Stream state

The **ios_base** class, from which **ios** derives,[3] defines four flags that you can use to test the state of a stream:

Flag	Meaning
badbit	Some fatal (perhaps physical) error occurred. The stream should be considered unusable.
eofbit	End-of-input has occurred (either by encountering the physical end of a file stream or by the user terminating a console

[3] For this reason, we can write **ios::failbit** instead of **ios_base::failbit** to save typing.

	stream, such as with Ctrl-Z or Ctrl-D).
failbit	An I/O operation failed, most likely because of invalid data (e.g., letters were found when trying to read a number). The stream is still usable. The **failbit** flag is also set when end-of-input occurs.
goodbit	All is well; no errors. End-of-input has not yet occurred.

You can test whether any of these conditions have occurred by calling corresponding member functions that return a Boolean value indicating whether any of these have been set. The **good( )** stream member function returns true if none of the other three bits are set. The **eof( )** function returns true if **eofbit** is set, which happens with an attempt to read from a stream that has no more data (usually a file). Because end-of-input happens in C++ when trying to read past the end of the physical medium, **failbit** is also set to indicate that the "expected" data was not successfully read. The **fail( )** function returns true if *either* **failbit** or **badbit** is set, and **bad( )** returns true only if the **badbit** is set. Comment

Once any of the error bits in a stream's state are set, they remain set, which is not always what you want. When reading a file for example, you might want to reposition to an earlier place in the file before end-of-file occurred. Just moving the file pointer doesn't automatically reset **eofbit** or **failbit**; you have to do it yourself with the **clear( )** function, like this: Comment

```
myStream.clear(); // Clears all error bits
```

After calling **clear( )**, **good( )** will return **true** if called immediately. As you saw in the **Date** extractor earlier, the **setstate( )** function sets the bits you pass it. It turns out that **setstate( )** doesn't affect any other bits—if they're already set, they stay set. If you want to set certain bits but at the same time reset all the rest, you can call an overloaded version of **clear( )**, passing it a bitwise expression representing the bits you want to set, as in: Comment

```
myStream.clear(ios::failbit | ios::eofbit);
```

Most of the time you won't be interested in checking the stream state bits individually. Usually you just want to know if everything is okay. This is the case when you read a file from beginning to end; you just want to know when the input data has been exhausted. In cases such as these, a conversion operator is defined for **void*** that is automatically called when a stream occurs in a Boolean expression. To read a stream until end-of-input using this idiom looks like the following: Comment

```
int i;
while (myStream >> i)
  cout << i << endl;
```

Remember that **operator>>( )** returns its stream argument, so the **while** statement above tests the stream as a Boolean expression. This particular example assumes that the input stream **myStream** contains integers separated by white space. The function **ios_base::operator void*( )** simply calls **good( )** on its stream and returns the result.[4] Because most stream operations return their stream, using this idiom is convenient. Comment

Streams and exceptions

Iostreams existed as part of C++ long before there were exceptions, so checking stream state manually was just the way things were done. For backward compatibility, this is still the status quo, but iostreams can throw exceptions instead. The **exceptions( )** stream member function takes a parameter representing the state bits for which you want exceptions to be thrown. Whenever the stream

---

[4] It is customary to use **operator void*( )** in preference to **operator bool( )** because the implicit conversions from **bool** to **int** may cause surprises, should you errantly place a stream in a context where an integer conversion can be applied. The **operator void*( )** function will only implicitly be called in the body of a Boolean expression.

encounters such a state, it throws an exception of type **std::ios_base::failure**, which inherits from **std::exception**. Comment

Although you can trigger a failure exception for any of the four stream states, it's not necessarily a good idea to enable exceptions for all of them. As Chapter 1 explains, use exceptions for truly exceptional conditions, but end-of-file is not only *not* exceptional—it's *expected*! For that reason, you might want to enable exceptions only for the errors represented by **badbit**, which you would do like this: Comment

```
myStream.exceptions(ios::badbit);
```

You enable exceptions on a stream-by-stream basis, since **exceptions( )** is a member function for streams. The **exceptions( )** function returns a bitmask[5] (of type **iostate**, which is some compiler-dependent type convertible to **int**) indicating which stream states will cause exceptions. If those states have already been set, an exception is thrown immediately. Of course, if you use exceptions in connection with streams, you had better be ready to catch them, which means that you need to wrap all stream processing with a **try** block that has an **ios::failure** handler. Many programmers find this tedious and just check states manually where they expect errors to occur (since, for example, they don't expect **bad( )** to return **true** most of the time anyway). This is another reason that having streams throw exceptions is optional and not the default. In any case, you can choose how you want to handle stream errors. Comment

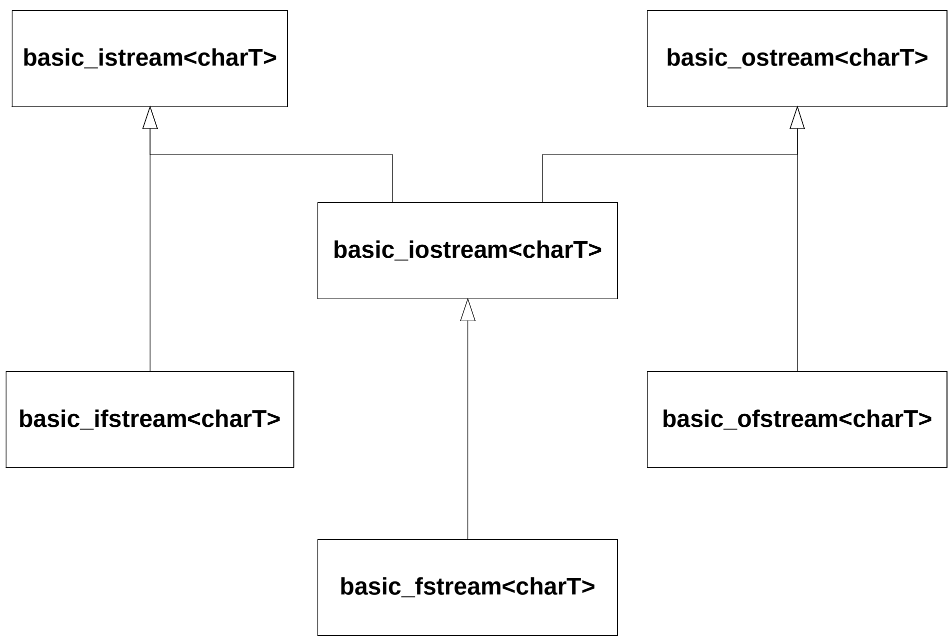
File iostreams

Manipulating files with iostreams is much easier and safer than using **stdio** 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

[5] An integral type used to hold single-bit flags.

(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. An **fstream** object can do both input and output. Comment

The file stream classes fit into the iostreams classes as shown in the following figure.



As before, the classes you actually use are template specializations defined by type definitions. For example, **ifstream**, which processes files of **char**, is defined as Comment

```
typedef basic_ifstream<char> ifstream;
```

A File-Processing Example

Here's an example that shows many of the features discussed so far. Notice the inclusion of **<fstream>** to declare the file I/O classes. Although on many platforms this will also include **<iostream>** automatically, compilers are not required to do so. If you want portable code, always include both headers. Comment

```
//: C04:Strfile.cpp
// Stream I/O with files
// The difference between get() & getline()
#include <fstream>
#include <iostream>
#include "../require.h"
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 in which the compiler expects a Boolean result, produces a value that indicates success or failure. Comment

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. The **get( )** function 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 default arguments. The first argument is the number of characters to throw away and defaults to one. The second argument is the character at which the **ignore( )** function quits (after extracting it) and defaults to **EOF**. Comment

Next, you see two output statements that look 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. Comment

To demonstrate **getline( )**, 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 with 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 do this, you can even reuse the **in** object by calling the **open( )** member function. Comment

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. Comment

This example, as well as most of the examples in this chapter, assumes that each call to any overload of **getline( )** will actually encounter a newline character. If this is not the case, the eofbit state of the stream will be set and the call to **getline( )** will return **false**, causing the program to lose the last line of input.

Open modes

You can control the way a file is opened by overriding the constructor's default arguments. The following table shows the flags that control the mode of the file: Comment

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 only.
ios::ate	Opens an existing file (either input or output) and seeks to the end.
ios::trunc	Truncates the old file, if it already exists.
ios::binary	Opens a file in *binary mode*. The default is *text mode*.

You can combine these flags using a bitwise *or* operation. Comment

The binary flag, while portable, only has an effect on some non-UNIX systems, such as operating systems derived from MS-DOS, that have special conventions for storing end-of-line delimiters. For example, on MS-DOS systems in text mode (which is the default), every time you output a newline character (**'\n'**), the file system actually outputs two characters, a carriage-return/linefeed pair (CRLF), which is the pair of ASCII characters **0x0D** and **0x0A**. Conversely, when you read such a file back into memory in text mode, each occurrence of this pair of bytes causes a **'\n'** to be sent to the program in its place. If you want to bypass this special processing, you open files in binary mode. Binary mode has nothing whatsoever to do with whether you *can* write raw bytes to a file—you *always* can (by calling **write( )**) . You should, however, open a file in binary mode when you'll be using **read( )** or **write( )**, because these functions take a byte count parameter. Having the extra **'\r'** characters will throw your byte count off in those instances. You should also open a file in binary mode if you're going to use the stream-positioning commands discussed later in this chapter. Comment

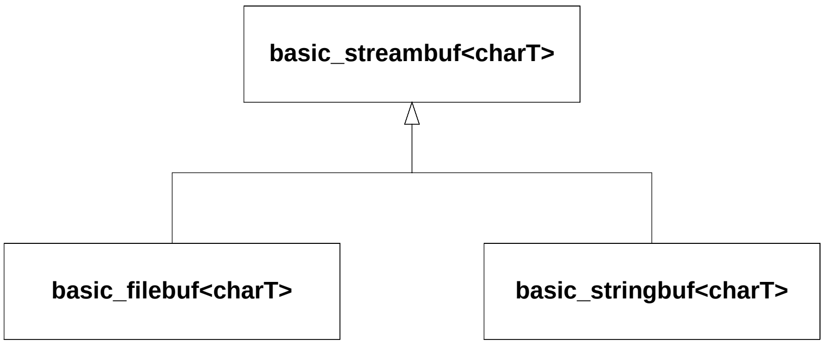
You can open a file for both input and output by declaring an **fstream** object. When declaring an **fstream** object, you must use enough of the open mode flags mentioned earlier to let the file system know whether you want to input, output, or both. To switch from output to input, you need to either flush the stream or change the file position. To change from input to output, you must change the file position. To create a file via an **fstream** object, you need to use the **ios::trunc** open mode flag in the constructor call if you will actually do both input and output. Comment

Iostream buffering

Good design practice dictates that whenever you create a new class, you should endeavor to hide the details of the underlying implementation as much possible from the user of the class. You show them only what they need to know and make the rest **private** to avoid confusion. When using inserters and extractors, you normally don't know or care where the bytes are being produced or consumed, whether you're dealing with standard I/O, files, memory, or some newly created class or device. Comment

A time comes, however, when it is important to communicate with the part of the iostream that produces and consumes bytes. To provide this part with a common interface and still hide its underlying implementation, the standard library abstracts it 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, and so on.) 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 by calling member functions for the **streambuf** object. Comment

Currently, the most important thing for you to know is that every iostream object contains a pointer to a **streambuf** object, and the **streambuf** object has some member functions you can call if necessary. For file and string streams, there are specialized types of stream buffers, as the following figure illustrates. Comment



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 characters from your object into the one on the left side of the **<<**. If you want to move all the characters from one iostream to another, you don't have to go through the tedium (and potential coding errors) of reading them one character or one line at a time. It's a much more elegant approach. Comment

For example, here's a simple program that opens a file and sends the contents to standard output (similar to the previous example): Comment

```
//: C04:Stype.cpp
// Type a file to standard output
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Stype.cpp");
  assure(in, "Stype.cpp");
  cout << in.rdbuf(); // Outputs entire file
} ///:~
```

An **ifstream** is created using the source code file for this program as an argument. The **assure( )** function reports a failure if the file cannot be opened. All the work really happens in the statement: Comment

```
cout << in.rdbuf();
```

which sends the entire contents of the file to **cout**. This is not only more succinct to code, it is often more efficient than moving the bytes one at a time. Comment

A form of **get( )** allows you to write directly into the **streambuf** of another object. The first argument is a reference to the destination **streambuf**, and the second is the terminating character (**'\n'** by default), which stops the **get( )** function. So there is yet another way to print a file to standard output: Comment

```
//: C04:Sbufget.cpp
// Copies a file to standard output
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Sbufget.cpp");
  assure(in);
  streambuf& sb = *cout.rdbuf();
  while (!in.get(sb).eof()) {
    if (in.fail())          // Found blank line
      in.clear();
    cout << char(in.get()); // Process '\n'
  }
} ///:~
```

The **rdbuf( )** function returns a pointer, so it must be dereferenced to satisfy the function's need to see an object. Stream buffers are not meant to be copied (they have no copy constructor), so we define **sb** as a *reference* to **cout**'s stream buffer. We need the calls to **fail( )** and **clear( )** in case the input file has a blank line (this one does). When this particular overloaded version of **get( )** sees two newlines in a row (evidence of a blank line), it sets the input stream's fail bit, so we must call **clear( )** to reset it so that the stream can continue to be read. The second call to **get( )** extracts and echoes each newline delimiter. (Remember, the **get( )** function doesn't extract its delimiter like **getline( )** does.) Comment

You probably won't need to use a technique like this often, but it's nice to know it exists.[6]Comment

[6] A more in-depth treatment of stream buffers and streams in general can be found in Langer & Kreft's, *Standard C++ IOStreams and Locales*, Addison-Wesley, 1999.

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 might want to move this stream position. You can do so 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. Comment

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 calls to **seekp( )** for an **ostream** or **seekg( )** for an **istream**, when you want to return to that position in the stream. Comment

The second approach is a relative seek and uses overloaded versions of **seekp( )** and **seekg( )**. The first argument is the number of characters to move: it can be positive or negative. The second argument is the seek direction: Comment

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 standard stream objects, such as **cin** and **cout**, explicitly disallow it): Comment

```
//: C04:Seeking.cpp
// Seeking in iostreams
#include <cassert>
#include <cstddef>
```

```
#include <cstring>
#include <fstream>
#include "../require.h"
using namespace std;

int main() {
  const int STR_NUM = 5, STR_LEN = 30;
  char origData[STR_NUM][STR_LEN] = {
    "Hickory dickory dus. . .",
    "Are you tired of C++?",
    "Well, if you are,",
    "That's just too bad,",
    "There's plenty more for us!"
  };
  char readData[STR_NUM][STR_LEN] = { 0 };
  ofstream out("Poem.bin", ios::out |
ios::binary);
  assure(out, "Poem.bin");
  for(size_t i = 0; i < STR_NUM; i++)
    out.write(origData[i], STR_LEN);
  out.close();
  ifstream in("Poem.bin", ios::in | ios::binary);
  assure(in, "Poem.bin");
  in.read(readData[0], STR_LEN);
  assert(strcmp(readData[0], "Hickory dickory dus.
. .")
    == 0);
  // Seek -STR_LEN bytes from the end of file
  in.seekg(-STR_LEN, ios::end);
  in.read(readData[1], STR_LEN);
  assert(strcmp(readData[1], "There's plenty more
for us!")
    == 0);
  // Absolute seek (like using operator[] with a
file)
  in.seekg(3 * STR_LEN);
  in.read(readData[2], STR_LEN);
  assert(strcmp(readData[2], "That's just too
bad,") == 0);
  // Seek backwards from current position
  in.seekg(-STR_LEN * 2, ios::cur);
  in.read(readData[3], STR_LEN);
```

```
  assert(strcmp(readData[3], "Well, if you have,")
== 0);
  // Seek from the begining of the file
  in.seekg(1 * STR_LEN, ios::beg);
  in.read(readData[4], STR_LEN);
  assert(strcmp(readData[4], "Have you had plenty
of C++?")
    == 0);
} ///:~
```

This program writes a (very clever) poem to a file using a binary output stream. Since we reopen it as an **ifstream**, we use **seekg( )** to position the "get pointer." As you can see, you can seek from the beginning or end of the file or from the current file position. Obviously, you must provide a positive number to move from the beginning of the file and a negative number to move back from the end. Comment

Now that you know about the **streambuf** and how to seek, you can understand an alternative method (besides using an **fstream** object) for creating 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. You can't write to an **ifstream**, of course, so you need to create an **ostream** with the underlying stream buffer: Comment

```
ifstream in("filename", ios::in | ios::out);
ostream out(in.rdbuf());
```

You might wonder what happens when you write to one of these objects. Here's an example: Comment

```
//: C04:Iofile.cpp
// Reading & writing one file
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
```

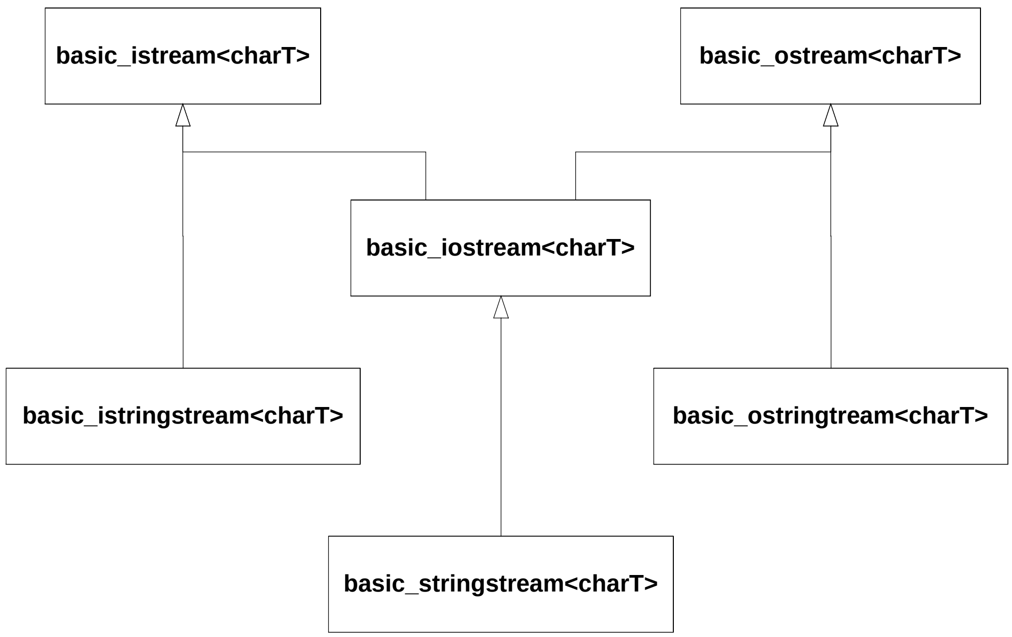
```
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("Iofile.out", ios::in);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::end);
  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 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 displayed. Of course, the file is automatically saved and closed when **out2** goes out of scope and its destructor is called. Comment

String iostreams

A string stream works directly with memory instead of a file or standard output. It allows you to use the same reading and formatting functions that you use with **cin** and **cout** 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*. Comment

The class names for string streams echo those for file streams. If you want to create a string stream to extract characters from, you create an **istringstream**. If you want to put characters into a string stream, you create an **ostringstream**. All declarations for string stream are in the standard header **<sstream>**. As usual, there are class templates that fit into the iostreams hierarchy, as shown in the following figure: Comment



Input string streams

To read from a string using stream operations, you create an **istringstream** object initialized with the string. The following program shows how to use an **istringstream** object.

```
//: C04:Istring.cpp
// Input string streams
```

```
#include <cassert>
#include <cmath>  // For fabs()
#include <iostream>
#include <limits> // For epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream s("47 1.414 This is a test");
  int i;
  double f;
  s >> i >> f; // Whitespace-delimited input
  assert(i == 47);
  double relerr = (fabs(f) - 1.414) / 1.414;
  assert(relerr <=
numeric_limits<double>::epsilon());
  string buf2;
  s >> buf2;
  assert(buf2 == "This");
  cout << s.rdbuf(); // " is a test"
} ///:~
```

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 such as **atof( )**, **atoi( )**, even though the latter may be more efficient for single conversions. Comment

In the expression **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 1 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 number into a whole number and a fraction part. Otherwise it would seem to be an error. The second **assert( )** calculates the relative error between what we read and what we expected; it's always better to do this than to compare floating-point numbers for equality. The constant returned by **epsilon( )**, defined in **<limits>**, represents the *machine epsilon* for double-precision numbers, which is the best tolerance you can expect comparisons of **doubles** to satisfy.[7] Comment

As you may already have guessed, **buf2** doesn't get the rest of the string, just the next white-space-delimited word. In general, it's best to use the extractor in iostreams 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. Comment

To test the **Date** extractor at the beginning of this chapter, we used an input string stream with the following test program:

```
//: C04:DateIOTest.cpp
//{L} ../C02/Date
#include <iostream>
#include <sstream>
#include "../C02/Date.h"
using namespace std;

void testDate(const string& s) {
  istringstream os(s);
  Date d;
  os >> d;
  if (os)
    cout << d << endl;
  else
    cout << "input error with \"" << s << "\"\n";
}

int main() {
  testDate("08-10-2002");
  testDate("8-10-2002");
  testDate("08 - 10 - 2002");
  testDate("A-10-2002");
```

[7] For more information on machine epsilon and floating-point computation in general, see Chuck's article, "The Standard C Library, Part 3", *C/C++ Users Journal*, March 1995, also available at www.freshsources.com/1995006a.htm.

```
  testDate("08%10/2002");
} ///:~
```

Each string literal in **main( )** is passed by reference to **testDate( )**, which in turn wraps it in an **istringstream** so we can test the stream extractor we wrote for **Date** objects. The function **testDate( )** also begins to test the inserter, **operator<<( )**. Comment

Output string streams

To create an output string stream to put data into, you just create an **ostringstream** object, which manages a dynamically sized character buffer to hold whatever you insert. To get the formatted result as a **string** object, you call the **str( )** member function. Here's an example: Comment

```
//: C04:Ostring.cpp
// Illustrates ostringstream
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  cout << "type an int, a float and a string: ";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Throw away white space
  string stuff;
  getline(cin, stuff); // Get rest of the line
  ostringstream os;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << "string = " << stuff << endl;
  string result = os.str();
  cout << result << endl;
} ///:~
```

This is similar to the **Istring.cpp** example earlier that fetched an **int** and a **float**. A sample execution follows (the keyboard input is in bold type). Comment

```
type an int, a float and a string: 10 20.5 the end
integer = 10
float = 20.5
string = the end
```

You can see that, like the other output streams, you can use the ordinary formatting tools, such as the **<<** operator and **endl**, to send bytes to the **ostringstream**. The **str( )** function returns a new **string** object every time you call it so the underlying **stringbuf** object owned by the string stream is left undisturbed. Comment

In the previous chapter, we presented a program, **HTMLStripper.cpp**, that removed all HTML tags and special codes from a text file. As promised, here is a more elegant version using string streams.

```
//: C04:HTMLStripper2.cpp
//{L} ../c03/replaceAll
// Filter to remove html tags and markers
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "../require.h"
using namespace std;

string& replaceAll(string& context, const string&
from,
  const string& to);

string& stripHTMLTags(string& s)
throw(runtime_error) {
  size_t leftPos;
```

```
  while ((leftPos = s.find('<')) != string::npos)
{
    size_t rightPos = s.find('>', leftPos+1);
    if (rightPos == string::npos) {
      ostringstream msg;
      msg << "Incomplete HTML tag starting in
position "
          << leftPos;
      throw runtime_error(msg.str());
    }
    s.erase(leftPos, rightPos - leftPos + 1);
  }
  // Remove all special HTML characters
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // Etc...
  return s;
}

int main(int argc, char* argv[]) {
  requireArgs(argc, 1,
    "usage: HTMLStripper2 InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  // Read entire file into string; then strip
  ostringstream ss;
  ss << in.rdbuf();
  try {
    string s = ss.str();
    cout << stripHTMLTags(s) << endl;
    return EXIT_SUCCESS;
  }
  catch (runtime_error& x) {
    cout << x.what() << endl;
    return EXIT_FAILURE;
  }
} ///:~
```

In this program we read the entire file into a string by inserting a **rdbuf( )** call to the file stream into an **ostringstream**. Now it's an easy matter to search for HTML delimiter pairs and erase them without having to worry about crossing line boundaries like we had to with the previous version in Chapter 3. Comment

The following example shows how to use a bidirectional (that is, read/write) string stream.

```
//: C04:StringSeeking.cpp
// Reads and writes a string stream
//{-bor}
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string text = "We will sell no wine";
  stringstream ss(text);
  ss.seekp(0, ios::end);
  ss << " before its time.";
  assert(ss.str() ==
    "We will sell no wine before its time.");
  // Change "sell" to "ship"
  ss.seekg(9, ios::beg);
  string word;
  ss >> word;
  assert(word == "ell");
  ss.seekp(9, ios::beg);
  ss << "hip";
  // Change "wine" to "code"
  ss.seekg(16, ios::beg);
  ss >> word;
  assert(word == "wine");
  ss.seekp(16, ios::beg);
  ss << "code";
  assert(ss.str() ==
    "We will ship no code before its time.");
  ss.str("A horse of a different color.");
  assert(ss.str() == "A horse of a different
color.");
} ///:~
```

As always, to move the put pointer, you call **seekp( )**, and to reposition the get pointer, you call **seekg( )**. Even though we didn't show it with this example, string streams are a little more forgiving than file streams in that you can switch from reading to writing or vice-versa at any time. You don't need to reposition the get or put pointers or flush the stream. This program also illustrates the overload of **str( )** that replaces the stream's underlying **stringbuf** with a new string. Comment

Output stream formatting

The goal of the iostreams design is to allow you to easily move and/or format characters. It certainly wouldn't be useful if you couldn't do most of the formatting provided by C's **printf( )** family of functions. In this section, you'll learn all the output formatting functions that are available for iostreams, so you can format your bytes the way you want them. Comment

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, a generic member function sets state flags to control formatting, such as left or right justification, to use uppercase letters for hex notation, to always use a decimal point for floating-point values, and so on. On the other hand, separate member functions set and read values for the fill character, the field width, and the precision. Comment

In an attempt to clarify all this, we'll first examine the internal formatting data of an iostream , along with the member functions that can modify that data. (Everything can be controlled through the member functions, if desired.) We'll cover the manipulators separately. Comment

Format flags

The class **ios** contains data members to store all the formatting information pertaining to a 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 an object of type **fmtflags** (usually a synonym for **long**) 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. Comment

```
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. Comment

The use of **setf( )** can seem somewhat confusing. To know which overloaded version to use, you must know what type of flag you're changing. There are two types of flags: those that are simply on or off, and those 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 shown in the following table. Comment

on/off flag	**Effect**
ios::skipws	Skip white space. (For input; this is the default.)
ios::showbase	Indicate the numeric base (as set, for example, by **dec**, **oct**, or **hex**) when printing an integral value. Input streams also recognize the base prefix when **showbase** is on.

on / off flag	Effect
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.

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)**. Comment

The **unitbuf** flag controls *unit buffering*, which means that each insertion is flushed to its output stream immediately. This is handy for error tracing, so that in case of a program crash, your data is still written to the log file. The following program illustrates unit buffering.

```
//: C04:Unitbuf.cpp
#include <cstdlib>  // For abort()
#include <fstream>
using namespace std;

int main() {
  ofstream out("log.txt");
  out.setf(ios::unitbuf);
  out << "one\n";
  out << "two\n";
  abort();
} ///:~
```

It is necessary to turn on unit buffering before any insertions are made to the stream. When we commented out the call to **setf( )**, one particular compiler had written only the letter 'o' to the file **log.txt**. With unit buffering, no data was lost. Comment

The standard error output stream **cerr** has unit buffering turned on by default. There is a cost for unit buffering, of course, so if an output stream is heavily used, don't enable unit buffering unless efficiency is not a consideration. Comment

Format fields

The second type of formatting flags work in a group. Only one of these flags can be , 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 that 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** and *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 **ios::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 that there are manipulators to provide equivalent functionality in all the places you would use **setf( )**. Comment

Here are the flag groups and their effects:

ios::basefield	**effect**
ios::dec	Format integral values in base 10 (decimal) (the default radix—no prefix is visible).
ios::hex	Format integral values in base 16 (hexadecimal).

ios::basefield	**effect**
ios::oct	Format integral values in base 8 (octal).

Comment

ios::floatfield	**effect**
ios::scientific	Display floating-point numbers in scientific format. Precision field indicates number of digits after the decimal point.
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.

Comment

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. (In other words, the sign, if printed, is left-justified while the number is right-justified).

Comment

Width, fill, and precision

The internal variables that control the width of the output field, the fill character used to pad an output field, and the precision for printing floating-point numbers are read and written by member functions of the same name. Comment

Function	effect
int ios::width( )	Returns the current width. (Default is 0.) Used for both insertion and extraction.
int ios::width(int n)	Sets the width, returns the previous width.
int ios::fill( )	Returns the current fill character. (Default is space.)
int ios::fill(int n)	Sets the fill character, returns the previous fill character.
int ios::precision( )	Returns 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."

Comment

The **fill** and **precision** values are fairly straightforward, but **width** requires some explanation. When the width is zero, inserting a value produces 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 fewer 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. Comment

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 rather an implicit argument to the inserters and extractors. If you want a constant width, you must call **width( )** after each insertion or extraction. Comment

An exhaustive example

To make sure you know how to call all the functions previously discussed, here's an example that calls them all: Comment

```
//: C04:Format.cpp
// Formatting Functions
#include <fstream>
#include <iostream>
#include "../require.h"
using namespace std;
#define D(A) T << #A << endl; A

int main() {
  ofstream T("format.out");
  assure(T);
  D(int i = 47;)
  D(float f = 2300114.414159;)
  char* s = "Is there any more?";

  D(T.setf(ios::unitbuf);)
  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase | ios::showpos);)
  D(T << i << endl;) // Default is dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  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.unsetf(ios::uppercase);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  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.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;
} ///:~
```

This example uses a trick to create a trace file so that you can monitor what's happening. The macro **D(a)** uses the preprocessor "stringizing" to turn **a** into a string to display. Then it reiterates **a** so the statement is executed. The macro sends all the information to a file called **T**, which is the trace file. The output is: Comment

```
int i = 47;
float f = 2300114.414159;
```

```
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
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;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
```

```
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
```

Studying this output should clarify your understanding of the iostream formatting member functions. Comment

Manipulators

As you can see from the previous program, calling the member functions for stream formatting operations 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 are a convenience because you can insert them for their effect within a containing expression; you don't have to create a separate function-call statement. Comment

Manipulators change the state of the stream instead of (or in addition to) processing data. When you insert **endl** in an output expression, for example, it not only inserts a newline character, but it also *flushes* the stream (that is, puts out all pending characters that have been stored in the internal stream buffer but not yet output). You can also just flush a stream like this: Comment

```
cout << flush;
```

which causes a call to the **flush( )** member function, as in

```
cout.flush();
```

as a side effect (nothing is inserted into the stream). Additional basic manipulators will change the number base to **oct** (octal), **dec** (decimal) or **hex** (hexadecimal): Comment

```
 cout << hex << "0x" << i << endl;
```

In this case, numeric output will continue in hexadecimal mode until you change it by inserting either **dec** or **oct** in the output stream.

There's also a manipulator for extraction that "eats" white space: Comment

```
cin >> ws;
```

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. The **<iostream>** header also includes **ws**, **endl**, and **flush** and the additional set shown here: Comment

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 display E for scientific values.

Manipulator	Effect
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**	Indicates the display preference for floating-point output (scientific notation vs. fixed-point decimal).

Comment

Manipulators with arguments

There are six standard manipulators, such as **setw( )**, that take arguments. These are defined in the header file <**iomanip>**, and are summarized in the following table.

Manipulator	effect
setiosflags (fmtflags n)	Equivalent to a call to **setf(n)**. The setting remains in effect until the next change, such as **ios::setf( )**.
resetiosflags(fmtflags n)	Clears only the format flags specified by **n**. The setting remains in effect until the next change, such as **ios::unsetf( )**.

Manipulator	effect
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**, such as **ios::fill( )**.
setprecision(int n)	Changes the precision to **n**, such as **ios::precision( )**.
setw(int n)	Changes the field width to **n**, such as **ios::width( )**.

Comment

If you're doing a lot of formatting, you can see how using manipulators instead of calling stream member functions can clean up your code. As an example, here's the program from the previous section rewritten to use the manipulators. (The **D( )** macro has been removed to make it easier to read.) Comment

```
//: C04:Manips.cpp
// Format.cpp using manipulators
#include <fstream>
#include <iomanip>
#include <iostream>
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::showbase | ios::uppercase
            | ios::showpos);
  trc << i << endl;
  trc << hex << i << endl
      << 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 << resetiosflags(ios::uppercase) <<
endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  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 << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;
} ///:~
```

You can see that a lot of the multiple statements have been condensed into a single chained insertion. Notice the call to

setiosflags( ) in which the bitwise-OR of the flags is passed. This could also have been done with **setf( )** and **unsetf( )** as in the previous example. Comment

When using **setw( )** with an output stream, the output expression is formatted into a temporary string that is padded with the current fill character if needed, as determined by comparing the length of the formatted result to the argument of **setw( )**. In other words, **setw( )** affects the *result string* of a formatted output operation. Likewise, using **setw( )** with input streams only is meaningful when reading *strings*, as the following example makes clear.

```
//: C04:InputWidth.cpp
// Shows limitations of setw with input
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream is("one 2.34 five");
  string temp;
  is >> setw(2) >> temp;
  assert(temp == "on");
  is >> setw(2) >> temp;
  assert(temp == "e");
  double x;
  is >> setw(2) >> x;
  double relerr = fabs(x - 2.34) / x;
  assert(relerr <=
numeric_limits<double>::epsilon());
} ///:~
```

If you attempt to read a string, **setw( )** will control the number of characters extracted quite nicely... up to a point. The first extraction gets two characters, but the second only gets one, even though we asked for two. That is because **operator>>( )** uses white space as a delimiter (unless you turn off the **skipws** flag). When trying to read a number, however, such as **x**, you cannot use **setw( )** to limit the characters read. With input streams, use only **setw( )** for extracting strings. Comment

Creating manipulators

Sometimes you'd like to create your own manipulators, and it turns out to be remarkably simple. A zero-argument manipulator such as **endl** is simply a function that takes as its argument an **ostream** reference and returns an **ostream** reference. The declaration for **endl** is Comment

```
ostream& endl(ostream&);
```

Now, when you say: Comment

```
cout << "howdy" << endl;
```

the **endl** produces the *address* of that function. So the compiler asks, "Is there a function I can call that takes the address of a function as its argument?" Predefined functions in **<iostream>** do this; they're called *applicators* (because they *apply* a function to a stream). The applicator calls its function argument, passing it the **ostream** object as its argument. You don't need to know how applicators work to create your own manipulator; you only need to know that they exist. Nonetheless, they're simple. Here's the (simplified) code for an **ostream** applicator:

```
ostream& ostream::operator<<(ostream&
(*pf)(ostream&)) {
  return pf(*this);
}
```

The actual definition is a little more complicated since it involves templates, but this code illustrates the technique. When a function such as ***pf** (that takes a stream parameter and returns a stream reference) is inserted into a stream, this applicator function is called, which in turn executes the function to which **pf** points. Applicators for **ios_base**, **basic_ios**, **basic_ostream**, and **basic_istream** are predefined in the standard C++ library. Comment

To illustrate the process, here's a trivial example that creates a manipulator called **nl** that is equivalent to just inserting a newline into a stream (i.e., no flushing of the stream occurs, as with **endl**):
Comment

```
//: C04: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;
} ///:~
```

When you insert **nl** into an output stream, such as **cout**, the following sequence of calls ensues:

```
cout.operator<<(nl) ➔ nl(cout)
```

The expression

```
os << '\n';
```

inside **nl( )** calls **ostream::operator(char)**, which of course returns the stream, which is what is ultimately returned from **nl( )**.[8] Comment

Effectors

As you've seen, zero-argument manipulators are easy to create. But what if you want to create a manipulator that takes arguments? If you inspect the **<iomanip>** header, you'll see a type called **smanip**, which is what the manipulators with arguments return.

[8] Before putting **nl** into a header file, make it an **inline** function (see Chapter 7).

You might be tempted to somehow use that type to define your own manipulators, but don't give in to the temptation. The **smanip** type is implementation-dependent, so using it would not be portable. Fortunately, you can define such manipulators in a straightforward way without any special machinery, based on a technique called an *effector* and introduced by Jerry Schwarz.[9] An effector is a simple class whose constructor formats a string representing the desired operation, along with an overloaded **operator<<** to insert that string into a stream. Here's an example with two effectors. The first outputs a truncated character string, and the second prints a number in binary. Comment

```
//: C04:Effector.cpp
// Jerry Schwarz's "effectors"
#include <cassert>
#include <limits>  // For max()
#include <sstream>
#include <string>
using namespace std;

// Put out a prefix of a string:
class Fixw {
  string str;
public:
  Fixw(const string& s, int width)
    : str(s, 0, width) {}
  friend ostream&
  operator<<(ostream& os, const Fixw& fw) {
    return os << fw.str;
  }
};

// Print a number in binary:
typedef unsigned long ulong;
class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
```

[9] Jerry Schwarz is the designer of iostreams.

