C++ Cookbook

By Jeff Cogswell, Christopher Diggins,
Ryan Stephens, Jonathan Turkanis

.....

Publisher: **O'Reilly**
Pub Date: **November 2005**
ISBN: **0-596-00761-2**
Pages: **592**

Table of Contents | Index

Overview

Despite its highly adaptable and flexible nature, C++ is also one of the more complex programming languages to learn. Once mastered, however, it can help you organize and process information with amazing efficiency and quickness.

The C++ Cookbook will make your path to mastery much shorter. This practical, problem-solving guide is ideal if you're an engineer, programmer, or researcher writing an application for one of the legions of platforms on which C++ runs. The algorithms provided in C++ Cookbook will jump-start your development by giving you some basic building blocks that you don't have to develop on your own.

Less a tutorial than a problem-solver, the book addresses many of the most common problems you're likely encounter--whether you've been programming in C++ for years or you're relatively new to the language. Here are just some of the time-consuming tasks this book contains practical solutions for:

- Reading the contents of a directory
- Creating a singleton class
- Date and time parsing/arithmetic
- String and text manipulation
- Working with files
- Parsing XML
- Using the standard containers

Typical of O'Reilly's "Cookbook" series, C++ Cookbook is written in a straightforward format, featuring recipes that contain problem statements and code solutions, and apply not to hypothetical situations, but those that you're likely to encounter. A detailed explanation then follows each recipe in order to show you how and why the solution works. This question-solution-discussion format is a proven teaching method, as any fan of the "Cookbook" series can attest to. This book will move quickly to the top of your list of essential C++ references.

NEXT

C++ Cookbook
By Jeff Cogswell, Christopher Diggins,
Ryan Stephens, Jonathan Turkanis
.....
Publisher: **O'Reilly**
Pub Date: **November 2005**
ISBN: **0-596-00761-2**
Pages: **592**

PREV NEXT

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Preface

C++ runs on virtually every platform and in an infinite number of applications. If you bought or might buy this book, you are probably an engineer or researcher writing one of these applications. But regardless of what you are writing and what platform you are targeting, odds are that you will be re-solving many of the same problems that other C++ programmers have been solving for years. What we have done in this book is solve many of these common problems and explain each of the solutions.

Whether you have been programming in C++ for years or are relatively new to the language, you are probably familiar with the things you have rewrite on each new project: Date and time parsing/arithmetic, manipulating string and text, working with files, parsing XML, using the standard containers, and so on. These are the kinds of problems this book contains solutions for. In some cases (e.g., date and time arithmetic), the standard library contains very little support. In others (e.g., string manipulation) the standard library contains functionally rich classes, but it can't do everything and some very common tasks are cumbersome.

The format is straightforward. Each recipe has a problem statement and a code solution, and most have a discussion that follows. We have tried to be pragmatic and solve the problems at hand without digressing too far, but in many cases there are related topics that are so useful (or just cool) that we have to provide a page or two of explanation.

This is a book about solving common problems with C++, but not a book about learning C++. We assume that you have at least a basic knowledge of C++ and object-oriented programming. In particular, it will be helpful if you have at least some familiarity with:

- C++ inheritance and virtual functions
- The standard library
- Components of the Standard Template Library (containers, iterators, and algorithms)
- Templates

These are not strict prerequisites for reading this book, but having at least a basic knowledge of them will help.

About the Examples

In crafting our code examples, we strove for simplicity, portability, and performance. The design for each solution followed a similar path: use standard C++ (language or library) if possible; if not, use a de facto standard as the replacement. For example, many of the recipes that deal with strings use the standard string class, and most of the mathematical and scientific recipes use standard numeric types, containers, and templates. The standard library has strong support for these areas, so standard facilities are a perfect fit. By comparison, however, C++ has little or no standardized support for multithreading or XML parsing. Thus, we used the multithreading support provided in the Boost Threads library and the XML parsing functionality provided by the Xerces parser.

Often, there are many ways to do the same thing in C++, which gives developers flexibility, but also invites some controversy. Most of the examples illustrate the best general solution we could come up with, but that doesn't mean that it's the best solution ever. If there are alternative solutions that are better in some ways and not as good in others (maybe the solution that uses the standard library is awkward or unintuitive; in this case, we may provide an alternative that uses Boost), we present the alternative anyway to give you some insight into the various solutions that are available.

Lots of the examples use templates. If you don't have much experience writing templates, you should get some soon. There is very little introductory material on templates in this book, except for two recipes in Chapter 8: Recipe 8.11 and Recipe 8.12. Most of the interesting developments in C++ are in the areas of template metaprogramming and policy-based design.

At the time of this writing, there is a lot of movement in the C++ community. The first technical report (called TR1) is more or less stable. It is a standardized list of features that will be eventually added to the next version of the C++ standard. It is not required that standard library implementations support it, but many vendors have already begun implementing TR1 and you can expect to see it appearing in shipped compilers soon. Many of the libraries in TR1 first appeared in the Boost project.

We use libraries from Boost a lot. Boost is a set of open source, peer-reviewed, portable libraries that fill in many of the gaps in the standard library. The current version as of this writing is 1.32, and 1.33 should be out any time now. We provide many pointers to specific Boost libraries in the examples. For more information on Boost in general, check out the project web site at www.boost.org.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, Unix utilities, commands, and command-line parameters.

<...>

Angle-brackets surround elements that you need to specify in commands and command-line parameters when those elements appear inline, in italics.

Constant width

Indicates code or fragments thereof. For example, class names, method names, and the like are rendered in constant width whenever they appear in the text.

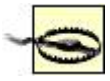
`Constant width bold`

Shows user-input in mixed, input/output examples.

Constant width italic

Indicates user-specified items in syntax examples.

Indicates a tip, suggestion, or general note.



Indicates a warning or caution.