```
  friend ostream& operator<<(ostream& os, const
Bin& b) {
    const ulong ULMAX =
numeric_limits<ulong>::max();
    ulong bit = ~(ULMAX >> 1); // Top bit set
    while(bit) {
      os << (b.n & bit ? '1' : '0');
      bit >>= 1;
    }
    return os;
  }
};

int main() {
  string words =
    "Things that make us happy, make us wise";
  for(int i = words.size(); --i >= 0;) {
    ostringstream s;
    s << Fixw(words, i);
    assert(s.str() == words.substr(0, i));
  }
  ostringstream xs, ys;
  xs << Bin(0xCAFEBABEUL);
  assert(xs.str() ==

"1100""1010""1111""1110""1011""1010""1011""1110");
  ys << Bin(0x76543210UL);
  assert(ys.str() ==

"0111""0110""0101""0100""0011""0010""0001""0000");
} ///:~
```

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, inserts it into the first argument, the **ostream**, and then returns the **ostream** so that it can be used in a chained expression. When you use **Fixw** in an expression like this:
Comment

```
cout << Fixw(string, i) << endl;
```

a *temporary object* is created by the call to the **Fixw** constructor, and that temporary object is passed to **operator<<**. The effect is that of a manipulator with arguments. The temporary **Fixw** object persists until the end of the statement. Comment

The **Bin** effector relies on the fact that shifting an unsigned number to the right shifts zeros into the high bits. We use **numeric_limits<unsigned long>::max()** (the largest **unsigned long** value, from the standard header **<limits>**) to produce a value with the high bit set, and this value is moved across the number in question (by shifting it to the right), masking each bit in turn. We've juxtaposed string literals in the code for readability; the separate strings are of course concatenated into one by the compiler. Comment

Historically, 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, this problem is eliminated. Comment

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 such as **sed** and **awk** from UNIX are perhaps the most well known, but a text editor also fits this category), they generally have some limitations. Both **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. Comment

Maintaining class library source code

Generally, when you create a class, you think in library terms: you make a header file **Name.h** for the class declaration, and you create a file in which the member functions are implemented, called

Name.cpp. These files have certain requirements: a particular coding standard (the program shown here uses 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.) Comment

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. Comment

```
//: C04:Cppcheck.cpp
// Configures .h & .cpp files to conform to style
// standard. Tests existing files for conformance.
#include <fstream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

bool startsWith(const string& base, const string&
key) {
  return base.compare(0, key.size(), key) == 0;
}

void cppCheck(string fileName) {
  enum bufs { BASE, HEADER, IMPLEMENT,
    HLINE1, GUARD1, GUARD2, GUARD3,
    CPPLINE1, INCLUDE, BUFNUM };
  string part[BUFNUM];
  part[BASE] = fileName;
  // Find any '.' in the string:
  size_t loc = part[BASE].find('.');
  if(loc != string::npos)
    part[BASE].erase(loc); // Strip extension
  // Force to upper case:
  for(size_t i = 0; i < part[BASE].size(); i++)
    part[BASE][i] = toupper(part[BASE][i]);
```

```
  // Create file names and internal lines:
  part[HEADER] = part[BASE] + ".h";
  part[IMPLEMENT] = part[BASE] + ".cpp";
  part[HLINE1] = "//" ": " + part[HEADER];
  part[GUARD1] = "#ifndef " + part[BASE] + "_H";
  part[GUARD2] = "#define " + part[BASE] + "_H";
  part[GUARD3] = "#endif // " + part[BASE] +"_H";
  part[CPPLINE1] = string("//") + ": "
    + part[IMPLEMENT];
  part[INCLUDE] = "#include \"" + part[HEADER] +
"\"";
  // First, try to open existing files:
  ifstream existh(part[HEADER].c_str()),
           existcpp(part[IMPLEMENT].c_str());
  if(!existh) { // Doesn't exist; create it
    ofstream newheader(part[HEADER].c_str());
    assure(newheader, part[HEADER].c_str());
    newheader << part[HLINE1] << endl
      << part[GUARD1] << endl
      << part[GUARD2] << endl << endl
      << part[GUARD3] << endl;
  } else { // Already exists; verify it
    stringstream hfile; // Write & read
    ostringstream newheader; // Write
    hfile << existh.rdbuf();
    // Check that first three lines conform:
    bool changed = false;
    string s;
    hfile.seekg(0);
    getline(hfile, s);
    bool lineUsed = false;
    // The call to good() is for Microsoft (later
too)
    for (int line = HLINE1; hfile.good() && line
<= GUARD2;
         ++line) {
      if(startsWith(s, part[line])) {
        newheader << s << endl;
        lineUsed = true;
        if (getline(hfile, s))
          lineUsed = false;
      } else {
```

```
        newheader << part[line] << endl;
        changed = true;
        lineUsed = false;
      }
    }
    // Copy rest of file
    if (!lineUsed)
      newheader << s << endl;
    newheader << hfile.rdbuf();
    // Check for GUARD3
    string head = hfile.str();
    if(head.find(part[GUARD3]) == string::npos) {
      newheader << part[GUARD3] << endl;
      changed = true;
    }
    // If there were changes, overwrite file:
    if(changed) {
      existh.close();
      ofstream newH(part[HEADER].c_str());
      assure(newH, part[HEADER].c_str());
      newH << "//@//\n"  // Change marker
        << newheader.str();
    }
  }
  if(!existcpp) { // Create cpp file
    ofstream newcpp(part[IMPLEMENT].c_str());
    assure(newcpp, part[IMPLEMENT].c_str());
    newcpp << part[CPPLINE1] << endl
      << part[INCLUDE] << endl;
  } else { // Already exists; verify it
    stringstream cppfile;
    ostringstream newcpp;
    cppfile << existcpp.rdbuf();
    // Check that first two lines conform:
    bool changed = false;
    string s;
    cppfile.seekg(0);
    getline(cppfile, s);
    bool lineUsed = false;
    for (int line = CPPLINE1;
         cppfile.good() && line <= INCLUDE;
         ++line) {
```

```
      if(startsWith(s, part[line])) {
        newcpp << s << endl;
        lineUsed = true;
        if (getline(cppfile, s))
          lineUsed = false;
      } else {
        newcpp << part[line] << endl;
        changed = true;
        lineUsed = false;
      }
    }
    // Copy rest of file
    if (!lineUsed)
      newcpp << s << endl;
    newcpp << cppfile.rdbuf();
    // If there were changes, overwrite file:
    if(changed){
      existcpp.close();
      ofstream newCPP(part[IMPLEMENT].c_str());
      assure(newCPP, part[IMPLEMENT].c_str());
      newCPP << "//@//\n"  // Change marker
        << newcpp.str();
    }
  }
}

int main(int argc, char* argv[]) {
  if(argc > 1)
    cppCheck(argv[1]);
  else
    cppCheck("cppCheckTest.h");
} ///:~
```

First notice the useful function **startsWith( )**, which does just what its name says—it returns **true** if the first string argument starts with the second argument. This is used when looking for the expected comments and include-related statements. Having the array of strings, **part**, allows for easy looping through the series of expected statements in source code. If the source file doesn't exist, we merely write the statements to a new file of the given name. If the file does exist, we search a line at a time, verifying that the expected lines occur. If they are not present, they are inserted. Special care has to be taken to make sure we don't drop existing lines (see where we use the Boolean variable **lineUsed**). Notice that we use a **stringstream** for an existing file, so we can first write the contents of the file to it and then read from it and search it. Comment

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. Comment

If you run this program without any arguments, the following two files are created:

```
// CPPCHECKTEST.h
#ifndef CPPCHECKTEST_H
#define CPPCHECKTEST_H

#endif // CPPCHECKTEST_H
```

```
// CPPCHECKTEST.cpp
#include "CPPCHECKTEST.h"
```

(We removed the colon after the double-slash in the first comment lines so as not to confuse the book's code extractor. It will appear in the actual output produced by **cppCheck.**)

You can experiment by removing selected lines from these files and re-running the program. Each time you will see that the correct lines are added back in. When a file is modified, the string "`//@//`" is placed as the first line of the file to bring the change to your attention. You will need to remove this line before you process the file again (otherwise **cppcheck** will assume the initial comment line is missing). Comment

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. When you run your compiler, it should generate error messages, and you should see all the numbers appear when you compile all the files. This program also appends the modified line to a special file so that you can easily locate any lines that don't generate errors. Comment

```
//: C04:Showerr.cpp
// Un-comment error generators
#include <cstddef>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;

const string 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.";

class Showerr {
  const int CHAP;
  const string MARKER, FNAME;
  // File containing error number counter:
  const string ERRNUM;
  // File containing error lines:
```

```
  const string ERRFILE;
  stringstream edited; // Edited file
  int counter;
public:
  Showerr(const string& f, const string& en,
    const string& ef, int c) : FNAME(f),
MARKER("//!"),
    ERRNUM(en), ERRFILE(ef), CHAP(c) { counter =
0; }
  void replaceErrors() {
    ifstream infile(FNAME.c_str());
    assure(infile, FNAME.c_str());
    ifstream count(ERRNUM.c_str());
    if(count) count >> counter;
    int linecount = 1;
    string buf;
    ofstream errlines(ERRFILE.c_str(), ios::app);
    assure(errlines, ERRFILE.c_str());
    while(getline(infile, buf)) {
      // Find marker at start of line:
      size_t pos = buf.find(MARKER);
      if(pos != string::npos) {
        // Erase marker:
        buf.erase(pos, MARKER.size() + 1);
        // Append counter & error info:
        ostringstream out;
        out << buf << " // (" << ++counter << ") "
            << "Chapter " << CHAP
            << " File: " << FNAME
            << " Line " << linecount << endl;
        edited << out.str();
        errlines << out.str(); // Append error
file
      }
      else
        edited << buf << "\n"; // Just copy
      linecount++;
    }
  }
  void saveFiles() {
    ofstream outfile(FNAME.c_str()); // Overwrites
    assure(outfile, FNAME.c_str());
```

```
    outfile << edited.rdbuf();
    ofstream count(ERRNUM.c_str()); // Overwrites
    assure(count, ERRNUM.c_str());
    count << counter; // Save new counter
  }
};

int main(int argc, char* argv[]) {
  const string ERRCOUNT("../errnum.txt"),
    ERRFILE("../errlines.txt");
  requireMinArgs(argc, 1, usage.c_str());
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(ERRCOUNT.c_str()); // Delete files
        remove(ERRFILE.c_str());
        return 0;
      default:
        cerr << usage << endl;
        return 1;
    }
  }
  if (argc == 3) {
    Showerr s(argv[1], ERRCOUNT, ERRFILE,
atoi(argv[2]));
    s.replaceErrors();
    s.saveFiles();
  }
} ///:~
```

You can replace the marker with one of your choice. Comment

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 string stream, **edited**. When the whole file is processed, it is closed (by reaching the end of a scope), it is reopened as an output file, and **edited** is poured into the file. Also notice the counter is saved in an external file. The next time this program is invoked, it continues to sequence the counter. Comment

A simple datalogger

This example shows an approach you might take to log data to disk and later retrieve it for processing. It is meant to produce a temperature-depth profile of the ocean at various points. To hold the data, a class is used: Comment

```
//: C04:DataLogger.h
// Datalogger record layout
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iosfwd>
#include <string>
using std::ostream;

struct Coord {
  int deg, min, sec;
  Coord(int d=0, int m=0, int s=0)
    : deg(d), min(m), sec(s) {}
  std::string toString() const;
};
ostream& operator<<(ostream&, const Coord&);

class DataPoint {
  std::time_t timestamp; // Time & day
  Coord latitude, longitude;
  double depth, temperature;
public:
  DataPoint(std::time_t ts, const Coord& lat,
            const Coord& lon, double dep, double
temp)
    : timestamp(ts), latitude(lat),
longitude(lon),
      depth(dep), temperature(temp) {}
  DataPoint() : timestamp(0), depth(0),
temperature(0) {}
  friend ostream& operator<<(ostream&, const
DataPoint&);
};
#endif // DATALOG_H ///:~
```

A **DataPoint** consists of a time stamp, which is stored as a **time**_t value as defined in **<ctime>**, longitude and latitude coordinates, and values for depth and temperature. We use inserters for easy formatting. Here's the implementation file: Comment

```
//: C04:Datalogger.cpp {O}
// Datapoint implementations
#include "DataLogger.h"
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

ostream& operator<<(ostream& os, const Coord& c) {
  return os << c.deg << '*' << c.min << '\''
            << c.sec << '"';
}

string Coord::toString() const {
  ostringstream os;
  os << *this;
  return os.str();
}

ostream& operator<<(ostream& os, const DataPoint&
d) {
  os.setf(ios::fixed, ios::floatfield);
  char fillc = os.fill('0'); // Pad on left with
'0'
  tm* tdata = localtime(&d.timestamp);
  os << setw(2) << tdata->tm_mon << '\\'
     << setw(2) << tdata->tm_mday << '\\'
     << setw(2) << tdata->tm_year+1900 << ' '
     << setw(2) << tdata->tm_hour << ':'
     << setw(2) << tdata->tm_min << ':'
     << setw(2) << tdata->tm_sec;
  os.fill(' '); // Pad on left with ' '
  streamsize prec = os.precision(4);
  os << " Lat:" << setw(9) <<
d.latitude.toString()
```

```
     << ", Long:" << setw(9) <<
d.longitude.toString()
     << ", depth:" << setw(9) << d.depth
     << ", temp:" << setw(9) << d.temperature;
  os.fill(fillc);
  os.precision(prec);
  return os;
} ///:~
```

The **Coord::toString( )** function is necessary because the **DataPoint** inserter calls **setw( )** before it prints the latitude and longitude. If we used the stream inserter for **Coord** instead, the width would only apply to the first insertion (that is, to **Coord::deg**), since width changes are always reset immediately. The call to **setf( )** causes the floating-point output to be fixed-precision, and **precision( )** sets the number of decimal places to four. Notice how we restore the fill character and precision to whatever they were before the inserter was called. Comment

To get the values from the time encoding stored in **DataPoint::timestamp**, we call the function **std::localtime( )**, which returns a static pointer to a **tm** object. The **tm struct** has the following layout:

```
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; // 0-11 months
  int tm_year; // Years since 1900
  int tm_wday; // Sunday == 0, etc.
  int tm_yday; // 0-365 day of year
  int tm_isdst; // Daylight savings?
};
```

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 the **DataPoint** inserter. You can also print it out to the screen, but it's easier to inspect in file form. Comment

```
//: C04:Datagen.cpp
// Test data generator
//{L} DataLogger
#include <cstdlib>
#include <cstring>
#include <fstream>
#include "../require.h"
#include "DataLogger.h"
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;
  Coord lat(45,20,31);
  Coord lon(22,34,18);
  // Seed random number generator:
  srand(time(&timer));
  for(int i = 0; i < 100; i++, timer += 55) {
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += 1.0 / fraction;
    double newtemp = 150 + rand()%200; // Kelvin
    fraction = rand() % 100 + 1;
    newtemp += 1.0 / fraction;
    const DataPoint d(timer, Coord(45,20,31),
                      Coord(22,34,18), newdepth,
                      newtemp);
    data << d << endl;
    bindata.write(reinterpret_cast<const
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. To illustrate the formatting used for the text file, here is the first line of **data.txt** (the line wraps because it's longer than this page will allow): Comment

```
07\28\2002 12:54:40 Lat:45*20'31", Long:22*34'18",
depth:  16.0164, temp: 242.0122
```

The Standard C library function **time( )** updates the **time_t** value its argument points to with an encoding of the current time, which on most platforms 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 also a convenient way to seed the random number generator with the Standard C library function **srand( )**, as is done here. Comment

After this, the **timer** is incremented by 55 seconds to give an interesting interval between readings in this simulation. Comment

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 a platform-dependent constant, **RAND_MAX**, defined in **<cstdlib>** (usually the value of the platform's largest unsigned integer). To put this in a desired range, use the remainder 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). Comment

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 a **char*** because that's what **write( )** expects for narrow streams. The second argument is the number of characters to write, which in this case is the size of the **DataPoint** object (again, because we're using *narrow streams*). 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*, which writes the data referred to by pointers and defines new pointers when read back in later. (We don't talk about serialization in this volume—most vendor class libraries have some sort of serialization structure built into them.) Comment

Verifying and viewing the data

To check the validity of the data stored in binary format, you can read it into memory with the **read( )** member function for input streams, and compare it to the text file created earlier by **Datagen.cpp**. The following example just writes the formatted results to **cout**, but you can redirect this to a file and then use a file comparison utility to verify that it is identical to the original. Comment

```
//: C04:Datascan.cpp
// Test data generator
//{L} DataLogger
#include <fstream>
#include <iostream>
#include "DataLogger.h"
#include "../require.h"
using namespace std;

int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  DataPoint d;
  while (bindata.read(reinterpret_cast<char*>(&d),
         sizeof d))
    cout << d << endl;
} ///:~
```

Internationalization

The software industry is now a healthy, worldwide economic market, and applications that can run in various languages and cultures are in demand. As early as the late 1980s, the C Standards Committee added support for non-U.S. formatting conventions with their *locale* mechanism. A locale is a set of preferences for displaying certain entities such as dates and monetary quantities. In the 1990s, the C Standards Committee approved an addendum to Standard C that specified functions to handle *wide characters* (denoted by the type **wchar_t**), which allow support for character sets other than ASCII and its commonly used Western European extensions. Although the size of a wide character is not specified, some platforms implement them as 32-bit quantities, so they can hold the encodings specified by the Unicode Consortium, as well as mappings to multi-byte characters sets defined by Asian standards bodies. C++ has integrated support for both wide characters and locales into the iostreams library. Comment

Wide Streams

A *wide stream* is a simply a stream class that handles wide characters. All the examples so far (except for the last traits example in Chapter 3) have used *narrow* streams, meaning streams that hold instances of **char**. Since stream operations are essentially the same, no matter the underlying character type, they are encapsulated generically as templates. As we mentioned earlier, all input streams, for example, are connected somehow to the **basic_istream** class template, which is defined as follows:

```
template<class charT, class traits =
char_traits<charT> >
class basic_istream {…};
```

In fact, all input stream types are specializations of this template, according to the following type definitions:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_istringstream<char> istringstream;
typedef basic_istringstream<wchar_t>
wistringstream;
```

All other stream types are defined in similar fashion.

In a "perfect" world, this is all you'd have to do to have streams of different character types. In reality, things aren't that simple. The reason is that the character-processing functions provided for **char** and **wchar_t** don't have the same names. To compare two narrow strings, for example, you use the **strcmp( )** function. For wide characters, that function is named **wcscmp( )**. (Remember these originated in C, which does not have function overloading, hence unique names are a must.) For this reason, a generic stream can't just call **strcmp( )** in response to a comparison operator. There needs to be a way for the correct low-level functions to be called automatically. Comment

The principle that guides the solution is well known. You simply "factor out" the differences into a new abstraction. The operations you can perform on characters have been abstracted into the **char_traits** template, which has predefined specializations for char and **wchar_t**, as we discussed at the end of the previous chapter. To compare two strings, then, **basic_string** just calls **traits::compare( )** (remember that **traits** is the second template parameter), which in turn calls either **strcmp( )** or **wcscmp( )**, depending on which specialization is being used (transparent to **basic_string**, of course). Comment

You only need to be concerned about **char_traits** if you must access the low-level character processing functions; most of the time you really don't care. You should consider, however, making your inserters and extractors more robust by defining them as templates, just in case someone wants to use them on a wide stream.

To illustrate, recall again the **Date** class inserter from the beginning of this chapter. We originally declared it as:

```
ostream& operator<<(ostream&, const Date&);
```

This accommodates only narrow streams. To make it generic, we simply make it a template based on **basic_ostream**:

```
template<class charT, class traits>
```

```
std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os,
           const Date& d) {
  charT fillc = os.fill(os.widen('0'));
  charT dash = os.widen('-');
  os << setw(2) << d.month << dash
     << setw(2) << d.day << dash
     << setw(4) << d.year;
  os.fill(fillc);
  return os;
}
```

Notice that we also have to replace **char** with the template parameter **charT** in the definition of **fillc**, since it could be either **char** or **wchar_t**, depending on the template instantiation being used. Comment

Since you don't know when you're writing the template which type of stream you have, you need a way to automatically convert character literals to the correct size for the stream. This is the job of the **widen( )** member function. The expression **widen('-')**, for example, converts its argument to **L'-'** (the literal syntax equivalent to the conversion **wchar_t('-')**) if the stream is a wide stream and leaves it alone otherwise. There is also a **narrow( )** function that converts to a **char** if needed. Comment

We can use **widen( )** to write a generic version of the **nl** manipulator we presented earlier in the chapter.

```
template<class charT, class traits>
basic_ostream<charT,traits>&
nl(basic_ostream<charT,traits>& os) {
  return os << charT(os.widen('\n'));
}
```

Locales

Perhaps the most notable difference in typical numeric computer output from country to country is the punctuator used to separate the integer and fractional parts of a real number. In the United States, a period is used to denote a decimal point, but in much of the world, a comma is expected instead. It would be quite inconvenient to do all your own formatting for locale-dependent displays. Once again, creating an abstraction that handles these differences solves the problem.

That abstraction is the *locale*. All streams have an associated locale object that they use for guidance on how to display certain quantities for different cultural environments. A locale manages the categories of culture-dependent display rules, which are defined as follows:

Category	Effect
collate	allows comparing strings according to different, supported collating sequences
ctype	abstracts the character classification and conversion facilities found in **<cctype>**
monetary	supports different displays of monetary quantities
numeric	supports different display formats of real numbers, including radix (decimal point) and grouping (thousands) separators
time	supports various international formats for display of date and time
messages	scaffolding to implement context-dependent message catalogs (such as for error messages in different languages)

The following program illustrates basic locale behavior:

```
//: C04:Locale.cpp
//{-g++}
//{-bor}
//{-edg}
// Illustrates effects of locales
```

```
#include <iostream>
#include <locale>
using namespace std;
int main() {
  locale def;
  cout << def.name() << endl;
  locale current = cout.getloc();
  cout << current.name() << endl;
  float val = 1234.56;
  cout << val << endl;
  // Change to French/France
  cout.imbue(locale("french"));
  current = cout.getloc();
  cout << current.name() << endl;
  cout << val << endl;

  cout << "Enter the literal 7890,12: ";
  cin.imbue(cout.getloc());
  cin >> val;
  cout << val << endl;
  cout.imbue(def);
  cout << val << endl;
} ///:~
```

Here's the output:

```
C
C
1234.56
French_France.1252
1234,56
Enter the literal 7890,12: 7890,12
7890,12
7890.12
```

The default locale is the "C" locale, which is what C and C++ programmers have been used to all these years (basically, English language and American culture). All streams are initially "imbued" with the "C" locale. The **imbue( )** member function changes the locale that a stream uses. Notice that the full ISO name for the "french" locale is displayed (that is, French used in France vs. French used in another country). This example shows that this locale uses a comma for a radix point in numeric display. We have to change **cin** to the same locale if we want to do input according to the rules of this locale.

Each locale category is divided into number of *facets*, which are classes encapsulating the functionality that pertains to that category. For example, the **time** category has the facets **time_put** and **time_get**, which contain functions for doing time and date **input** and **output** respectively. The **monetary** category has facets **money_get**, **money_put**, and **moneypunct**. (The latter facet determines the currency symbol.) The following program illustrates the **moneypunct** facet. (The **time** facet requires a sophisticated use of iterators which is beyond the scope of this chapter.)

```
//: C04:Facets.cpp
//{-bor}
//{-g++}
#include <iostream>
#include <locale>
#include <string>
using namespace std;

int main() {
  // Change to French/France
  locale loc("french");
  cout.imbue(loc);

  string currency =
    use_facet<moneypunct<char>
>(loc).curr_symbol();
  char point =
    use_facet<moneypunct<char>
>(loc).decimal_point();
  cout << "I made " << currency << 12.34 << "
today!"
       << endl;
} ///:~
```

The output shows the French currency symbol and decimal separator:

```
I made Ç12,34 today!
```

You can also define your own facets to construct customized locales[10]. Be aware that the overhead for locales is considerable. In fact, some library vendors provide different "flavors" of the standard C++ library to accommodate environments that have limited space.[11]

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. However, be aware that some additional features in iostreams are not used often, but you can discover them by looking at the iostream header files and by reading your compiler's documentation on iostreams or the references mentioned in this chapter and in the book's preface. Comment

Exercises

1. Open a file by creating an **ifstream** object called in. Make an **ostringstream** object called **os**, and read the entire contents into the **os** using the **rdbuf( )** member function. Get the underlying **string** from **os** 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.
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.

---

[10] See the Langer & Kreft book mentioned earlier for more detailed information.

[11] See, for example, Dinkumware's Abridged library at www.dinkumware.com. This library omits locale support. and exception support is optional.

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.
5. Open a file by creating an **ifstream** object. Make an **ostringstream** object and read the entire contents into the **ostringstream** using the **rdbuf( )** member function. Extract a **string** copy of the underlying buffer and capitalize every character in the file using the Standard C **toupper( )** macro defined in **<cctype>**. Write the result out to a new file.
6. 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 write out the lines (with line numbers) that match to the new file.
7. Write a program that adds a copyright notice to the beginning of all source-code files indicated by the program's command-line arguments.
8. We know that **setw()** allows for a minimum of characters read in, but what if you wanted to read a maximum? Write an effector that allows the user to specify a maximum number of characters to extract.
9. Demonstrate to yourself that if the fail or bad bit is set, and you subsequently turn on stream exceptions, that the stream will immediately throw an exception.
10. String streams accommodate easy conversions, but they come with a price. Write a program that races **atoi( )** against the **stringstream** conversion system to see the overhead involved with **stringstream**. Comment

5: Templates in depth

Intro stuff

intro stuffComment

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:Comment

```
//: C05: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 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.Comment

Default template arguments

The typename keyword

Consider the following:Comment

```
//: C05: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**.Comment

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 something 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.Comment

The **typename** keyword tells the compiler to interpret a particular name as a type. It must be used for a name that:Comment

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.Comment

The short version of the rule is: if your type is a qualified name that involves a template argument, you must use **typename**.Comment

Typedefing a typename

The **typename** keyword does not automatically create a **typedef**. A line which reads:Comment

```
typename Seq::iterator It;
```

causes a variable to be declared of type **Seq::iterator**. If you mean to make a **typedef**, you must say:Comment

```
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:Comment

```
//: C05: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 relatively long time after templates were introduced.Comment

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.Comment

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.[1] 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.
Comment

A string conversion system

Comment

```
//: C05:stringConv.h
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>
```

[1] See *C++ Inside & Out* (Osborne/McGraw-Hill, 1993) by the author, Chapter 10.

```
template<typename T>
T fromString(const std::string& s) {
  std::istringstream is(s);
  T t;
  is >> t;
  return t;
}

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: Comment

```
//: C05: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:Comment

```
i == "1234"
x == "567.89"
c == "(1,2)"

i == 1234
x == 567.89
c == (1,2)
```

Comment

A memory allocation system

There are 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:Comment

```
//: C05: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).Comment

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.Comment

If the number of elements requested is zero, the storage is freed. There's an additional function template **freemem( )** that aliases this behavior.Comment

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:Comment

```
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:Comment

```
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( )**.Comment

Here's a program to test **getmem( )**. It allocates storage and fills it up with values, then increases that amount of storage:Comment

```
//: C05: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. Comment

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.Comment

Type induction in function templates

As a simple but very useful example, consider the following:Comment

```
//: :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:Comment

```
//: C05:ArraySize.cpp
//{-msc}
//{-bor}
// 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.Comment

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[2]:Comment

```
//: C05:TemplateFunctionAddress.cpp
// Taking the address of a function generated
// from a template.
//{L} ../TestSuite/Test

template <typename T> void f(T*) {}

void h(void (*pf)(int*)) {}

template <class T>
  void g(void (*pf)(T*)) {}
```

[2] I am indebted to Nathan Myers for this example.

```
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**. Comment

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. Comment

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:Comment

```
//: C05: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.Comment

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.Comment

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.Comment

To test the various overloaded versions of **apply( )**, the class **Gromit**[3] is created containing functions with different numbers of arguments:Comment

```
//: C05:Gromit.h
// The techno-dog. Has member functions
// with various numbers of arguments.
#include <iostream>

class Gromit {
  int arf;
public:
  Gromit(int arf = 1) : arf(arf + 1) {}
  void speak(int) {
    for(int i = 0; i < arf; i++)
      std::cout << "arf! ";
    std::cout << std::endl;
```

[3] A reference to the British animated short *The Wrong Trousers* by Nick Park.

```
  }
  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:Comment

```
//: C05:applyGromit.cpp
// Test applySequence.h
//{L} ../TestSuite/Test
#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 implementationComment

Template-templates

```
//: C05:TemplateTemplate.cpp
//{-bor}
//{-msc}
//{-g++}
#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);
} ///:~
```

Comment

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:Comment

```
dogs.apply(&Gromit::sit);
```

This is analogous to the act (in Chapter XX) of bringing ordinary functions inside a class.[4]Comment

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:Comment

```
//: C05: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()) {
```

[4] Check your compiler version information to see if it supports member function templates.

```
      ((*it)->*f)();
      it++;
    }
  }
  // 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.Comment

You can see how the function calls are also simpler for the client programmer:Comment

```
//: C05:applyGromit2.cpp
// Test applyMember.h
//{L} ../TestSuite/Test
//{-g++}
//{-msc}
#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.Comment

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:Comment

```
//: C05:Sorted.h
// Template specialization
#ifndef SORTED_H
```

```
#define SORTED_H
#include <string>
#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(std::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.Comment

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.Comment

Here's a test for **Sorted.h** that uses the unique random number generator introduced earlier in the chapter:Comment

```
//: C05:Sorted.cpp
// Testing template specialization
//{L} ../TestSuite/Test
//{-msc}
#include "Sorted.h"
#include "Urand.h"
```

```
#include "../arraySize.h"
#include <iostream>
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.Comment

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 toComment

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.)Comment

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:Comment

```
//: C05:Nobloat.h
// Templatized InheritStack.cpp
#ifndef NOBLOAT_H
#define NOBLOAT_H
#include "../C0B/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.Comment

```
//: C05:NobloatTest.cpp
//{L} ../TestSuite/Test
#include "Nobloat.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  ifstream in("NobloatTest.cpp");
  assure(in, "NobloatTest.cpp");
  NBStack<string> 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;
```

```
  }
} ///:~
```

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:Comment

```
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:Comment

```
//: C05:ExplicitInstantiation.cpp
//{L} ../TestSuite/Test
//{-msc}
#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.Comment

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:Comment

```
//: C05: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.Comment

The inclusion vs. separation models

The export keyword

Template programming idioms

The "curiously-recurring template"

Implementing Locales

Traits

Template Metaprogramming

Expression Templates

Compile-time Assertions

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. Comment

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. Comment

Exercises

1. Exercise 1
2. Exercise 2
3. Exercise 3
4. Etc.

Comment

6: Generic algorithms

Algorithms are at the core of computing. To be able to write an algorithm once and for all to work with any type of sequence makes your programs both simpler and safer. The ability to customize algorithms at runtime has revolutionalized software development.

The subset of the standard C++ library known as the Standard Template Library (STL) was originally designed around *generic algorithms*—code that processes sequences of any type of values in a type-safe manner. The goal was to use predefined algorithms for almost every task, instead of hand-coding loops every time you need to process a collection of data. This power comes with a bit of a learning curve, however. By the time you get to the end of this chapter, you should be able to decide for yourself whether you find the algorithms addictive or too confusing to remember. If you're like most people, you'll resist them at first but then tend to use them more and more. Comment

A first look

Among other things, the generic algorithms in the standard library 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 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 just say **copy( )**. This is the kind of thing we've been doing in computer programming from the beginning—creating high-level abstractions to express *what* you're doing and spending less time saying *how* you're doing it. The *how* has been solved once and for all and is hidden in the algorithm's code, ready to be reused on demand. Comment

Here's an example of how to use the **copy** algorithm:

```
//: C06:CopyInts.cpp
// Copies ints without an explicit loop
#include <algorithm>
#include <cassert>
#include <cstddef>  // For size_t
using namespace std;

int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  copy(a, a + SIZE, b);
  for (int i = 0; i < SIZE; ++i)
    assert(a[i] == b[i]);
} ///:~
```
Comment

The **copy** algorithm's first two parameters represent the *range* of the input sequence—in this case the array **a**. Ranges are denoted by a pair of pointers. The first points to the first element of the sequence, and the second points one position *past the end* of the array (right after the last element). This may seem strange at first, but it is an old C idiom that comes in quite handy. For example, the difference of these two pointers yields the number of elements in the sequence. More important, in implementing **copy( )**, the second pointer can act as a sentinel to stop the iteration through the sequence. The third argument refers to the beginning of the output sequence, which is the array **b** in this example. It is assumed that the array that **b** represents has enough space to receive the copied elements. Comment

The **copy( )** algorithm wouldn't be very exciting if it could only process integers. It can in fact copy any sequence. The following example copies **string** objects. Comment

```
//: C06:CopyStrings.cpp
// Copies strings
#include <algorithm>
#include <cassert>
#include <cstddef>
```

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

int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  copy(a, a + SIZE, b);
  assert(equal(a, a + SIZE, b));
} ///:~
```
Comment

This example introduces another algorithm, **equal( )**, which returns **true** only if each element in the first sequence is equal (using its **operator==( )**) to the corresponding element in the second sequence. This example traverses each sequence twice, once for the copy, and once for the comparison, without a single explicit loop! Comment

Generic algorithms achieve this flexibility because they are function templates, of course. If you guessed that the implementation of **copy( )** looked something like the following, you'd be "almost" right. Comment

```
template<typename T>
void copy(T* begin, T* end, T* dest) {
  while (begin != end)
    *begin++ = *dest++;
}
```
Comment

We say "almost," because **copy( )** can actually process sequences delimited by anything that acts like a pointer, such as an iterator. In this way, **copy( )** can duplicate a vector, as in the following example. Comment

```
//: C06:CopyVector.cpp
// Copies the contents of a vector
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <vector>
using namespace std;
```

```
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2(SIZE);
  copy(v1.begin(), v1.end(), v2.begin());
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

Comment

The first vector, **v1**, is initialized from the sequence of integers in the array **a**. The definition of the vector **v2** uses a different vector constructor that makes room for **SIZE** elements, initialized to zero (the default value for integers).

As with the array example earlier, it's important that **v2** have enough space to receive a copy of the contents of **v1**. For convenience, a special library function, **back_inserter( )**, returns a special type of iterator that *inserts* elements instead of *overwriting* them, so memory is expanded automatically by the container as needed. The following example uses **back_inserter( ),** so it doesn't have to expand the size of the output vector, **v2**, ahead of time. Comment

```
//: C06:InsertVector.cpp
// Appends the contents of a vector to another
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iterator>
#include <vector>
using namespace std;

int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  vector<int> v1(a, a + SIZE);
  vector<int> v2;  // v2 is empty here
  copy(v1.begin(), v1.end(), back_inserter(v2));
  assert(equal(v1.begin(), v1.end(), v2.begin()));
} ///:~
```

The **back_inserter( )** function is defined in the **<iterator>** header. We'll explain how insert iterators work in depth in the next chapter. Comment

Since iterators are identical to pointers in all essential ways, you can write the algorithms in the standard library in such a way as to allow both pointer and iterator arguments. For this reason, the implementation of **copy( )** looks more like the following code. Comment

```
template<typename Iterator>
void copy(Iterator begin, Iterator end, Iterator
dest) {
  while (begin != end)
    *begin++ = *dest++;
}
```

Whichever argument type you use in the call, **copy( )** assumes it properly implements the indirection and increment operators. If it doesn't, you'll get a compile-time error. Comment

Predicates

At times, you might want to copy only a well-defined subset of one sequence to another, such as only those elements that satisfy a certain condition. To achieve this flexibility, many algorithms have alternate calling sequences that allow you to supply a *predicate*, which is simply a function that returns a Boolean value based on some criterion. Suppose, for example, that you only want to extract from a sequence of integers those numbers that are less than or equal to 15. A version of **copy( )** called **remove_copy_if( )** can do the job, like this: Comment

```
//: C06:CopyInts2.cpp
// Ignores ints that satisfy a predicate
#include <algorithm>
#include <cstddef>
#include <iostream>
using namespace std;
// You supply this predicate
```

```
bool gt15(int x) {
  return 15 < x;
}
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  int b[SIZE];
  int* endb = remove_copy_if(a, a+SIZE, b, gt15);
  int* beginb = b;
  while (beginb != endb)
    cout << *beginb++ << endl; // Prints 10 only
} ///:~
```

Comment

The **remove_copy_if( )** function template takes the usual range-delimiting pointers, followed by a predicate of your choosing. The predicate must be a pointer to function[1] that takes a single argument of the same type as the elements in the sequence, and it must return a **bool**. In this case, the function **gt15** returns **true** if its argument is greater than 15. The **remove_copy_if( )** algorithm applies **gt15( )** to each element in the input sequence and ignores those elements when writing to the output sequence. Comment

The following program illustrates yet another variation of the **copy** algorithm.

```
//: C06:CopyStrings2.cpp
// Replaces strings that satisfy a predicate
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;
// The predicate
bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}
int main() {
  string a[] = {"read", "my", "lips"};
```

[1] Or something that is callable as a function, as you'll see shortly.

```
  const size_t SIZE = sizeof a / sizeof a[0];
  string b[SIZE];
  string* endb =
    replace_copy_if(a, a + SIZE, b, contains_e,
                    string("kiss"));
  string* beginb = b;
  while (beginb != endb)
    cout << *beginb++ << endl;
} ///:~
```
Comment

Instead of just ignoring elements that don't satisfy the predicate, **replace_copy_if( )** substitutes a fixed value for such elements when populating the output sequence. The output in this case is

```
kiss
my
lips
```

because the original occurrence of "read", the only input string containing the letter *e*, is replaced by the word "kiss", as specified in the last argument in the call to **replace_copy_if( )**. Comment

The **replace_if( )** algorithm changes the original sequence in place, instead of writing to a separate output sequence, as the following program shows.

```
//: C06:ReplaceStrings.cpp
// Replaces strings in-place
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
using namespace std;
bool contains_e(const string& s) {
  return s.find('e') != string::npos;
}
int main() {
  string a[] = {"read", "my", "lips"};
  const size_t SIZE = sizeof a / sizeof a[0];
  replace_if(a, a + SIZE, contains_e,
string("kiss"));
```

```
  string* p = a;
  while (p != a + SIZE)
    cout << *p++ << endl;
} ///:~
```

Stream iterators

Like any good software library, the Standard C++ Library attempts to provide convenient ways to automate common tasks. We mentioned in the beginning of this chapter that you can use generic algorithms in place of looping constructs. So far, however, our examples have still used an explicit loop to print their output. Since printing output is one of the most common tasks, you would hope for a way to automate that too. Comment

That's where *stream iterators* come in. A stream iterator allows you to use a stream as either an input or an output sequence. To eliminate the output loop in the **CopyInts2.cpp** program, for instance, you can do something like the following. Comment

```
//: C06:CopyInts3.cpp
// Uses an output stream iterator
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <iterator>
using namespace std;
bool gt15(int x) {
  return 15 < x;
}
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout,
"\n"), gt15);
} ///:~
```
Comment

In this example we've replaced the output sequence **b** in the third argument to **remove_copy_if( )** with an *output* stream iterator, which is an instance of the **ostream_iterator** class template declared in the **<iterator>** header. Output stream iterators overload their copy-assignment operators to write to their stream. This particular instance of **ostream_iterator** is attached to the output stream **cout**. Every time **remove_copy_if( )** assigns an integer from the sequence **a** to **cout** through this iterator, the iterator writes the integer to **cout** and also automatically writes an instance of the separator string found in its second argument, which in this case contains just the newline character.

It is just as easy to write to a file instead of to **cout**, of course. All you have to do is provide an output file stream instead of **cout**: Comment

```
//: C06:CopyIntsToFile.cpp
// Uses an output file stream iterator
#include <algorithm>
#include <cstddef>
#include <fstream>
#include <iterator>
using namespace std;
bool gt15(int x) {
  return 15 < x;
}
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  ofstream outf("ints.out");
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(outf,
"\n"), gt15);
} ///:~
```
Comment

An *input* stream iterator allows an algorithm to get its input sequence from an input stream. This is accomplished by having both the constructor and **operator++( )** read the next element from the underlying stream and by overloading **operator*( )** to yield the value previously read. Since algorithms require two pointers to delimit an input sequence, you can construct an **istream_iterator** in two ways, as you can see in the program that follows. Comment

```
//: C06:CopyIntsFromFile.cpp
// Uses an input stream iterator
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;
bool gt15(int x) {
  return 15 < x;
}
int main() {
  ifstream inf("someInts.dat");
  assure(inf, "someInts.dat");
  remove_copy_if(istream_iterator<int>(inf),
                 istream_iterator<int>(),
                 ostream_iterator<int>(cout,
"\n"), gt15);
} ///:~
```
Comment

The first argument to **replace_copy_if( )** in this program attaches an **istream_iterator** object to the input file stream containing **int**s. The second argument uses the default constructor of the **istream_iterator** class. This call constructs a special value of **istream_iterator** that indicates end-of-file, so that when the first iterator finally encounters the end of the physical file, it compares equal to the value **istream_iterator<int>( )**, allowing the algorithm to terminate correctly. Note that this example avoids using an explicit array altogether. Comment

Algorithm complexity

Using a software library is a matter of trust. You trust the implementers to not only provide correct functionality, but you also hope that the functions execute as efficiently as possible. It's better to write your own loops than to use algorithms that degrade performance. Comment

To guarantee quality library implementations, the C++ standard not only specifies what an algorithm should do, but how fast it should do it and sometimes how much space it should use. Any algorithm that does not meet the performance requirements does not conform to the standard. The measure of an algorithm's operational efficiency is called its *complexity*. Comment

When possible, the standard specifies the exact number of operation counts an algorithm should use. The **count_if( )** algorithm, for example, returns the number of elements in a sequence satisfying a given predicate. The following call to **count_if( )**, if applied to a sequence of integers similar to the examples earlier in this chapter, yields the number of integer elements that are greater than 15: Comment

```
size_t n = count_if(a, a + SIZE, gt15);
```

Since **count_if( )** must look at every element exactly once, it is specified to make a number of comparisons exactly equal to the number of elements in the sequence. Naturally, the **copy( )** algorithm has the same specification. Comment

Other algorithms can be specified to take *at most* a certain number of operations. The **find( )** algorithm searches through a sequence in order until it encounters an element equal to its third argument: Comment

```
int* p = find(a, a + SIZE, 20);
```

It stops as soon as the element is found and returns a pointer to that first occurrence. If it doesn't find one, it returns a pointer one position past the end of the sequence (**a+SIZE** in this example). Therefore, **find** is said to make at most a number of comparisons equal to the number of elements in the sequence. Comment

Sometimes the number of operations an algorithm takes cannot be measured with such precision. In such cases, the standard specifies the algorithm's *asymptotic complexity*, which is a measure of how the algorithm behaves with large sequences compared to well-known formulas. A good example is the **sort( )** algorithm, which the standard says takes "approximately **n log n** comparisons on average" (**n** is the number of elements in the sequence)[2]. Such complexity measures give a "feel" for the cost of an algorithm and at least give a meaningful basis for comparing algorithms. As you'll see in the next chapter, the **find( )** member function for the **set** container has logarithmic complexity, which means that the cost of searching for an element in a **set** will, for large sets, be proportional to the logarithm of the number of elements. This is much smaller than the number of elements for large **n**, so it is always better to search a **set** by using its **find( )** member function rather than by using the generic **find( )** algorithm. Comment

Function objects

As you study some of the examples earlier in this chapter, you will probably notice the limited utility of the function **gt15( )**. What if you want to use a number other than 15 as a comparison threshold? You may need a **gt20( )** or **gt25( )** or others as well. Having to write a separate function for each such comparison has two distasteful difficulties:

1. You may have to write a lot of functions!

2. You must know all required values when you write your application code.

The second limitation means that you can't use runtime values[3] to govern your searches, which is downright unacceptable. Overcoming this difficulty requires a way to pass information to predicates at runtime. For example, you would need a greater-than function that you can initialize with an arbitrary comparison value. Unfortunately, you can't pass that value as a function parameter, because unary predicates, such as our **gt15( )**, are applied to each

---

[2] This is simply an English rendition of **O(n log n)**, which is the mathematical way of saying that for large **n**, the number of comparisons grows in direct proportion to the function **f(n) = n log n**.

[3] Unless you do something ungainly like use global variables.

value in a sequence individually and must therefore take only one parameter.

The way out of this dilemma is, as always, to create an abstraction. In this case, we need an abstraction that can act like a function as well as store state, without disturbing the number of function parameters it accepts when used. This abstraction is called a *function object*[4].

A function object is an instance of a class that overloads **operator( )**, the function call operator. This operator allows an object to be used with function call syntax. As with any other object, you can initialize it via its constructors. Here is a function object that can be used in place of **gt15( )**:

```
//: C06:GreaterThanN.cpp
#include <iostream>
using namespace std;
class gt_n {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};
int main() {
  gt_n f(4);
  cout << f(3) << endl;  // Prints 0 (for false)
  cout << f(5) << endl;  // Prints 1 (for true)
} ///:~
```

The fixed value to compare against (4) is passed when the function object **f** is created. The expression **f(3)** is then evaluated by the compiler as the following function call:

```
f.operator()(3);
```

[4] Function objects are also called *functors*, after a mathematical concept with similar behavior.

which returns the value of the expression **3 > value**, which is false when **value** is 4, as it is in this example.

Since such comparisons apply to types other than **int**, it would make sense to define **gt_n( )** as a class template. It turns out you don't have to do it yourself, though—the standard library has already done it for you. The following descriptions of function objects should not only make that topic clear, but also give you a better understanding of how the generic algorithms work. Comment

Classification of function objects

The standard C++ library classifies function objects based on the number of arguments that their **operator( )** takes and the kind of value it returns. This classification is organized according to whether a function object's **operator( )** takes zero, one, or two arguments, as the following definitions illustrate. Comment

Generator: A type of function object that takes *no arguments* and returns a value of an arbitrary type. A random number generator is an example of a generator. The standard library provides one generator, the function **rand( )** declared in **<cstdlib>**, and has some algorithms, such as **generate_n( )**, which apply generators to a sequence. Comment

Unary Function: A type of function object that takes a *single argument* of any type and returns a value that may be of a different type (which may be **void**). Comment

Binary Function: A type of function object that takes *two arguments* of any two (possibly distinct) types and returns a value of any type (including **void**). Comment

Unary Predicate: A Unary Function that returns a **bool**.

Binary Predicate: A Binary Function that returns a **bool**.

Strict Weak Ordering: A binary predicate that allows for a more general interpretation of "equality." Some of the standard containers consider two elements equivalent if neither is less than the other (using **operator<( )**). This is important when comparing floating-point values, and objects of other types where **operator==( )** is unreliable or unavailable. This notion also applies if you want to sort a sequence of data records (**struct**s) on a subset of the **struct**'s fields, that comparison scheme is considered a strict weak ordering because two records with equal keys are not really "equal" as total objects, but they are equal as far as the comparison you're using is concerned. The importance of this concept will become clearer in the next chapter. Comment

In addition, certain algorithms make assumptions about the operations available for the types of objects they process. We will use the following terms to indicate these assumptions: Comment

LessThanComparable: A class that has a less-than **operator<**. Comment

Assignable: A class that has a copy-assignment **operator=** for its own type. Comment

EqualityComparable: A class that has an equivalence **operator==** for its own type. Comment

We will use these terms later in this chapter to describe the generic algorithms in the standard library.

Automatic creation of function objects

The **<functional>** header defines a number of useful generic function objects. They are admittedly simple, but you can use them to compose more complicated function objects. Consequently, in many instances, you can construct complicated predicates without writing a single function yourself! You do so by using *function object adapters* to take the simple function objects and adapt them for use with other function objects in a chain of operations. Comment

To illustrate, let's use only standard function objects to accomplish what **gt15( )** did earlier. The standard function object, **greater**, is a *binary* function object that returns **true** if its first argument is greater than its second argument. We cannot apply this directly to a sequence of integers through an algorithm such as **remove_copy_if( )**, because **remove_copy_if( )** expects a *unary* predicate. No problem. We can construct a unary predicate on the fly that uses **greater** to compare its first argument to a *fixed value*. We fix the value of the second parameter that **greater** will use to be 15 with the function object adapter **bind2nd**, like this:
Comment

```
//: C06:CopyInts4.cpp
// Uses a standard function object and adapter
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  remove_copy_if(a, a + SIZE,
                 ostream_iterator<int>(cout,
"\n"),
                 bind2nd(greater<int>(), 15));
} ///:~
```
Comment

This program accomplishes the same thing as **CopyInts3.cpp**, but without our having to write our own predicate function **gt15( )**. The function object adapter **bind2nd( )** is a template function that creates a function object of type **binder2nd**, which simply stores the two arguments passed to **bind2nd( )**, the first of which must be a binary function or function object (that is, anything that can be called with two arguments). The **operator( )** function in **binder2nd**, which is itself a unary function, calls the binary function it stored, passing it its incoming parameter and the fixed value it stored. Comment

To make the explanation concrete for this example, let's call the instance of **binder2nd** created by **bind2nd( )** by the name **b**. When **b** is created, it receives two parameters (**greater<int>()** and 15) and stores them. Let's call the instance of **greater<int>** by the name **g**. For convenience, let's also call the instance of the output stream iterator by the name **o**. Then the call to **remove_copy_if( )** earlier becomes the following: Comment

```
remove_copy_if(a, a + SIZE, o, b(g,
15).operator());
```

As **remove_copy_if( )** iterates through the sequence, it calls **b** on each element, to determine whether to ignore the element when copying to the destination. If we denote the current element by the name **e**, that call inside **remove_copy_if( )** is equivalent to Comment

```
if (b(e))
```

but **binder2nd**'s function call operator just turns around and calls **g(e,15)**, so the earlier call is the same as Comment

```
if (greater<int>(e, 15))
```

which is the comparison we were seeking. There is also a **bind1st( )** adapter that creates a **binder1st** object, which fixes the *first* argument of the associated input binary function. Comment

As another example, let's count the number of elements in the sequence not equal to 20. This time we'll use the algorithm **count_if( )**, introduced earlier. There is a standard binary function object, **equal_to**, and also a function object adapter, **not1( )**, that take a unary function object as a parameter and invert its truth value. The following program will do the job. Comment

```
//: C06:CountNotEqual.cpp
// Count elements not equal to 20
#include <algorithm>
#include <cstddef>
#include <functional>
```

```
#include <iostream>
using namespace std;
int main() {
  int a[] = {10, 20, 30};
  const size_t SIZE = sizeof a / sizeof a[0];
  cout << count_if(a, a + SIZE,
                   not1(bind1st(equal_to<int>(),
20)));// 2
} ///:~
```
Comment

As **remove_copy_if( )** did in the previous example, **count_if( )** calls the predicate in its third argument (let's call it **n**) for each element of its sequence and increments its internal counter each time **true** is returned. If, as before, we call the current element of the sequence by the name **e**, the statement Comment

```
if (n(e))
```

in the implementation of **count_if** is interpreted as

```
if (!bind1st(equal_to<int>, 20)(e))
```

which of course ends up as

```
if (!equal_to<int>(20, e))
```

because **not1( )** returns the logical negation of the result of calling its unary function argument. The first argument to **equal_to** is 20 in this case because we used **bind1st( )** instead of **bind2nd( )**. Since testing for equality is symmetric in its arguments, we could have used either **bind1st( )** or **bind2nd( )** in this example. Comment

The following table shows the templates that generate the standard function objects, along with the kinds of expressions to which they apply. Comment

Name	Type	Result produced
plus	BinaryFunction	arg1 + arg2
minus	BinaryFunction	arg1 - arg2
multiplies	BinaryFunction	arg1 * arg2
divides	BinaryFunction	arg1 / arg2
modulus	BinaryFunction	arg1 % arg2
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
unary_negate	Unary Logical	!(UnaryPredicate(arg1))
binary_negate	Binary Logical	!(BinaryPredicate(arg1, arg2))

Comment

Adaptable function objects

Standard function adapters such as **bind1st( )** and **bind2nd( )** make some assumptions about the function objects they process. To illustrate, consider the following expression from the last line of the earlier **CountNotEqual.cpp** program: Comment

```
not1(bind1st(equal_to<int>(), 20))
```

The **bind1st( )** adapter creates a unary function object of type **binder1st**, which simply stores an instance of **equal_to<int>** and the value 20. The **binder1st::operator()** function needs to know its argument type and its return type; otherwise, it will not have a valid declaration. The convention to solve this problem is to expect all function objects to provide nested type definitions for these types. For unary functions, the type names are **argument_type** and **result_type**; for binary function objects they are **first_argument_type**, **second_argument_type**, and **result_type**. Looking at the implementation of **bind1st( )** and **binder1st** in the **<functional>** header reveals these expectations. First inspect **bind1st( )**, as it might appear in a typical library implementation: Comment

```
template <class Op, class T>
binder1st<Op>
bind1st(const Op& f, const T& val)
{
  typedef typename Op::first_argument_type Arg1_t;
  return binder1st<Op>(f, Arg1_t(val));
}
```

Note that the template parameter, **Op**, which represents the type of the binary function being adapted by **bind1st( )**, must have a nested type named **first_argument_type**. (Note also the use of **typename** to inform the compiler that it is a member *type* name, as explained in Chapter 5.) Now notice how **binder1st** uses the type names in **Op** in its declaration of its function call operator: Comment

```
// Inside the implementation for binder1st<Op>…
```

```
typename Op::result_type
operator()(const typename
Op::second_argument_type& x)
  const;
```

Function objects whose classes provide these type names are called *adaptable function objects.* Comment

Since these names are expected of all standard function objects as well as of any function objects you create that you want to use with the function object adapters, the **<functional>** header provides two templates that define these types for you: **unary_function** and **binary_function**. You simply derive from these classes while filling in the argument types as template parameters. Suppose, for example, that we want to make the function object **gt_n**, defined earlier in this chapter, adaptable. All we need to do is the following: Comment

```
class gt_n : public unary_function<int, bool> {
  int value;
public:
  gt_n(int val) : value(val) {}
  bool operator()(int n) {
    return n > value;
  }
};
```

Comment

All **unary_function** does is to provide the appropriate type definitions, which it infers from its template parameters as you can see in its definition: Comment

```
template <class Arg, class Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};
```

These types become accessible through **gt_n** because it derives publicly from **unary_function**. The **binary_function** template behaves in a similar manner. Comment

More function object examples

The following **FunctionObjects.cpp** example provides simple tests for most of the built-in basic function object templates. This way, you can see how to use each template, along with their resulting behavior. This example uses one of the following generators for convenience: Comment

```
//: C06:Generators.h
// Different ways to fill sequences
#ifndef GENERATORS_H
#define GENERATORS_H
#include <set>
#include <cstdlib>
#include <cstring>
#include <ctime>
// Microsoft namespace work-around
#ifndef _MSC_VER
using std::rand;
using std::srand;
using std::time;
#endif
// 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 limit;
public:
  URandGen(int lim) : limit(lim) {
```

```
    srand(time(0));
  }
  int operator()() {
    while(true) {
      int i = int(rand()) % limit;
      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() { srand(time(0)); }
  char operator()() {
    return source[rand() % len];
  }
};

// Statics created here for convenience, but
// will cause problems if multiply included:
const char* CharGen::source = "ABCDEFGHIJK"
  "LMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
const int CharGen::len = strlen(source);
#endif // GENERATORS_H ///:~
```

We'll be using these generating functions in various examples throughout this chapter. The **SkipGen** function object returns the next number of an arithmetic sequence whose common difference is held in its **skp** data member. A **URandGen** object generates a unique random number in a specified range. (It uses a **set** container, which we'll discuss in the next chapter.) A **CharGen** object returns a random alphabetic character. Here is the sample program we promised, which uses **URandGen**. Comment

```
//: C06:FunctionObjects.cpp
```

```
//{-bor}
// Illustrates selected predefined function object
// templates from the standard C++ library
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
using namespace std;

template<class Iter>
void print(Iter b, Iter e, char* msg = "") {
  if(msg != 0 && *msg != 0)
    cout << msg << ":" << endl;
  typedef typename Iter::value_type T;
  copy(b, e, 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.begin(), r.end(), \
  "After " #EXPR);
// For Boolean tests:
#define B(EXPR) EXPR; print(br.begin(), br.end(),
\
```

```
  "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.begin(), x.end(), "x");
  print(y.begin(), y.end(), "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.begin(), b1.end(), "b1");
  print(b2.begin(), b2.end(), "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 short, we used a few handy tricks. The **print( )** template is designed to print any sequence, along with an optional message. Since **print( )** uses the **copy( )** algorithm to send objects to **cout** via an **ostream_iterator**, the **ostream_iterator** must know the type of object it is printing, which we infer from the **value_type** member of the iterator passed[5]. As you can see in **main( )**, however, the compiler can deduce the type of **T** when you hand it a **vector<T>**, so you don't have to specify that template argument explicitly; you just say **print(x)** to print the **vector<T> x**. Comment

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. The **testUnary( )** function takes a source vector, a destination vector, and a unary function object to apply to the source vector to produce the destination vector. In **testBinary( )**, two source vectors 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, which applies the unary function/function object found in its fourth parameter to each sequence element, writing the result to the sequence indicated by its third parameter, which in this case is the same as the input sequence. Comment

---

[5] All standard iterators define a number of nested types, including **value_type**, which represents the type the iterator refers to. See Chapter 7 for more detail.

For each test, you want to see a string describing the test, 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. After evaluating the expression, they pass the appropriate range to **print( )**. To produce the message the expression is "string-ized" using the preprocessor. That way you see the code of the expression that is executed followed by the result vector. Comment

The last little tool, **BRand**, 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+1)/2**. If the random numbers are evenly distributed, this should happen half the time. Comment

In **main( )**, three vectors of **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** from **Generators.h** is used. The standard **generate_n( )** algorithm populates the sequence specified in its first argument by invoking its third argument (which must be a generator) a given number of times (specified in its second argument). Since there is one operation in which elements of **x** are divided by elements of **y**, we must ensure that there are no zero values of **y**. This is accomplished by once again 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: Comment

```
bind2nd(plus<int>(), 1)
```

This expression uses the **plus** function object to add 1 to its first argument. As we did earlier in this chapter, we use a binder adapter to make this a unary function so it can applied to the sequence by a single call to **transform( )**. Comment

Another test 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. Comment

Once the two vectors is printed, **T( )** is used to test each of the function objects that produces a numeric 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. Here is the output from an execution of **FunctionObjects.cpp**:

Comment

```
x:
4 8 18 36 22 6 29 19 25 47
y:
4 14 23 9 11 32 13 15 44 30
After testBinary(x, y, r, plus<int>()):
8 22 41 45 33 38 42 34 69 77
After testBinary(x, y, r, minus<int>()):
0 -6 -5 27 11 -26 16 4 -19 17
After testBinary(x, y, r, multiplies<int>()):
16 112 414 324 242 192 377 285 1100 1410
After testBinary(x, y, r, divides<int>()):
1 0 0 4 2 0 2 1 0 1
After testBinary(x, y, r, limit<int>()):
0 8 18 0 0 6 3 4 25 17
After testUnary(x, r, negate<int>()):
-4 -8 -18 -36 -22 -6 -29 -19 -25 -47
After testBinary(x, y, br, equal_to<int>()):
1 0 0 0 0 0 0 0 0 0
After testBinary(x, y, br, not_equal_to<int>()):
0 1 1 1 1 1 1 1 1 1
After testBinary(x, y, br, greater<int>()):
0 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less<int>()):
0 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br, greater_equal<int>()):
1 0 0 1 1 0 1 1 0 1
After testBinary(x, y, br, less_equal<int>()):
1 1 1 0 0 1 0 0 1 0
After testBinary(x, y, br,
not2(greater_equal<int>())):
0 1 1 0 0 1 0 0 1 0
After testBinary(x,y,br,not2(less_equal<int>())):
0 0 0 1 1 0 1 1 0 1
```

```
b1:
0 1 1 0 0 0 1 0 1 1
b2:
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_and<int>()):
0 1 1 0 0 0 1 0 1 1
After testBinary(b1, b2, br, logical_or<int>()):
0 1 1 0 0 0 1 0 1 1
After testUnary(b1, br, logical_not<int>()):
1 0 0 1 1 1 0 1 0 0
After testUnary(b1, br, not1(logical_not<int>())):
0 1 1 0 0 0 1 0 1 1
```

A binder doesn't have to produce a unary *predicate*; it can also create any 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: Comment

```
//: C06:FBinder.cpp
// Binders aren't limited to producing predicates
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
#include "Generators.h"
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);
  transform(v.begin(), v.end(), v.begin(),
            bind2nd(multiplies<int>(), 10));
  copy(v.begin(), v.end(), out);
} ///:~
```

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. Comment

The "bound" argument to a binder cannot be a function object, but it does not have to be a compile-time constant. For example: Comment

```
//: C06:BinderValue.cpp
// The bound argument can vary
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
using namespace std;

int boundedRand() { return rand() % 100; }

int main() {
  const int SZ = 20;
  int a[SZ], b[SZ] = {0};
  generate(a, a + SZ, boundedRand);
  int val = boundedRand();
  int* end = remove_copy_if(a, a + SZ, b,

bind2nd(greater<int>(), val));
  // Sort for easier viewing:
  sort(a, a + SZ);
  sort(b, end);
  ostream_iterator<int> out(cout, " ");
  cout << "Original Sequence:\n";
  copy(a, a + SZ, out); cout << endl;
  cout << "Values less <= " << val << endl;
  copy(b, end, out); cout << endl;
} ///:~
```

Here, an array is filled with 20 random numbers between 0 and 100, and the user provides a value on the command line. In the **remove_copy_if( )** call, you can see that the bound argument to **bind2nd( )** is random number in the same range as the sequence. The output of a sample execution follows. Comment

```
Original Sequence:
```

```
4 12 15 17 19 21 26 30 47 48 56 58 60 63 71 79 82
90 92 95
Values less <= 41
4 12 15 17 19 21 26 30
```

Function pointer adapters

Wherever a function-like entity is expected by an algorithm, you can supply either a pointer to an ordinary function or a function object. When the algorithm issues a call, if it is through a function pointer, than the native function-call mechanism is used. If it through a function object, then that objects **operator( )** member executes. You saw earlier, for example, that we passed a raw function, **gt_15( )**, as a predicate to **remove_copy_if( )** in the program **CopyInts2.cpp**. We also passed pointers to functions returning random numbers to **generate( )** and **generate_n( )**. Comment

You cannot, however, use raw functions with function object adapters, such as **bind2nd( )**, because they assume the existence of type definitions for the argument and result types. Instead of manually converting your native functions into function objects yourself, the standard library provides a family of adapters to do the work for you. 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—they must only be used with unary functions or binary functions. Comment

The following program uses **ptr_fun( )** to wrap a unary function.

```
//: C06:PtrFun1.cpp
// Using ptr_fun() with a unary function
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
```

```
int d[] = {123, 94, 10, 314, 315};
const int DSZ = sizeof d / sizeof *d;
bool isEven(int x) {
  return x % 2 == 0;
}
int main() {
  vector<bool> vb;
  transform(d, d + DSZ, back_inserter(vb),
    not1(ptr_fun(isEven)));
  copy(vb.begin(), vb.end(),
    ostream_iterator<bool>(cout, " "));
  cout << endl;
  // Output: 1 0 0 0 1
} ///:~
```

We can't simply pass **isEven** to **not1**, because **not1** needs to know the actual argument type and return type its argument uses. The **ptr_fun( )** adapter deduces those types through template argument deduction. The definition of the unary version of **ptr_fun( )** looks something like this: Comment

```
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*fptr)(Arg))
{
  return pointer_to_unary_function<Arg,
Result>(fptr);
}
```

As you can see, this version of **ptr_fun( )** deduces the argument and result types from **fptr** and uses them to initialize a **pointer_to_unary_function** object that stores **fptr**. The function call operator for **pointer_to_unary_function** just calls **fptr**, as you can see by the last line of its code: Comment

```
template <class Arg, class Result>
class pointer_to_unary_function
: public unary_function<Arg, Result> {
  Result (*fptr)(Arg); // stores the f-ptr
public:
  pointer_to_unary_function(Result (*x)(Arg))
```

```
    : fptr(x) {}
  Result operator()(Arg x) const {return fptr(x);}
};
```

Since **pointer_to_unary_function** derives from **unary_function**, the appropriate type definitions come along for the ride and are available to **not1**. Comment

There is also a binary version of **ptr_fun( ),** which returns a **pointer_to_binary_function** object (which derives from **binary_function**, of course) that behaves analogously to the unary case. The following program uses the binary version of **ptr_fun( )** to raise numbers in a sequence to a power. It also reveals a "gotcha" when passing overloaded functions to **ptr_fun( )**. Comment

```
//: C06:PtrFun2.cpp
// Using ptr_fun() for a binary function
#include <algorithm>
#include <cmath>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
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<double, double, double>(pow),
2.0));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, " "));
  cout << endl;
} ///:~
```

The **pow( )** function is overloaded in the standard C++ header **<cmath>** for each of the floating-point data types, as follows:

```
float pow(float, int);  // efficient int power
versions…
double pow(double, int);
long double pow(long double, int);
float pow(float, float);
double pow(double, double);
long double pow(long double, long double);
```

Since there are multiple versions of **pow( )**, the compiler has no way of knowing which to choose. In this case, we have to help the compiler by using explicit function template specialization, as explained in the previous chapter. Comment

An even trickier problem is that of converting a member function into a function object suitable for using with the generic algorithms. As a simple example, suppose we have the classical "shape" problem and want to apply the **draw( )** member function to each pointer in a container of **Shape**: Comment

```
//: C06:MemFun1.cpp
// Applying pointers to member functions
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>
#include "../purge.h"
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( )** algorithm does just what it sounds like it: it passes each element in a sequence to the function object denoted by its third argument. In this case, we want the function object to wrap 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 a member as its argument. The **purge( )** function is just a little something we wrote that calls delete on every element of sequence. Comment

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( )** is 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 that out; all you need to remember is when to use **mem_fun( )** vs. **mem_fun_ref( )**. Comment

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, use the second overloaded form of the **transform( )** algorithm: Comment

```
//: C06:MemFun2.cpp
// Calling member functions through an object
reference
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>
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;
  // Output: 0 20 60 120 200
} ///:~
```

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 the 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 that 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( )**. Comment

Most any member function works with **mem_fun_ref( )**. You can also use standard library member functions, if your compiler doesn't add any default arguments beyond the normal arguments specified in the standard[6]. For example, suppose you'd like to read a file and search for blank lines; your compiler may allow you to use the **string::empty( )** member function like this: Comment

```
//: C06:FindBlanks.cpp
// Demonstrates mem_fun_ref() with string::empty()
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <fstream>
#include <functional>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;
```

[6] If a compiler were to define **string::empty** with default arguments (which is allowed), then the expression **&string::empty** would define a pointer to a member function taking the total number of arguments. Since there is no way for the compiler to provide the extra defaults, there would be a "missing argument" error when an algorithm applied **string::empty** via **mem_fun_ref**.

```
typedef vector<string>::iterator LSI;

int main(int argc, char* argv[]) {
  char* fname = "FindBlanks.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  vector<string> vs;
  string s;
  while(getline(in, s))
    vs.push_back(s);
  vector<string> cpy = vs; // For testing
  LSI lsi = find_if(vs.begin(), vs.end(),
     mem_fun_ref(&string::empty));
  while(lsi != vs.end()) {
    *lsi = "A BLANK LINE";
    lsi = find_if(vs.begin(), vs.end(),
      mem_fun_ref(&string::empty));
  }
  for(size_t i = 0; i < cpy.size(); i++)
    if(cpy[i].size() == 0)
      assert(vs[i] == "A BLANK LINE");
    else
      assert(vs[i] != "A BLANK LINE");
} ///:~
```

This example 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 vector, the process is repeated to find every blank line in the file. 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 to select the current string. Comment

Writing your own function object adapters

Consider how to write a program that converts strings representing floating-point numbers to their actual numeric values. To get things started, here's a generator that creates the strings: Comment

```
//: C06: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 = sizeof n / sizeof *n;
    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 strings 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. Comment

The following program 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**. Comment

```
//: C06:MemFun3.cpp
// Using mem_fun()
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
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;
} ///:~
```

This program does two transformations: one to convert strings to C-style strings (arrays of characters), and one to convert the C-style strings to numbers via **atof( )**. It would be nice to combine these two operations into one. After all, we can compose functions in mathematics, so why not C++? Comment

The obvious approach takes the two functions as arguments and applies them in the proper order:

```
//: C06:ComposeTry.cpp
// A first attempt at implementing function
composition
```

```
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <string>
using namespace std;

template<typename R, typename E, typename F1,
typename F2>
class unary_composer {
   F1 f1;
   F2 f2;
public:
   unary_composer(F1 fone, F2 ftwo) : f1(fone),
f2(ftwo) {}
   R operator()(E x) {
      return f1(f2(x));
   }
};
template<typename R, typename E, typename F1,
typename F2>
unary_composer<R, E, F1, F2> compose(F1 f1, F2 f2)
{
   return unary_composer<R, E, F1, F2>(f1, f2);
}
int main()
{
  double x =
    compose<double, const string&>(atof,
      mem_fun_ref(&string::c_str))("12.34");
  assert(x == 12.34);
} ///:~
```

The **unary_composer** object in this example stores the function pointers **atof** and **string::c_str** such that the latter function is applied first when its **operator( )** is called. The **compose( )** function adapter is a convenience, so we don't have to supply all four template arguments explicitly—**F1** and **F2** are deduced from the call. Comment

It would be much better, of course, if we didn't have to supply any template arguments at all. This is achieved by adhering to the convention for type definitions for adaptable function objects; in other words, we will assume that the functions to be composed are adaptable. This requires that we use **ptr_fun( )** for **atof( )**. For maximum flexibility, we also make **unary_composer** adaptable in case it gets passed to a function adapter. The following program does so and easily solves the original problem. Comment

```
//: C06:ComposeFinal.cpp
// An adaptable composer
#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "NumStringGen.h"
using namespace std;

template<typename F1, typename F2>
class unary_composer
  : public unary_function<typename
F2::argument_type,
                          typename
F1::result_type> {
public:
   unary_composer(F1 f1, F2 f2) : f1(f1), f2(f2)
{}
   typename F1::result_type
     operator()(typename F2::argument_type x) {
      return f1(f2(x));
   }
private:
   F1 f1;
   F2 f2;
};
template<typename F1, typename F2>
unary_composer<F1, F2> compose(F1 f1, F2 f2) {
```

```
    return unary_composer<F1, F2>(f1, f2);
}
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),
    compose(ptr_fun(atof),
mem_fun_ref(&string::c_str)));
  copy(vd.begin(), vd.end(),
    ostream_iterator<double>(cout, "\t"));
  cout << endl;
} ///:~
```

Once again we must use **typename** to let the compiler know that the member we are referring to is a nested type. Comment

Some implementations[7] support composition of function objects as an extension, and the C++ standards committee is likely to add these capabilities to the next version of standard C++. Comment

A catalog of STL algorithms

This section provides a quick reference for when you're searching for the appropriate algorithm. We leave the full exploration of all the STL algorithms to other references (see the end of this chapter, and Appendix A), along with the more intimate details of performance, and so on. Our goal here is for you to become rapidly comfortable and facile with the algorithms, and we'll assume you

---

[7] STLPort, for instance, which comes with version 6 of Borland C++ Builder and is based on SGI STL.

will look into the more specialized references if you need more depth of detail. Comment

Although you will often see the algorithms described using their full template declaration syntax, we're 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. We think you'll find this form is easier to read, and you can quickly find the full declaration in the template header file if for some reason you feel the need. Comment

The reason for all the fuss about iterators is to accommodate any type of container that meets the requirements in the standard library. So far we have illustrated the generic algorithms with only arrays and vectors as sequences, but in the next chapter you'll see a full complement of data structures that support less robust iteration. For this reason, the algorithms are categorized in part by the types of iteration facilities they require. Comment

The names of the iterator classes describe the iterator type to which they must conform. There are no interface base classes to enforce these iteration operations—they are just expected to be there. If they are not, your compiler will complain. The various flavors of iterators are described briefly as follows. Comment

InputIterator. An input iterator only allows *reading* elements of its sequence in a single, forward pass using **operator++** and **operator*.** Input iterators can also be tested with **operator==** and **operator!=**. That's all. Comment

OutputIterator. An output iterator only allows *writing* elements to a sequence in a single, forward pass using **operator++** and **operator*. 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 such as are the type of iterator returned by **back_inserter( )**). Comment

There ius no way to determine whether multiple **InputIterators** or **OutputIterators** point within the same range, so there is no way to us multiple such iterators in concert. Just think in terms of iterators to support **istream**s and **ostream**s, and **InputIterator** and **OutputIterator** will make perfect sense. Also note that algorithms that use **InputIterators** or **OutputIterators** 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. Comment

ForwardIterator. Because you can only read from an **InputIterator** and write to an **OutputIterator**, you can't use either of them to simultaneously read and modify a range, and you can't 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 compare such iterators in the same range for equality. Since forward iterators can both read and write, they can of course be used wherever an input or output iterator is called for. Comment

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--**. Comment

RandomAccessIterator. This type of iterator supports all the 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 you can compare iterators to see which is greater using **operator<**,

operator>, and so on. If you're implementing a sorting routine or something similar, random access iterators are necessary to be able to create an efficient algorithm. Comment

The names used for the template parameter types in the algorithm descriptions later in this chapter consist of the listed iterator types (sometimes with a '1' or '2' appended to distinguish different template arguments) and can also include other arguments, often function objects. Comment

When describing the group of elements that an operation is performed on, mathematical "range" notation is often used. In this, the square bracket means "includes the end point," and 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 by 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)**, in which **first** is the iterator pointing to the initial element, and **last** is the past-the-end iterator. Comment

Most books and discussions of the STL algorithms arrange them according to side-effects: *non-mutating* algorithms don't change the elements in the range, *mutating* algorithms do change the elements, and so on. These descriptions are based primarily on the underlying behavior or implementation of the algorithm—that is, on the designer's perspective. In practice, we don't find this a useful categorization, so instead we'll 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, and so on. This should help you find the algorithm you want more easily. Comment

Note that all the algorithms are in the namespace **std**. If you do not see a different header such as **<utility>** or **<numeric>** above the function declarations, it appears in **<algorithm>**.Comment

Support tools for example creation

It's useful to create some basic tools with which to test the algorithms. In the examples that follow we'll use the generators mentioned earlier in **Generators.h**, as well as what appears below. Comment

Displaying a range is something that will be done constantly, so here is a templatized function that let you print any sequence, regardless of the type in that sequence: Comment

```
//: C06:PrintSequence.h
// Prints the contents of any sequence
#ifndef PRINTSEQUENCE_H
#define PRINTSEQUENCE_H
#include <iostream>
#include <iterator>

template<typename InputIter>
void print(InputIter first, InputIter last,
  char* nm = "", char* sep = "\n",
  std::ostream& os = std::cout) {
  if(nm != 0 && *nm != '\0')
    os << nm << ": " << sep;
  while(first != last)
    os << *first++ << sep;
  os << std::endl;
}
#endif // PRINTSEQUENCE_H ///:~
```

The default prints to **cout** with newlines as separators, but you can change that. You can also provide a message to print at the head of the output. Comment

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 relative order of those elements that are equivalent as far as the comparison function is concerned. 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 use a stable sort, you will get **{ a(1), a(2), b(1), b(2), c(1), c(2) }**. The STL **sort( )** algorithm uses a variation of *quicksort* and is therefore unstable, but a **stable_sort( )** is also provided[8]. Comment

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. Comment

```
//: C06:NString.h
// A "numbered string" that indicates which
// occurrence this is of a particular word
#ifndef NSTRING_H
#define NSTRING_H
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
typedef std::pair<std::string, int> psi;

// Only compare on the first element
bool operator==(const psi& l, const psi& r) {
  return l.first == r.first;
}

class NString {
  std::string s;
  int thisOccurrence;
```

[8] The **stable_sort( )** algorithm uses *mergesort*, which is indeed stable, but tends to run slower than *quicksort* on average.

```
  // Keep track of the number of occurrences:
  typedef std::vector<psi> vp;
  typedef vp::iterator vpit;
  static vp words;
  void addString(const std::string& x) {
    psi p(x, 0);
    vpit it = std::find(words.begin(),
words.end(), p);
    if(it != words.end())
      thisOccurrence = ++it->second;
    else {
      thisOccurrence = 0;
      words.push_back(p);
    }
  }
public:
  NString() : thisOccurrence(0) {}
  NString(const std::string& x) : s(x) {
    addString(x);
  }
  NString(const char* x) : s(x) {
    addString(x);
  }
  // The implicit operator= and
  // copy-constructor are OK here
  friend std::ostream& operator<<(
    std::ostream& os, const NString& ns) {
    return os << ns.s << " ["
      << ns.thisOccurrence << "]";
  }
  // 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;
  }
  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 normally be done in a
// separate cpp file:
NString::vp NString::words;
#endif // NSTRING_H ///:~
```

We would normally use a **map** container to associate a string with its number of occurrences, but maps don't appear until the next chapter, so we use a **vector** of pairs instead. You'll see plenty of similar examples there. Comment

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 call it. Comment

As this is just a demonstration class, we are taking the liberty of placing the definition of the static members **words** and **occurrences** in this header file, but this will break down if the header file is included in more than one place, so you should normally place all **static** definitions to **cpp** files. Comment