Using Code Examples

This book is designed to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "C++ Cookbook by D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, and Jeff Cogswell. Copyright 2006 O'Reilly Media, Inc., 0-596-00761-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

PREV NEXT

Comments and Questions

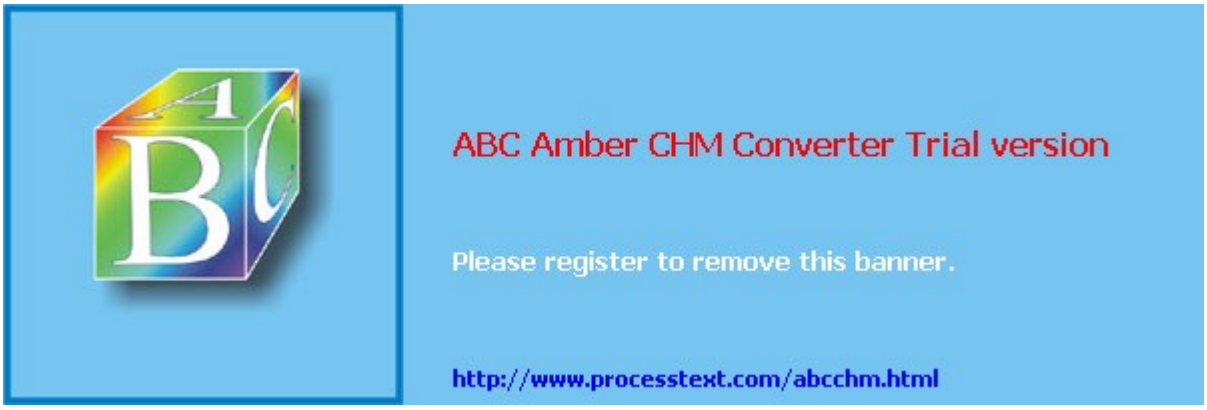
Please address comments and questions concerning this book to the publisher:
O'Reilly Media, Inc.1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:
http://www.oreilly.com/catalog/cpluspluscbk

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

PREV NEXT



PREV NEXT

Safari Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

PREV NEXT

Acknowledgments

From D. Ryan Stephens

The most important people I have to thank are my wife, Daphne, and my children, Jesse, Pascal, and Chloe. Writing a book is hard work, but above all it is time-consuming work, and my family has been supportive and has tolerated my late nights in the office in the best possible way.

I also have to thank the technical reviewers, who made this book better than it otherwise would have been. As with so many things, it is always helpful to have a second, third, and fourth set of eyes look over something for clarity and correctness. Many thanks to Dan Saks, Uwe Schnitker, and David Theese.

Finally, I have to thank my editor, Jonathan Gennick, for his advice, which was mostly grammatical, frequently stylistic, occasionally psychologically supportive, and always good.

From Christopher Diggins

I wish to thank Kris Unger, Jonathan Turkanis, Jonathan Gennick, and Ryan Stephens for their helpful suggestions and critiques, and making me a better writer for it. A very special thanks to my wife Mélanie Charbonneau for brightening my life.

From Jonathan Turkanis

Because my chapters touched on so many different commerical products and open source projectsand because I had so many questions about each of themI have an unusually large number of people to thank.

Let me first thank Ron Liechty, Howard Hinnant, and the engineers at Metrowerks for answering every conceivable question and for providing me with several versions of CodeWarrior.

I'd also like to thank the Boost.Build developers, especially Vladimir Prus, Rene Rivera, and David Abrahams, not just for answering my questions but also for putting together the Boost build system, which was the single most important source of information for Chapter 1.

Thanks also to Walter Bright at Digital Mars; Greg Comeau at Comeau Computing; P. J. Plauger at Dinkumware; Colin Laplace at Bloodshed Software; Ed Mulroy and Pavel Vozenilek at the borland.public.* newsgroups; Arnaud Debaene and Igor Tandetnik at microsoft.public.vc.languages; Earnie Boyd, Greg Chicares, Adib Taraben, John Vandenberg, and Lennart Borgman at the MinGW/MSYS mailing list; Christopher Faylor, Larry Hall, Igor Pechtchanski, Joshua Daniel Franklin, and Dave Korn at the Cygwin list; Mike Stump and Geoffrey Keating at the GCC developers list; Mark Goodhand at DecisionSoft; and David N. Bertoni at apache.org.

I'm also indebted to Robert Mecklenburg, whose book Managing Projects with GNU make, Third Edition (O'Reilly) provided the foundation for my treatment of GNU make.

In addition, Vladimir Prus, Matthew Wilson, Ryan Stephens, and Christopher Diggins provided detailed criticism of early drafts of the manuscript.

Finally, I must thank my editor, Jonathan Gennick, my wife, Jennifer, and my Grandfather, Louis S. Goodman, who taught me how to write.

PREV NEXT


ABC Amber CHM Converter Trial version
Please register to remove this banner.
http://www.processtext.com/abcchm.html

Chapter 1. Building C++ Applications

Introduction to Building

This chapter contains recipes for transforming C++ source code into executable programs and libraries. By working through these recipes, you'll learn about the basic tools used to build C++ applications, the various types of binary files involved in the build process, and the systems that have been developed to make building C++ applications manageable.

If you look at the titles of the recipes in this chapter, you might get the impression that I solve the same problems over and over again. You'd be right. That's because there are many ways to build C++ applications, and while I can't cover them all, I try to cover some of the most important methods. In the first dozen or so recipes, I show how to accomplish three fundamental tasksbuilding static libraries, building dynamic libraries, and building executablesusing a variety of methods. The recipes are grouped by method: first, I look at building from the command line, then with the Boost build system (Boost.Build), and then with an Integrated Development Environment (IDE), and finally with GNU *make* .

Before you start reading recipes, be sure to read the following introductory sections. I'll explain some basic terminology, provide an overview of the command-line tools, build systems and IDEs covered in the chapter, and introduce the source code examples.