Filling and generating

These algorithms let you automatically fill a range with a particular value or generate a set of values for a particular range. The "fill" functions insert a single value multiple times into the container, and the "generate" functions use an object called a *generator* (described earlier) to create the values to insert into the container. Comment

```
void fill(ForwardIterator first, ForwardIterator
last, const T& value);
void fill_n(OutputIterator first, Size n, const T&
value);
```
Comment

fill( ) assigns **value** to every element in the range **[first, last)**. **fill_n( )** assigns **value** to **n** elements starting at **first**.Comment

```
void generate(ForwardIterator first,
ForwardIterator last, Generator gen);
void generate_n(OutputIterator first, Size n,
Generator gen);
```
Comment

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**.Comment

Example

The following example fills and generates into **vector**s. It also shows the use of **print( )**:Comment

```
//: C06: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.begin(), v1.end(), "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.begin(), v3.end(), "v3", " ");
```

```
  vector<int> v4;
  generate_n(back_inserter(v4),15, URandGen(30));
  print(v4.begin(), v4.end(), "v4", " ");
} ///:~
```

A **vector<string>** is created with a predefined 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**. Also, the default newline separator is replaced with a space. Comment

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. Comment

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. Comment

Counting

All containers have a member function **size( )** that will tells you how many elements they hold. The following two algorithms count objects that satisfy certain criteria. Comment

---

```
IntegralValue count(InputIterator first,
InputIterator last, const EqualityComparable&
value);
```
Comment

Produces the number of elements in **[first, last)** that are equivalent to **value** (when tested using **operator==**). Comment

---

```
IntegralValue count_if(InputIterator first,
InputIterator last, Predicate pred);
```
Comment

Produces the number of elements in **[first, last)** that each cause **pred** to return **true**. Comment

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 characters, which are then displayed: Comment

```
//: C06:Counting.cpp
// The counting algorithms
#include <algorithm>
#include <functional>
#include <iterator>
#include <set>
#include <vector>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

int main() {
  vector<char> v;
  generate_n(back_inserter(v), 50, CharGen());
  print(v.begin(), v.end(), "v", "");
  // Create a set of the characters in v:
  set<char> cs(v.begin(), v.end());
  typedef set<char>::iterator sci;
  for(sci it = cs.begin(); it != cs.end(); it++) {
    int n = count(v.begin(), v.end(), *it);
    cout << *it << ": " << n << ", ";
  }
  int lc = count_if(v.begin(), v.end(),
    bind2nd(greater<char>(), 'a'));
  cout << "\nLowercase letters: " << lc << endl;
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "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. Comment

Manipulating sequences

These algorithms let you move sequences around. Comment

```
OutputIterator copy(InputIterator first,
InputIterator last, OutputIterator
destination);
```
Comment

Using assignment, copies from **[first, last)** to **destination**, incrementing **destination** after each assignment. This is essentially a “shuffle-left” operation, and so the source sequence must not contain the 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 by using **inserter( )** in the case of an associative container). Comment

```
BidirectionalIterator2
copy_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last,
BidirectionalIterator2 destinationEnd);
```
Comment

Like **copy( )**, but actually copies the elements in reverse order. This is essentially a “shuffle-right” operation, and, like **copy( )**, the source sequence must not contain the destination. 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.Comment

```
void reverse(BidirectionalIterator first,
BidirectionalIterator last);
OutputIterator reverse_copy(BidirectionalIterator
first, BidirectionalIterator last, OutputIterator
destination);
```
Comment

Both forms of this function reverse the range **[first, last)**. **reverse( )** reverses the range in place, and **reverse_copy( )**

leaves the original range alone and copies the reversed elements into **destination**, returning the past-the-end iterator of the resulting range. Comment

```
ForwardIterator2 swap_ranges(ForwardIterator1
first1, ForwardIterator1 last1, ForwardIterator2
first2);
```
Comment

Exchanges the contents of two ranges of equal size by swapping corresponding elements.Comment

```
void rotate(ForwardIterator first, ForwardIterator
middle, ForwardIterator last);
OutputIterator rotate_copy(ForwardIterator first,
ForwardIterator middle, ForwardIterator last,
OutputIterator destination);
```

Moves the contents of **[first, middle)** to the end of the sequence, and the contents of **[middle, last)** to the beginning. 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. Comment

```
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);
```
Comment

A *permutation* is one unique ordering of a set of elements. If you have **n** unique elements, there are **n!** (**n** factorial) distinct possible combinations of those elements. All these combinations can be conceptually sorted into a sequence using a lexicographical (dictionary-like) 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.Comment

The **next_permutation( )** and **prev_permutation( )** functions rearrange the elements into their next or previous permutation and if successful return **true**. If there are no more "next" permutations, the elements are in sorted order; so **next_permutation( )** returns **false**. If there are no more "previous" permutations, the elements are in descending sorted order; so **previous_permutation( )** returns **false**.Comment

The versions of the functions that have a **StrictWeakOrdering** argument perform the comparisons using **binary_pred** instead of **operator<**.Comment

---

```
void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last);
void random_shuffle(RandomAccessIterator first,
RandomAccessIterator last RandomNumberGenerator&
rand);Comment
```

This function randomly rearranges the elements in the range. It yields uniformly distributed results if the random-number generator does. The first form uses an internal random number generator, and the second uses a user-supplied random-number generator. The generator must return a value in the range [0, n) for some positive n. Comment

---

```
BidirectionalIterator
partition(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);
BidirectionalIterator
stable_partition(BidirectionalIterator first,
BidirectionalIterator last, Predicate pred);Comment
```

The “partition” functions move elements that satisfy **pred** to the beginning of the sequence. An iterator pointing one past the last of those elements is returned (which is, in effect, and “end” iterator” for the initial subsequence of elements that satisfy **pred**). This location is often called the “partition point”. Comment

With **partition( )**, the order of the elements in each resulting subsequence 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. Comment

Example

This gives a basic demonstration of sequence manipulation:Comment

```
//: C06: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.begin(), v1.end(), "v1", " ");
  vector<int> v2(v1.size());
  copy_backward(v1.begin(), v1.end(), v2.end());
  print(v2.begin(), v2.end(), "copy_backward", "
");
  reverse_copy(v1.begin(), v1.end(), v2.begin());
  print(v2.begin(), v2.end(), "reverse_copy", "
");
  reverse(v1.begin(), v1.end());
  print(v1.begin(), v1.end(), "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.begin(), v1.end(), "swap_ranges", " ");
  // Start with fresh sequence:
  generate(v1.begin(), v1.end(), SkipGen());
  print(v1.begin(), v1.end(), "v1", " ");
  int third = v1.size() / 3;
  for(int i = 0; i < 10; i++) {
    rotate(v1.begin(), v1.begin() + third,
      v1.end());
    print(v1.begin(), v1.end(), "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.begin(), ns.end(), "ns", " ");
  vector<NString>::iterator it =
    partition(ns.begin(), ns.end(),
      bind2nd(greater<NString>(), "b"));
  cout << "Partition point: " << *it << endl;
  print(ns.begin(), ns.end(), "", " ");
  // 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.begin(), ns.end(), "", " ");
} ///:~
```

The best way to see the results of this program is to run it. (You'll probably want to redirect the output to a file.)Comment

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 reverse order.Comment

reverse_copy( ), however, actually does create a reversed copy, and **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.Comment

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.Comment

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).Comment

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 can be easily used with many of the STL algorithms. Of course, an array of **char** could also have been used.Comment

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 that is automatically used.Comment

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.Comment

Searching and replacing

All these algorithms are used for searching for one or more objects within a range defined by the first two iterator arguments.Comment

```
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, **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.Comment

```
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. Comment

```
ForwardIterator adjacent_find(ForwardIterator
first,
  ForwardIterator last);
ForwardIterator adjacent_find(ForwardIterator
first,
  ForwardIterator last, BinaryPredicate
binary_pred);
```
Comment

Like **find( )**, performs a linear search through the range, but instead of looking for only one element, it searches for two adjacent elements that are equivalent. 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. An iterator to the first of the two elements is returned if a pair is found; otherwise, **last** is returned. Comment

```
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);
```
Comment

Like **find( )**, performs a linear search through the range. Both forms search for an element in the second range that's equivalent to one in the first, the first form using **operator==**, and the second using the supplied predicate. In the second form, the current element from the first range becomes the first argument to **binary_pred**, and the element from the second range becomes the second argument.Comment

```
ForwardIterator1 search(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2,
ForwardIterator2 last2);
ForwardIterator1 search(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2,
ForwardIterator2 last2 BinaryPredicate
binary_pred);
```
Comment

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==**, and the second checks to see if each pair of objects being compared causes **binary_pred** to return **true**.Comment

```
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);
```
Comment

The forms and arguments are just like **search( )** in that they look for the second range appearing as a subset of the first range, but while **search( )** looks for the first occurrence of the subset, **find_end( )** looks for the *last* occurrence and returns an iterator to its first element. Comment

```
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);
```
Comment

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.Comment

```
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator min_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate
binary_pred);Comment
```

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 **[first, r)**. 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 **[first, r)**.Comment

```
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last);
ForwardIterator max_element(ForwardIterator first,
ForwardIterator last, BinaryPredicate
binary_pred);Comment
```

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 **[first, r)**. 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 **[first, r)**.Comment

```
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);
```
Comment

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; **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).Comment

Example

To provide easy viewing of the results, this example manipulates **vector**s of **int**. Again, not every possible version of each algorithm is shown. (Some that should be obvious have been omitted.)Comment

```
//: C06:SearchReplace.cpp
// The STL search and replace algorithms
#include <algorithm>
#include <functional>
#include <vector>
#include "PrintSequence.h"
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.begin(), v.end(), "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 + 1, 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.begin(), v2.end(), "replace_copy 8 ->
47", " ");
  replace_if(v.begin(), v.end(),
    bind2nd(greater_equal<int>(), 7), -1);
  print(v.begin(), v.end(), "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.Comment

In **main( )**, an array **a** is created and fed to the constructor for **vector<int> v**. This vector is used as the target for the search and replace activities, and you'll note that there are duplicate elements—these are discovered by some of the search/replace routines.Comment

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.Comment

The **find_if( )** algorithm uses a predicate to determine if it has discovered the correct element. In this 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.Comment

Since two identical objects appear next to each other in a number of cases in **v**, 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 + 1** as the first argument (this way, it will still find two pairs in a triple).Comment

You might look at the **while** loop and think that you can do it a bit more cleverly, to wit:Comment

```
while(it != v.end()) {
  cout << "adjacent_find: " << *it++
    << ", " << *it++ << endl;
  it = adjacent_find(it, v.end());
}
```

Of course, this is exactly what we tried first. However, we did not get the output we expected, on any compiler. This is because there is no guarantee about when the increments occur in this expression. Comment

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.Comment

The **find_first_of( )** algorithm 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**.Comment

The **search( )** algorithm 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 presents some interesting possibilities—here, the **PlusOne** predicate causes the range **{ 4, 5, 6 }** to be found.Comment

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.Comment

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 we'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 says "find me 6 consecutive values that, 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.Comment

The **min_element( )** and **max_element( )** algorithms 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.Comment

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.Comment

Comparing ranges

These algorithms provide ways to compare two ranges. At first glance, the operations they perform seem close to the **search( )** function. However, **search( )** tells you where the second sequence appears within the first, and **equal( )** and **lexicographical_compare( )** simply tell you how two sequences compare. 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.Comment

```
bool equal(InputIterator first1, InputIterator
last1, InputIterator first2);
bool equal(InputIterator first1, InputIterator
last1, InputIterator first2 BinaryPredicate
binary_pred);Comment
```

In both 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.Comment

```
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);
```
Comment

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 comparison, 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, **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, range 1 is *not* less than range 2, so the function returns **false**.Comment

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 "abc" precedes "abcd". If the algorithm reaches the end of one of the ranges without a mismatch, then the shorter range comes first. In that case, if the shorter range is the first range, the result is **true**, otherwise it is **false**. Comment

In the first version of the function, **operator<** is used to perform the comparisons, and in the second version, **binary_pred** is used.Comment

```
pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1
last1, InputIterator2 first2);
pair<InputIterator1, InputIterator2>
```

```
mismatch(InputIterator1 first1, InputIterator1
last1, InputIterator2 first2, BinaryPredicate
binary_pred);
```
Comment

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 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. The **pair** template class is a **struct** with two elements denoted by the member names **first** and **second** and is defined in the **<utility>** header. Comment

As in **equal( )**, the first function tests for equality using **operator==** while the second one uses **binary_pred**.Comment

Example

Because the standard C++ **string** class is built like a container (it has **begin( )** and **end( )** member functions that 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.Comment

```
//: C06:Comparison.cpp
// The STL range comparison algorithms
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include "PrintSequence.h"
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'.Comment

To understand the output of the **lexicographical_compare( )** tests, you must remember two things: first, the comparison is performed character-by-character, and, second, on our platform, capital letters "precede" lowercase letters. In the first test, **s1** is compared to **s1**. These are exactly equivalent. 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**.Comment

To further examine lexicographical comparison, the next test in this 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 **s3** (the copy of **s1**), the characters, which are exactly equal up to that point, no longer count. Because **s3** is shorter than **s2,** it lexicographically precedes **s2**.Comment

The final test uses **mismatch( )**. 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.Comment

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, and so on, 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, **[first, new_last)** is the sequence without any of the removed elements, and **[new_last, last)** is the sequence of removed elements.Comment

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:Comment

```
c.erase(remove(c.begin(), c.end(), value),
c.end());
```

You can also use the **resize()** member function that belongs to all standard sequences (more on this in the next chapter). Comment

The return value of **remove( )** is the **new_last** iterator, so **erase( )** deletes all the removed elements from **c**.Comment

The iterators in **[new_last, last)** are dereferenceable, but the element values are unspecified and should not be used.Comment

```
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);
```
Comment

Each of the "remove" forms moves through the range **[first, last)**, finding values that match a removal criterion and copying the unremoved elements over the removed elements (thus effectively removing them). The original order of the unremoved 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.Comment

The "if" versions pass each element to **pred( )** to determine whether it should be removed. (If **pred( )** returns **true**, the element is removed.) The "copy" versions do not modify the original sequence, but instead copy the unremoved values into a range beginning at **result** and return an iterator indicating the past-the-end value of this new range.Comment

---

```
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);
```
Comment

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 unremoved elements is maintained. The return value is an iterator pointing past the end of the range that has the adjacent duplicates removed.Comment

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.Comment

For each iterator value **i** in the input range, the versions containing **binary_pred** call:Comment

```
binary_pred(*i, *(i-1));
```

and if the result is **true**, ***i** is considered a duplicate.Comment

The "copy" versions do not modify the original sequence, but instead copy the unremoved values into a range beginning at **result** and return an iterator indicating the past-the-end value of this new range.Comment

Example

This example gives a visual demonstration of the way the "remove" and "unique" functions work.Comment

```
//: C06:Removing.cpp
// The removing algorithms
#include <algorithm>
#include <cctype>
#include <string>
#include "Generators.h"
#include "PrintSequence.h"
using namespace std;

struct IsUpper {
  bool operator()(char c) {
    return isupper(c);
  }
};

int main() {
  string v;
  v.resize(25);
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v original", "");
  // Create a set of the characters in v:
```

```
  string us(v.begin(), v.end());
  sort(us.begin(), us.end());
  string::iterator it = us.begin(), cit = v.end(),
    uend = unique(us.begin(), us.end());
  // Step through and remove everything:
  while(it != uend) {
    cit = remove(v.begin(), cit, *it);
    print(v.begin(), v.end(), "Complete v", "");
    print(v.begin(), cit, "Pseudo v ", " ");
    cout << "Removed element:\t" << *it
         << "\nPsuedo Last Element:\t"
         << *cit << endl << endl;
    it++;
  }
  generate(v.begin(), v.end(), CharGen());
  print(v.begin(), v.end(), "v", "");
  cit = remove_if(v.begin(), v.end(), IsUpper());
  print(v.begin(), cit, "v after remove_if
IsUpper", " ");
  // Copying versions are not shown for remove
  // and remove_if.
  sort(v.begin(), cit);
  print(v.begin(), cit, "sorted", " ");
  string v2;
  v2.resize(cit - v.begin());
  unique_copy(v.begin(), cit, v2.begin());
  print(v2.begin(), v2.end(), "unique_copy", " ");
  // Same behavior:
  cit = unique(v.begin(), cit, equal_to<char>());
  print(v.begin(), cit, "unique equal_to<char>", "
");
} ///:~
```

The string **v**, which is a container of characters, as you know, is filled with randomly generated characters. Each character is used in a **remove** statement, but the entire string **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**).Comment

To demonstrate **remove_if( )**, the address of the standard C library function **isupper( )** (in **<cctype>** is called inside the function object class **IsUpper**, an object of which is passed as the predicate for **remove_if( )**. This returns **true** only 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 noncopying versions, which you should be able to use without an example.Comment

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.Comment

Sorting and operations on sorted ranges

A significant category of STL algorithms requires that the range they operate on be in sorted order.Comment

STL provides a number of separate sorting algorithms, depending on whether the sort should be stable, partial, or just regular (non-stable). 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. Comment

Once your sequence is sorted, you can perform many operations on that sequence, from simply locating an element or group of elements to merging with another sorted sequence or manipulating sequences as mathematical sets.Comment

Each algorithm involved with sorting or operations on sorted sequences has two versions. The first uses the object's own **operator<** to perform the comparison, and the second uses **operator( )(a, b)** to determine the relative order of **a** and **b**. Other than this, there are no differences, so this distinction will not be pointed out in the description of each algorithm.Comment

Sorting

The sort algorithms require ranges delimited by random-access iterators, such as a **vector** or **deque**. The **list** container has its own built-in **sort( )** function, since it only supports bi-directional iteration. Comment

```
void sort(RandomAccessIterator first,
RandomAccessIterator last);
void sort(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);Comment
```

Sorts **[first, last)** into ascending order. The first form uses **operator<** and the second form uses the supplied comparator object to determine the order.Comment

```
void stable_sort(RandomAccessIterator first,
RandomAccessIterator last);
void stable_sort(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);Comment
```

Sorts **[first, last)** into ascending order, preserving the original ordering of equivalent elements. (This is important if elements can be equivalent but not identical.) Comment

```
void partial_sort(RandomAccessIterator first,
RandomAccessIterator middle, RandomAccessIterator
last);
void partial_sort(RandomAccessIterator first,
RandomAccessIterator middle, RandomAccessIterator
last, StrictWeakOrdering binary_pred);Comment
```

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. Comment

```
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);
```
Comment

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)**, the smaller number of elements is used. Comment

```
void nth_element(RandomAccessIterator first,
RandomAccessIterator nth, RandomAccessIterator
last);
void nth_element(RandomAccessIterator first,
RandomAccessIterator nth, RandomAccessIterator
last, StrictWeakOrdering binary_pred);
```
Comment

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 satisfy the predicate (**operator<** by default, as usual), and all the elements in the range **(nth, last]** will not. However, neither subrange is in any particular order, unlike **partial_sort( )** which has the first range in sorted order.Comment

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( )**.Comment

Locating elements in sorted ranges

Once a range is sorted, you can use a group of operations to find elements within those ranges. In the following functions, there are always two forms. One assumes the intrinsic **operator<** has been used to perform the sort, and the second 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.Comment

```
bool binary_search(ForwardIterator first,
ForwardIterator last, const T& value);
bool binary_search(ForwardIterator first,
ForwardIterator last, const T& value,
StrictWeakOrdering binary_pred);
```
Comment

Tells you whether **value** appears in the sorted range **[first, last)**.Comment

```
ForwardIterator lower_bound(ForwardIterator first,
ForwardIterator last, const T& value);
ForwardIterator lower_bound(ForwardIterator first,
ForwardIterator last, const T& value,
StrictWeakOrdering binary_pred);
```
Comment

Returns an iterator indicating the first occurrence of **value** in the sorted range **[first, last)**. If **value** is not present, an iterator to where it would fit in the sequence is returned. Comment

```
ForwardIterator upper_bound(ForwardIterator first,
ForwardIterator last, const T& value);
ForwardIterator upper_bound(ForwardIterator first,
ForwardIterator last, const T& value,
StrictWeakOrdering binary_pred);
```
Comment

Returns an iterator indicating one past the last occurrence of **value** in the sorted range **[first, last)**. If **value** is not present, an iterator to where it would fit in the sequence is returned.Comment

```
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);
```
Comment

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 the location where **value** would fit if it is not found. Comment

Example

The following example turns each input word into an **NString** and added to a **vector<NString>**. The **vector** is then used to demonstrate the various sorting and searching algorithms. Comment

```
//: C06:SortedSearchTest.cpp
// Test searching in sorted ranges
#include <algorithm>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cstddef>
#include <fstream>
#include <iostream>
#include <iterator>
#include <vector>
#include "NString.h"
#include "PrintSequence.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  typedef vector<NString>::iterator sit;
  char* fname = "test.txt";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  srand(time(0));
  cout.setf(ios::boolalpha);
  vector<NString> original;
  copy(istream_iterator<string>(in),
```

```
    istream_iterator<string>(),
back_inserter(original));
  require(original.size() >= 4, "Must have four
elements");
  vector<NString> v(original.begin(),
original.end()),
    w(original.size() / 2);
  sort(v.begin(), v.end());
  print(v.begin(), v.end(), "sort");
  v = original;
  stable_sort(v.begin(), v.end());
  print(v.begin(), v.end(), "stable_sort");
  v = original;
  sit it = v.begin(), it2;
  // Move iterator to middle
  for(size_t i = 0; i < v.size() / 2; i++)
    it++;
  partial_sort(v.begin(), it, v.end());
  cout << "middle = " << *it << endl;
  print(v.begin(), v.end(), "partial_sort");
  v = original;
  // Move iterator to a quarter position
  it = v.begin();
  for(size_t i = 0; i < v.size() / 4; i++)
    it++;
  // Less elements to copy from than to the
destination
  partial_sort_copy(v.begin(), it, w.begin(),
w.end());
  print(w.begin(), w.end(), "partial_sort_copy");
  // Not enough room in destination
  partial_sort_copy(v.begin(), v.end(), w.begin(),
    w.end());
  print(w.begin(), w.end(), "w
partial_sort_copy");
  // v remains the same through all this process
  assert(v == original);
  nth_element(v.begin(), it, v.end());
  cout << "The nth_element = " << *it << endl;
  print(v.begin(), v.end(), "nth_element");
  string f = original[rand() % original.size()];
  cout << "binary search: "
```

```
    << binary_search(v.begin(), v.end(), f)
    << endl;
  sort(v.begin(), v.end());
  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");
} ///:~
```

This example uses the **NString** class seen earlier, which stores an occurrence number with copies of a string. The call to stable_sort( ) shows how the original order for objects with equal strings is preserved. 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."Comment

Notice in the call to **nth_element( )** 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'll see that the elements before the nth element will be less than or equal to the nth element.Comment

This example also illustrates all three binary search algorithms. As advertised, **lower_bound( )** refers to the first element in the sequence equal to a given key, **upper_bound( )** points one past the last, and **equal_range( )** returns both results as a pair. Comment

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.Comment

```
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);
```
Comment

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.Comment

```
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last);
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last, StrictWeakOrdering
binary_pred);
```
Comment

This assumes that **[first, middle)** and **[middle, last)** are each sorted ranges in the same sequence. The two ranges are merged so that the resulting range **[first, last)** contains the combined ranges in sorted order. Comment

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 our own **print** template) work with arrays as well as containers. Comment

```
//: C06: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));
  a[3] = 4;
  a[4] = 4;
  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", " ");
  // Reset b
  for(int i = 0; i < sz*2; i++)
    b[i] = 0;
  inplace_merge(a, a + sz, a + sz*2);
  print(a, a + sz*2, "inplace_merge", " ");
  int* end = set_union(a, a + sz, a + sz, a +
sz*2, b);
  print(b, end, "set_union", " ");
} ///:~
```

In **main( )**, instead of creating two separate arrays, both ranges are 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, exept that it removes duplicates from the second set. Finally, **inplace_merge( )** is used to combine both parts of **a**. Comment

Set operations on sorted ranges

Once ranges have been sorted, you can perform mathematical set operations on them.Comment

```
bool includes(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2,
InputIterator2 last2);
bool includes(InputIterator1 first1,
InputIterator1 last1, InputIterator2 first2,
```

```
InputIterator2 last2, StrictWeakOrdering
binary_pred);
```
Comment

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, **[first1, last1)** must also hold at least **n** elements if the result is to be **true**.Comment

---

```
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);
```
Comment

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, the resulting set will contain the larger number of identical values.Comment

---

```
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);
```
Comment

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, the resulting set will contain the smaller number of identical values.Comment

```
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);
```
Comment

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), the resulting set will contain **max(n-m, 0)** copies of that value.Comment

```
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);
```
Comment

Constructs, in **result**, the set containing:Comment

4. All the elements in set 1 that are not in set 2

5. 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), the resulting set will contain **abs(n-m)** copies of that value, in which **abs( )** is the absolute value. The return value is the end of the output range.Comment

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.Comment

```
//: C06:SetOperations.cpp
// Set operations on sorted ranges
#include <vector>
#include <algorithm>
#include "PrintSequence.h"
#include "Generators.h"
using namespace std;

int main() {
  const int sz = 30;
  char v[sz + 1], v2[sz + 1];
  CharGen g;
  generate(v, v + sz, g);
  generate(v2, v2 + sz, g);
  sort(v, v + sz);
  sort(v2, v2 + sz);
  print(v, v + sz, "v", "");
  print(v2, v2 + sz, "v2", "");
  bool b = includes(v, v + sz, v + sz/2, v + sz);
  cout.setf(ios::boolalpha);
  cout << "includes: " << b << endl;
  char v3[sz*2 + 1], *end;
  end = set_union(v, v + sz, v2, v2 + sz, v3);
  print(v3, end, "set_union", "");
  end = set_intersection(v, v + sz,
    v2, v2 + sz, v3);
  print(v3, end, "set_intersection", "");
  end = set_difference(v, v + sz, v2, v2 + sz,
v3);
  print(v3, end, "set_difference", "");
  end = set_symmetric_difference(v, v + sz,
    v2, v2 + sz, v3);
  print(v3, end, "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 array **v3** holds 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. Comment

Heap operations

A heap is an array-like data structure used to implement a "priority queue", which is just a range that is organized in a way that accommodates retrieving elements by priority according to some comparison function. The heap operations in the standard library allow a sequence to be treated as a "heap" data structure, which always efficiently returns the element of highest priority, without fully ordering the entire sequence. Comment

As with the "sort" operations, there are two versions of each function. The first uses the object's own **operator<** to perform the comparison; the second uses an additional **StrictWeakOrdering** object's **operator( )(a, b)** to compare two objects for **a < b**. Comment

---

```
void make_heap(RandomAccessIterator first,
RandomAccessIterator last);
void make_heap(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);Comment
```

Turns an arbitrary range into a heap. Comment

```
void push_heap(RandomAccessIterator first,
RandomAccessIterator last);
void push_heap(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);Comment
```

Adds the element *(**last-1)** to the heap determined by the range **[first, last-1)**. In other words, it places the last element in its proper location in the heap.Comment

```
void pop_heap(RandomAccessIterator first,
RandomAccessIterator last);
void pop_heap(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);
```
Comment

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.Comment

```
void sort_heap(RandomAccessIterator first,
RandomAccessIterator last);
void sort_heap(RandomAccessIterator first,
RandomAccessIterator last, StrictWeakOrdering
binary_pred);
```
Comment

This could be thought of as the complement of **make_heap( )**. 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.Comment

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, and **transform( )** places the results of each operation into a destination sequence (which can be the original sequence).Comment

```
UnaryFunction for_each(InputIterator first,
InputIterator last, UnaryFunction f);
```
Comment

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, 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**.Comment

```
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( )**, **transform( )** 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.)Comment

The first form of **transform( )** simply calls **f(*first)**, where first ranges through the input sequence. Similarly, the second form calls **f(*first1, *first2)**. (Note that 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.Comment

Examples

Since much of what you do with objects in a container is to apply an operation to all those objects, these are fairly important algorithms and merit several illustrations.Comment

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. If you look in your compiler's header file at the template defining **for_each( )**, you'll see something like this:Comment

```
template <class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last,
                  Function f) {
    while (first != last) f(*first++);
    return f;
}
```

The following example shows several 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: Comment

```
//: C06: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[9]. In addition, each **Counted** keeps a **char*** identifier to make tracking the output easier. Comment

The **CountedVector** is derived 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. Comment

```
//: C06:ForEach.cpp
// Use of STL for_each() algorithm
#include <algorithm>
#include <iostream>
#include <vector>
#include "Counted.h"
using namespace std;

// 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; }
```

[9] We're ignoring the copy constructor and assignment operator in this example, since they don't apply.

```
int main() {
  CountedVector B("two");
  for_each(B.begin(),B.end(),DeleteT<Counted>());
  CountedVector C("three");
  for_each(C.begin(), C.end(), wipe<Counted>);
} ///:~
```

You can't just use a simple function taking a **Counted*** to clean up, like the following:

```
void destroy(Counted* fp) { delete fp; }
```

The reason is that **vector** iterators are not necessarily pointers to their respective type, and so the following would fail:

```
CountedVector A("one");
for_each(A.begin(), A.end(), destroy);
```

The template argument cannot be deduced properly because the argument to **destroy( )** is not an iterator. The obvious solution is to use templates, which is shown above. Comment

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? You 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 this example, this approach would be especially useful since it's more appropriate to assign to each pointer the safe value of zero after calling **delete** for that pointer. **Transform( )** can easily do this:Comment

```
//: C06: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; }

#ifdef _MSC_VER
// Microsoft needs explicit instantiation
template Counted* deleteP(Counted* x);
#endif


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 five null pointers, which is safer since any duplicate **delete**s will have no effect.Comment

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 do the following: Comment

```
for_each(a.begin(), a.end(), ptr_fun(operator
delete));
```

This has the same problem as the call to **destroy( )** did earlier: **operator delete( )** takes a void*, but iterators aren't void pointers (or pointers at all). Even if you could make it compile, what you'd 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**. Comment

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, the return value is not very useful, but if it is a function object, that function object may have internal member data that it uses to accumulate information about all the objects that it sees during **for_each( )**. Comment

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: Comment

```
//: C06:Inventory.h
#ifndef INVENTORY_H
#define INVENTORY_H
#include <iostream>
#include <cstdlib>
#include <ctime>

#ifndef _MSC_VER
// Microsoft namespace work-around
using std::rand;
using std::srand;
using std::time;
#endif

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() { srand(time(0)); }
  Inventory operator()() {
    static char c = 'a';
    int q = rand() % 100;
    int v = rand() % 500;
    return Inventory(c++, q, v);
  }
};
#endif // INVENTORY_H ///:~
```

Member functions get the item name and get and set quantity and value. An **operator<<** prints the **Inventory** object to an **ostream**. A generator creates objects that have sequentially labeled items and random quantities and values. Comment

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:Comment

```
//: C06: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.begin(), vi.end(), "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 that you can then examine; in this case, it is simply printed. Comment

You can do most things to the **Inventory** objects using **for_each( )**. For example, **for_each( )** can handily increase all the prices by 10%. 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:Comment

```
//: C06:TransformNames.cpp
// More use of transform()
#include <algorithm>
#include <cctype>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
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.begin(), vi.end(), "vi");
  transform(vi.begin(), vi.end(), vi.begin(),
    NewImproved());
  print(vi.begin(), vi.end(), "vi");
} ///:~
```

Notice that the resulting range is the same as the input range; that is, the transformation is performed in place.Comment

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 any number of special lists need to be generated. 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( )**:Comment

```
//: C06:SpecialList.cpp
// Using the second version of transform()
#include <algorithm>
```

```
#include <cstdlib>
#include <ctime>
#include <vector>
#include "Inventory.h"
#include "PrintSequence.h"
using namespace std;

struct Discounter {
  Inventory operator()(const Inventory& inv,
    float discount) {
    return Inventory(inv.getItem(),
      inv.getQuantity(),
      int(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.begin(), vi.end(), "vi");
  vector<float> disc;
  generate_n(back_inserter(disc), 15, DiscGen());
  print(disc.begin(), disc.end(), "Discounts:");
  vector<Inventory> discounted;
  transform(vi.begin(),vi.end(), disc.begin(),
    back_inserter(discounted), Discounter());
  print(discounted.begin(), discounted.end(),
        "discounted");
} ///:~
```

Given an **Inventory** object and a discount percentage, the Discounter function object produces a new **Inventory** with the discounted price. The **DiscGen** function object just generates random discount values between 1% and 10% 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**.Comment

Numeric algorithms

These algorithms are all tucked into the header **<numeric>**, since they are primarily useful for performing numeric calculations.Comment

```
<numeric>
T accumulate(InputIterator first, InputIterator
last, T result);
T accumulate(InputIterator first, InputIterator
last, T result, BinaryFunction f);Comment
```

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**, in which **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. Comment

Note the similarity between the second form of **transform( )** and the second form of **accumulate( )**. Comment

```
<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);Comment
```

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. Thus, if you have two sequences {1, 1, 2, 2} and {1, 2, 3, 4}, the inner product becomesComment

```
(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.Comment

The second form simply applies a pair of functions to its sequence. 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 sequence, the result would be the following operations:Comment

```
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.Comment

---

```
<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**; 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},** the generated sequence is **{1, 1 + 1, 1 + 1 + 2, 1 + 1 + 2 + 2, 1 + 1 + 2 + 2 + 3}**, that is, **{1, 2, 4, 6, 9}**.Comment

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 sequence, the output is **{1, 1, 2, 4, 12}**. Note that the first output value is always the same as the first input value. Comment

The return value is the end of the output range **[result, result + (last - first))**. Comment

```
<numeric>
OutputIterator adjacent_difference(InputIterator
first, InputIterator last, OutputIterator result);
OutputIterator adjacent_difference(InputIterator
first, InputIterator last, OutputIterator result,
BinaryFunction op);
```
Comment

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 unchanged). 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}**. Comment

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 sequence, the output is **{1, 1, 2, 4, 6}**. Comment

The return value is the end of the output range **[result, result + (last - first))**. Comment

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 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.Comment

```
//: C06:NumericTest.cpp
//{L} ../TestSuite/Test
#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.Comment

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( )**.Comment

General utilities

Finally, here are some basic tools that are used with the other algorithms; you may or may not use them directly yourself.Comment

---

```
<utility>
struct pair;
make_pair( );Comment
```

This was described and used earlier in this chapter. 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**, in which **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.Comment

If you want to create a **pair** “on the fly,”, you typically use the template function **make_pair( )** rather than explicitly constructing a **pair** object.Comment

---

```
<iterator>
distance(InputIterator first, InputIterator
last);Comment
```

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.Comment

---

```
<iterator>
void advance(InputIterator& i, Distance n);
```
Comment

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.Comment

---

```
<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);
```
Comment

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 will be shown in some detail in the next chapter.Comment

---

```
const LessThanComparable& min(const
LessThanComparable& a, const LessThanComparable&
b);
const T& min(const T& a, const T& b,
BinaryPredicate binary_pred);
```
Comment

Returns the lesser of its two arguments, or returns 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.Comment

```
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.Comment

```
void swap(Assignable& a, Assignable& b);
void iter_swap(ForwardIterator1 a,
ForwardIterator2 b);
```
Comment

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.Comment

The **iter_swap( )** function swaps the values that its two arguments reference.Comment

Creating your own STL-style algorithms

Once you become comfortable with the STL algorithm style, you can begin to create your own generic algorithms. Because these will conform to the conventions of all the other algorithms in the STL, they're easy to use for programmers who are familiar with the STL, and thus they become a way to "extend the STL vocabulary." Comment

The easiest way to approach the problem is to go to the **<algorithm>** header file, find something similar to what you need, and pattern your code after that[10]. (Virtually all STL

[10] Without violating any copyright laws, of course.

implementations provide the code for the templates directly in the header files.)

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. Comment

If you take a close look at the list of algorithms in the standard C++ library, you might notice a glaring omission: there is no **copy_if( )** algorithm. Although it's true that you can accomplish the same thing with **remove_copy_if( )**, this is not quite as convenient because you have to invert the condition. (Remember, **remove_copy_if( )** only copies those elements that *don't* match its predicate, in effect *removing* those that do.) You might be tempted to write a function object adapter that negates its predicate before passing it to **remove_copy_if( )**, by including a statement something like this:

```
// Assumes pred is the incoming condition
replace_copy_if(begin, end, not1(pred));
```

This seems reasonable, but when you remember that you want to be able to use predicates that are pointers to raw functions, you see why this won't work—**not1** expects an adaptable function object. The only solution is to write a **copy_if( )** algorithm from scratch. Since you know from inspecting the other copy algorithms that conceptually you need separate iterators for input and output, the following example will do the job. Comment

```
//: C06: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 ///:~
```

Summary

The goal of this chapter was to give you a practial understanding of the 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. Comment

Although this chapter did show some examples of creating your own tools, we 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. This was in part because of space limitations, but mostly because it is beyond the charter of this book; our goal here is to give you practical understanding that will affect your day-to-day programming skills. Comment

A number of books are dedicated solely to the STL (these are listed in the appendices), but we especially recommend Matthew H. Austern's *Generic Programming and the STL* (Addison-Wesley, 1999) and Scott Meyers's *Effective STL* (Addison-Wesley, 2002). Comment

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. Using **transform( )** and **toupper( )** (in **<cctype>**), write a single function call that will convert a string to all uppercase letters.
3. Create a **Sum** function object template that will accumulate all the values in a range when used with **for_each( )**.
4. Write an anagram generator that takes a word as a command-line argument and produces all possible permutations of the letters.
5. 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 and just moves them around.)
6. 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 **vector<B*>,** and fill it with **B** and **D** objects. Use **for_each( )** to call **f( )** for each of the objects in your vector.
7. Modify **FunctionObjects.cpp** so that it uses **float** instead of **int**.
8. 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.)
9. 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.

10. 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 **vector<string>**. Force all the words to lowercase, sort them, remove all the duplicates, and print the results.
11. 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( )**.
12. 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>**, and then use **partial_sum( )** with a standard function object.
13. 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.
14. Create template function objects that perform bitwise operations for &, |, ^, and ~.
15. 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>**).
16. Write a program to compare the speed of sorting a list using **list::sort( )** vs. using **std::sort( )** (the STL algorithm version of **sort( )**).
17. Create an STL-style algorithm **transform_if( )** following the first form of **transform( )** that performs transformations only on objects that satisfy a unary predicate. Objects that don't satisfy the predicate are omitted from the result. It needs to return a new "end" iterator.
18. Create an STL-style algorithm that is an overloaded version of **for_each( )** which 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 that it applies to each object of the first range.

19. Create a **Matrix** class that is made from a **vector<vector<int> >**. Provide it with a friend **ostream& operator<<(ostream&, const Matrix&)** to display the matrix. Create the following binary operations using the STL function objects where possible: **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. (You might need to look up the mathematical meanings of the matrix operations if you don't remember them.) Demonstrate each.
20. Templatize the **Matrix** class and associated operations from the previous example so they will work with any appropriate type.
21. Using these names:

    Jon Brittle
    Jane Brittle
    Mike Brittle
    Sharon Brittle
    George Jensen
    Evelyn Jensen

    Find all the possible ways to arrange them for a wedding picture.
22. After being separated for one picture, the bride and groom decided they wanted to be together for all of them. Find all the possible ways to arrange the people for the picture if the bride and groom (Jon Brittle and Jane Brittle) are to be next to each other.
23. A travel company wants to find out the average number of days people take to travel from one end of the continent to another. The problem is that in the survey, some people did not take a direct route and took much longer than is needed (such unusual data points are

called "outliers"). Using the following generator following, generate travel days into a vector. Use **remove_if()** to remove all the outliers in your vector. Take the average of the data in the vector to find out how long people generally take to travel.