Even if you'll be using a build system or IDE, you should start by reading the recipes on building from the command line: these recipes introduce some essential concepts that you'll need to understand later in this chapter.

Basic Terminology

The three basic tools used to build C++ applications are the *compiler*, the *linker*, and the *archiver* (or *librarian*). A collection of these programs and possibly other tools is called a *toolset*.

The compiler takes C++ source files as input and produces *object files* , which contain a mixture of machine-executable code and symbolic references to functions and data. The archiver takes a collection of object files as input and produces a *static library*, or *archive*, which is simply a collection of object files grouped for convenient use. The linker takes a collection of object files and libraries and *resolves* their symbolic references to produce either an *executable* or *dynamic library* . Roughly speaking, the linker operates by matching each use of a symbol to its definition. When an executable or dynamic library is created, it is said to be *linked*; the libraries used to build the executable or dynamic library are said to be *linked against*.

An executable, or *application*, is simply any program that can be executed by the operating system. A dynamic library, also called a *shared library*, is like an executable except that it can't be run on its own; it consists of a body of machine-executable code that is loaded into memory after an application is started and can be shared by one or more applications. On Windows, dynamic libraries are also called *dynamic link libraries* (DLLs).

The object files and static libraries on which an executable depends are needed only when the executable is built. The dynamic libraries on which an executable depends, however, must be present on a user's system when the executable is run.

Table 1-1 shows the file extensions typically associated with these four basic types of files on Microsoft Windows and Unix. When I mention a file that has a different extension on Windows and Unix, I'll sometimes omit the extension if it's clear from the context.

Recipe 1.1. Obtaining and Installing GCC

Problem

You wish to obtain GCC, the free GNU C/C++ compiler.

Solution

The solution depends on your operating system.

Windows

Install MinGW, Cygwin, or both.

To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named *MinGW-<version>.exe*.

Next, run the installation program. It will ask you to specify where you want to install MinGW. It may also ask you which packages you wish to install; at a minimum, you must install *gcc-core*, *gcc-g++*, *binutils,* and the MinGW runtime, but you may wish to install more. When the installation is complete, you will be able to run *gcc*, *g++*, *ar*, *ranlib*, *dlltool*, and several other GNU tools from the Windows command line. You may wish to add the *bin* subdirectory of your MinGW installation to your PATH environment variable so that you can specify these tools on the command line by their simple names rather than by their full pathnames.

To install Cygwin, go to the Cygwin homepage, www.cygwin.com, and follow the link InstallCygwin Now to download the Cygwin installation program. Next, run the installation program. It will ask you to make a series of choices, such as where Cygwin should be installed.

I'm explaining the Cygwin installation process in detail because it can be a bit complicated, depending on what you want to install. The process may have changed by the time you read this, but if it has, it will probably have been made easier.

The most important choice you must make is the selection of packages. If you have enough disk space and a high-speed Internet connection, I recommend that you install all of the packages. To do this, click once on the word Default next to the word All at the top of the hierarchical display of packages. After a possibly long pause, the word Default should change to Install.

If you are short on disk space, or if you have a slow Internet connection, you can choose a smaller selection of packages. To select just the development tools, click once on the word Default next to the word Devel. After a possibly long pause, the word Default should change to Install. For an even smaller collection of packages, expand the list of development packages by clicking on the + icon next to the word Devel. Select the packages *gcc-core*, *gcc-g++*, and *make* by clicking on the word Skip, opposite each package, causing Skip to change to Install.

When you are done selecting packages, press *Finish*. When the installation program completes, the Cygwin installation directory should contain a file named *cygwin.bat* . Running this script will display the Cygwin shell, a command-line environment from which you can run *gcc*, *g++*, *ar*, *ranlib*, *dlltool*, *make*, and any other utilities you chose to install. The installation process adds the *bin* subdirectory of the

PREV NEXT


ABC Amber CHM Converter Trial version
Please register to remove this banner.
http://www.processtext.com/abcchm.html

Recipe 1.2. Building a Simple "Hello, World" Application from the Command Line

Problem

You want to build a simple "Hello, World" program, such as that in Example 1-4.

Example 1-4. A simple "Hello, World" program

hello.cpp

```
#include <iostream>

int main( )
{
    std::cout << "Hello, World!\n";
}
```

Solution

Follow these steps:

1.

1. Set any environment variables required by your toolset.
2.

2. Enter a command telling your compiler to compile and link your program.

Scripts for setting environment variables are listed in Table 1-5; these scripts are located in the same directory as your command-line tools (Table 1-3). If your toolset does not appear in Table 1-5, you can skip the first step. Otherwise, run the appropriate script from the command line, if you are using Windows, or source the script, if you are using Unix.

Table 1-5. Scripts for setting environment variables required by your command-line tools

Toolset	Script
Visual C++	*vcvars32.bat*
Intel (Windows)	*iclvars.bat*[2]
Intel (Linux)	*iccvars.sh* or *iccvars.csh*
Metrowerks (Mac OS X)	*mwvars.sh* or *mwvars.csh*[3]
Metrowerks (Windows)	*cwenv.bat*
Comeau	Same as the backend toolset

[2] With earlier version of the Intel compiler, this script was named *iccvars.bat*.

Recipe 1.3. Building a Static Library from the Command Line

Problem

You wish to use your command-line tools to build a static library from a collection of C++ source files, such as those listed in Example 1-1.

Solution

First, use your compiler to compile the source files into object files. If your source files include headers located in other directories, you may need to use the *-I* option to instruct your compiler where to search for headers; for more information, see Recipe 1.5. Second, use your archiver to combine the object files into a static library.

To compile each of the three source files from Example 1-1, use the command lines listed in Table 1-8, modifying the names of the input and output files as needed. To combine the resulting object files into a static library, use the commands listed in Table 1-10.

Table 1-10. Commands for creating the archive libjohnpaul.lib or libjohnpaul.a

Toolset	Command line
GCC (Unix)Intel (Linux)Comeau (Unix)	*ar ru libjohnpaul.a john.o paul.o johnpaul.oranlib libjohnpaul.a*
GCC (Windows)	*ar ru libjohnpaul.a john.o paul.o johnpaul.o*
Visual C++Comeau (with Visual C++)	*lib -nologo -out:libjohnpaul.lib john.obj paul.obj johnpaul.obj*
Intel (Windows)	*xilib -nologo /out:libjohnpaul.lib john.obj paul.obj johnpaul.obj*
Metrowerks (Windows)	*mwld -library -o libjohnpaul.lib john.obj paul.obj johnpaul.obj*
Metrowerks (Mac OS X)	*mwld -library -o libjohnpaul.a john.o paul.o johnpaul.o*
Borland	*tlib libjohnpaul.lib /u /a /C +john +paul +johnpaul*
Digital Mars	*lib -c -n libjohnpaul.lib john.obj paul.obj johnpaul.obj*

For example, to compile *john.cpp*, *paul.cpp*, and *johnpaul.cpp* into object files using GCC, change to

Recipe 1.4. Building a Dynamic Library from the Command Line

Problem

You wish to use your command-line tools to build a dynamic library from a collection of C++ source files, such as those listed in Example 1-2.

Solution

Follow these steps:

1.

1. Use your compiler to compile the source files into object files. If you're using Windows, use the *-D* option to define any macros necessary to ensure that your dynamic library's symbols will be exported. For example, to build the dynamic library in Example 1-2, you need to define the macro GEORGERINGO_DLL. If you're building a third-party library, the installation instructions should tell you what macros to define.

2.

2. Use your linker to create a dynamic library from the object files created in step 1.

If your dynamic library depends on other libraries, you'll need to tell the compiler where to search for the library headers, and to tell the linker the names of the other libraries and where to find them. This is discussed in detail in Recipe 1.5.

The basic commands for performing the first step are given Table 1-8; you'll need to modify the names of the input and output files appropriately. The commands for performing the second step are given in Table 1-11. If you're using a toolset that comes with static and dynamic variants of its runtime libraries, direct the compiler and linker to use a dynamically linked runtime, as described in Recipe 1.23.

Table 1-11. Commands for creating the dynamic library libgeorgeringo.so, libgeorgeringo.dll, or libgeorgeringo.dylib

Toolset	Command line
GCC	*g++ -shared -fPIC -o libgeorgeringo.so george.o ringo.o georgeringo.o*
GCC (Mac OS X)	*g++ -dynamiclib -fPIC -o libgeorgeringo.dylib george.o ringo.o georgeringo.o*
GCC (Cygwin)	*g++ -shared -o libgeorgeringo.dll -Wl,out-implib,libgeorgeringo.dll.a-W1,export-all-symbols -Wl,enable-auto-image-base george.o ringo.o georgeringo.o*

Recipe 1.5. Building a Complex Application from the Command Line

Problem

You wish to use your command-line tools to build an executable that depends on several static and dynamic libraries.

Solution

Start by building the static and dynamic libraries on which your application depends. Follow the instructions distributed with the libraries, if they are from a third party; otherwise, build them as described in Recipe 1.3 and Recipe 1.4.

Next, compile your application's *.cpp* files into object files as described in "Building a Simple "Hello, World" Program from the Command Line. You may need to use the *-I* option to tell your compiler where to search for the headers needed by your application, as shown in Table 1-12.

Table 1-12. Specifying directories to search for headers

Toolset	Option
All	*-I<directory>*

Finally, use your linker to produce an executable from the collection of object files and libraries. For each library, you must either provide a full pathname or tell the linker where to search for it.

At each stage of this process, if you are using a toolset which comes with static and dynamic variants of its runtime libraries, and if your program uses at least one dynamic library, you should direct the compiler or linker to use a dynamically linked runtime library, as described in Recipe 1.23.

Table 1-13 presents commands for linking the application *hellobeatles* from Example 1-3. It assumes that:

-
- The current directory is *hellobeatles*.
-
- The static library *libjohnpaul.lib* or *libjohnpaul.a* was created in the directory *johnpaul*.
-
- The dynamic library *georgeringo.dll*, *georgeringo.so*, or *georgeringo.dylib* and its import library, if any, were created in the directory *georgeringo*.

Since Comeau can't build dynamic libraries, as mentioned in Recipe 1.4, the entry for Comeau in Table 1-13 assumes that libgeorgeringo has been built as a static library rather than as a dynamic library. To build libgeorgeringo as a static library, remove the modifier GEORGERINGO_DECL from the declaration of the function georgeringo( ) in Example 1-2.

Recipe 1.6. Installing Boost.Build

Problem

You want to obtain and install Boost.Build.

Solution

Consult the Boost.Build documentation at www.boost.org/boost-build2 or follow these steps:

1.

1. Go to the Boost homepage, www.boost.org, and follow the Download link to Boost's SourceForge download area.
2.

2. Download and unpack either the latest release of the package *boost* or the latest release of the package *boost-build*. The former includes the full collection of Boost libraries, while the latter is a standalone release of Boost.Build. Place the unpacked files in a suitable permanent location.
3.

3. Download and unpack the latest version of the package *boost-jam* for your platform; this package includes a prebuilt *bjam* executable. If the package *boost-jam* is not available for your platform, follow the instructions provided with the package you downloaded in step 2 to build the executable from the source.
4.

4. Copy *bjam* to a location in your PATH environment variable.
5.

5. Permanently set the environment variable BOOST_BUILD_PATH to the Boost. Build root directory. If you downloaded the package *boost* in step 1, the root directory is the subdirectory *tools/build/v2* of your Boost installation; otherwise, it is the directory *boost-build*.
6.