```
int travelTime() {
  // The "outlier"
  if(rand() % 10 == 0)
    return rand() % 100;
  // Regular route
  return rand() % 10 + 10;
}
```

24. Determine how much faster **binary_search()** is to **find()** when it comes to searching *sorted* ranges.
25. The army wants to recruit people from its selective service list. They have decided to recruit those that signed up for the service in 1997 starting from the oldest down to the youngest. Generate an arbitrary amount of people (give them data members such as **age** and **yearEnrolled**) into a vector. Partition the vector so that those who enrolled in 1997 are ordered at the beginning of the list, starting from the youngest to the oldest, and leave the remaining part of the list sorted according to age.
26. Make a **class** called Town with population, altitude, and weather data members. Make the weather an **enum** with **{ RAINY, SNOWY, CLOUDY, CLEAR }**. Make a class that generates Town objects. Generate town names (whether they make sense or not it doesn't matter) or pull them off the internet. Ensure that the whole town name is lower case and there are no duplicate names. For simplicity, we recommend keeping your town names to one word. For the population, altitudes, and weather fields, make a generator that will randomly generate weather conditions, populations within the range [100 to 1,000,000) and altitudes between [0, 8000) feet. Fill a

vector with your Town objects. Rewrite the vector out to a new file called Towns.txt.

27. There was a baby boom, resulting in a 10% population increase in every town. Update your town data using **transform()**, rewrite your data back out to file.
28. Find the towns with the highest and lowest population. Temporarily implement **operator<** for your town object for this exercise. Also try implementing a function that returns true if its first parameter is less than its second. Use it as a predicate to call the algorithm you use.
29. Find all the towns within the altitudes 2500-3500 feet inclusive. Implement equality operators for the **Town** class as needed.
30. We need to place an airport in a certain altitude, but location is not a problem. Organize your list of towns so that there are no duplicate (duplicate meaning that no two altitudes are within the same 100 ft range. Such classes would include [100, 199), [200, 199), etc. altitudes. Sort this list in ascending order in at least two different ways using the function objects in **<functional>**. Do the same for descending order. Implement relational operators for the **Town class** as needed.
31. Generate an arbitrary number of random numbers in a stack-based array. Use **max_element()** to find the largest number in array. Swap it with the number at the end of your array. Find the next largest number and place it in the array in the position before the previous number. Continue doing this until all elements have been moved. When the algorithm is complete, you will have a sorted array. (This is a "selection sort".)
32. Write a program that will take phone numbers from a file (that also contains names and other suitable information) and change the numbers that begin with 222 to 863. Be sure to save the old numbers. The file format is be as follows:
    222 8945
    756 3920

222 8432
etc.

33. Write a program that given a last name will find everyone with that last name with his or her corresponding phone number. Use the algorithms that deal with ranges (**lower_bound**, **upper_bound**, **equal_range**, etc.). Sort with the last name acting as a primary key and the first name acting as a secondary key. Assume that you will read the names and numbers from a file where the format will be as follows. (Be sure to order them so that the last names are ordered, and the first names are ordered within the last names.):

```
John Doe        345 9483
Nick Bonham     349 2930
Jane Doe        283 2819
```

34. Given a file with data similar to the following, pull all the state acronyms from the file and put them in a separate file. (Note that you can't depend on the line number for the type of data. The data is on random lines.)

```
ALABAMA
AL
AK
ALASKA
ARIZONA
AZ
ARKANSAS
AR
CA
CALIFORNIA
CO
COLORADO
etc.
```

When complete, you should have a file with all the state acronyms which are:

AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA ME MD MA MI MN MS MO MT NE NV NH NJ NM NY NC

ND OH OK OR PA RI SC SD TN TX UT VT VA WA WV WI WY

35. Make an **Employee** class with two data members: **hours** and **hourlyPay**. Employee shall also have a **calcSalary()** function which returns the pay for that employee. Generate random hourly pay and hours for an arbitrary amount of employees. Keep a **vector<Employee*>**. Find out how much money the company is going to spend for this pay period.
36. Race **sort(), partial_sort(),** and **nth_element()** against each other and find out if it's really worth the time saved to use one of the weaker sorts if they're all that's needed.

7: Generic containers

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.Comment

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.Comment

The C++ approach to containers is based, of course, on templates. The containers in the standard C++ library represent a full complement of data structures designed to work well with the standard algorithms and to meet common software development needs.

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. Comment

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), expands 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. Comment

Fortunately, a good object-oriented programming language comes with a set of containers. In C++, it's the Standard Template Library. 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 containers can include sets, queues, hash tables, trees, stacks, and so on. Comment

All containers have some way to put things in and get things out. The way 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 an entity is array-like, 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? Comment

The solution, of course, 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, an iterator also provides a level of abstraction, which you can use 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 lets you traverse that sequence without worrying about the underlying structure—that is, whether it's a vector, a linked list, a set, or something else. This gives you the flexibility to easily change the underlying data structure without disturbing the code in your program that traverses the container. Separating iteration from the control of the container traversed also allows having multiple iterators that traverse the same container simultaneously. Comment

From a 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 your needs, there'd be no reason to have different kinds. You need a choice of containers for two reasons. First, containers provide different types of interfaces and external behavior. A stack has an interface and a behavior that is different from that of a queue, which is different from 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. Compare a vector to a list, as an example. Both are simple sequences that can have nearly 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, it is expensive to move through a linked list to randomly select an element, and it takes longer to find an element if it is farther 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. The efficiencies of these and other operations depend on 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, code that merely traverses sequences is insulated from changes in the underlying sequence implementation. Comment

Remember that a container is only a storage cabinet in which to put objects. If that cabinet solves all your needs, it probably doesn't really matter *how* it is implemented. If you're working in a programming environment that has built-in overhead due to other factors, 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. Comment

STL reference documentation

As in the previous chapter, you will notice that this chapter does not contain exhaustive documentation describing each of the member functions in each STL container. Although we describe the member functions we use, we've left the full descriptions to others. We recommend the online resources available for the Dinkumware, Silicon Graphics, and STLPort STL implementations.[1] Comment

A first look

Here's an example using the **set** class, a container modeled after a traditional mathematical set and which does not accept duplicate values. This simple **set** was created to work with **int**s: Comment

```
//: C07:Intset.cpp
// Simple use of STL set
#include <cassert>
#include <set>
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 duplicates:
        intset.insert(j);
  assert(intset.size() == 10);
} ///:~
```

[1] Visit http://www.dinkumware.com, http://www.sgi.com/tech/stl, or http://www.stlport.org.

The **insert( )** member does all the work: it attempts to insert an element and ignores it if it's already there. Often the only activities involved in using a set are simply insertion and testing to see whether it contains the element. You can also form a union, an intersection, or a 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, but only the 10 unique instances are accepted. Comment

Now consider taking the form of **Intset.cpp** and modifying it to display a list of the words used in a document. The solution becomes remarkably simple. Comment

```
//: C07:WordSet.cpp
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

void wordSet(char* fileName) {
  ifstream source(fileName);
  assure(source, fileName);
  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;
}

int main(int argc, char* argv[]) {
  if(argc > 1)
    wordSet(argv[1]);
  else
    wordSet("WordSet.cpp");
} ///:~
```

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**. Not only does the output reveal that duplicates have been ignored, but because of the way **set** is implemented, the words are automatically sorted. Comment

A set is an example of an *associative container*, one of the three categories of containers provided by the standard C++ library. The containers and their categories are summarized in the following table.

Category	Containers
Simple Sequences	vector, list, deque
Container Adapters	queue, stack, priority_queue
Associative Containers	set, map, multiset, multimap

All the containers in the standard library hold objects and expand their resources as needed. The key difference between one container and another is the way the objects are stored in memory and what operations are available to the user. Comment

A **vector**, as you already know, 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 (just as it is with an array), and it's also expensive when doing so allocates additional storage. A **deque** ((double-ended-queue, pronounced "deck") also 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 the front as well as the back of the sequence. A **list** is a doubly linked list, so it's expensive to move around randomly but cheap to insert an element anywhere. Thus **list**, **deque** and **vector** are 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 experiment with the others only if you're tuning for efficiency. Comment

Many of the problems you set out to solve will only require a simple linear sequence such as a **vector**, **deque,** or **list**. All three have a member function **push_back( )** that you use to insert a new element at the back of the sequence (**deque** and **list** also have **push_front( )**).Comment

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**. You can use iterators on all three sequences to access elements. Each container provides the appropriate type of iterator for accessing its elements. Comment

One more observation and then we'll be ready for another example. Even though the containers hold objects by value (that is, they hold copies of whole objects), sometimes you'll want to store pointers so that you can refer to objects from a hierarchy and therefore take advantage of the polymorphic behavior of the classes represented. Consider the classic "shape" example in which 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 **Shape** types created on the heap: Comment

```
//: C07: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** objects, but where to put them? In an STL container, of course. For convenience, this **typedef**:Comment

```
typedef std::vector<Shape*> Container;
```

creates an alias for a **vector** of **Shape***, and this **typedef**:Comment

```
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, and so on), 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 percent of the time. And that's what is done in the previous example: after a container is created, it's filled with different types of **Shape** pointers. 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***. 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. Comment

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 common to write a loop like: Comment

```
for(Iter i = shapes.begin(); i != shapes.end();
i++)
```

This says "take me through every element in the sequence." Comment

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: Comment

```
(*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: Comment

```
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 that proper cleanup occurs. Comment

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: Comment

```
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 is little inheritance in the STL, which may come as a surprise), 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** or any other container that supports the same interface and see which one works fastest for your needs. Comment

Containers of strings

In the previous example, at the end of **main( )**, it was necessary to move through the whole list and **delete** all the **Shape** pointers: Comment

```
for(Iter j = shapes.begin();
      j != shapes.end(); j++)
    delete *j;
```

This highlights what could be seen as an oversight 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. Comment

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? Comment

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>**:Comment

```
//: C07:StringVector.cpp
// A vector of strings
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include "../require.h"
```

```
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "StringVector.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  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**: Comment

```
  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**. Comment

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: Comment

```
*w = ss.str() + ": " + *w;
```

You may be surprised that you can assign back into the container via the iterator , but it's a tribute to the careful design of the STL. Comment

Because the **vector<string>** contains the objects themselves, a number of interesting things take place. First, no explicit cleanup of the **string** objects on your part 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. Comment

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** because it stores *copies* of the objects you give it. Thus your coding is significantly cleaned up. Comment

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 safe to say that you *never* want to do this. Comment

Inheriting from STL containers

The power of instantly creating a sequence of elements is amazing, and it makes you realize how much time you may have 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. You might as well take the functionality in **StringVector.cpp** and package it into a class for later reuse. Comment

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 so many existing algorithms 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. Comment

```
//: C07:FileEditor.h
// File editor tool
#ifndef FILEEDITOR_H
#define FILEEDITOR_H
#include <iostream>
#include <string>
#include <vector>

class FileEditor :
  public std::vector<std::string> {
public:
  void open(const char* filename);
  FileEditor(const char* filename) {
    open(filename);
  }
  FileEditor() {};
  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. Comment

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 the reference. Comment

The implementation is quite simple: Comment

```
//: C07:FileEditor.cpp {O}
#include "FileEditor.h"
#include <fstream>
#include "../require.h"
```

```
using namespace std;

void FileEditor::open(const 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 and then discover some commonly used functionality within the program that can be turned into a class. Comment

The line-numbering program can now be rewritten using **FileEditor**: Comment

```
//: C07:FEditTest.cpp
//{L} FileEditor
// Test the FileEditor tool
#include <sstream>
#include "FileEditor.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  FileEditor file;
  if(argc > 1) {
    file.open(argv[1]);
  } else {
    file.open("FEditTest.cpp");
  }
  // 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: Comment

```
FileEditor file(argv[1]);
```

(or in the **open( )** member function), and writing happens in the single line (which defaults to sending the output to **cout**):Comment

```
file.write();
```

The bulk of the program is involved with actually modifying the file in memory.Comment

A plethora of iterators

As mentioned earlier in this and the previous chapter, an iterator is an abstraction that allows code to be generic, that is, to work with different types of containers without knowing the underlying structure of those containers. Most containers support iterators[2]. You can always say: Comment

```
<ContainerType>::iterator
<ContainerType>::const_iterator
```

to produce the types of the iterators produced by that container. Every container has a **begin( )** member function that produces an iterator indicating the beginning of the elements in the container,

[2] The container adaptors, stack, queue, and priority_queue do not support iterators, since they do not behave as sequences from the user's point of view.

and an **end( )** member function that produces an iterator which is the as the *past-the-end* marker of the container. If the container is **const**¸ **begin( )** and **end( )** produce **const** iterators, which disallow changing the elements pointed to (because the appropriate operators are **const**). Comment

All iterators can advance within their sequence (via **operator++**) and allow **==** and **!=** comparisons. Thus, to move an iterator **it** forward without running it off the end, you say something like: Comment

```
while(it != pastEnd) {
  // Do something
  it++;
}
```

in which **pastEnd** is the past-the-end marker produced by the container's **end( )** member function. Comment

An iterator can be used to produce the element that it is currently selecting within a container through the dereferencing operator (**operator***). 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, you can say either: Comment

```
(*it).f();
```

or Comment

```
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: Comment

```
//: C07:Apply.cpp
// Using simple iteration
#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)(); // 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);
} ///:~
```

You can't use **operator->** in this case, because the resulting statement would be

```
(it->*f)();
```

which attempts to use the iterator's **operator->***, which is not provided by the iterator classes[3].

It is much easier to use either **for_each( )** or **transform( )** to apply functions to sequences anyway, as you saw in the previous chapter. Comment

Iterators in reversible containers

A container may also be *reversible*, which means that it can produce iterators that move backward from the end, as well as the iterators that move forward from the beginning. All standard containers support iterators such bidirectional iteration. Comment

A reversible container has the member functions **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, rbegin( )** and **rend( )** will produce **const_reverse_iterator**s.Comment

The following example uses **vector**, but will work with all containers that support iteration:Comment

```
//: C07:Reversible.cpp
// Using reversible containers
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main() {
  ifstream in("Reversible.cpp");
  assure(in, "Reversible.cpp");
  string line;
```

[3] It will only work for implementations of vector that uses a *pointer* (a **T***) as the iterator type, like STLPort does.

```
  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.Comment

Iterator categories

The iterators are classified into "categories" that describe their capabilities. The order in which they are generally described moves from the categories with the most restricted behavior to those with the most powerful behavior.Comment

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. A special constructor defines the past-the-end value. In summary, you can dereference it for reading (once only for each value) and move it forward.Comment

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.Comment

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.Comment

Bidirectional: operator--

The bidirectional iterator has all the functionality of the forward iterator, and in addition it can be moved backward one location at a time using **operator--**.Comment

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 random-access 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 **<**, **>=**, and so on.Comment

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:Comment

6. 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).

7. You use the STL algorithms (the subject of the previous chapter). Each of the algorithms has requirements that it places on the iterators with which it works. 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 require only the most primitive iterator category (input or output), your algorithm will work with *everything* (**copy( )** is an example of this).

A hierarchy of iterator tag classes identify the category of an iterator. The class names correspond to the iterator categories, as you would expect, and their derivation reflects the relationship between them:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public
input_iterator_tag {};
struct bidirectional_iterator_tag : public
  forward_iterator_tag {};
struct random_access_iterator_tag : public
  bidirectional_iterator_tag {};
```

The class **forward_iterator_tag** derives only from **input_iterator_tag**, not from **output_iterator_tag**, because we need to have past-the-end iterators in algorithms that use forward iterators, and output iterators don't have them. For efficiency, certain algorithms provide different implementations for different iterator types, which they infer from the iterator tag defined by the iterator. We will use some of these tag classes later in this chapter when we define our own iterator types. Comment

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).Comment

The *insertion iterators* are necessary because some of the STL algorithms—**copy( ),** for example—use the assignment **operator=** 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 **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 helper 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.Comment

An **insert_iterator** lets you insert elements in the middle of the sequence, again replacing the meaning of **operator=**, but this time by automatically calling **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 in addition to the container object. The shorthand function **inserter( )** produces the same object.Comment

The following example shows the use of the different types of inserters:Comment

```
//: C07: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 insertions (even though, as you shall see later, **insert( )** is not an efficient operation for **vector**).Comment

More on stream iterators

We introduced the use of the stream iterators **ostream_iterator** (an output iterator) and **istream_iterator** (an input iterator) in conjunction with **copy( )** in the previous chapter.Remember 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 you need 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 that is the past-the-end sentinel. In the following program this object is named **end**:Comment

```
//: C07:StreamIt.cpp
// Iterators for istreams and ostreams
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include "../require.h"
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), **init** becomes equivalent to **end,** and the **copy( )** terminates.Comment

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 insert a newline along with each assignment.Comment

Although 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[4]. Although these are templates, the only template arguments they will accept are either **char** or **wchar_t**. The following example lets you compare the behavior of the stream iterators with the streambuf iterators:Comment

```
//: C07:StreambufIterator.cpp
// istreambuf_iterator & ostreambuf_iterator
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include "../require.h"
using namespace std;
```

[4] 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 the 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 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 numbers.[5]Comment

Manipulating raw storage

The **raw_storage_iterator** is defined in **<algorithm>** and is an output iterator. It is provided to enable algorithms to store their results in 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 that creates **Noisy**

[5] We are indebted to Nathan Myers for explaining this.

objects, which print trace statements for their construction, assignment, and destruction (we'll show the class definition later):Comment

```
//: C07:RawStorageIterator.cpp
// Demonstrate the raw_storage_iterator
//{-bor}
#include <iostream>
#include <iterator>
#include <algorithm>
#include "Noisy.h"
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.Comment

The basic sequences: vector, list, deque

Sequences keep objects in whatever order you store them. They differ in the efficiency of their operations, however, so if you are going to manipulate a sequence in a particular fashion, choose the appropriate container for those types of manipulations. So far in this book we've been using **vector** as the container of choice. This is quite often the case in applications. However, when you start making more sophisticated uses of containers, it becomes important to know more about their underlying implementations and behavior so that you can make the right choices.Comment

Basic sequence operations

Using a template, the following example shows the operations that all the basic sequences, **vector**, **deque,** and **list**, support. Comment

```
//: C07:BasicSequenceOperations.cpp
// The operations available for all the
// basic sequence Containers.
#include <deque>
#include <iostream>
#include <list>
#include <vector>
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( )** member functions that return iterators. Comment

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 are an assignment **operator=** and two kinds of

assign( ) member functions. One takes a quantity and an initial value, and the other takes a beginning and ending iterator.Comment

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( )**. When you call resize( ) to expand a sequence, the new elements use the default constructor of the type of element in the sequence, or if they are built-in types, they are zero-initialized. Comment

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. Comment

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** supports only 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, **operator+** and **operator-** could have been used to move the iterators in big jumps.)Comment

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( )**.Comment

The naming of the member function **swap( )** is a little confusing, since there's also a nonmember **swap( )** algorithm that switches two elements of a container. The member **swap( )** swaps everything in one container for another (if the containers hold the same type), effectively swapping the containers themselves. It does this efficiently by swapping the contents of each container, which is mostly just pointers. The nonmember **swap( )** algorithm normally uses assignment to interchange its arguments (an expensive operation for an entire container), but is customized through template specialization to call the member **swap( )** for the standard containers.Comment

The following sections discuss the particulars of each type of sequence container.Comment

vector

The **vector** class template is intentionally made to look like a souped-up array, since it has array-style indexing, but also can expand dynamically. The **vector** class template is so fundamentally useful that it was introduced in a primitive way early in this book and was used regularly in previous examples. This section will give a more in-depth look at **vector**.Comment

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 lightning-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**. In addition, when a **vector** runs out of preallocated storage, 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 approach produces a number of unpleasant side-effects.Comment

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 rapid and efficient. (If you *do* know how many objects to expect, you can preallocate storage using **reserve( ).**) But eventually you will put in one too many objects, and, unbeknownst to you, the **vector** responds by:Comment

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 expensive if you overfill your vector a lot. To see what happens when you're filling a **vector**, here is the **Noisy** class mentioned earlier that prints out information about its creations, destructions, assignments, and copy-constructions:Comment

```
//: C07: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 << "]" << std::endl;
  }
  Noisy(const Noisy& rv) : id(rv.id) {
    std::cout << "c[" << id << "]" << std::endl;
    copycons++;
  }
  Noisy& operator=(const Noisy& rv) {
    std::cout << "(" << id << ")=[" <<
      rv.id << "]" << std::endl;
    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 << "]" << std::endl;
    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 the following definitions, 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 **static** variables 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.Comment

To support certain operations such as 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 **ostream** inserter follows the standard form and simply prints the **id**.Comment

NoisyGen produces a function object (since it has an **operator( )**) that is used to automatically generate **Noisy** objects during testing.Comment

NoisyReport is a singleton object, because it is responsible for printing out the results at program termination. It has a **private** constructor, therefore, so no additional **NoisyReport** objects can be created, and it has 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**.Comment

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.Comment

Using **Noisy.h**, the following program will show the behaviors that occur when a **vector** overflows its currently allocated storage:Comment

```
//: C07:VectorOverflow.cpp
//{-bor}
// Shows the copy-construction and destruction
// That occurs when a vector must reallocate
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include "Noisy.h"
using namespace std;

int main(int argc, char* argv[]) {
  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 use the default value of 1000, or you can use your own value by putting it on the command line.Comment

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.Comment

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 preallocate, 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 using the **vector** constructor with an integral first argument; the latter initializes each element using the default copy-constructor.Comment

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 if 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** and 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).Comment

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 (that is, 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:Comment

```
//: C07:VectorCoreDump.cpp
// Invalidating an iterator
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;
```

```
int main() {
  vector<int> vi(10, 0);
  ostream_iterator<int> out(cout, " ");
  vector<int>::iterator i = vi.begin();
  *i = 47;
  copy(vi.begin(), vi.end(), out);
  cout << endl;
  // Force it to move memory (could also just add
  // enough objects):
  vi.resize(vi.capacity() + 1);
  // Now i points to wrong memory:
  *i = 48;  // Access violation
  copy(vi.begin(), vi.end(), out); // No change to
vi[0]
} ///:~
```

This illustrates the concept of *iterator invalidation*. Certain operations cause internal changes to a container's underlying data, so any iterators in effect before such changes may no longer be valid afterward. 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 this observation with the awareness of the potential expense for 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. The standard C++ library documents which container operations invalidate iterators.Comment

You may observe that using **vector** as the "basic" container in the earlier chapters of this book might 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.Comment

Inserting and erasing elements

The **vector** is most efficient if:Comment

5. You **reserve( )** the correct amount of storage at the beginning so the **vector** never has to reallocate.

6. 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 this is:Comment

```
//: C07:VectorInsertAndErase.cpp
// Erasing an element from a vector
//{-bor}
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include "Noisy.h"
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 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 11, 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 clean up the lvalue.) Last, the object on the end of the array is deleted.Comment

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.Comment

deque

The **deque** container is a basic sequence 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, a typical implementation of **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 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 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:Comment

```
//: C07:StringDeque.cpp
// Converted from StringVector.cpp
#include <cstddef>
#include <ctime>
#include <deque>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "StringDeque.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  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(fname);
```

```
  assure(in2, fname);
  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(size_t 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(size_t j = 0; j < dstrings.size(); j++) {
    ostringstream ss;
    ss << j;
    dstrings[j] = ss.str() + ": " + dstrings[j];
  }
  ticks = clock() - ticks;
  cout << "Indexing deque: " << 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 deque: " << ticks << endl;
} ///:~
```

Knowing now what you do about the inefficiency of adding things to **vector** because of storage reallocation, you might expect dramatic differences between the two. However, on a 1.7MB text file, one compiler's program produced the following (measured in platform/compiler specific clock ticks, not seconds):Comment

```
Read into vector: 8350
Read into deque: 7690
Indexing vector: 2360
Indexing deque: 2480
Iterating vector: 2470
Iterating deque: 2410
```

A different compiler and platform roughly agreed with this. It's not so dramatic, is it? This points out some important issues:Comment

1. We (programmers) are typically 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, you'll see a more "pure" comparison of performance between **vector**, **deque,** and **list**.Comment

Converting between sequences

Sometimes you need the behavior or efficiency of one kind of container for one part of your program, and you need 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.Comment

The following example reads objects into a **deque** and then converts to a **vector**:Comment

```
//: C07:DequeConversion.cpp
// Reading into a Deque, converting to a vector
//{-bor}
#include <algorithm>
#include <cstdlib>
#include <deque>
#include <iostream>
#include <vector>
#include "Noisy.h"
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 this is unneccessary because the constructor used for **v1** determines the memory need ahead of time.Comment

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**:Comment

```
//: C07:DequeOverflow.cpp
//{-bor}
// 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 <cstdlib>
#include <deque>
#include "Noisy.h"
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 have relatively few (if any) destructors called 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 of each of its data blocks. (Thus, no additional copy-constructions and destructions occur.) The **deque** 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. (The index block that holds the data blocks together may need to be reallocated, however.) Insertions in the middle of a **deque**, however, could be even messier than for **vector** (but not as costly).Comment

Because a **deque** never moves its storage, an existing iterator never becomes invalid after you add new things to either end of a deque, as it was demonstrated to do with **vector** (in **VectorCoreDump.cpp**). 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.Comment

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( )**:Comment

```
//: C07:IndexingVsAt.cpp
// Comparing "at()" to operator[]
#include <ctime>
#include <deque>
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  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:
  try {
    di.at(vi.size() + 1);
  } catch(...) {
    cerr << "Exception thrown" << endl;
  }
} ///:~
```

As you saw in Chapter 1, 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[ ]**.Comment

list

A **list** is implemented as a doubly linked list data structure and is thus designed for rapid insertion and removal of elements *anywhere* in 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 vice-versa), 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.Comment

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.Comment

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 an existing 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:Comment

```
//: C07:ListStability.cpp
// Things don't move around in lists
//{-bor}
#include "Noisy.h"
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
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 move the pointer instead of copying it. On the other hand, the **swap( )** function is a generic algorithm and doesn't know about **list** in particular, so it uses the copying approach for swapping two elements. In general you should use the member version of an algorithm if it is supplied instead of its generic algorithm equivalent. In particular, use the generic **sort( )** and **reverse( )** algorithms only with arrays, **vector**s, and **deque**s.Comment

If you have large, complex objects, you might 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.Comment

Special list operations

The **list** has some special built-in operations 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:Comment

```
//: C07:ListSpecialFunctions.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <list>
#include "Noisy.h"
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 displays results. After filling four **list**s with **Noisy** objects, one list is spliced into another in three 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)**.**Comment

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.Comment

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).Comment

A **unique( )** member function removes all duplicates, but only if the **list** has been sorted first:Comment

```
//: C07:UniqueList.cpp
// Testing list's unique() function
#include <iostream>
#include <iterator>
#include <list>
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 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.Comment

The **unique( )** function will remove only *adjacent* duplicate elements, and thus sorting is necessary before calling **unique( )**.Comment

Four additional **list** member functions 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).Comment

list **vs.** set

Looking at the previous example, you might 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:Comment

```
//: C07:ListVsSet.cpp
// Comparing list and set performance
```

```
#include <algorithm>
#include <iostream>
#include <list>
#include <set>
#include <cstdlib>
#include <ctime>
using namespace std;

class Obj {
  int a[20]; // To take up extra space
  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!Comment

Swapping basic sequences

We mentioned earlier that all basic sequences have a member function **swap( )** that's designed to switch one sequence with another (but only 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:Comment

```
//: C07:Swapping.cpp
//{-bor}
// All basic sequence containers can be swapped
#include "Noisy.h"
#include <algorithm>
#include <deque>
#include <iostream>
#include <iterator>
#include <list>
#include <vector>
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 can 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 resources of one object for another.Comment

The STL algorithms also contain a **swap( )**, and when this function is applied to two containers of the same type, it uses 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 fas—it turns out that fast sorting of a container of containers was a design goal of the STL.Comment

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 tree data structure to provide rapid lookups, thus producing sorted results when you traverse it.) The first two examples in this chapter used **set**s.Comment

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 nonalpha characters, which must be stripped off to generate proper words. One solution to this problem is to use the Standard C library function macros **isalpha( )** and **isspace( )** to extract only the characters you want. You can replace all such characters with spaces so that you can easily extract valid words from each line you read.:Comment

```
//: C07:WordList.cpp
// Display a list of words used in a document
#include <algorithm>
#include <cctype>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <sstream>
#include <string>
#include "../require.h"
using namespace std;
char replaceJunk(char c) {
   // Only keep alphas, space (as a delimiter),
and '
   return (isalpha(c) || isspace(c) || c == '\'')
? c

: ' ';
```

```
}
int main(int argc, char* argv[]) {
  char* fname = "WordList.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  set<string> wordlist;
  string line;
  while(getline(in, line)) {
    transform(line.begin(), line.end(),
line.begin(),
              replaceJunk);
    istringstream is(line);
    string word;
    while (is >> word)
      wordlist.insert(word);
    }
  // Output results:
  copy(wordlist.begin(), wordlist.end(),
       ostream_iterator<string>(cout, "\n"));
} ///:~
```

The call to **transform( )** replaces each character to be ignored with a space. The set container not only ignores duplicate words, but compares the words it keeps according to the function object **less<string>** (the default second template argument for the **set** container), which in turn uses **string::operator<( )**, so the words emerge in alphabetical order. Comment

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. Using a set is particularly handy when you just want to do lookup, since its **find( )** member function has logarithmic complexity and therefore is much faster than the generic **find( )** algorithm. Comment

The following version shows how to build the list of words with an **istreambuf_iterator** that moves the characters from one place (the input stream) to another (a **string** object), depending on whether the Standard C library function **isalpha( )** is true: Comment

```
//: C07:WordList2.cpp
// Illustrates istreambuf_iterator and insert
iterators
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "WordList2.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  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 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.Comment

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 hold the resulting words.Comment

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**.Comment

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).Comment

After the **while** loop sets everything up, it begins by looking for the first alpha character, incrementing **start** until that character is found. It then copies characters from one iterator to the other, stopping when a nonalpha character is found. Each **word**, assuming it is nonempty, is added to **wordlist**.Comment

A completely reusable tokenizer

The word list examples use different approaches to extract tokens from a stream, neither of which are very flexible. Since the STL containers and algorithms all revolve around iterators, the most flexible solution will itself use 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 certainly a type of 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 following **TokenIterator** is also a good example of how you can design your own iterators.[6]Comment

The **TokenIterator** class 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 that is a function object whose **operator( )** takes a **char** and decides whether it should be in the token. 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.Comment

The following header file contains two basic predicates, **Isalpha** and **Delimiters**, along with the template for **TokenIterator**:Comment

```
//: C07:TokenIterator.h
#ifndef TOKENITERATOR_H
#define TOKENITERATOR_H
#include <algorithm>
#include <cctype>
#include <functional>
#include <iterator>
#include <string>

struct Isalpha : std::unary_function<char, bool> {
  bool operator()(char c) {
    using namespace std;
    return isalpha(c);
  }
};

class Delimiters : std::unary_function<char, bool>
{
  std::string exclude;
public:
```

[6] This is another example coached by Nathan Myers.

```
  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,
std::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 ///:~
```

The **TokenIterator** class is inherited from the **std::iterator** template. It might appear that some kind of functionality 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 the **iterator_category** 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, which means you must design all the other functionality in yourself.Comment

The **TokenIterator** class may look a little strange at first, because the first constructor requires both a "begin" and an "end" iterator as arguments, along with the predicate. Remember, this is a "wrapper" iterator that has no idea 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.Comment

The core of the behavior happens in **operator++**. This erases the current value of **word** using **string::resize( )** and 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, input characters are copied into **word**. Finally, the **TokenIterator** object is returned and must be dereferenced to access the new token.Comment

The postfix increment requires a proxy object to hold the value before the increment, so it can be returned. 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==**.Comment

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 word list problem is solved:Comment

```
//: C07:TokenIteratorTest.cpp
//{-msc}
#include "TokenIterator.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <vector>
#include <deque>
#include <set>
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "TokenIteratorTest.cpp";
  if(argc > 1) fname = argv[1];
```

```
  ifstream in(fname);
  assure(in, fname);
  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 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.Comment

The strangest thing in the previous program is the declaration of **wordIter2**. Note the extra parentheses in the first argument to the constructor. Without these, a conforming compiler will think that **wordIter2** is a prototype for a function that has three arguments and returns a **TokenIterator<IsbIt, Delimiters>**[7]. (Microsoft's Visual C++ .NET compiler accepts it without the extra parentheses, but it shouldn't.) Comment

stack

The **stack**, along with the **queue** and **priority_queue**, are classified as *adapters*, which means they adapt one of the basic sequence containers to store their data. This 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 at times it's useful to know that you

[7] For a detailed explanation of this oddity, see Items 6 and 29 in Scott Meyer's *Effective STL*.

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, we'll 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. In most cases you won't need to concern yourself with the underlying implementation.Comment

The following example shows **stack<string>** implemented in the three ways: the default (which uses **deque**), with a **vector,** and with a **list**:Comment

```
//: C07:Stack1.cpp
// Demonstrates the STL stack
#include <fstream>
#include <iostream>
#include <list>
#include <stack>
#include <string>
#include <vector>
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() {
  ifstream in("Stack1.cpp");
  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 nonintuitive 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. More important, it is *exception safe*, as we discussed in Chapter 1. If **pop( )** both changed the state of the stack and attempted to return the top element, an exception in the element's copy-constructor could cause the element to be lost. When you're using a **stack** (or a **priority_queue**, described later), you can efficiently refer to **top( )** as many times as you want and then discard the top element explicitly using **pop( ).** (Perhaps if some term other than the familiar "pop" had been used, this would have been a bit clearer.)Comment

The **stack** template has a simple interface—essentially the member functions you saw earlier. Since it only makes sense to access a stack at its top, no iterators are available for traversing it. Nor are there sophisticated forms of initialization, but if you need that, you can use the underlying container upon which the **stack** is implemented. 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 kind of source-code reformatting.)Comment

```
//: C07:Stack2.cpp
// Converting a list to a stack
#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() {
  ifstream in("Stack2.cpp");
  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 and 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.)Comment

In **main( )**, the input file is read into a **list<Line>**, and then each line in the list is copied into a **stack** that is sent to **stackOut( )**.Comment

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( )** member functions, and then you have all the additional functionality of the **vector**. **Stack1.cpp** can be rewritten to show this:Comment

```
//: C07:Stack3.cpp
// Using a vector as a stack; modified Stack1.cpp
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
  ifstream in("Stack3.cpp");
  vector<string> textlines;
  string line;
  while(getline(in, line))
    textlines.push_back(line + "\n");
  while(!textlines.empty()) {
    cout << textlines.back();
    textlines.pop_back();
  }
} ///:~
```

This produces the same output as **Stack1.cpp**, but you can now perform **vector** operations as well. Of course, **list** can also 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.)Comment

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.Comment

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.Comment

Queues are often used when modeling systems in which 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": customers arrive at random intervals, gett into a line, and then are served by a set of tellers. Since the customers arrive randomly and each takes 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.Comment

A problem in performing this simulation is that, in effect, each customer and teller should be run by a separate process. What we'd like is a multithreaded environment so that 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.Comment

In multithreading, multiple threads of control run simultaneously in the same address space. (Multithreading differs from *multitasking*, in which different processes each run in their own address space.) The trick is that you have fewer CPUs than you do threads (and often only one CPU). To give the illusion that each thread has its own CPU, a *time-slicing* mechanism says "OK, current thread, you've had enough time. I'm going to stop you and give time to some other thread." This automatic stopping and starting of threads is called *preemptive,* and it means you don't need to manage the threading process at all.Comment

An alternative approach is for each thread to voluntarily yield the CPU to the scheduler, which then 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. 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). Each teller will have an infinite-looping **run( )** member function 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:Comment

```
//: C07:BankTeller.cpp
// Using a queue and simulated multithreading
// To model a bank teller system
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <list>
#include <queue>
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;
  enum { 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
  clock_t ticks = clock();
  // Run simulation for at least 5 seconds:
  while(clock() < ticks + 5 * CLK_TCK) {
    // 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. Comment

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.Comment

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.Comment

The driver for the simulation is the **while** loop in **main( )**, which uses processor ticks (defined in **<ctime>**) to determine if the simulation has run for at least 5 seconds. At the beginning of each pass through the loop, a random number of customers is 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. In this adaptation section of the program you can experiment with policies regarding the optimal addition and removal of tellers. If this is the only section that you're modifying, you might want to encapsulate policies inside different objects. We'll revisit this problem with a multithreaded solution in Chapter 10. Comment

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 you can 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.Comment

Like **stack** and **queue**, **priority_queue** is an adapter that is built on top of one of the basic sequences—the default is **vector**.Comment

It's trivial to make a **priority_queue** that works with **int**s:Comment

```
//: C07:PriorityQueue1.cpp
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <queue>
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:Comment

```
//: C07:PriorityQueue2.cpp
// Changing the priority
#include <cstdlib>
#include <ctime>
```

```
#include <functional>
#include <iostream>
#include <queue>
using namespace std;
int main() {
  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();
  }
} ///:~
```

A more interesting problem is a to-do list, in which each object contains a **string** and a primary and secondary priority value:Comment

```
//: C07: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();
  }
} ///:~
```

The **ToDoItem**'s **operator<** must be a nonmember function for it to work with **less< >**. Other than that, everything happens automatically. The output is:Comment

```
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
```

You cannot iterate through a **priority_queue**, but it's possible to simulate 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 that **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 that provides access to the underlying implementation:Comment

```
//: C07:PriorityQueue4.cpp
// Manipulating the underlying implementation
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
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:Comment

```
//: C07:PriorityQueue5.cpp
// Building your own priority queue
```

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
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:Comment

```
//: C07:PriorityQueue6.cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
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, the **vector** is not organized as a heap, but instead as a sorted sequence. The **assureHeap( )** function guarantees that it's put back into heap form before performing any heap operations on it. Comment

The first **for** loop in **main( )** now has the additional quality that it displays the heap as it's being built.Comment

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 member functions 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:Comment

```
//: C07:PriorityQueue7.cpp
// A priority queue that will hand you a vector
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
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();
  }
} ///:~
```

The **PQV** class template 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). It then sorts that copy (leaveing **PQV**'s **vector** untouched), and reverses the order so that traversing the new **vector** produces the same effect as popping the elements from the priority queue. Comment

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: Comment

```
//: C07:PriorityQueue8.cpp
// A more compact version of PriorityQueue7.cpp
```

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <queue>
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**.Comment

Holding bits

Because C was a language that purported to be "close to the hardware," many have 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.Comment

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[8]. The primary differences between these types are:Comment

4. 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.