6. Configure Boost.Build for your toolsets and libraries by editing the configuration file *user-config.jam*, located in the Boost.Build root directory. The file *user-config.jam* contains comments explaining how to do this.

Discussion

The most difficult part of using Boost.Build is downloading and installing it. Eventually Boost may provide a graphical installation utility, but for the time being, you must follow the above steps.

The purpose of step five is to help the build tool, *bjam*, find the root directory of the build system. This step is not strictly necessary, however, since there is another way to accomplish the same thing: simply create a file called *boost-build.jam*, with the single line:

```
boost-build boost-build-root ;
```

and place it in the root directory of your project or in any of its parent directories. The second method may be preferable if you wish to distribute Boost.Build with your source code, since it makes the installation process easier for end users.

The sixth step is potentially the most complex, but in practice it is usually simple. If you have just a single

Recipe 1.7. Building a Simple "Hello, World" Application Using Boost.Build

Problem

You want to use Boost.Build to build a simple "Hello, World" program, such as the one in Example 1-4.

Solution

Create a text file named *Jamroot* in the directory where you wish the executable and any accompanying intermediate files to be created. In the file *Jamroot*, invoke two *rules*, as follows. First, invoke the exe rule to declare an executable target, specifying your *.cpp* file as a *source*. Next, invoke the install rule, specifying the executable target name and the location where you want the install directory. Finally, run *bjam* to build your program.

For example, to build an executable *hello* or *hello.exe* from the file *hello.cpp* in Example 1-1, create a file named *Jamroot* with the following content in the directory containing *hello.cpp*, as shown in Example 1-8.

Example 1-8. Jamfile for project hello

```
# jamfile for project hello

exe hello : hello.cpp ;

install dist : hello : <location>. ;
```

Next, change to the directory containing *hello.cpp* and *Jamroot* and enter the following command:

```
> bjam hello
```

This command builds the executable *hello* or *hello.exe* in a subdirectory of the current directory. Finally, enter the command:

```
> bjam dist
```

This command copies the executable to the directory specified by the location property, which in this case is the current directory.

As this book goes to press, the Boost.Build developers are preparing for the official release of Boost.Build version 2. By the time you read this, Version 2 will probably already have been released; if not, you can enable the behavior described in this chapter by passing the command-line option *v2* to bjam. For example, instead of entering **bjam hello**, enter **bjam --v2 hello**.

Discussion

The file *Jamroot* is an example of a *Jamfile*. While a small collection of C++ source files might be managed using a single Jamfile, a large codebase will typically require many Jamfiles, organized hierarchically. Each Jamfile resides in a separate directory and corresponds to a separate *project*. Most Jamfiles are simply named *Jamfile*, but the highest-level Jamfilethe Jamfile that resides in a directory that is an ancestor of the directories containing all the other Jamfilesis named *Jamroot*. The project defined by this highest-level Jamfile is known as the *project root*. Each project except the project root has a *parent project* , defined as the project in the nearest ancestor directory containing a Jamfile.

Recipe 1.8. Building a Static Library Using Boost.Build

Problem

You want to use Boost.Build to build a static library from a collection of C++ source files, such as those listed in Example 1-1.

Solution

Create a *Jamroot* file in the directory where you wish the static library to be created. In the file *Jamroot*, invoke the lib rule to declare a library target, specifying your *.cpp* files as sources and the property <link>static as a requirement. Add a usage requirement of the form <include>*path* to specify the library's include directory, i.e., the directory with respect to which include directives for library headers should be resolved. You may need to add one or more requirements of the form <include>*path* to tell the compiler where to search for included headers. Finally, run *bjam* from the directory containing *Jamroot*, as described in Recipe 1.7.

For example, to build a static library from the source files listed in Example 1-1, your *Jamroot* might look like Example 1-11.

Example 1-11. A Jamfile to build the static library libjohnpaul.lib or libjohnpaul.a

```
# Jamfile for project libjohnpaul

lib libjohnpaul
    : # sources
      john.cpp paul.cpp johnpaul.cpp
    : # requirements
      <link>static
    : # default-build
    : # usage-requirements
      <include>..
    ;
```

To build the library, enter:

```
> bjam libjohnpaul
```

Discussion

The lib rule is used to declare a target representing a static or dynamic library. It takes the same form as the exe rule, as illustrated in Example 1-9. The usage requirement <include>.. frees projects that depend on your library from having to explicitly specify your library's include directory in their requirements. The requirement <link>static specifies that your target should always be built as a static library. If you want the freedom to build a library target either as static or as dynamic, you can omit the requirement <link>static. Whether the library is built as static or dynamic can then be specified on the command line, or in the requirements of a target that depends on the library target. For example, if the requirement <link>static were omitted in Example 1-11, you could build the target libjohnpaul as a static library by entering the command:

```
> bjam libjohnpaul link=static
```

Writing source code for a library that can be built either as static or dynamic is a bit tricky, however, as discussed in Recipe 1.9.

See Also

Recipe 1.3, Recipe 1.11, and Recipe 1.16

Recipe 1.9. Building a Dynamic Library Using Boost.Build

Problem

You wish to use Boost.Build to build a dynamic library from a collection of C++ source files, such as those listed in Example 1-2.

Solution

Create a *Jamroot* file in the directory where you wish the dynamic libraryand the import library, if anyto be created. In the file *Jamroot*, invoke the lib rule to declare a library target, specifying your *.cpp* files as sources and the properties <link>shared as a requirement. Add a usage requirement of the form <include>*path* to specify the library's include directory, i.e., the directory with respect to which include directives for library headers should be resolved. If your source files include headers from other libraries, you may need to add several requirements of the form <include>*path* to tell the compiler where to search for included headers. You may also need to add one or more requirements of the form <define>*symbol* to ensure that your dynamic library's symbols will be exported using _ _declspec(dllexport) on Windows. Finally, run *bjam* from the directory containing *Jamroot*, as described in Recipe 1.7.

For example, to build a dynamic library from the source files listed in Example 1-2, create a file named *Jamroot* in the directory *georgeringo*, as shown in Example 1-12.

Example 1-12. A Jamfile to build the dynamic library georgeringo.so, georgeringo.dll, or georgeringo.dylib

```
# Jamfile for project georgringo

lib libgeorgeringo
    : # sources
      george.cpp ringo.cpp georgeringo.cpp
    : # requirements
      <link>shared
      <define>GEORGERINGO_DLL
    : # default-build
    : # usage-requirements
      <include>..
    ;
```

To build the library, enter:

```
> bjam libgeorgeringo
```

Discussion

As discussed in Recipe 1.8, the lib rule is used to declare a target representing a static or dynamic library. The usage requirement <include>.. frees projects which depend on your library from having to explicitly specify your library's include directory in their requirements. The requirement <link>shared specifies that the target should always be built as a dynamic library. If you want the freedom to build a library target either as static or as dynamic, you can omit the requirement <link>shared and specify this property on the command line, or in the requirements of a target that depends on the library target. Writing a library which can be built as either static or dynamic requires some care, however, because of the preprocessor directives necessary to ensure that symbols are properly exported on Windows. Rewriting Example 1-2 so that it can be built as either static or dynamic makes a good exercise.

See Also

Recipe 1.4, Recipe 1.12, Recipe 1.16, and Recipe 1.19

Recipe 1.10. Building a Complex application Using Boost.Build

Problem

You wish to use Boost.Build to build an executable that depends on several static and dynamic libraries.

Solution

Follow these steps:

1.
1. For each library on which the executable dependsunless it is distributed as a prebuilt binarycreate a Jamfile as described in Recipe 1.8 and Recipe 1.9.
2.
2. Create a *Jamroot* file in the directory where you want the executable to be created.
3.
3. In the file *Jamroot*, invoke the exe rule to declare an executable target. Specify your *.cpp* files and the library targets on which the executable depends as sources. Also, add properties of the form <include>*path* as sources, if necessary, to tell the compiler where to search for library headers.
4.
4. In the file *Jamroot*, invoke the install rule, specifying the properties <install-dependencies>on, <install-type>EXE, and <install-type>SHARED_LIB as requirements.
5.
5. Run *bjam* from the directory containing *Jamroot* as described in Recipe 1.7.

For example, to build an executable from the source files listed in Example 1-3, create a file named *Jamroot* in the directory *hellobeatles* as shown in Example 1-13.

Example 1-13. A Jamfile to build the executable hellobeatles.exe or hellobeatles

```
 # Jamfile for project hellobeatles

exe hellobeatles
    : # sources
      ../johnpaul//libjohnpaul
      ../georgeringo//libgeorgeringo
      hellobeatles.cpp
    ;

install dist
    : # sources
      hellobeatles
    : # requirements
      <install-dependencies>on
      <install-type>EXE
      <install-type>SHARED_LIB
      <location>.
    ;
```

Now enter:

```
 > bjam hellobeatles
```

Recipe 1.11. Building a Static Library with an IDE

Problem

You wish to use your IDE to build a static library from a collection of C++ source files, such as those listed in Example 1-1.

Solution

The basic outline is as follows:

1. Create a new project and specify that you wish to build a static library rather than an executable or a dynamic library.
2. Choose a build configuration (e.g., debug versus release, single-threaded versus multithreaded).
3. Specify the name of your library and the directory in which it should be created.
4. Add your source files to the project.
5. If necessary, specify one or more directories where the compiler should search for included headers. See Recipe 1.13.
6. Build the project.

The steps in this outline vary somewhat depending on the IDE; for example, with some IDEs, several steps are combined into one or the ordering of the steps is different. The second step is covered in detail in Recipe 1.21, Recipe 1.22, and Recipe 1.23. For now, you should use default settings as much as possible.

For example, here's how to build a static library from the source code in Example 1-1 using the Visual C++ IDE.

Select New⟶ Project from the File menu, select Visual C++[9] in the left pane, select Win32 Console Application, and enter *libjohnpaul* as your project's name. From the Win32 Application Wizard go to Application Settings, select Static library, uncheck Precompiled header, and press Finish. You should now have an empty project with two build configurations, Debug and Release, the former being the active configuration.

[9] In versions of Visual C++ prior to Visual C++ 2005, this option was labeled Visual C++ Projects.

Next, display your project's property pages by right-clicking on the project's name in the Solution Explorer and selecting Properties. Go to Configuration Properties⟶ Librarian⟶ General and enter the pathname of your project's output file in the field labeled Output File. The directory portion of the pathname should point to the directory *binaries* which you created at the beginning of this chapter; the file name portion should be *libjohnpaul.lib*.

Finally, use Add Existing Item... from the Project menu to add the source files listed in Example 1-1 to

Recipe 1.12. Building a Dynamic Library with an IDE

Problem

You wish to use your IDE to build a dynamic library from a collection of C++ source files, such as those listed in Example 1-2.

Solution

The basic outline is as follows:

1.
1. Create a new project and specify that you wish to build a dynamic library rather than static library or an executable.
2.
2. Choose a build configuration (e.g., debug versus release, single-threaded versus multithreaded).
3.
3. Specify the name of your library and the directory where it should be created.
4.
4. Add your source files to the project.
5.
5. On Windows, define any macros necessary to ensure that your dynamic library's symbols will be exported using _ _declspec(dllexport).
6.
6. If necessary, specify one or more directories where the compiler should search for included headers. See Recipe 1.13.
7.
7. Build the project.

As with Recipe 1.11, the steps in this outline vary somewhat depending on the IDE. The second step is covered in detail in Recipe 1.21, Recipe 1.22, and Recipe 1.23. For now, you should use default settings wherever possible.