5. 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 number of bits, being a template parameter, is known at compile time and allows the underlying integral array to be stored on the runtime stack. The **vector<bool>** container, on the other

[8] Chuck designed and provided the reference implementations for **bitset** and also **bitstring**, the precursor to **vector<bool>**, while an active member of the C++ standards committee in the early 1990s.

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. Furthermore, neither is a traditional "STL container." The **bitset** template class has an interface for bit-level operations and in no way resembles the STL containers we've discussed up to this point. The **vector<bool>** specialization of **vector** tries to be an STL-like container, but it falls short as discussed below. Comment

bitset<n>

The template for **bitset** accepts an unsigned integral template argument that is the number of bits to represent. Thus, **bitset<10>** is a different type than **bitset<20>**, and you cannot perform comparisons, assignments, and so on between the two.Comment

A **bitset** provides virtually any bit operation that you could ask for, in an efficient form. However, each **bitset** is implemented by logically packing bits in an array of **unsigned 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**.Comment

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.Comment

```
//: C07:BitSet.cpp
//{-bor}
// Exercising the bitset class
#include <bitset>
```

```
#include <climits>
#include <cstdlib>
#include <ctime>
#include <iostream>
#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();
  cout << a << endl;
  // Converting a string to a bitset:
  string cbits("111011010110111");
  cout << "as a string = " << 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])
    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 are combined using the **operator|=**.Comment

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, and so on. 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**.Comment

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 **stream inserter** 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( )**.Comment

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 **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).Comment

You'll notice that **bitset** only has three nonmember operators: *and* (**&**), *or* (**|**), and *exclusive-or* (**^**). Each of these create a new **bitset** as their return value. All the member operators opt for the more efficient **&=**, **|=**, and so on form in which 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, we 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.Comment

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 in the other documentation mentioned earlier in this chapter.Comment

vector<bool>

The **vector<bool>** container 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***.Comment

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.Comment

```
//: C07:VectorOfBool.cpp
// Demonstrate the vector<bool> specialization
#include <bitset>
#include <cstddef>
#include <iostream>
#include <iterator>
#include <sstream>
#include <vector>
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(size_t 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.Comment

The **vector<bool>** specialization is a "crippled" STL container in the sense that certain guarantees that other containers provide are missing. For example, with the other containers the following identities hold:

```
// Let c be an STL container other than
vector<bool>:
T& r = c.front();
T* p = &*c.begin();
```

For all other containers, the **front( )** function yields an lvalue (something you can get a non-const reference to), and **begin( )** must yield something you can dereference and then take the address of. Neither is possible because bits are not addressable.

Both **vector<bool>** and **bitset** use a proxy class (**reference**, mentioned earlier) to read and set bits as necessary.

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 with 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.Comment

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:Comment

```
//: C07:AssociativeBasics.cpp
//{-bor}
// Basic operations with sets and maps
#include <cstddef>
#include <iostream>
#include <iterator>
#include <map>
#include <set>
#include "Noisy.h"
using namespace std;

int main() {
  Noisy na[7];
  // 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(size_t 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 you can check to see if an object is already in an associative container in a couple of ways. The **count( )** member function, 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 will produce 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 will 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.)Comment

Designing an **operator[ ]** always presents a bit of a dilemma. Because it's intended to be treated as an array-indexing operation, 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 the member functions **count( )** (to see if it's there) or **find( )** (to get an iterator to it).Comment

A number of problems are associated with the **for** loop that prints out the values of the container using **operator[ ]**. 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 zero to the size of the container, and if some spots 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:Comment

```
mapped_type& operator[] (const key_type& k) {
  value_type tmp(k,T());
  return (*((insert(tmp)).first)).second;
}
```

The **map::insert( )** function takes a key-value pair and does nothing if there is already an entry in the map with the given key—otherwise it inserts an entry for the key. In either case, it returns a new key-value pair holding an iterator to the inserted pair as its first element and holding true as the second element if an insertion actually took place. The members **first** and **second** give the key and value, respectively, because **map::value_type** is really just a **typedef** for a **std::pair**:Comment

```
typedef pair<const Key, T> value_type;
```

We've seen the **std::pair** template before, which just holds two values of independent types, as you can see by its definition:Comment

```
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);
};
```

The **pair** template class is 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**.Comment

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:Comment

```
//: C07:NoisyMap.cpp
// Mapping Noisy to Noisy
//{L} ../TestSuite/Test
#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**, in which **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. Comment

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 hold only 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. Comment

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. Comment

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. Comment

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):Comment

```
//: C07: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) {
  while(n-- > 0)
    a.insert(val);
}

template<class Assoc, class Count, class Gen>
void assocGen_n(Assoc& a, Count n, Gen g) {
  while(n-- > 0)
    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).Comment

This code demonstrates two valuable lessons. The first 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.Comment

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** or **back_insert_iterator**, because 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.)Comment

```
//: C07: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++);
```

```
  }
};

namespace std {
// A generic global operator<<
// for printing any STL pair<>:
template<typename F, typename S> ostream&
operator<<(ostream& os, const pair<F,S>& p) {
  return os << p.first << "\t"
    << p.second << 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( )**.Comment

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<<**. (It is in namespace **std** for the strange name lookup reasons discussed in Chapter 5.) As you can see in the following, this allows the use of **copy( )** to output the **map**:Comment

```
//: C07: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 iterator, 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.Comment

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>**.Comment

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 that 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.Comment

In a single-item container such as a **vector** or a **list**, only one thing is 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**.Comment

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 uses 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:Comment

```
//: C07:WordCount.cpp
// Count occurrences of words using a map
#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[]) {
  char* fname = "WordCount.cpp";
```

```
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  WordMap wordmap;
  string word;
  while(in >> word)
    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:Comment

```
wordmap[word]++;
```

This increments the **Count** object associated with that word. If there isn't such a word yet in the map, a key-value pair for the word is automatically inserted, with the **Count** object initialized to zero by its default constructor. Thus, when it's incremented, the **Count** becomes 1.Comment

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.Comment

If you want to find the count for a particular word, you can use the array index operator, like this:Comment

```
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!)Comment

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>**:Comment

```
//: C07: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(const std::string defaults[][2], int
sz);
  void parse(int argc, char* argv[],
    const 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 nonflag arguments at the beginning of the command line). Finally, **print( )** displays the values. Here is the implementation:Comment

```
//: C07:ProgVals.cpp {O}
#include "ProgVals.h"
using namespace std;

ProgVals::ProgVals(
  const 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[],
  const 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 **=** (reporting an error if it isn't there) and then is broken into two strings, the **name** that appears before the **=** and the **value** that appears after. The **operator[ ]** is then used to change the existing value to the new value.Comment

Here's an example to test the tool:Comment

```
//: C07: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.**After the **usage** string, you can see that 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.Comment

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.Comment

Many command-line programs also use a style of beginning a flag with a hyphen, and sometimes they use single-character flags.Comment

The STL **map** is used in numerous places throughout the rest of this book.Comment

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. Comment

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:Comment

```
//: C07:WildLifeMonitor.cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <map>
#include <sstream>
#include <string>
#include <vector>
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() const { return x; }
  int getY() const { return y; }
  const time_t* getTime() const { 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;
  enum { 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;
  }
};

// Display a menu of animals, allow the user to
// select one, return the index value:
int menu() {
  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();
  return i;
}

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:
  for(int count = 1; count < 10; count++) {
    // Use menu to get selection:
    // int i = menu();
    // Generate randomly (for automated testing):
    int i = rand() % 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.Comment

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.Comment

The key-value pairs that make up a **Sighting** are the **string,** which names the type of animal, and the **DataPoint,** which 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.Comment

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.Comment

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 for which they want to see all the sightings . If you press **q,** the program will quit, but if you select an animal number, the **equal_range( )** member function is invoked. This returns an iterator (**DMIter**) to the beginning of the set of matching pairs and an iterator 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.Comment

Multisets

You've seen the **set**, which allows only 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," in which you can ask, "Is 'it' in this set?" If there can be more than one "it," what does that question mean?Comment

With some thought, you can see that it makes little 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 key during the comparison. That is, to the comparison operation, the objects look the same, but they actually contain some differing internal state.Comment

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 can of course substitute your own comparison function.Comment

Consider a simple class that contains one element that is used in the comparison and another that is not:Comment

```
//: C07:MultiSet1.cpp
// Demonstration of multiset behavior
#include <algorithm>
#include <ctime>
#include <iostream>
#include <iterator>
#include <set>
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:
  enum { 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, and 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.Comment

As a second example, a (possibly) more elegant version of **WordCount.cpp** can be created using **multiset**:Comment

```
//: C07:MultiSetWordCount.cpp
// Count occurrences of words using a multiset
#include <fstream>
#include <iostream>
#include <iterator>
#include <set>
#include <string>
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  char* fname = "MultiSetWordCount.cpp";
  if(argc > 1) fname = argv[1];
  ifstream in(fname);
  assure(in, fname);
  multiset<string> wordmset;
  string word;
```

```
  while(in >> word)
    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. The **equal_range( )** member function (not generic algorithm) produces the starting and ending iterators of the word that's currently selected, and the algorithm **distance( )** (defined in **<iterator>**) is used to count the number of elements in that range. The iterator **it** is then moved forward to the end of the range, which puts it at the next word. 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 such as **Count** has a lot of appeal.Comment

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. 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 criteria, you'd probably just use a **vector**, **deque,** or **list**. Comment

Combining STL containers

When using a thesaurus, you want to know all the words that are similar to a particular word. 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**:Comment

```
//: C07:Thesaurus.cpp
// A map of vectors
//{-msc}
//{-g++}
#include <map>
#include <vector>
#include <string>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <ctime>
#include <cstdlib>
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");

// Ask for a "word" to look up:
string menu(Thesaurus& thesaurus) {
  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') exit(0); // Quit
    if(thesaurus.find(reply) == thesaurus.end())
      continue; // Not in list, try again
    return reply;
  }
}

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"));
  // Create a list of the keys:
  string keys[10];
  int i = 0;
  for(TIter it = thesaurus.begin();
    it != thesaurus.end(); it++)
    keys[i++] = (*it).first;
  for(int count = 0; count < 10; count++) {
    // Enter from the console:
    // string reply = menu(thesaurus);
    // Generate randomly
    string reply = keys[rand() % 10];
    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**.Comment

In **main( )**, a **Thesaurus** is created, filled with 10 entries and printed using the **copy( )** algorithm. The **menu( )** function asks the user 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[ ],** which will automatically make a new entry if it doesn't find a match!) If so, **operator[ ]** is used to fetch out the **vector<string>** that is displayed.Comment

In the previous code, the selection of the **reply** string is generated randomly, to allow automated testing.Comment

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, and so on. For that matter, you can combine any of the STL containers this way.Comment

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:Comment

```
//: :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.Comment

Although 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.Comment

Here is **Stlshape.cpp**, modified to use the **purge( )** function:Comment

```
//: C07:Stlshape2.cpp
// Stlshape.cpp with the purge() function
#include <iostream>
#include <vector>
#include "../purge.h"
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, 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.Comment

Creating your own containers

With the STL as a foundation, you can 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.Comment

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:Comment

```
//: C07: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>{
    typename 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);
    }
    typename 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 most of the coding is in the iterator. 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.Comment

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 nontrivial exercise.) Although this can seem limiting, consider **stack**, **queue,** and **priority_queue**, which don't produce any iterators at all!Comment

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 enough time to add these components, and thus they were left out[9].Comment

Fortunately, alternatives are freely available. One of the nice things about the STL is that it establishes a basic model for creating STL-

[9] They will likely appear in the next revision of Standard C++.

like classes, so anything built using the same model is easy to understand if you are already familiar with the STL.Comment

The SGI STL from Silicon Graphics[10] 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, SGI has 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).Comment

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**:Comment

```
//: C07: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
// (It is included with the g++ distribution)
//{-bor} You can add the header by hand
//{-msc} You can add the header by hand
//{-g++} You can add the header by hand
#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;
```

[10] Available at http://www.sgi.com/tech/stl.

```
  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 we 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**, consider a **hash_map**.Comment

Non-STL containers

There are two "non-STL" containers in the standard library: **bitset** and **valarray**[11]. We say "non-STL" because neither of these containers fulfills all the requirements of STL containers. The **bitset** container, which we covered earlier in this chapter, packs bits into integers and does not allow direct addressing of its members. The **valarray** template class is a vector-like container that is optimized for efficient numeric computation. Neither container provides iterators. Although you can populate a **valarray** with nonnumeric data, it has mathematical functions that only operate with numeric data, such as **sin**, **cos**, **tan**, and so on. Most of **valarray**'s functions and operators operate on a **valarray** as a whole, as the following example illustrates.

```
//: C07:Valarray1.cpp
//{-bor}
// Illustrates basic valarray functionality
#include <iostream>
#include <valarray>
using namespace std;

double f(double x) {
    return 2.0*x - 1.0;
}

template<class T>
void print(const char* lbl, const valarray<T>& a)
{
    cout << lbl << ": ";
    for (size_t i = 0; i < a.size(); ++i)
        cout << a[i] << ' ';
    cout << endl;
}

int main() {
  double n[] = {1.0, 2.0, 3.0, 4.0};
```

[11] As we explained earlier, the **vector<bool>** specialization is also a non-STL container to some degree.

```
  valarray<double> v(n, sizeof n / sizeof n[0]);
  print("v", v);
  valarray<double> sh(v.shift(1));
  print("shift 1", sh);
  valarray<double> acc(v + sh);
  print("sum", acc);
  valarray<double> trig(sin(v) + cos(acc));
  print("trig", trig);
  valarray<double> p(pow(v, 3.0));
  print("3rd power", p);
  valarray<double> app(v.apply(f));
  print("f(v)", app);
  valarray<bool> eq(v == app);
  print("v == app?", eq);
  double x = v.min();
  double y = v.max();
  double z = v.sum();
  cout << "x = " << x << ", y = " << y
    << ", z = " << z  << endl;
} ///:~
```

The **valarray** class provides a constructor that takes an array of the target type and the count of elements in the array to be used to initialize the new **valarray**. The **shift( )** member function shifts each **valarray** element one position to the left (or to the right, if its argument is negative) and fills in holes with the default value for the type (zero in this case). There is also a **cshift( )** member function that does a circular shift (or "rotate"). All mathematical operators and functions are overloaded to operate on **valarray**s, and binary operators require **valarray** arguments of the same type and size. The **apply( )** member function, like the **transform( )** algorithm, applies a function to each element, but the result is collected into a result **valarray**. The relational operators return suitably sized instances of **valarray<bool>** that indicate the result of element-by-element comparisons, such as with **eq** above. Most operations return a new result array, but a few, such as **min( )**, **max( )**, and **sum( )**, return a single scalar value for obvious reasons.

The most interesting thing you can do with **valarray**s is reference subsets of their elements, not only for extracting information, but for updating it. A subset of a **valarray** is called a *slice*, and certain operators use slices to do their work. The following sample program uses slices.

```
//: C07:Valarray2.cpp
// From "Thinking in C++, 2nd Edition, Volume 2"
// by Bruce Eckel & Chuck Allison, (c) 2001
MindView, Inc.
// Available at www.BruceEckel.com.
// Illustrates slices and masks
//{-bor}
#include <valarray>
#include <iostream>
using namespace std;

template<class T>
void print(const char* lbl, const valarray<T>& a)
{
  cout << lbl << ": ";
  for (size_t i = 0; i < a.size(); ++i)
    cout << a[i] << ' ';
  cout << endl;
}

int main() {
  int data[] = {1,2,3,4,5,6,7,8,9,10,11,12};
  valarray<int> v(data, 12);
  valarray<int> r1(v[slice(0, 4, 3)]);
  print("slice(0,4,3)", r1);
  // Extract conditionally
  valarray<int> r2(v[v > 6]);
  print("elements > 6", r2);
  // Square first column
  v[slice(0, 4, 3)] *= valarray<int>(v[slice(0, 4,
3)]);
  print("after squaring first row", v);
  // Restore it
  int idx[] = {1,4,7,10};
  valarray<int> save(idx, 4);
  v[slice(0, 4, 3)] = save;
  print("v restored", v);
  // Extract a 2-d subset: {{1,3,5}, {7, 9, 11}}
```

```
  valarray<size_t> siz(2);
  siz[0] = 2;
  siz[1] = 3;
  valarray<size_t> gap(2);
  gap[0] = 6;
  gap[1] = 2;
  valarray<int> r3(v[gslice(0, siz, gap)]);
  print("2-d slice", r3);
  // Extract a subset via a boolean mask (bool
elements)
  valarray<bool> mask(false, 5);
  mask[1] = mask[2] = mask[4] = true;
  valarray<int> r4(v[mask]);
  print("v[mask]", r4);
  // Extract a subset via an index mask (size_t
elements)
  size_t idx2[] = {2,2,3,6};
  valarray<size_t> mask2(idx2, 4);
  valarray<int> r5(v[mask2]);
  print("v[mask2]", r5);
  // Use an index mask in assignment
  valarray<char> text("now is the time", 15);
  valarray<char> caps("NITT", 4);
  valarray<size_t> idx3(4);
  idx3[0] = 0;
  idx3[1] = 4;
  idx3[2] = 7;
  idx3[3] = 11;
  text[idx3] = caps;
  print("capitalized", text);
} ///:~
```

A **slice** object takes three arguments: the starting index, the number of elements to extract, and the "stride," which is the gap between elements of interest. Slices can be used as indexes into an existing **valarray**, and a new **valarray** containing the extracted elements is returned. A **valarray** of **bool**, such as is returned by the expression **v > 6**, can be used as an index into another **valarray**; the elements corresponding to the true slots are extracted. As you can see, you can also use slices and masks as indexes on the left side of an assignment. A **gslice** object (for "generalized slice") is like a slice, except that the counts and strides are themselves arrays, which allows you to interpret a **valarray** as a multidimensional array. The example above extracts a 2 by 3 array from **v**, where the numbers start at zero and the numbers for the first dimension are found six slots apart in **v**, and the others two apart, which effectively extracts the matrix

```
1 3 5
7 9 11
```

Here is the complete output for this program:

```
slice(0,4,3): 1 4 7 10
elements > 6: 7 8 9 10
after squaring v: 1 2 3 16 5 6 49 8 9 100 11 12
v restored: 1 2 3 4 5 6 7 8 9 10 11 12
2-d slice: 1 3 5 7 9 11
v[mask]: 2 3 5
v[mask2]: 3 3 4 7
capitalized: N o w   I s   T h e   T i m e
```

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.) Our 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.Comment

Exercises

1. Create a **set<char>**, 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. Are there any letters in the alphabet that are not used in that particular file?

2. Create three sequences of **Noisy** objects, a **vector**, **deque**, and **list**. Sort them. Now write a function template to receive the **vector** and **deque** sequences as a parameter to sort them and record the sorting time. Write a specialized template function to do the same for **list** (ensure to call its member **sort( )** instead of the generic algorithm). Compare the performance of the different sequence types.
3. Create a generator that produces random **int** values between 0 and 20 inclusive, and use it to fill a **multiset<int>**. Count the occurrences of each value, following the example given in **MultiSetWordCount.cpp**.
4. Change **StlShape.cpp** so that it uses a **deque** instead of a **vector**.
5. Modify **Reversible.cpp** so it works with **deque** and **list** instead of **vector**.
6. Use a **stack<int>** and populate it with a Fibonacci sequence. 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.
7. Using only three **stack**s (*source*, *sorted*, and *losers*), sort a random sequence of numbers by placing the numbers initially on the *source* stack. Assume the number on the top of the *source* is the largest, and push it on the *sorted* stack. Continue to pop the *source* stack comparing it with the top of the *sorted* stack. Whichever number is the smallest, pop it from its stack and push it onto the on the *losers*' stack. Once the *source* stack is empty, repeat the process using the *loser's* stack as the *source* stack, and use the *source* stack as the *losers*' stack. The algorithm completes when all the numbers have been placed into the *winners*' stack.

8. Open a text file whose name is provided on the command line. Read the file a word at a time, and use a **multiset<string>** to create a word count for each word.
9. Modify **WordCount.cpp** so that it uses **insert( )** instead of **operator[ ]** to insert elements in the map.
10. 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, and then pull the elements out to show they are in the proper order.
11. Rewrite **Ring.cpp** so it uses a **deque** instead of a **list** for its underlying implementation.
12. Modify **Ring.cpp** so that the underlying implementation can be chosen using a template argument. (Let that template argument default to **list**.)
13. Create an iterator class called **BitBucket** that just absorbs whatever you send to it without writing it anywhere.
14. 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 display 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.
15. 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**.
16. Compare the performance of **stack** based on whether it is implemented with **vector**, **deque**, or **list**.
17. Create a template that implements a singly linked list called **SList**. Provide a default constructor and **begin( )**

and **end( )** functions (via an appropriate nested iterator), **insert( )**, **erase( )** and a destructor.

18. Generate a sequence of random integers storing them into an array of **int**. Initialize a **valarray<int>** with its contents. Compute the sum, minimum value, maximum value, average, and median of the integers using **valarray** operations.
19. Create a **valarray<int>** with 12 random values. Create another **valarray<int>** with 20 random values. You will interpret the first **valarray** as a 3 x 4 matrix of **int**s and the second as a 4 x 5 matrix of **int**s, and multiply them by the rules of matrix multiplication. Store the result in a **valarray<int>** of size 15, representing the 3 x 5 result matrix. Use slices to multiply the rows of the first matrix time the columns of the second. Print the result in rectangular matrix form.

Part 3: Special Topics

Comment

8: Runtime type identification

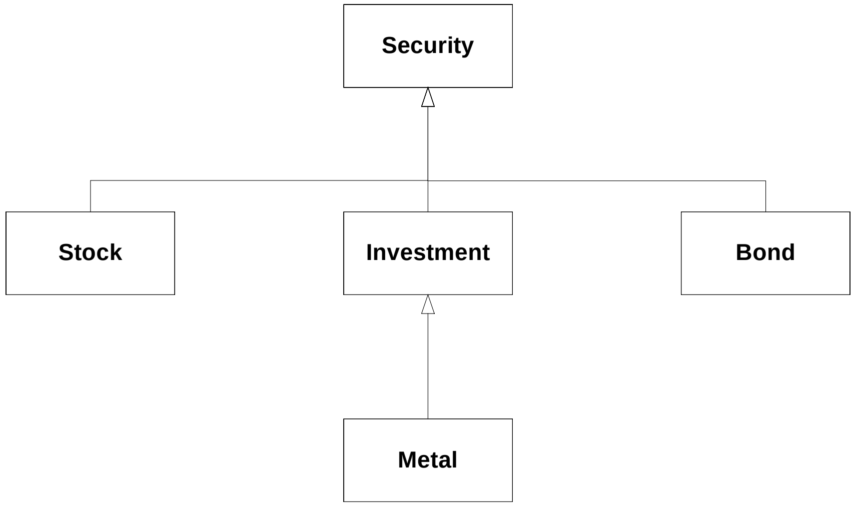
Runtime type identification (RTTI) lets you find the dynamic type of an object when you have only a pointer or a reference to the base type.

This can be thought of as a “secondary” feature in C++, a pragmatism to help out when you get into rare 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 automatically. On occasion, however, it’s useful to know the exact *runtime* (that is, most derived) 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 information about the runtime type of 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. Comment

Runtime casts

One way to determine the runtime type of an object through a pointer is to employ a *runtime cast*, which verifies that the attempted conversion is valid. This is useful when you need to cast a base-class pointer to a derived type. Since inheritance hierarchies are typically depicted with base classes above derived classes, such a cast is called a *downcast*.

Consider the following class hierarchy.



In the code that follows, the **Investment** class has an extra operation that the other classes do not, so it is important to be able to know at runtime whether a **Security** pointer refers to a **Investment** object or not. To implement checked runtime casts, each class keeps an integral identifier to distinguish it from other classes in the hierarchy. Comment

```
//: C08:CheckedCast.cpp
// Checks casts at runtime
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Security {
protected:
  enum {baseID = 0};
public:
  virtual bool isA(int id) {
      return (id == baseID);
  }
};

class Stock : public Security {
```

```
  typedef Security Super;
protected:
  enum {OFFSET = 1, typeID = baseID + OFFSET};
public:
  bool isA(int id) {
    return id == typeID || Super::isA(id);
  }
  static Stock* dynacast(Security* s) {
    return (s->isA(typeID)) ?
      static_cast<Stock*>(s) : 0;
  }
};

class Bond : public Security {
  typedef Security Super;
protected:
  enum {OFFSET = 2, typeID = baseID + OFFSET};
public:
  bool isA(int id) {
    return id == typeID || Super::isA(id);
  }
  static Bond* dynacast(Security* s) {
    return (s->isA(typeID)) ?
      static_cast<Bond*>(s) : 0;
  }
};

class Investment : public Security {
  typedef Security Super;
protected:
  enum {OFFSET = 3, typeID = baseID + OFFSET};
public:
  bool isA(int id) {
    return id == typeID || Super::isA(id);
  }
  static Investment* dynacast(Security* s) {
    return (s->isA(typeID)) ?
      static_cast<Investment*>(s) : 0;
  }
  void special() {
    cout << "special Investment function\n";
  }
```

```
};

class Metal : public Investment {
  typedef Investment Super;
protected:
  enum {OFFSET = 4, typeID = baseID + OFFSET};
public:
  bool isA(int id) {
    return id == typeID || Super::isA(id);
  }
  static Metal* dynacast(Security* s) {
    return (s->isA(typeID)) ?
      static_cast<Metal*>(s) : 0;
  }
};

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  for (vector<Security*>::iterator it =
         portfolio.begin();
       it != portfolio.end(); ++it) {
    Investment* cm = Investment::dynacast(*it);
    if(cm) cm->special();
    else cout << "not a Investment" << endl;
    it++;
  }
  cout << "cast from intermediate pointer:\n";
  Security* sp = new Metal;
  Investment* cp = Investment::dynacast(sp);
  if(cp) cout << "  it's a Investment\n";
  Metal* mp = Metal::dynacast(sp);
  if(mp) cout << "  it's a Metal too!\n";
  purge(portfolio);
} ///:~
```

The polymorphic **isA( )** function checks to see if its argument is compatible with its type argument (**id**), which means that either **id** matches the object's **typeID** exactly or that of one of its ancestors in the hierarchy (hence the call to **Super::isA( )** in that case). The **dynacast( )** function, which is static in each class, calls **isA( )** for its pointer argument to check if the cast is valid. If **isA( )** returns true, the cast is valid, and a suitably cast pointer is returned. Otherwise, the null pointer is returned, which tells the caller that the cast is not valid, meaning that the original pointer is not pointing to an object compatible with (convertible to) the desired type. All this machinery is necessary to be able to check intermediate casts, such as from a **Security** pointer that refers to a **Metal** object to a **Investment** pointer in the previous example program.[1] Comment

Although for most programs downcasting is not needed (and indeed is discouraged, since everyday polymorphism solves most problems in object-oriented application programs), the ability to check a cast to a more derived type is important for utility programs such as debuggers, class browsers, and databases. C++ provides such a checked cast with the **dynamic_cast** operator. The following program is a rewrite of the previous example using **dynamic_cast**. Comment

```
//: C08:CheckedCast2.cpp
// Uses RTTI's dynamic_cast
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;

class Security {
public:
  virtual ~Security(){}
};
class Stock : public Security {};
class Bond : public Security {};
class Investment : public Security {
public:
```

[1] With Microsoft's compilers you will have to enable RTTI; it's disabled by default. The command-line option enable it is **/GR**.

```
  void special() {
    cout << "special Investment function\n";
  }
};
class Metal : public Investment {};

int main() {
  vector<Security*> portfolio;
  portfolio.push_back(new Metal);
  portfolio.push_back(new Investment);
  portfolio.push_back(new Bond);
  portfolio.push_back(new Stock);
  vector<Security*>::iterator it =
    portfolio.begin();
  while(it != portfolio.end()) {
    Investment* cm =
dynamic_cast<Investment*>(*it);
    if(cm) cm->special();
    else cout << "not a Investment" << endl;
    it++;
  }
  cout << "cast from intermediate pointer:\n";
  Security* sp = new Metal;
  Investment* cp = dynamic_cast<Investment*>(sp);
  if(cp) cout << "  it's a Commodity\n";
  Metal* mp = dynamic_cast<Metal*>(sp);
  if(mp) cout << "  it's a Metal too!\n";
  purge(portfolio);
} ///:~
```

This example is much shorter, since most of the code in the original example was just the overhead for checking the casts. The target type of a **dynamic_cast** is placed in angle brackets, like the other new-style C++ casts (**static_cast**, and so on), and the object to cast appears as the operand. There is one important change to the **Security** class to take note of: it has a virtual destructor. We had to do this because **dynamic_cast** requires that the types you use it with be *polymorphic* if you want safe downcasts[2]. This in turn requires that the class must have at least one virtual function. We chose to use a destructor in this case so we didn't have to invent some other extraneous function to get the job done. Comment

You can also use **dynamic_cast** with references instead of pointers, but since there is no such thing as a null reference, you need another way to know if the cast fails. That "other way" is to catch a **bad_cast** exception, as follows:

```
Metal m;
Security& s = m;
try {
  Investment& c = dynamic_cast<Investment&>(s);
  cout << "  it's a Investment\n";
}
catch (bad_cast&) {
  cout << "s is not a Investment type\n";
}
```

The **bad_cast** class is defined in the **<typeinfo>** header, and, like most of the standard library, is declared in the **std** namespace. Comment

The typeid operator

The other way to get runtime information for an object is through the **typeid** operator. This operator returns an object of class **type_info**, which yields information about the type of object to which it was applied. If the type is polymorphic, it gives information about the most derived type that applies (the *dynamic type*); otherwise it yields static type information. One use of the **typeid** operator is to get the name of the dynamic type of an object as a **const char***, as you can see in the following example. Comment

[2] Compilers typically insert a pointer to a class's RTTI table inside of its virtual function table.

```
//: C08:TypeInfo.cpp
// Illustrates the typeid operator
#include <typeinfo>
#include <iostream>
using namespace std;

struct PolyBase {virtual ~PolyBase(){}};
struct PolyDer : PolyBase {};
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase
{NonPolyDer(int){}};
int main() {
  // Test polymorphic Types
  const PolyDer pd;
  const PolyBase* ppb = &pd;
  cout << typeid(ppb).name() << endl;
  cout << typeid(*ppb).name() << endl;
  cout << boolalpha << (typeid(*ppb) ==
typeid(pd))
    << endl;
  cout << (typeid(PolyDer) == typeid(const
PolyDer))
    << endl;
  // Test non-polymorphic Types
  const NonPolyDer npd(1);
  const NonPolyBase* nppb = &npd;
  cout << typeid(nppb).name() << endl;
  cout << typeid(*nppb).name() << endl;
  cout << (typeid(*nppb) == typeid(npd))
    << endl;
  // Test a built-in type
  int i;
  cout << typeid(i).name() << endl;
} ///:~
```

The output from this program is

```
struct PolyBase const *
struct PolyDer
1
1
struct NonPolyBase const *
```

```
struct NonPolyBase
0
int
```

The first output line just echoes the static type of **ppb** because it is a pointer. To get RTTI to kick in, you need to look at the object a pointer or reference is connected to, which is illustrated in the second line. Notice that RTTI ignores top-level **const** and **volatile** qualifiers. With non-polymorphic types, you just get the static type (the type of the pointer itself). As you can see, built-in types are also supported. Comment

It turns out that you can't store the result of a **typeid** operation in a **type_info** object, because there are no accessible constructors and assignment is disallowed; you must use it as we have shown. In addition, the actual string returned by **type_info::name( )** is compiler dependent. Some compilers return "class C" instead of just "C", for instance, for a class named **C**. Applying **typeid** to an expression that dereferences a null pointer will cause a **bad_typeid** exception (also defined in **<typeinfo>**) to be thrown. Comment

The following example shows that the class name that **type_info::name( )** returns is fully qualified. Comment

```
//: C08: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;
} ///:~
```

Since **Nested** is a member type of the **One** class, the result is **One::Nested**. Comment

You can also ask a **type_info** object if it precedes another **type_info** object in the implementation-defined "collation sequence," using **before(type_info&)**, which returns true or false. When you say,Comment

```
if(typeid(me).before(typeid(you))) // ...
```

you're asking if **me** occurs before **you** in the collation sequence. This is useful should you store pointers or references to **type_info** objects as keys. Comment

Casting to intermediate levels

As you saw in the earlier program that used the hierarchy of Security classes, **dynamic_cast** can detect both exact types and, in an inheritance hierarchy with multiple levels, intermediate types. Here is another example.Comment

```
//: C08:IntermediateCast.cpp
#include <cassert>
#include <typeinfo>
using namespace std;

class B1 {
public:
  virtual ~B1() {}
};

class B2 {
public:
  virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};
```

```
int main() {
  B2* b2 = new Mi2;
  Mi2* mi2 = dynamic_cast<Mi2*>(b2);
  MI* mi = dynamic_cast<MI*>(b2);
  B1* b1 = dynamic_cast<B1*>(b2);
  assert(typeid(b2) != typeid(Mi2*));
  assert(typeid(b2) == typeid(B2*));
  delete b2;
} ///:~
```

This example has the extra complication of multiple inheritance (more on this later in this chapter). If you create an **Mi2** and upcast it to the root (in this case, one of the two possible roots is chosen), the **dynamic_cast** back to either of the derived levels **MI** or **Mi2** is successful. Comment

You can even cast from one root to the other:Comment

```
  B1* b1 = dynamic_cast<B1*>(b2);
```

This is successful because **B2** is actually pointing to an **Mi2** object, which contains a subobject of type **B1**.Comment

Casting to intermediate levels brings up an interesting difference between **dynamic_cast** and **typeid( )**. The **typeid( )** operation always produces a reference to a static **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 **b2** as a pointer to the derived type, like **dynamic_cast** does:Comment

```
typeid(b2) != typeid(Mi2*)
```

The type of **D2** is simply the exact type of the pointer:Comment

```
typeid(b2) == typeid(B2*)
```

void **pointers**

RTTI only works for complete types, meaning that all class information must be available when **typeid** is used. In particular, it doesn't work with **void** pointers:Comment

```
//: C08: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."[3]Comment

Using RTTI with templates

Class templates work well with RTTI, since all they do is generate classes. As usual, RTTI provides a convenient way to obtain the name of the class you're in. The following example prints the order of constructor and destructor calls:Comment

```
//: C08:ConstructorOrder.cpp
// Order of constructor calls
```

[3] A **dynamic_cast<void*>** always gives the address of the full object—not a subobject. This will be explained more fully in the next chapter.

```
#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; } ///:~
```

This template uses a constant **int** to differentiate one class from another, but type 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. The output is: Comment

```
Announce<0> constructor
Announce<1> constructor
Announce<2> constructor
X::X()
X::~X()
Announce<2> destructor
Announce<1> destructor
Announce<0> destructor
```

Multiple inheritance

Of course, the RTTI mechanisms must work properly with all the complexities of multiple inheritance, including **virtual** base classes (discussed in depth in the next chapter):Comment

```
//: C08: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
} ///:~
```

The **typeid( )** operation 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:Comment

```
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**.Comment

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 difficult not to organize programs into sets of **switch** statements. They could accomplish this with RTTI and thus lose the important value of polymorphism in code development and maintenance. The intent of C++ is that you use virtual functions throughout your code and that you only use RTTI when you must.Comment

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 derive 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 the main body that requires your new feature, you'll have to detect your particular type.Comment

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. Comment

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.Comment

A trash recycler

The following program simulates a trash recycler. Different kinds of "trash" are inserted into a single container and then later sorted according to their dynamic types. Comment

```
//: C08:Recycle.cpp
// A Trash Recycler
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <typeinfo>
#include <vector>
#include "../purge.h"
using namespace std;

class Trash {
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() { cout << "~Trash()\n"; }
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float 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(float 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(float 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() %
1000)/10.0));
        break;
```

```
      case 1 :
        bin.push_back(new Paper((rand() %
1000)/10.0));
        break;
      case 2 :
        bin.push_back(new Glass((rand() %
1000)/10.0));
        break;
    }
  // Note: bins hold exact type of object, not
base type:
  vector<Glass*> glassBin;
  vector<Paper*> paperBin;
  vector<Aluminum*> alumBin;
  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) alumBin.push_back(ap);
    else if(pp) paperBin.push_back(pp);
    else if(gp) glassBin.push_back(gp);
    sorter++;
  }
  sumValue(alumBin, cout);
  sumValue(paperBin, cout);
  sumValue(glassBin, cout);
  sumValue(bin, cout);
  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. Comment

We can do even better by using a **map** that associates pointers to **type_info** objects with a vector of **Trash** pointers. Since a map requires an ordering predicate, we provide one named **TInfoLess** that calls **type_info::before( )**. As we insert **Trash** pointers into the map, they are associated automatically with their **type_info** key. Comment

```
//: C08:Recycle2.cpp
// A Trash Recycler
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <map>
#include <typeinfo>
#include <utility>
#include <vector>
#include "../purge.h"
using namespace std;

class Trash {
  float _weight;
public:
  Trash(float wt) : _weight(wt) {}
  virtual float value() const = 0;
  float weight() const { return _weight; }
  virtual ~Trash() { cout << "~Trash()\n"; }
};

class Aluminum : public Trash {
  static float val;
public:
  Aluminum(float wt) : Trash(wt) {}
  float value() const { return val; }
  static void value(float 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(float 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(float newval) {
    val = newval;
  }
};
float Glass::val = 0.23;

// Comparator for type_info pointers
struct TInfoLess {
  bool operator()(const type_info* t1, const
type_info* t2)
  const {
    return t1->before(*t2);
  }
};
typedef map<const type_info*, vector<Trash*>,
TInfoLess>
  TrashMap;

// Sums up the value of the Trash in a bin:
void sumValue(const TrashMap::value_type& p,
ostream& os) {
  vector<Trash*>::const_iterator tally =
p.second.begin();
  float val = 0;
  while(tally != p.second.end()) {
    val += (*tally)->weight() * (*tally)->value();
    os << "weight of "
        << p.first->name()  // type_info::name()
        << " = " << (*tally)->weight() << endl;
    tally++;
  }
```

```
  os << "Total value = " << val << endl;
}

int main() {
  srand(time(0)); // Seed random number generator
  TrashMap bin;
  // Fill up the Trash bin:
  for(int i = 0; i < 30; i++) {
    Trash* tp;
    switch(rand() % 3) {
      case 0 :
        tp = new Aluminum((rand() % 1000)/10.0);
        break;
      case 1 :
        tp = new Paper((rand() % 1000)/10.0);
        break;
      case 2 :
        tp = new Glass((rand() % 1000)/10.0);
        break;
    }
    bin[&typeid(*tp)].push_back(tp);
  }
  // Print sorted results
  for (TrashMap::iterator p = bin.begin();
       p != bin.end(); ++p) {
    sumValue(*p, cout);
    purge(p->second);
  }
} ///:~
```

We've modified **sumValue( )** to call **type_info::name( )** directly, since the **type_info** object is now available there as the first member of the **TrashMap::value_type** pair. This avoids the extra call to **typeid** to get the name of the type of **Trash** being processed that was necessary in the previous version of this program. Comment

Mechanism and overhead of RTTI

Typically, RTTI is implemented by placing an additional pointer in a class's virtual function table. This pointer points to the **type_info** structure for that particular type. The effect of a **typeid( )** expression is quite simple: the virtual function table pointer is used to fetch the **type_info** pointer, and a reference to the resulting **type_info** structure is produced. Since this is just a two-pointer dereference operation, this is a constant time operation.Comment

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. A library routine then determines whether **source_pointer**'s type is of type **destination*** or a base class of **destination***. The pointer it returns may be 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 in which a base type may appear more than once in an inheritance hierarchy and virtual base classes are used.Comment

Because the library routine used for **dynamic_cast** must check through a list of base classes, the overhead for **dynamic_cast** may be higher than **typeid( )** (but of course you get different information, which may be essential to your solution), and 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**.Comment

Summary

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 dynamic type of the object pointed to by a base pointer, and that's what RTTI provides. 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.Comment

Exercises

1. Modify **C16:AutoCounter.h** in Volume 1 of this series so that it becomes a useful debugging tool. It will be used as a nested member of each class that you are interested in tracing. Turn **AutoCounter** into a template that takes the class name of the surrounding class as the template argument, and in all the error messages use RTTI to print out the name of the class.
2. 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.
3. Modify the **Instrument** hierarchy from Chapter 14 of Volume 1 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 with various types of Instrument objects created using the **new** operator. 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. Comment

9: Multiple inheritance

The basic concept of multiple inheritance (MI) sounds simple enough. 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 can simple as well.

Or maybe not! MI can introduce a number of ambiguities and strange situations, which are covered in this chapter. But first, it will be helpful to get a little perspective on the subject. Comment

Perspective

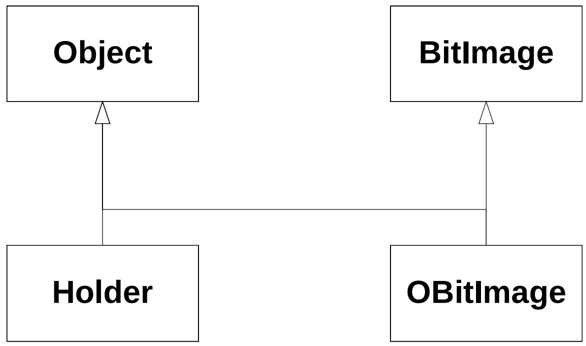
Before C++, the most successful object-oriented language was Smalltalk. Smalltalk was created from the ground up as an OO language. It is often referred to as *pure*, whereas C++, because it was built on top of C, is called a *hybrid* language (it supports more than just the OO paradigm). One of the 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. The Smalltalk class hierarchy is therefore a single monolithic tree.Comment

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.Comment

It was not a crystal-clear, however, that programmers could not get by without, multiple inheritance, and there was (and still is) a lot of disagreement about whether it 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 and exceptions) 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 is seldom involved in your daily design decisions.Comment

One of the most pressing issues at the time 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.Comment

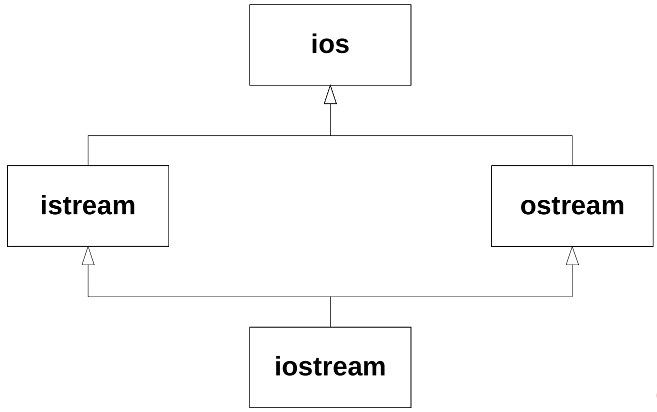
Now consider the situation in C++. Suppose vendor **A** creates an object-based hierarchy that includes a useful set of containers including one you want to use called **Holder**. Now you come across vendor **B**'s class hierarchy that contains some other class that is important to you, a **BitImage** 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**:Comment



Comment

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 5, the addition of templates has changed the way containers are created, so this situation isn't a driving issue for MI.Comment

The other reason you may need MI is related to design. You can intentionally use MI to make a design more flexible or useful (or at least seemingly so). An example of this is in the original **iostream** library design (which still persists in today's template design, as you saw in Chapter 4):Comment



Comment

Both **istream** and **ostream** are useful classes by themselves, but they can also be derived from simultaneously by a class that combines both their characteristics and behaviors. The class **ios** provides what is common to all stream classes, and so in this case MI is a code factoring mechanism. Comment

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.Comment

Interface Inheritance

One use of multiple inheritance that is not controversial pertains to *interface inheritance*. In C++, all inheritance is *implementation inheritance*, because everything in a base class, interface and implementation, becomes part of a derived class. It is not possible to only inherit part of a class (the interface alone, say). As chapter 14 of Volume 1 explains, private and protected inheritance make it possible to restrict access to members inherited from base classes when used by clients of a derived class object, but this doesn't affect the derived class; it still contains all base class data and can access all non-private base class members. Comment

Interface inheritance, on the other hand, only adds member function declarations to a derived class interface, and is not directly supported in C++. The usual technique to simulate interface inheritance in C++ is to derive from an *interface class*, which is a class that contains only declarations (no data or function bodies). These declarations will be pure virtual functions, of course. Here is an example. Comment

```
//: C09:Interfaces.cpp
// Multiple interface inheritance
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const = 0;
```

```
};

class Intable {
public:
  virtual ~Intable() {}
  virtual int toInt() const = 0;
};

class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const = 0;
};

class Able : public Printable,
             public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) {
    myData = x;
  }
  void print(ostream& os) const {
    os << myData;
  }
  int toInt() const {
    return myData;
  }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}
void testIntable(const Intable& n) {
  int i = n.toInt() + 1;
  cout << i << endl;
```

```
}
void testStringable(const Stringable& s) {
  string buf = s.toString() + "th";
  cout << buf << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```
Comment

Only pure virtual functions are inherited from classes **Printable**, **Intable**, and **Stringable**, which must therefore be implemented in derived class overrides, which the **Able** class provides. This gives **Able** objects multiple "is-a" relationships. The object **a** can act as a **Printable** object because its class **Able** derives publicly from **Printable**, and provides an implementation for **print( )**. The test functions have no need to know the most-derived type of their parameter; they just need an object that is substitutable for their parameter's type. Comment

As usual, a template solution is more compact:

```
//: C09:Interfaces2.cpp
// Implicit interface inheritance via templates
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Able {
  int myData;
public:
  Able(int x) {
    myData = x;
  }
  void print(ostream& os) const {
    os << myData;
  }
```

```
  int toInt() const {
    return myData;
  }
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  }
};

template<class Printable>
void testPrintable(const Printable& p) {
  p.print(cout);
  cout << endl;
}
template<class Intable>
void testIntable(const Intable& n) {
  int i = n.toInt() + 1;
  cout << i << endl;
}
template<class Stringable>
void testStringable(const Stringable& s) {
  string buf = s.toString() + "th";
  cout << buf << endl;
}

int main() {
  Able a(7);
  testPrintable(a);
  testIntable(a);
  testStringable(a);
} ///:~
```
Comment

The names **Printable**, **Intable** and **Stringable** are now just template parameters that assume the existence of the operations indicated in their respective contexts. Some people are more comfortable with the first version, because the type names guarantee by inheritance that the expected interfaces are implemented. Others are content with the fact that if the operations required by the test functions are not satisfied by their template type arguments, the error is still caught at compile time. The latter approach is technically a “weaker” form of type checking than the former (inheritance) approach, but the effect on the programmer (and the program) is the same. This is one form of weak typing that is acceptable to modern C++ programmers. Comment

Implementation Inheritance

As we stated earlier, C++ only provides implementation inheritance, meaning that you inherit everything from all of your base classes. This can be Good Thing, of course, because it frees you from having to implement *everything* in the derived class, like we had to do with the interface inheritance examples above. A common use of multiple inheritance involves using **mixin classes**, which are classes not intended to be instantiated independently, but exist to add capabilities to other classes through inheritance. Comment

As an example, suppose we are clients of a class that supports access to a database. We will likely only have a header file available (which is part of the point we are about to make), but for illustration, assume the following, simple implementation of a **Database** class: Comment

```
//: C09:Database.h
// A prototypical resource class
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>
using std::cout;
using std::string;
using std::runtime_error;

struct DatabaseError : runtime_error {
  DatabaseError(const string& msg) :
runtime_error(msg)
  {}
};
```

```
class Database {
public:
  Database(const string& dbStr) : dbid(dbStr) {}
  virtual ~Database(){}
  void open() throw(DatabaseError) {
    cout << "connected to " << dbid << '\n';
  }
  void close() {
    cout << dbid << " closed\n";
  }
  //Other database functions...
private:
  string dbid;
};
#endif ///:~
```

We're leaving out actual database functionality (storing, retrieving, etc.), but that's actually not important here. Using this class requires a database connection string, and that you call **Database::open( )** to connect and **Database::close( )** to disconnect: Comment

```
//: C09:UseDatabase.cpp
#include "Database.h"
int main() {
  Database db("MyDatabase");
  db.open();
  // Use other db functions...
  db.close();
}
/* Output:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```
Comment

In a typical client-server situation, a client will have multiple objects sharing a connection to a database. It is important that the database eventually be closed, but only after access to it is no longer required. It is common to encapsulate this behavior through a class that tracks the number of client entities using the database connection, and to automatically terminate the connection when that count goes to zero. To add reference counting to the **Database** class, we create a mixin class named **Countable**, and mix it into the **Database** class by creating a new class, **DBConnection**, through multiple inheritance. Here's the **Countable** mixin class: Comment

```
//: C09:Countable.h
// A "mixin" class
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
public:
  long attach() { return ++count; }
  long detach() {
    return (--count > 0) ? count : (delete this,
0);
  }
  long refCount() const { return count; }
protected:
  Countable() { count = 0; }
  virtual ~Countable() { assert(count == 0); }
private:
  long count;
};
#endif ///:~
```
Comment

It is evident that this is not a stand-alone class because its constructor is **protected**; it therefore requires a friend or derived class to use it. It is important that the destructor is virtual, because it is only called from the **delete this** statement in **detach( )**, and we of course want the complete derived object to be destroyed. The **DBConnection** derives from both **Database** and **Countable**, and provides a static **create( )** function that initializes its **Countable** subobject. (This is an example of the Factory Method design pattern, discussed in the next chapter.) Comment

```
//: C09:DBConnection.h
// Uses a "mixin" class
#ifndef DBCONNECTION_H
```

```
#define DBCONNECTION_H
#include "Countable.h"
#include "Database.h"
#include <cassert>
#include <string>
using std::string;

class DBConnection : public Database, public
Countable {
public:
  static DBConnection* create(const string& dbStr)
  throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
// Other added functionality as desired...
protected:
  DBConnection(const string& dbStr)
throw(DatabaseError)
  : Database(dbStr) {
    open();
  }
  ~DBConnection() {
    close();
  }
private:
  // Disallow copy
  DBConnection(const DBConnection&);
  DBConnection& operator=(const DBConnection&);
};
#endif ///:~
```
Comment

We now have a reference-counted database connection without modifying the **Database** class, and can safely assume that it will not be surreptitiously terminated. The opening and closing is done using the Resource Acquisition is Initialization idiom (RAII) mentioned in Chapter 1 via the **DBConnection** constructor and destructor. This makes using a **DBConnection** easy to use, as the following program shows. Comment

```
//: C09:UseDatabase2.cpp
// Tests the Countable "mixin" class
#include <cassert>
#include "DBConnection.h"

class DBClient {
public:
  DBClient(DBConnection* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() {
    db->detach();
  }
  // Other database requests using db…
private:
  DBConnection* db;
};

int main() {
  DBConnection* db =
DBConnection::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
  // Use database, then release attach from
original create
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```
Comment

The call to **DBConnection::create( )** calls **attach( )**, so when we're finished we must explicitly call **detach( )** to release the original hold on the connection. Note that the **DBClient** class also uses RAII to manage its use of the connection. When the program terminates, the destructors for the two **DBClient** objects will decrement the reference count (by calling **detach( )**, which **DBConnection** inherited from **Countable**), and the database connection will be closed when the count reaches zero after the object **c1** is destroyed. (This is because of **Countable**'s virtual destructor, as we explained earlier). Comment

A template approach is commonly used for mixin inheritance, allowing the user to specify at compile time which flavor of mixin is desired. This way we can use different reference-counting approaches without explicitly defining **DBConnection** twice. Here's how it's done. Comment

```
//: C09:DBConnection2.h
// A parameterized mixin
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include "Database.h"
#include <cassert>
#include <string>
using std::string;

template<class Counter>
class DBConnection : public Database, public
Counter {
public:
  static DBConnection* create(const string& dbStr)
  throw(DatabaseError) {
    DBConnection* con = new DBConnection(dbStr);
    con->attach();
    assert(con->refCount() == 1);
    return con;
  }
// Other added functionality as desired...
protected:
  DBConnection(const string& dbStr)
throw(DatabaseError)
  : Database(dbStr) {
    open();
  }
  ~DBConnection() {
    close();
  }
private:
  // Disallow copy
```

```
  DBConnection(const DBConnection&);
  DBConnection& operator=(const DBConnection&);
};
#endif ///:~
```
Comment

The only change here is the template prefix to the class definition (and renaming **Countable** to **Counter** for clarity). We could also make the database class a template parameter (had we multiple database access classes to choose from), but it is not a mixin, per se, since it is a stand-alone class. The following example uses the original **Countable** as the **Counter** mixin type, but we could use any type that implements the appropriate interface (**attach( )**, **detach( )**, etc.). Comment

```
//: C09:UseDatabase3.cpp
// Tests a parameterized "mixin" class
#include <cassert>
#include "Countable.h"
#include "DBConnection2.h"

class DBClient {
public:
  DBClient(DBConnection<Countable>* dbCon) {
    db = dbCon;
    db->attach();
  }
  ~DBClient() {
    db->detach();
  }
private:
  DBConnection<Countable>* db;
};

int main() {
  DBConnection<Countable>* db =
    DBConnection<Countable>::create("MyDatabase");
  assert(db->refCount() == 1);
  DBClient c1(db);
  assert(db->refCount() == 2);
  DBClient c2(db);
  assert(db->refCount() == 3);
```

```
  db->detach();
  assert(db->refCount() == 2);
} ///:~
```
Comment

The general pattern for multiple parameterized mixins is simply:

```
template<class Mixin1, class Mixin2, … , class
MixinK>
class Subject : public Mixin1,
                public Mixin2,
                …
                public MixinK {..};
```

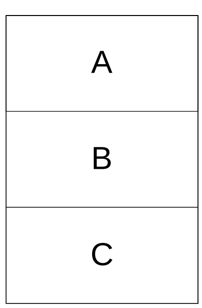
Duplicate subobjects

When you inherit from a base class, you get a copy of all the data members of that base class in your derived class. The following program shows how multiple base subobjects might be laid out in memory. Comment

```
//: C09:Offset.cpp
// Illustrates layout of subobjects with MI
#include <iostream>
using namespace std;

class A {
  int x;
};

class B {
  int y;
};

class C : public A, public B {
  int z;
};

int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl;
  cout << "sizeof(B) == " << sizeof(B) << endl;
  cout << "sizeof(C) == " << sizeof(C) << endl;
```

```
  C c;
  cout << "&c == " << &c << endl;
  A* ap = &c;
  B* bp = &c;
  cout << "ap == " << static_cast<void*>(ap) <<
endl;
  cout << "bp == " << static_cast<void*>(bp) <<
endl;
  C* cp = static_cast<C*>(bp);
  cout << "cp == " << static_cast<void*>(cp) <<
endl;
  cout << "bp == cp? " << boolalpha << (bp == cp)
<< endl;
  cp = 0;
  bp = cp;
  cout << bp << endl;
}
/* Output:
sizeof(A) == 4
sizeof(B) == 4
sizeof(C) == 12
&c == 1245052
ap == 1245052
bp == 1245056
cp == 1245052
bp == cp? true
0
*/ ///:~ Comment
```

As you can see, the **B** portion of the object **c** is offset 4 bytes from the beginning of the entire object, suggesting the following layout:



The object **c** begins with it's **A** subobject, then the **B** portion, and finally the data from the complete type **C** itself. Since a **C** is-a **A** and is-a **B**, it is possible to upcast to either base type. When upcasting to an **A**, the pointer points to the **A** portion, which happens to be at the beginning of the object, so the address **ap** is the same as the expression **&c**. When upcasting to a **B**, however, it must point to where the **B** subobject actually resides, because class B knows nothing about class **C** (or class **A**, for that matter). In other words, the object pointed to by **bp** must be able to behave as a stand-alone **B** object (except for any required polymorphic behavior, of course). Comment

When casting **bp** back to a **C***, since the original object was a **C** in the first place, it knows where the **B** subobject resides, and adjusts the pointer back to the original address of the complete object. If **bp** had been pointing to a stand-alone **B** object instead of a **C** object in the first place, the cast would be illegal[1]. Furthermore, in the comparison **bp == cp**, **cp** is implicitly converted to a **B***, since that is the only way to make the comparison meaningful in general (i.e., upcasting is always allowed), hence the **true** result. So when converting back and forth between subobjects and complete types, the appropriate offset is applied. Comment

The null pointer requires special handling, obviously, since blindly subtracting an offset when converting to or from a **B** subobject will result in an invalid address if the pointer were 0 to start with. For this reason, when casting to or from a **B***, the compiler generates logic to check first to see if the pointer is zero. If it isn't, it applies the offset; otherwise it leaves it as zero. Comment

If you have multiple base classes, and if those base classes in turn have a common base class, you will have two copies of the top-level base, as you can see in the following example. Comment

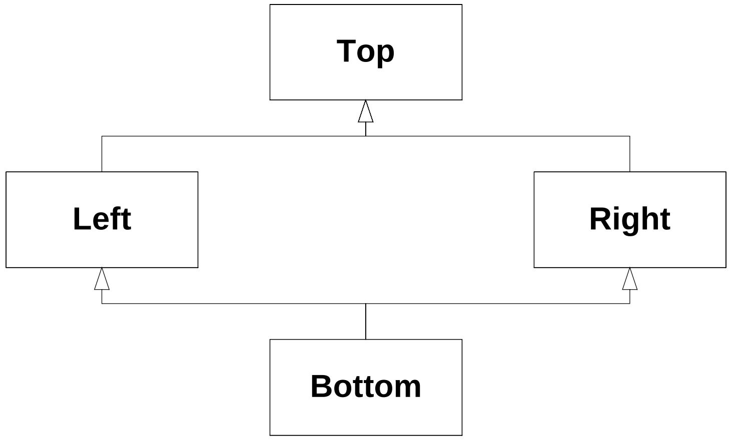
```
//: C09:Duplicate.cpp
```

[1] But not detected as an error. **dynamic_cast**, however can solve this problem. See the next chapter for details.

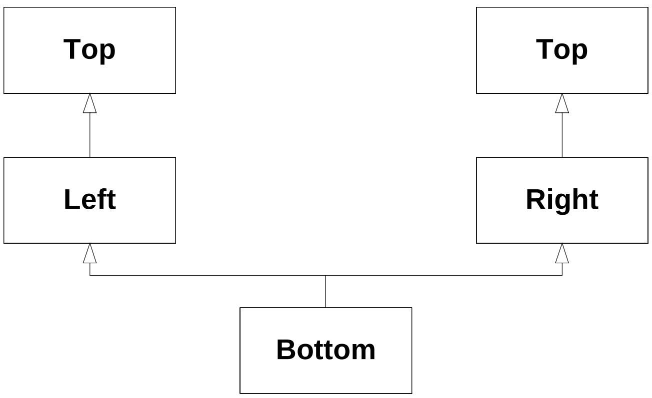
```
// Shows duplicate subobjects
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
};

class Left : public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Left(i, k), Right(j, k) { w = m; }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl; // 20
} ///:~
```

Comment

The fact that the size of **b** is 20 bytes reveals the fact that there are five integers altogether in a complete **Bottom** object. A typical class diagram for this scenario is usually appears as:



This is the so-called "diamond inheritance", but it would be better rendered as: Comment



The awkwardness of this design surfaces in the constructor for the **Bottom** class in the code above. The user only thinks four integers are required, but which arguments should be passed to the two parameters that **Left** and **Right** require? Although this design is not inherently "wrong", it is usually not what an application calls for. It also presents a problem when trying to convert a pointer to a **Bottom** object to a **Top***. As we showed earlier, the address may need to be adjusted, depending on where the subobject resides within the complete object, but in this case there are *two* **Top** subobjects to choose from. The compiler doesn't know which to choose, so such an upcast is ambiguous, and therefore not allowed. The same reasoning explains why a **Bottom** object would not be able to call a function that is only defined in **Top**. If such a function **Top::f( )** existed, calling **b.f( )** above would need to refer to a **Top** subobject as a context in which to execute, and there are two to choose between. Comment

Virtual base classes

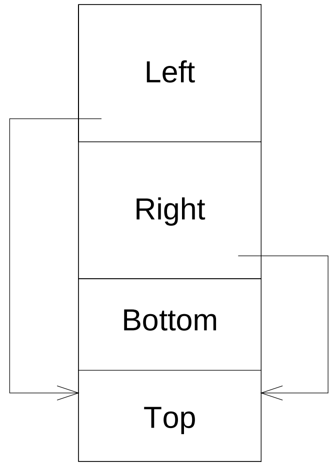
What we usually want in such cases is *true* diamond inheritance, where a single **Top** object is shared by both **Left** and **Right** subobjects within a complete **Bottom** object, which is what the first class diagram depicts. This is achieved by making **Top** a *virtual base class* of **Left** and **Right**: Comment

```
//: C09:VirtualBase.cpp
// Shows a shared subobject via a virtual base
#include <iostream>
using namespace std;

class Top {
protected:
  int x;
public:
  Top(int n) { x = n; }
  virtual ~Top(){}
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
protected:
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
};

class Right : virtual public Top {
```

```
protected:
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << b.x << ',' << b.y << ',' << b.z
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << sizeof b << endl;
  cout << b << endl;
  cout << static_cast<void*>(&b) << endl;
  Top* p = static_cast<Top*>(&b);
  cout << *p << endl;
  cout << static_cast<void*>(p) << endl;
  cout << dynamic_cast<void*>(p) << endl;
} ///:~
```
Comment

A virtual base class only allows one subobject of its type to exist in any complete derived object, no matter where it appears in the hierarchy[2]. This means that when a **Bottom** object is instantiated, the object layout may look something like this: Comment

[2] We use the term "hierarchy" because everyone else does, but the graph representing multiple inheritance relationships is in general a *directed acyclic graph* (DAG), also called a *lattice*, for obvious reasons.



The **Left** and **Right** subobjects each have a pointer (or some conceptual equivalent) to the shared **Top** subobject, and all references to that subobject in **Left** and **Right** member functions will go through those these pointers[3]. In this case there is no ambiguity when upcasting from a **Bottom** to a **Top** object, since there is only one **Top** object to convert to. Comment

The output of the program above is as follows:

```
36
1,2,3,4
1245032
1
1245060
1245032
```

The addresses printed suggest that this particular implementation does indeed store the **Top** subobject at the end of the complete object (although it's not really important where it goes). Even more

[3] The presence of these pointers explains why the size of **b** is much larger than the size of four integers. This is (part of) the cost of virtual base classes. There is also vptr overhead due to the virtual constructor.

important is the result of the **dynamic_cast** to **void***. Such a cast always resolves to the address of the complete object. Comment

We made the **Top** destructor virtual so we could apply the **dynamic_cast** operator. If you remove that virtual destructor (and the **dynamic_cast** statement so the program will compile), the size of **Bottom** decreases to 24 bytes. That seems to be a decrease equivalent to the size of three pointers. What gives? Comment

It's important not to take these numbers too literally. Other compilers we use manage only to increase the size by four bytes when the virtual constructor is added. Not being compiler writers, we can't tell you their secrets. We can tell you, however, that with multiple inheritance, a derived object must behave as if it has multiple VPTR's, one for each of its direct base classes that also have virtual functions. It's as simple as that. Compilers can make whatever optimizations its authors can invent, but the behavior must be the same. Comment

Certainly the strangest thing in the code above is the initializer for **Top** in the **Bottom** constructor. Normally one doesn't worry about initializing subobjects beyond direct base classes, since all classes take care of initializing their own bases. There are, however, multiple paths from **Bottom** to **Top**, so relying on the intermediate classes **Left** and **Right** to pass along the necessary initialization data results in an ambiguity (whose responsibility is it?)! For this reason, it always the responsibility of the *most derived class* to initialize a virtual base. But what about the expressions in the **Left** and **Right** constructors that also initialize **Top**? They are certainly necessary when crating stand-alone **Left** or **Right** objects, but must be *ignored* when a **Bottom** object is created (hence the zeroes in their initializers in the **Bottom** constructor—any values in those slots are ignored when the **Left** and **Right** constructors execute n the context of a **Bottom** object). The compiler takes care of all this for you, but it's important to understand where the responsibility lies. Always make sure that *all concrete (non-abstract) classes* in a multiple inheritance hierarchy are aware of any virtual bases and initialize them appropriately. Comment

These rules of responsibility apply not only to initialization but to all operations that span the class hierarchy. Consider the stream inserter in the code above. We made the data protected so we could “cheat” and access inherited data in **operator<<(ostream&, const Bottom&)**. It usually makes more sense to assign the work of printing each subobject to its corresponding class, and have the derived class call its base class functions as needed. What would happen if we tried that with **operator<<( )**, as the following code illustrates? Comment

```
//: C09:VirtualBase2.cpp
// Shows how not to implement operator<<
#include <iostream>
using namespace std;

class Top {
  int x;
public:
  Top(int n) { x = n; }
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream&
  operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ','
<< l.y;
  }
};

class Right : virtual public Top {
  int z;
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream&
```

```
  operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ','
<< r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    return os << static_cast<const Left&>(b)
      << ',' << static_cast<const Right&>(b)
      << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,1,3,4
} ///:~
```

You can't just blindly share the responsibility upward in the usual fashion because the **Left** and **Right** stream inserters each call the **Top** inserter, and again there will be duplication of data. Instead you need to mimic what the compiler automatically does with initialization. One solution is to provide special functions in the classes that know about the virtual base class, which ignore the virtual base when printing (leaving the job to the most derived class): Comment

```
//: C09:VirtualBase3.cpp
// A correct stream inserter
#include <iostream>
using namespace std;

class Top {
  int x;
public:
```

```
  Top(int n) { x = n; }
  friend ostream&
  operator<<(ostream& os, const Top& t) {
    return os << t.x;
  }
};

class Left : virtual public Top {
  int y;
protected:
  void specialPrint(ostream& os) const {
    // Only print Left's part
    os << ','<< y;
  }
public:
  Left(int m, int n) : Top(m) { y = n; }
  friend ostream&
  operator<<(ostream& os, const Left& l) {
    return os << static_cast<const Top&>(l) << ','
<< l.y;
  }
};

class Right : virtual public Top {
  int z;
protected:
  void specialPrint(ostream& os) const {
    // Only print Right's part
    os << ','<< z;
  }
public:
  Right(int m, int n) : Top(m) { z = n; }
  friend ostream&
  operator<<(ostream& os, const Right& r) {
    return os << static_cast<const Top&>(r) << ','
<< r.z;
  }
};

class Bottom : public Left, public Right {
  int w;
public:
```

```
  Bottom(int i, int j, int k, int m)
  : Top(i), Left(0, j), Right(0, k) { w = m; }
  friend ostream&
  operator<<(ostream& os, const Bottom& b) {
    os << static_cast<const Top&>(b);
    b.Left::specialPrint(os);
    b.Right::specialPrint(os);
    return os << ',' << b.w;
  }
};

int main() {
  Bottom b(1, 2, 3, 4);
  cout << b << endl;  // 1,2,3,4
} ///:~
```
Comment

The **specialPrint( )** functions are **protected** since they will only be called by **Bottom**. They only print their own data and ignore their **Top** subobject, because the **Bottom** inserter is in control when these functions are called. The **Bottom** inserter must know about the virtual base, just as a **Bottom** constructor needs to. This same reasoning applies to assignment operators in a hierarchy with a virtual base, as well as to any function, member or not, that wants to share the work throughout all classes in the hierarchy. Comment

Having discussed virtual base classes we can now illustrate the “full story” of object initialization. Since virtual bases give rise to shared subojects, it makes sense that they should be available before the sharing takes place, so the order of initialization of subobjects follows these rules (recursively, as needed, of course): Comment

1. All virtual base class subobjects are initialized, in top-down, left-to-right order according to where they appear in class definitions.

2. Non-virtual base classes are then initialized in top-down, left-to-right order.

3. All member objects are initialized in declaration order.

4. The complete object's constructor executes.

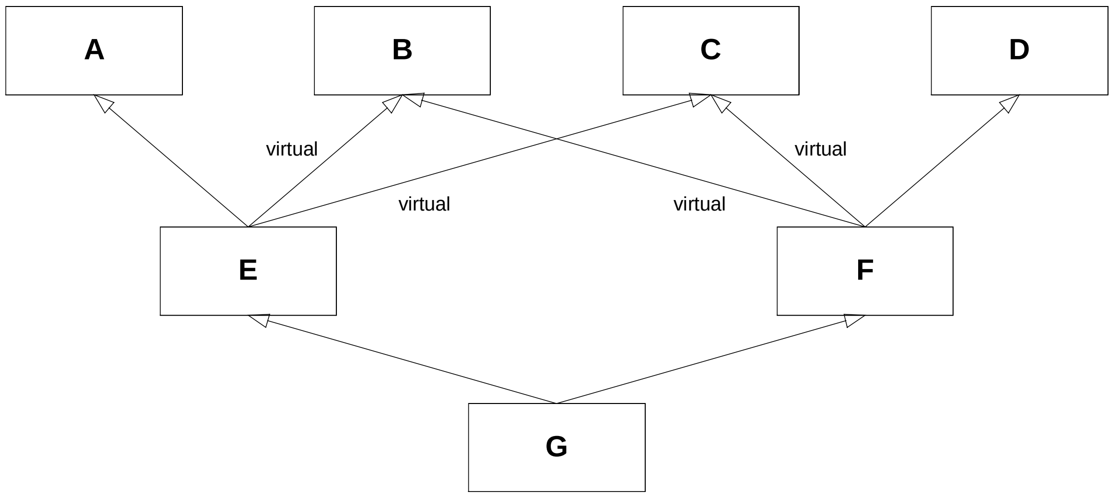
The following program reveals this behavior. Comment

```
//: C09:VirtInit.cpp
// Illustrates initialization order with virtual
bases
#include <iostream>
#include <string>
using namespace std;

class M {
public:
  M(const string& s) {
    cout << "M " << s << endl;
  }
};

class A{
  M m;
public:
  A(const string& s) : m("in A") {
     cout << "A " << s << endl;
  }
};

class B
{
  M m;
public:
  B(const string& s) : m("in B")  {
    cout << "B " << s << endl;
  }
};

class C
{
  M m;
public:
  C(const string& s) : m("in C")  {
    cout << "C " << s << endl;
  }
```

```
};

class D
{
  M m;
public:
  D(const string& s) : m("in D") {
    cout << "D " << s << endl;
  }
};

class E : public A, virtual public B, virtual
public C
{
  M m;
public:
  E(const string& s)
  : A("from E"), B("from E"), C("from E"), m("in
E") {
    cout << "E " << s << endl;
  }
};

class F : virtual public B, virtual public C,
public D
{
  M m;
public:
  F(const string& s)
  : B("from F"), C("from F"), D("from F"), m("in
F") {
    cout << "F " << s << endl;
  }
};

class G : public E, public F
{
  M m;
public:
  G(const string& s)
  : B("from G"), C("from G"), E("from G"),
    F("from G"), m("in G") {
```

```
    cout << "G " << s << endl;
  }
};

int main() {
  G g("from main");
} ///:~
```

The classes in this code can be represented by the following diagram: Comment



Each class has an embedded member of type **M**. Note that only four derivations are virtual: **E** from **B** and **C**, and **F** from **B** and **C**. The output of this program is

```
M in B
B from G
M in C
C from G
M in A
A from E
M in E
E from G
M in D
```

```
D from F
M in F
F from G
M in G
G from main
```

The initialization of **g** requires its **E** and **F** part to first be initialized. The **E** subobject requires **A**, **B**, and **C** subobjects. Since **B** and **C** are virtual bases, they go first, and are initialized from **G**'s initializer, **G** being the most-derived class. The class **B** has no base classes, so according rule 3 above its member object **m** is initialized, then its constructor prints "**B** from **G**", and similarly for the **C** subject of **E**. Next the **A** subobject of the **E** subobject is initialized, and then the **E** subobject itself. The same scenario repeats for g's **F** subobject, but without duplicating the initialization of the virtual bases. Comment

Name Lookup Issues

The ambiguities we have illustrated with subobjects apply of course to any names, including function names. If a class has multiple direct base classes that share member functions of the same name, the compiler doesn't know which one to choose. The following sample program would report such an error. Comment

```
// C09:AmbiguousName.cpp
class Top {};

class Left : virtual public Top {
public:
   void f(){}
};

class Right : virtual public Top {
public:
   void f(){}
};

class Bottom : public Left, public Right {};

int main() {
```

```
   Bottom b;
   b.f();     // error here
}
```

The class **Bottom** has inherited two functions of the same name (the signature is irrelevant, since name lookup occurs before overload resolution), and there is no way to choose between them. The usual technique to disambiguate the call is to inject the name of choice to break the tie: Comment

```
//: C09:BreakTie.cpp
class Top {};

class Left : virtual public Top {
public:
   void f(){}
};

class Right : virtual public Top {
public:
   void f(){}
};

class Bottom : public Left, public Right {
public:
  using Left::f;
};

int main() {
   Bottom b;
   b.f();     // calls Left::f()
} ///:~
```

The name **Left::f** is now found in the scope of **Bottom**, so the name **Right::f** is not even considered. Of course, if you want to introduce extra functionality beyond what **Left::f( )** provides, you would implement a **Bottom::f( )** function that calls **Left::f( )**, in addition to other things. Comment

The reason that the first version (**AmbiguousName.cpp**) had a problem was because **Left::f( )** and **Right::f( )** were at equal levels

in the hierarchy. The following hierarchy has no such problem:
Comment

```
//: C09:Dominance.cpp
class Top {
public:
  virtual void f() {}
};

class Left : virtual public Top {
public:
   void f(){}
};

class Right : virtual public Top {
};

class Bottom : public Left, public Right {};

int main() {
   Bottom b;
   b.f(); // calls Left::f()
} ///:~
```

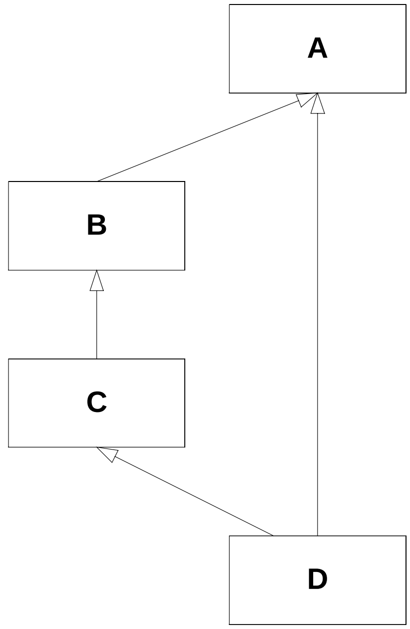
In this case there is no **Right::f( )**, so **Left::f( )**, being the most derived, is the one that is chosen. Why? Well, pretend that **Right** did not exist, giving the single-inheritance hierarchy **Top <= Left <= Bottom**. You would certainly expect **Left::f( )** to be the function called by the expression **b.f( )**, because of normal scope rules (a derived class is considered to be a nested scope of a base class). In general, a name **A::f** is said to *dominate* the name **B::f** if **A** derives from **B**, directly or indirectly, or in other words, if **A** is "more derived" in the hierarchy than **B**[4]. The following program further illustrates the dominance principle. Comment

```
//: C09:Dominance2.cpp
```

[4] Note that the virtual inheritance is crucial to this example. If **Top** were not a virtual base class, there would be multiple **Top** subobjects, and the ambiguity would remain. Dominance with multiple inheritance only comes into play with virtual base classes.