For example, here's how to build a dynamic library from the source code in Example 1-2 using the Visual C++ IDE.

Select New⟶ Project from the File menu, select Visual C++[10] in the left pane, select Win32 Console Application and enter *libgeorgeringo* as your project's name. From the Win32 Application Wizard go to Application Settings, select DLL and Empty Project, and press Finish. You should now have an empty project with two build configurations, Debug and Release, the former being the active configuration.

[10] In versions of Visual C++ prior to Visual C++ 2005, this option was labeled Visual C++ Projects.

Next, display your project's property pages by right-clicking on the project's name in the Solution Explorer and selecting Properties. Go to Configuration Properties⟶ Linker⟶ General and enter the pathname of your project's output file in the field labeled Output File. The directory portion of the

Recipe 1.13. Building a Complex Application with an IDE

Problem

You wish to use your IDE to build an executable that depends on several static and dynamic libraries.

Solution

The basic outline is as follows:

1.
    1. If you are building the dependent libraries from the source, and they don't come with their own IDE projects or makefiles, create projects for them, as described in Recipe 1.11 and Recipe 1.12.
    2.
    2. Create a new project and specify that you wish to build an executable rather than a library.
    3.
    3. Choose a build configuration (e.g., debug versus release, single-threaded versus multithreaded).
    4.
    4. Specify the name of your executable and the directory in which it should be created.
    5.
    5. Add your source files to the project.
    6.
    6. Tell the compiler where to find the headers for the dependent libraries.
    7.
    7. Tell the linker what libraries to use and where to find them.
    8.
    8. If your IDE supports project groups, add all the projects mentioned above to a single project group and specify the dependency relationships between them.
    9.
    9. If your IDE supports project groups, build the project group from step 8. Otherwise, build the projects individually, taking care to build each project before the projects that depend on it.

As with Recipe 1.11 and Recipe 1.12, the steps in this outline vary somewhat depending on the IDE. The third step is covered in detail in Recipes Recipe 1.21, Recipe 1.22, and Recipe 1.23. For now, you should use the default settings wherever possible.

For example, here's how to build an executable from the source code in Example 1-3 using the Visual C++ IDE.

Select New⟶ Project from the File menu, select Visual C++[11] in the left pane, select Win32 Console Application and enter *hellobeatles* as your project's name. From the Win32 Application Wizard go to Application Settings, select Console Application and Empty Project, and press Finish. You should now have an empty project *hellobeatles.vcproj* with two build configurations, Debug and Release, the former being the active configuration. You should also have a solution *hellobeatles.sln* containing the single project *hellobeatles.vcproj*.

Recipe 1.14. Obtaining GNU make

Problem

You want to obtain and install the GNU *make* utility, useful for building libraries and executables from source code.

Solution

The solution depends on your operating system.

Windows

While you can obtain prebuilt binaries for GNU *make* from several locations, to get the most out of GNU *make* it should be installed as part of a Unix-like environment. I recommend using either Cygwin or MSYS, which is a part of the MinGW project.

Cygwin and MinGW are described in Recipe 1.1.

If you installed Cygwin, as described in Recipe 1.1, you already have GNU *make*. To run it from the Cygwin shell, simply run the command *make*.

To install MSYS, begin by installing MinGW, as described in Recipe Recipe 1.1. A future version of the MinGW installer may give you the option of installing MSYS automatically. For now, follow these additional steps.

First, from the MinGW homepage, http://www.mingw.org, go to the MinGW download area and download the latest stable version of the MSYS installation program. The name of the installation program should be *MSYS-<version>.exe*.

Next, run the installation program. You will be asked to specify the location of your MinGW installation and the location where MSYS should be installed. When the installation program completes, the MSYS installation directory should contain a file named *msys.bat*. Running this script will display the MSYS shell, a port of the *bash* shell from which you can run GNU *make* and other mingw programs such as g++, *ar*, *ranlib*, and *dlltool*.

To use MSYS it is not necessary for the *bin* subdirectories of either your MinGW installation or your MSYS installation to be in your PATH environment variable.

Unix

First, check whether GNU *make* is installed on your system by running *make -v* from the command line. If GNU *make* is installed, it should print a message like the following:

```
 GNU Make 3.80
Copyright (C) 2002  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
...
```

Recipe 1.15. Building A Simple "Hello, World" Application with GNU make

Problem

You want to use GNU *make* to build a simple "Hello, World" program, such as that in Example 1-4.

Solution

Before you write your first makefile, you'll need to know a little terminology. A makefile consists of a collection of rules of the form

```
targets: prerequisites
    command-script
```

Here *targets* and *prerequisites* are space-separated strings, and *command-script* consists of zero or more lines of text, each of which begins with a Tab character. Targets and prerequisites are usually files names, but sometimes they are simply formal names for actions for *make* to perform. The command script consists of a sequence of commands to be passed to a shell. Roughly speaking, a rule tells *make* to generate the collection of targets from the collection of prerequisites by executing the command script.

Whitespace in makefiles is significant. Lines containing command scripts must begin with a Tab rather than a Space this is a source of some of the most common beginner errors. In the following examples, lines which begin with a Tab are indicated by an indentation of four characters.

Now you're ready to begin. Create a text file named *makefile* in the directory containing your source file. In this file, declare four targets. Call the first target all, and specify the name of the executable you wish to build as its sole prerequisite. It should have no command script. Give the second target the same name as your executable. Specify your application's source file as its prerequisite, and specify the command line needed to build the executable from the source file as your target's command script. The third target should be called install. It should have no prerequisites, and should have a command script to copy the executable from the directory containing the makefile to the directory where you want it installed. The last target should be called clean. Like install, it should have no prerequisites. Its command script should remove the executable and the intermediate object file from the current directory. The clean and install targets should both be labeled as phonytargets, using the PHONY attribute.