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

class A {
public:
   virtual void f() {cout << "A::f\n";}
};

class B : virtual public A {
public:
   void f() {cout << "B::f\n";}
};

class C : public B {};
class D : public C, virtual public A {};

int main()
{
   B* p = new D;
   p->f();  // calls B::f()
} ///:~
```

The class diagram for this hierarchy is as follows. Comment



Even though there is a shorter path from **D** to **A** than from **D** to **B**, **A** is a (direct, in this case) base class for **B**, and so **B::f** dominates **A::f**. Comment

Avoiding MI

When the question of whether to use multiple inheritance comes up, you should ask at least two questions:Comment

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? Comment

2. Do I need to upcast to both of the base classes? (This applies when you have more than two base classes, of course.) Comment

If you can't answer "no" to both questions, you can avoid using MI and should probably do so.Comment

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[5]. However, type conversion can't be used for normal member selection; that requires inheritance.Comment

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.Comment

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.Comment

For example, here's the header file for a library you acquire:Comment

---

[5] Jerry Schwarz, the author of IOStreams, has remarked to both of us on separate occasions that if he had it to do over again, he would probably remove MI from the design of IOStreams and use multiple stream buffers and conversion operators instead.

```
//: C09: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:Comment

```
//: C09:Vendor.cpp {O}
// Implementation of VENDOR.H
// This is compiled and unavailable to you
#include <iostream>
#include "Vendor.h"
using namespace std;

void Vendor::v() const {
  cout << "Vendor::v()\n";
}
void Vendor::f() const {
  cout << "Vendor::f()\n";
```

```
}
Vendor::~Vendor() {
  cout << "~Vendor()\n";
}
void Vendor1::v() const {
  cout << "Vendor1::v()\n";
}
void Vendor1::f() const {
  cout << "Vendor1::f()\n";
}
Vendor1::~Vendor1() {
  cout << "~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 with the equivalent file suffixes for your system).Comment

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.Comment

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:Comment

```
//: C09:Paste.cpp
//{L} Vendor
// Fixing a mess with MI
#include <iostream>
#include "Vendor.h"
using namespace std;

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() { cout << "~MyBase()\n"; }
};

class Paste1 : public MyBase, public Vendor1 {
public:
  void v() const {
    cout << "Paste1::v()\n";
    Vendor1::v();
  }
  void f() const {
    cout << "Paste1::f()\n";
    Vendor1::f();
  }
  void g() const {
    cout << "Paste1::g()\n";
  }
  ~Paste1() { cout << "~Paste1()\n"; }
};

int main() {
  Paste1& p1p = *new Paste1;
  MyBase& mp = p1p; // Upcast
  cout << "calling f()\n";
  mp.f();  // Right behavior
  cout << "calling g()\n";
```

```
  mp.g(); // New behavior
  cout << "calling A(p1p)\n";
  A(p1p); // Same old behavior
  cout << "calling B(p1p)\n";
  B(p1p);  // Same old behavior
  cout << "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( )**Comment

```
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:Comment

```
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.Comment

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.Comment

Summary

The reason MI exists in C++ and not in some 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.Comment

If no "diamonds" appear in your class hierarchy, MI is fairly simple (although identical function signatures in base classes must still 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.Comment

Multiple inheritance has been called the "goto of the 90's".[6] 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.Comment

[6] A phrase coined by Zack Urlocker.

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 take a single **int** argument. Now derive **A** from **Y** and **Z**. 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.

10: Design patterns

"... describe a problem which occurs over and over again in our environment, and then describe 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 nontraditional "patterns" approach to program design.

Probably the most important step forward in recent times in object-oriented design is the "design patterns" movement, chronicled in *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995).[1] That book shows 23 solutions to particular classes of problems. In this chapter, we discuss the basic concepts of design patterns and provide code examples that illustrate selected patterns. This should whet your appetite for reading *Design Patterns* (a source of what has now become an essential, almost mandatory vocabulary for object-oriented programming). Comment

The pattern concept

At first blush, you can think of a pattern as an especially clever and insightful way to solve 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. Furthermore, the pattern exists independently of any particular implementation and, indeed, can be implemented in a number of ways. Comment

[1] Also known as the "Gang of Four" book (GoF). Conveniently, the examples are in C++.

Although they're called "design patterns," they really aren't tied to the realm of design only. 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 span the analysis phase and high-level design phase. Because a pattern has a direct implementation in code, 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). Comment

The basic concept of a pattern can also be seen as the basic concept of program design in general: adding layers of abstraction. Whenever you abstract something, you're isolating particular details, and one of the most compelling motivations for 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 easier to maintain, but it also renders code easier to read and understand (which invariably results in lowered costs over time). Comment

The most difficult part of developing an elegant and maintainable design is often discovering what we call "the vector of change." (Here, "vector" refers to the maximum gradient as understood in the sciences, 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. Comment

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 have not been classified as design patterns. Comment

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 (Standard Template Library); it hides the particular implementation of the container as you're stepping through and selecting the elements one by one. Iterators allow you to write generic code that performs an operation on all 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. Comment

The singleton

Possibly the simplest design pattern is the Singleton, which is a way to provide one and only one instance of a class:

```
//: C10: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& instance() {
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};
```

```
Singleton Singleton::s(47);

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

The key to creating a singleton is to prevent the client programmer from having any control over the lifetime of the object. To do this, you must declare all constructors **private**, and you must prevent the compiler from implicitly generating any constructors. Note that the copy constructor and assignment operator are declared private to prevent any sort of copies being made. Comment

You must also decide how you're going to create the object. Here, it's created statically, but you can also wait until the client programmer asks for one and create it on demand. In the latter case, you'll have to use a pointer instead of a reference, but that allows the user to inadvertently delete the pointer, so the implementation above is considered safest. In any case, the object should be stored privately. You provide access through public methods. Here, **instance( )** produces a reference to the **Singleton** object. The rest of the interface (**getValue( )** and **setValue( )**) is the regular class interface. Comment

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. Comment

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, a problem is associated with static objects (member or not), and that's the order of initialization, as described in Volume 1 of this book. If one static object depends on another, it's important that the order of initialization proceed correctly. Comment

In Volume 1, you were shown how a static object defined inside a function can be used to control initialization order. 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 log file upon the first call to a function that returns a reference to that log file. This header file will do the trick: Comment

```
//: C10:LogFile.h
#ifndef LOGFILE_H
#define LOGFILE_H
#include <fstream>

std::ofstream& logfile();

#endif // LOGFILE_H ///:~
```

The implementation *must not be inlined*, because that would mean that the whole function, including the static object definition within, could be duplicated in any translation unit where it's included, and you'd end up with multiple copies of the static object. This would most certainly foil the attempts to control the order of initialization (but potentially in a very subtle and hard-to-detect fashion). So the implementation must be separate: Comment

```
//: C10:LogFile.cpp {O}
#include "LogFile.h"
std::ofstream& logfile() {
```

```
  static std::ofstream log("Logfile.log");
  return log;
} ///:~
```

Now the **log** object will not be initialized until the first time **logfile( )** is called. So if you use the function in one file: Comment

```
//: C10:UseLog1.h
#ifndef USELOG1_H
#define USELOG1_H
void f();
#endif // USELOG1_H ///:~

//: C10:UseLog1.cpp {O}
#include "UseLog1.h"
#include "LogFile.h"
void f() {
  logfile() << __FILE__ << std::endl;
} ///:~
```

And again in another file:

```
//: C10:UseLog2.cpp
//{L} LogFile UseLog1
#include "UseLog1.h"
#include "LogFile.h"
using namespace std;

void g() {
  logfile() << __FILE__ << endl;
}

int main() {
  f();
  g();
} ///:~
```

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[2]: Comment

```
//: C10:SingletonPattern2.cpp
// Meyers' Singleton
#include <iostream>
using namespace std;

class Singleton {
  int i;
  Singleton(int x) : i(x) { }
  void operator=(Singleton&);
  Singleton(const Singleton&);
public:
  static Singleton& instance() {
    static Singleton s(47);
    return s;
  }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};

int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
} ///:~
```

An especially interesting case is if two singletons depend on each other, like this:

```
//: C10:FunctionStaticSingleton.cpp
class Singleton1 {
  Singleton1() {}
public:
  static Singleton1& ref() {
```

[2] This is known as Meyers' Singleton, after its creator, Scott Meyers.

```
    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 fewer problems. Comment

Yet another variation on Singleton allows you to separate the "singleton-ness" of an object from its implementation. This is achieved through templates, using the Curiously Recurring Template Pattern mentioned in Chapter 5. The template class Singleton is really a "holder" for any class you want to restrict to a single object. Comment

```
// C10:CuriousSingleton.cpp
// Separates a class from its singleton-ness
#include <iostream>
using namespace std;
// The "holder"
template<class T>
class Singleton {
```

```
public:
  static T& instance() {
    static T theInstance;
    return theInstance;
  }
protected:
  Singleton(){}
  virtual ~Singleton(){}
private:
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);
};
// A sample class to be made into a Singleton
class MyClass : public Singleton<MyClass> {
  int x;
public:
  MyClass() { x = 0; }
  void setValue(int n) { x = n; }
  int getValue() const { return x; }
};

int main() {
  MyClass& m = MyClass::instance();
  cout << m.getValue() << endl;
  m.setValue(1);
  cout << m.getValue() << endl;
} ///:~
```

MyClass is made a Singleton simply by inheriting from **Singleton<MyClass>**. This self-referencing may sound implausible, but it works because there is no non-static data dependent on the template argument in the **Singleton** template. In other words, the code for the class **Singleton<MyClass>** can be instantiated by the compiler because it is not dependent on the size of **MyClass**. It's only later, when **Singleton<MyClass>::instance( )** is first called, that the size of

MyClass is needed, and of course by then compilation is over and its value is known[3]. Comment

It's interesting how intricate implementing such a simple pattern as Singleton can be. We haven't even addressed issues of thread safety, and yet many pages have elapsed since the beginning of this section. The last thing we wish to say about Singleton is that it should be used sparingly. True singleton objects arise very rarely, and the last thing a Singleton should be used for is to replace a global variable[4]. Comment

Classifying patterns

Design Patterns discusses 23 patterns, classified under three purposes (all of which revolve around the particular aspect that can vary):

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. Comment

2. **Structural**: designing objects to satisfy particular project constraints. These affect the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections. Comment

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

[3] Andrei Alexandrescu develops a superior Singleton holder in *Modern C++ Design*.

[4] For more information, see the article "Once is Not Enough" by Hyslop and Sutter in the March 2003 issue of *CUJ*.

request, moving through a sequence (as in an iterator), or implementing an algorithm. This chapter contains examples of the Observer and the Visitor patterns. Comment

Design Patterns includes 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. Comment

Features, idioms, patterns

Work is continuing beyond what is in the GoF book, of course; hence, there are more patterns and a more refined process on defining design patterns in general.[5] This is important because it is not easy to identify new patterns or to properly describe them. There has been some confusion in the popular literature on what a design pattern is, for example. Patterns are not trivial nor are they typically represented by features that are built into a programming language. Constructors and destructors, for example, could be called the "guaranteed initialization and cleanup design pattern." These are important and essential constructs, but they're routine language constructs and are not rich enough to be considered a design pattern. Comment

Another non-example comes from various forms of aggregation. Aggregation is a completely fundamental principle in object-oriented programming: you make objects out of other objects. Yet sometimes this idea is erroneously classified as a pattern. This is unfortunate because it pollutes the idea of the design pattern and suggests that anything that surprises you the first time you see it should be made into a design pattern. Comment

[5] For up-to-date information, visit http://hillside.net/patterns.

Yet another misguided example is found in the Java language; the designers of the JavaBeans specification decided to refer to a the simple "get/set" naming convention as a design pattern (for example, **getInfo( )** returns an **Info** property and **setInfo( )** changes it). This is just a commonplace naming convention and in no way constitutes a design pattern. Comment

Building complex objects

The class that will be created in the next example models a bicycle that can have a choice of parts, according to its type (mountain bike, touring bike, or racing bike). This is called the **Builder** design pattern. A builder class is associated with each flavor of bicycle, each of which implements the interface specified in the abstract class **BicycleBuilder**. A separate class, **BicycleTechnician**, uses a concrete **BicycleBuilder** object to construct a **Bicycle** object. Comment

```
//: C10:Bicycle.h
// Defines classes to build bicycles
// Illustrates the Builder Design Pattern
#ifndef BICYCLE_H
#define BICYCLE_H

#include <iosfwd>
#include <string>
#include <vector>

class BicyclePart {
public:
  enum BPart {FRAME, WHEEL, SEAT, DERAILEUR,
    HANDLEBAR, SPROCKET, RACK, SHOCK, NPARTS};
  BicyclePart(BPart);
  friend std::ostream&
  operator<<(std::ostream&, const BicyclePart&);
private:
  BPart id;
  static std::string names[NPARTS];
};
```

```
class Bicycle {
public:
  ~Bicycle();
  void addPart(BicyclePart*);
  friend std::ostream&
  operator<<(std::ostream&, const Bicycle&);
private:
  std::vector<BicyclePart*> parts;
};

class BicycleBuilder {
public:
  BicycleBuilder() {
    product = 0;
  }
  void createProduct() {
    product = new Bicycle;
  }
  virtual void buildFrame() = 0;
  virtual void buildWheel() = 0;
  virtual void buildSeat() = 0;
  virtual void buildDerailleur() = 0;
  virtual void buildHandlebar() = 0;
  virtual void buildSprocket() = 0;
  virtual void buildRack() = 0;
  virtual void buildShock() = 0;
  virtual std::string getBikeName() const = 0;
  Bicycle* getProduct() {
    Bicycle* temp = product;
    product = 0;  // relinquish product
    return temp;
  }
protected:
  Bicycle* product;
};

class MountainBikeBuilder : public BicycleBuilder
{
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
```

```
  void buildDeraileur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const {
    return "MountainBike";
  }
};

class TouringBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDeraileur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const {
    return "TouringBike";
  }
};

class RacingBikeBuilder : public BicycleBuilder {
public:
  void buildFrame();
  void buildWheel();
  void buildSeat();
  void buildDeraileur();
  void buildHandlebar();
  void buildSprocket();
  void buildRack();
  void buildShock();
  std::string getBikeName() const {
    return "RacingBike";
  }
};

class BicycleTechnician {
public:
```

```
  BicycleTechnician() {
    builder = 0;
  }
  void setBuilder(BicycleBuilder* b) {
    builder = b;
  }
  void construct();
private:
  BicycleBuilder* builder;
};

#endif ///:~
```

A **Bicycle** holds a vector of pointers to **BicyclePart** representing the parts used to construct the bicycle. To initiate the construction of a bicycle, a technician calls **BicycleBuilder::createproduct( )** on a derived **BicycleBuilder** object. The **BicycleTechnician::construct( )** function calls all the functions in the **BicycleBuilder** interface (since it doesn't know what type of concrete builder it has). The concrete builder classes omit (via empty function bodies) those actions that do not apply to the type of bicycle they build, as you can see in the following implementation file. Comment

```
//: C10:Bicycle.cpp {O}
// Defines classes to build bicycles
// Illustrates the Builder Design Pattern
#include <cassert>
#include <cstddef>
#include <iostream>
#include "Bicycle.h"
#include "../purge.h"
using namespace std;

// BicyclePart implementation
BicyclePart::BicyclePart(BPart bp) {
  id = bp;
}
ostream&
operator<<(ostream& os, const BicyclePart& bp) {
  return os << bp.names[bp.id];
```

```
}
std::string BicyclePart::names[NPARTS] = {
  "Frame", "Wheel", "Seat", "Derailleur",
  "Handlebar", "Sprocket", "Rack", "Shock"};

// Bicycle implementation
Bicycle::~Bicycle() {
  purge(parts);
}
void Bicycle::addPart(BicyclePart* bp) {
  parts.push_back(bp);
}
ostream&
operator<<(ostream& os, const Bicycle& b) {
  os << "{ ";
  for (size_t i = 0; i < b.parts.size(); ++i)
    os << *b.parts[i] << ' ';
  return os << '}';
}

// MountainBikeBuilder implementation
void MountainBikeBuilder::buildFrame() {
  product->addPart(new
BicyclePart(BicyclePart::FRAME));
}
void MountainBikeBuilder::buildWheel() {
  product->addPart(new
BicyclePart(BicyclePart::WHEEL));
}
void MountainBikeBuilder::buildSeat() {
  product->addPart(new
BicyclePart(BicyclePart::SEAT));
}
void MountainBikeBuilder::buildDerailleur() {
  product->addPart(
    new BicyclePart(BicyclePart::DERAILLEUR));
}
void MountainBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void MountainBikeBuilder::buildSprocket() {
```

```
  product->addPart(new
BicyclePart(BicyclePart::SPROCKET));
}
void MountainBikeBuilder::buildRack() {}
void MountainBikeBuilder::buildShock() {
  product->addPart(new
BicyclePart(BicyclePart::SHOCK));
}

// TouringBikeBuilder implementation
void TouringBikeBuilder::buildFrame() {
  product->addPart(new
BicyclePart(BicyclePart::FRAME));
}
void TouringBikeBuilder::buildWheel() {
  product->addPart(new
BicyclePart(BicyclePart::WHEEL));
}
void TouringBikeBuilder::buildSeat() {
  product->addPart(new
BicyclePart(BicyclePart::SEAT));
}
void TouringBikeBuilder::buildDerailleur() {
  product->addPart(new
BicyclePart(BicyclePart::DERAILLEUR));
}
void TouringBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void TouringBikeBuilder::buildSprocket() {
  product->addPart(new
BicyclePart(BicyclePart::SPROCKET));
}
void TouringBikeBuilder::buildRack() {
  product->addPart(new
BicyclePart(BicyclePart::RACK));
}
void TouringBikeBuilder::buildShock() {}

// RacingBikeBuilder implementation
void RacingBikeBuilder::buildFrame() {
```

```
  product->addPart(new
BicyclePart(BicyclePart::FRAME));
}
void RacingBikeBuilder::buildWheel() {
  product->addPart(new
BicyclePart(BicyclePart::WHEEL));
}
void RacingBikeBuilder::buildSeat() {
  product->addPart(new
BicyclePart(BicyclePart::SEAT));
}
void RacingBikeBuilder::buildDeraileur() {}
void RacingBikeBuilder::buildHandlebar() {
  product->addPart(
    new BicyclePart(BicyclePart::HANDLEBAR));
}
void RacingBikeBuilder::buildSprocket() {
  product->addPart(new
BicyclePart(BicyclePart::SPROCKET));
}
void RacingBikeBuilder::buildRack() {}
void RacingBikeBuilder::buildShock() {}

// BicycleTechnician implementation
void BicycleTechnician::construct()
{
  assert(builder);
  builder->createProduct();
  builder->buildFrame();
  builder->buildWheel();
  builder->buildSeat();
  builder->buildDeraileur();
  builder->buildHandlebar();
  builder->buildSprocket();
  builder->buildRack();
  builder->buildShock();
}; ///:~
```

The **Bicycle** stream inserter calls the corresponding inserter for each **BicyclePart**, and that prints out its type name so that you can see what a **Bicycle** contains. The power of this pattern is that it separates the algorithm for assembling parts into a complete product from the parts themselves and allows different algorithms for different products via different implementations of a common interface. Here is a sample program, along with the resulting output, that uses these classes. Comment

```
//: C10:BuildBicycles.cpp
//{L} Bicycle
#include <cstddef>
#include <iostream>
#include <map>
#include <vector>
#include "../purge.h"
#include "Bicycle.h"
using namespace std;

// Constructs a bike via a concrete builder
Bicycle*
buildMeABike(BicycleTechnician& t, BicycleBuilder*
builder) {
  t.setBuilder(builder);
  t.construct();
  Bicycle* b = builder->getProduct();
  cout << "Built a " << builder->getBikeName() <<
endl;
  return b;
}

int main() {
  // Create an order for some bicycles
  map <string, size_t> order;
  order["mountain"] = 2;
  order["touring"] = 1;
  order["racing"] = 3;

  // Build bikes
  vector<Bicycle*> bikes;
  BicycleBuilder* m = new MountainBikeBuilder;
  BicycleBuilder* t = new TouringBikeBuilder;
  BicycleBuilder* r = new RacingBikeBuilder;
  BicycleTechnician tech;
  map<string, size_t>::iterator it =
order.begin();
```

```
  while (it != order.end()) {
    BicycleBuilder* builder;
    if (it->first == "mountain")
      builder = m;
    else if (it->first == "touring")
      builder = t;
    else if (it->first == "racing")
      builder = r;
    for (size_t i = 0; i < it->second; ++i)
      bikes.push_back(buildMeABike(tech,
builder));
    ++it;
  }
  delete m;
  delete t;
  delete r;

  // Display inventory
  for (size_t i = 0; i < bikes.size(); ++i)
    cout << "Bicycle: " << *bikes[i] << endl;
  purge(bikes);
}

/* Output:
Built a MountainBike
Built a MountainBike
Built a RacingBike
Built a RacingBike
Built a RacingBike
Built a TouringBike
Bicycle: { Frame Wheel Seat Derailleur Handlebar
Sprocket Shock }
Bicycle: { Frame Wheel Seat Derailleur Handlebar
Sprocket Shock }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Handlebar Sprocket }
Bicycle: { Frame Wheel Seat Derailleur Handlebar
Sprocket Rack } */ ///:~
```
Comment

Factories: encapsulating object creation

When you discover that you need to add new types to a system, the most sensible first step is to use polymorphism to create a common interface to those new types. This separates the rest of the code in your system from the knowledge of the specific types that you are adding. New types can be added without disturbing existing code … or so it seems. At first it would appear that you need to change the code in such a design only in 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 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. Comment

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, all you must do when you add a new object is modify the factory. This design is a variation of the pattern commonly known as *Factory Method*. Since every object-oriented program creates objects, and since it's likely you will extend your program by adding new types, factories may be the most useful of all design patterns. Comment

As an example, let's revisit the **Shape** system. One approach to implementing a factory is to define a **static** method in the base class:

```
//: C10:ShapeFactory1.cpp
#include <iostream>
#include <stdexcept>
```

```
#include <string>
#include <vector>
#include "../purge.h"
using namespace std;

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
      : logic_error("Cannot create type " + type)
    {}
  };
  static Shape* factory(const 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(const 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(size_t i = 0; i < shapes.size(); i++) {
    shapes[i]->draw();
    shapes[i]->erase();
  }
  purge(shapes);
} ///:~
```

The **factory( )** function 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 will not be a hard-coded array as in the previous example.) Comment

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.**) Comment

Polymorphic factories

The **static factory( )** method in the previous example forces all the creation operations to be focused in one spot, so that's the only place you need to change the code. This is certainly a reasonable solution, as it nicely encapsulates the process of creating objects. However, *Design Patterns* emphasizes that the reason for the Factory Method pattern is so that different types of factories can be derived from the basic factory. Factory Method is in fact a special type of polymorphic factory. However, *Design Patterns* 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: Comment

```
//: C10:ShapeFactory2.cpp
// Polymorphic factory methods
#include <iostream>
#include <map>
#include <stdexcept>
#include <string>
#include <vector>
#include "../purge.h"
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 ShapeFactoryInitializer;
  class BadShapeCreation : public logic_error {
  public:
    BadShapeCreation(string type)
      : logic_error("Cannot create type " + type)
```

```
    {}
  };
  static Shape*
  createShape(const 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"; }
private:
  friend class ShapeFactoryInitializer;
  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
    friend class ShapeFactoryInitializer;
  };
};

class Square : public Shape {
  Square() {}
public:
  void draw() { cout << "Square::draw\n"; }
  void erase() { cout << "Square::erase\n"; }
  ~Square() { cout << "Square::~Square\n"; }
private:
  friend class ShapeFactoryInitializer;
  class Factory;
```

```
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Square; }
    friend class ShapeFactoryInitializer;
  };
};

// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
  static ShapeFactoryInitializer si;
  ShapeFactoryInitializer() {
    ShapeFactory::factories["Circle"] =
      new Circle::Factory;
    ShapeFactory::factories["Square"] =
      new Square::Factory;
  }
};

// Static member definition:
ShapeFactoryInitializer
  ShapeFactoryInitializer::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(size_t 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 member function, which means it cannot be called directly but can be overridden. The subclasses of **Shape** must each create their own subclasses of **ShapeFactory** and override the **create( )** method to create an object of their own type. These factories are private, so that they are only accessible from the main Factory Method. This way, all client programmers are forced to go through the Factory Method in order to create objects. Comment

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 in which 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. Comment

Notice that the **ShapeFactory** must be initialized by loading its **map** with factory objects, which takes place in the singleton **ShapeFactoryInitializer**. So to add a new type to this design you must inherit the type, create a factory, and modify **ShapeFactoryInitializer** 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. Comment

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 when you create the factory object, you decide how all the objects created by that factory will be used. The example 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, a button, a slider, and so on, 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. Comment

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: Comment

```
//: C10: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();
}
/* Output:
Kitty has encountered a Puzzle
KungFuGuy now battles against a NastyWeapon */
///:~
```

In this environment, **Player** objects interact with **Obstacle** objects, but the types of players and obstacles depend on thef game. 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 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. Comment

This example also illustrates *double dispatching*, 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." Comment

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**. The **draw( )** function should be a virtual function, a message to the **Shape** that it should draw itself appropriately, depending on whether it is a circle, a square, a line, and so on. However, this doesn't work inside the constructor, virtual functions resolve to the "local" function bodies when called in constructors. Comment

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. Comment

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. Comment

Coplien[6] 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, creates an object of that specific type (on the heap), and then 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: Comment

```
//: C10:VirtualConstructor.cpp
#include <iostream>
#include <string>
#include <exception>
#include <vector>
using namespace std;

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
```

[6] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

```
    }
    cout << "delete s: ";
    delete s; // The polymorphic deletion
  }
  class BadShapeCreation : public exception {
    string reason;
  public:
    BadShapeCreation(string type) {
      reason = "Cannot create type " + type;
    }
    ~BadShapeCreation() throw() {}
    const char *what() const throw() {
      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. Comment

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. Comment

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 can use other information—for example, in a parser the output of the scanner can be handed to the virtual constructor, which then uses that information to determine which token to create. Comment

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( )**. Comment

The default constructor in this scheme has at least one important chore—it must set the value of the **s** pointer to zero. This may sound 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). Comment

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). Comment

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. Comment

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 functions through the pointer to the "letter," the system operates properly. Comment

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**. Comment

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.
Comment

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 earlier. 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.
Comment

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( )**. Comment

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. Comment

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. Comment

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 in which **s** is set to zero. Comment

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. Comment

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 another object's change of state, but is also the basis of event management. Comment

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. Comment

Two types of objects are 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**. Comment

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. Comment

You can implement the observer pattern in a number of ways, 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: Comment

```
//: C10: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 want 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. Comment

The **Observer** object 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 its 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. Comment

The "observed object" that decides when and how to do the updating will be called the **Observable**:

```
//: C10: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 a way 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 this book, there is no support for multithreading in the standard C++ libraries, so this design would need to be modified in a multithreaded environment. Comment

The **Observable** object 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 notify the **Observer**s only 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. Comment

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. Comment

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 that 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. It then moves through the **set** and calls back to the **update( )** member function of each **Observer**. Comment

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 the observers will, in fact, get notified. *Where* you call **setChanged( )** depends on the logic of your program. Comment

Now we encounter a dilemma. Objects that are being observed may have more than one such item of interest. For example, if you're dealing with a 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 these events to different observers, each of which is interested in a different type of event. Comment

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." Comment

The "interface" idiom

The "inner class" idiom

Here's a situation in which 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 we've lifted from Java, which takes C++'s nested class one step further. Java has a built-in feature called an *inner class*, which is like a nested class in C++, but it has access to the nonstatic data of its containing class by implicitly using the "this" pointer of the class object it was created within.[7] Comment

To implement the inner class idiom in C++, we must obtain and use a pointer to the containing object explicitly. Here's an example:

```
//: C10: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) {
```

[7] There is some similarity between inner classes and *subroutine closures*, which save the reference environment of a function call so it can be reproduced later.

```
  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. Comment

The **Outer** constructor contains some private data (**name**), and it wants 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; you'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. Comment

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. Comment

Just to push a point, we'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. Comment

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:

```
//: C10:ObservedFlower.cpp
// Demonstration of "observer" pattern
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include "Observable.h"
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. The **OpenNotifier** and **CloseNotifier** events 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** we were 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. Comment

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. Comment

In **main( )**, you can see one of the primary benefits of the Observer pattern: the ability to change behavior at runtime by dynamically registering and unregistering **Observer**s with **Observable**s. Comment

If you study the previous code, 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. Comment

Multiple dispatching

When dealing with multiple types that 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**, and so, 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**, how can you get them to interact properly? Comment

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. Comment

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, you'll need a virtual call in each hierarchy. Comment

Here's an example of multiple dispatching:

```
//: C10:PaperScissorsRock.cpp
// Demonstration of multiple dispatching
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iterator>
#include <vector>
#include "../purge.h"
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);
} ///:~
```

Multiple dispatching with Visitor

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? Comment

The design pattern that solves this kind of problem is called a "visitor" (the final one in *Design Patterns*), and it builds on the double-dispatching scheme shown in the previous section. Comment

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 on the primary type. The objects of the primary type simply "accept" the visitor and then call the visitor's dynamically bound member function. Comment

```
//: C10:BeeAndFlowers.cpp
// Demonstration of "visitor" pattern
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>
#include <vector>
#include "../purge.h"
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);
} ///:~
```

Exercises

1. Using **SingletonPattern.cpp** as a starting point, create a class that manages a fixed number of its own objects. Assume the objects are database connections and you only have a license to use a fixed quantity of these at any one time.
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, and 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. Explain how **AbstractFactory.cpp** demonstrates *Double Dispatching* and the *Factory Method*.
5. Modify **ShapeFactory2.cpp** so that it uses an *Abstract Factory* to create different sets of shapes (for example, one particular type of factory object creates “thick shapes,” another creates “thin shapes,” but each factory object can create all the shapes: circles, squares, triangles, and so on).

6. 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.
7. Modify the example in exercise 6 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."

11: Concurrent programming

A: Recommended reading

[Note that some or all of these were listed in the first edition, so I think most might be replaced with new entries (but you might want to check to make sure).]Comment

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. Comment

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.Comment

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.Comment

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.Comment

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.Comment

Large Scale C++ (?) by John Lakos.Comment

C++ Gems, Stan Lippman, editor. SIGS publications.Comment

The Design & Evolution of C++, by Bjarne StroustrupComment

Bruce's books

Not all of these are currently available.Comment

Computer Interfacing with Pascal & C (Self-published via the Eisys imprint; only available via the Web site)Comment

Using C++Comment

C++ Inside & OutComment

Thinking in C++, 1st editionComment

Black Belt C++, the Master's Collection (edited by Bruce Eckel) (out of print).Comment

Thinking in Java, 2nd editionComment

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.Comment

Effective C++ and More Effective C++, by Scott Meyers.Comment

Ruminations on C++ by Koenig & Moo.Comment

The STL

Design Patterns

Comment

Comment

B: Etc

This appendix contains files that are required to build the files in Volume 2.

```
//: :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) {
    fputs(msg, stderr);
    fputs("\n", stderr);
    exit(EXIT_FAILURE);
  }
}

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);
     fputs("\n", stderr);
     exit(EXIT_FAILURE);
   }
}

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);
    fputs("\n", stderr);
    exit(EXIT_FAILURE);
  }
}

inline void assure(std::ifstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr,
      "Could not open file %s\n", filename);
    exit(EXIT_FAILURE);
  }
}

inline void assure(std::ofstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr,
      "Could not open file %s\n", filename);
    exit(EXIT_FAILURE);
  }
}
// Do we need this???
inline void assure(std::fstream& in,
  const char* filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr,
      "Could not open file %s\n", filename);
    exit(EXIT_FAILURE);
  }
}
#endif // REQUIRE_H ///:~
```

From Volume 1, Chapter 9:Comment

```
//: C0B:Stack4.h
// With inlines
```

```
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack(){ head = 0; }
  ~Stack(){
    require(head == 0, "Stack not empty");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() { return head->data; }
  void* pop(){
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

Comment

```
//: C0B:Dummy.cpp
// To give the makefile at least one target
// for this directory
int main() {} ///:~
```

Comment

The **Date** class files:

```
//: C02:Date.h
```

```
#ifndef DATE_H
#define DATE_H
#include <string>
#include <stdexcept>
#include <iosfwd>

class Date {
public:
  // A class for date calculations
  struct Duration {
    int years, months, days;
    Duration(int y, int m, int d)
    : years(y), months(m) ,days(d) {}
  };
  // An exception class
  struct DateError : public std::logic_error {
   DateError(const std::string& msg = "")
     : std::logic_error(msg) {}
  };
  Date();
  Date(int, int, int) throw(DateError);
  Date(const std::string&) throw(DateError);
  int getYear() const;
  int getMonth() const;
  int getDay() const;
  std::string toString() const;
  friend Duration duration(const Date&, const
Date&);
  friend bool operator<(const Date&, const Date&);
  friend bool operator<=(const Date&, const
Date&);
  friend bool operator>(const Date&, const Date&);
  friend bool operator>=(const Date&, const
Date&);
  friend bool operator==(const Date&, const
Date&);
  friend bool operator!=(const Date&, const
Date&);
  friend std::ostream& operator<<(std::ostream&,
                                  const Date&);
  friend std::istream& operator>>(std::istream&,
                                  Date&);
```

```
private:
  int year, month, day;
  int compare(const Date&) const;
  static int daysInPrevMonth(int year, int mon);
};

#endif ///:~
```

```
//: C02:Date.cpp {O}
#include "Date.h"
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <string>
#include <algorithm> // for swap()
#include <ctime>
#include <cassert>
#include <iomanip>
using namespace std;

namespace {
  const int daysInMonth[][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}};
  inline bool isleap(int y) {
    return y%4 == 0 && y%100 != 0 || y%400 == 0;
  }
}

Date::Date() {
  // Get current date
  time_t tval = time(0);
  struct tm *now = localtime(&tval);
  year = now->tm_year + 1900;
  month = now->tm_mon + 1;
  day = now->tm_mday;
}

Date::Date(int yr, int mon, int dy)
throw(Date::DateError) {
  if (!(1 <= mon && mon <= 12))
    throw DateError("Bad month in Date ctor");
```

```
  if (!(1 <= dy && dy <=
daysInMonth[isleap(year)][mon]))
    throw DateError("Bad day in Date ctor");
  year = yr;
  month = mon;
  day = dy;
}

Date::Date(const std::string& s)
throw(Date::DateError) {
  // Assume YYYYMMDD format
  if (!(s.size() == 8))
    throw DateError("Bad string in Date ctor");
  for(int n = 8; --n >= 0;)
    if (!isdigit(s[n]))
      throw DateError("Bad string in Date ctor");
  string buf = s.substr(0, 4);
  year = atoi(buf.c_str());
  buf = s.substr(4, 2);
  month = atoi(buf.c_str());
  buf = s.substr(6, 2);
  day = atoi(buf.c_str());
  if (!(1 <= month && month <= 12))
    throw DateError("Bad month in Date ctor");
  if (!(1 <= day && day <=
    daysInMonth[isleap(year)][month]))
    throw DateError("Bad day in Date ctor");
}

int Date::getYear() const {
  return year;
}

int Date::getMonth() const {
  return month;
}

int Date::getDay() const {
  return day;
}

string Date::toString() const {
```

```
  ostringstream os;
  os.fill('0');
  os << setw(4) << year
     << setw(2) << month
     << setw(2) << day;
  return os.str();
}

int Date::compare(const Date& d2) const {
  int result = year - d2.year;
  if (result == 0) {
    result = month - d2.month;
    if (result == 0)
      result = day - d2.day;
  }
  return result;
}

int Date::daysInPrevMonth(int year, int month) {
  if (month == 1) {
    --year;
    month = 12;
  }
  else
    --month;
  return daysInMonth[isleap(year)][month];
}

bool operator<(const Date& d1, const Date& d2) {
  return d1.compare(d2) < 0;
}
bool operator<=(const Date& d1, const Date& d2) {
  return d1 < d2 || d1 == d2;
}
bool operator>(const Date& d1, const Date& d2) {
  return !(d1 < d2) && !(d1 == d2);
}
bool operator>=(const Date& d1, const Date& d2) {
  return !(d1 < d2);
}
bool operator==(const Date& d1, const Date& d2) {
  return d1.compare(d2) == 0;
```

```
}
bool operator!=(const Date& d1, const Date& d2) {
  return !(d1 == d2);
}

Date::Duration
duration(const Date& date1, const Date& date2) {
  int y1 = date1.year;
  int y2 = date2.year;
  int m1 = date1.month;
  int m2 = date2.month;
  int d1 = date1.day;
  int d2 = date2.day;

  // Compute the compare
  int order = date1.compare(date2);
  if (order == 0)
    return Date::Duration(0,0,0);
  else if (order > 0) {
    // Make date1 precede date2 locally
    using std::swap;
    swap(y1, y2);
    swap(m1, m2);
    swap(d1, d2);
  }

  int years = y2 - y1;
  int months = m2 - m1;
  int days = d2 - d1;
  assert(years > 0 ||
     years == 0 && months > 0 ||
     years == 0 && months == 0 && days > 0);

  // Do the obvious corrections (must adjust days
  // before months!) - This is a loop in case the
  // previous month is February, and days < -28.
  int lastMonth = m2;
  int lastYear = y2;
  while (days < 0) {
    // Borrow from month
    assert(months > 0);
    days += Date::daysInPrevMonth(
```

```
      lastYear, lastMonth--);
    --months;
  }

  if (months < 0) {
    // Borrow from year
    assert(years > 0);
    months += 12;
    --years;
  }
  return Date::Duration(years, months, days);
}

ostream& operator<<(ostream& os, const Date& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.getMonth() << ‘-‘
     << setw(2) << d.getDay() << ‘-‘
     << setw(4) << setfill(fillc) << d.getYear();
  return os;
}

istream& operator>>(istream& is, Date& d) {
  is >> d.month;
  char dash;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.day;
  is >> dash;
  if (dash != '-')
    is.setstate(ios::failbit);
  is >> d.year;
  return is;
} ///:~
```

The file test.txt used in Chapter 6:

```
//: C06:Test.txt
f a f d A  G f d F a A F h f A d f f a a
///:~
```

Index