For example, to build an executable from the source code in Example 1-4 using GCC, your makefile might look as shown in Example 1-14.

Example 1-14. Makefile to build the executable hello with GCC

```
 # This is the default target, which will be built when
# you invoke make
.PHONY: all
all: hello

# This rule tells make how to build hello from hello.cpp
hello: hello.cpp
    g++ -o hello hello.cpp

# This rule tells make to copy hello to the binaries subdirectory,
# creating it if necessary
.PHONY: install
install:
```

Recipe 1.16. Building a Static Library with GNU Make

Problem

You want to use GNU make to build a static library from a collection of C++ source files, such as those listed in Example 1-1.

Solution

First, create a makefile in the directory where you want your static library to be created, and declare a phony target all whose single prerequisite is the static library. Next, declare your static library target. Its prerequisites should be the object files that the library will contain, and its command script should be a command line to build the library from the collection of object files, as demonstrated in Recipe 1.3. If you are using GCC or a compiler with similar command-line syntax, customize the implicit patterns rules, if necessary, by modifying one or more of the variables CXX, CXXFLAGS, etc. used in *make's* database of implicit rules, as shown in Recipe 1.15. Otherwise, write a pattern rule telling *make* how to compile *.cpp* files into object files, using the command lines from Table 1-4 and the pattern rule syntax explained in Recipe 1.16. Next, declare targets indicating how each of your library's source files depends on the headers it includes, directly or indirectly. You can write these dependencies by hand or arrange for them to be generated automatically. Finally, add install and clean targets as demonstrated in Recipe 1.15.

For example, to build a static library from the source files listed in Example 1-2 using GCC on Unix, create a makefile in the directory *johnpaul*, as shown in Example 1-20.

Example 1-20. Makefile for libjohnpaul.a using GCC on Unix

```
 # Specify extensions of files to delete when cleaning
CLEANEXTS   = o a

# Specify the target file and the install directory
OUTPUTFILE  = libjohnpaul.a
INSTALLDIR  = ../binaries

# Default target
.PHONY: all
all: $(OUTPUTFILE)

# Build libjohnpaul.a from john.o, paul.o, and johnpaul.o
$(OUTPUTFILE): john.o paul.o johnpaul.o
    ar ru $@ $^
    ranlib $@

# No rule to build john.o, paul.o, and johnpaul.o from .cpp
# files is required; this is handled by make's database of
# implicit rules

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    for file in $(CLEANEXTS); do rm -f *.$$file; done

# Indicate dependencies of .ccp files on .hpp files
john.o: john.hpp
paul.o: paul.hpp
johnpaul.o: john.hpp paul.hpp johnpaul.hpp
```

Recipe 1.17. Building a Dynamic Library with GNU Make

Problem

You wish to use GNU *make* to build a dynamic library from a collection of C++ source files, such as those listed in Example 1-2.

Solution

First, create a makefile in the directory where you want your dynamic library to be created, and declare a phony target all whose single prerequisite is the dynamic library. Next, declare your dynamic library target. Its prerequisites should be the object files from which the library will be built, and its command script should be a command line to build the library from the collection of object files, as demonstrated in Recipe 1.4. If you are using GCC or a compiler with similar command-line syntax, customize the implicit patterns rules, if necessary, by modifying one or more of the variables CXX, CXXFLAGS, etc. used in *make's* database of implicit rules, as shown in Recipe 1.15. Otherwise, write a pattern rule telling *make* how to compile *.cpp* files into object files, using the command lines from Table 1-4 and the pattern rule syntax explained in Recipe 1.16. Finally, add install and clean targets, as demonstrated in Recipe 1.15, and machinery to automatically generate source file dependencies, as demonstrated in Recipe 1.16.

For example, to build a dynamic library from the source files listed Example 1-2 using GCC on Unix, create a makefile in the directory *georgeringo*, as shown in Example 1-22.

Example 1-22. Makefile for libgeorgeringo.so using GCC

```
 # Specify extensions of files to delete when cleaning
CLEANEXTS   = o so

# Specify the source files, the target files,
# and the install directory
SOURCES     = george.cpp ringo.cpp georgeringo.cpp
OUTPUTFILE  = libgeorgeringo.so
INSTALLDIR  = ../binaries

.PHONY: all
all: $(OUTPUTFILE)

# Build libgeorgeringo.so from george.o, ringo.o,
# and georgeringo.o; subst is the search-and-replace
# function demonstrated in Recipe 1.16
$(OUTPUTFILE): $(subst .cpp,.o,$(SOURCES))
    $(CXX) -shared -fPIC $(LDFLAGS) -o $@ $^

.PHONY: install
install:
    mkdir -p $(INSTALLDIR)
    cp -p $(OUTPUTFILE) $(INSTALLDIR)

.PHONY: clean
clean:
    for file in $(CLEANEXTS); do rm -f *.$$file; done

# Generate dependencies of .ccp files on .hpp files
include $(subst .cpp,.d,$(SOURCES))

%.d: %.cpp
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
rm -f $@.$$$$
```

Recipe 1.18. Building a Complex Application with GNU make

Problem

You wish to use GNU *make* to build an executable which depends on several static and dynamic libraries.

Solution

Follow these steps:

1.
   1. Create makefiles for the libraries used by your application, as described in Recipe 1.16 and Recipe 1.17. These makefiles should reside in separate directories.
2.
   2. Create a makefile in yet another directory. This makefile can be used to build your application, but only after the makefiles in step 1 have been executed. Give this makefile a phony target all whose prerequisite is your executable. Declare a target for your executable with prerequisites consisting of the libraries which your application uses, together with the object files to be built from your application's *.cpp* files. Write a command script to build the executable from the collection libraries and object files, as described in Recipe 1.5. If necessary, write a pattern rule to generate object files from *.cpp* files, as shown in Recipe 1.16. Add install and clean targets, as shown in Recipe 1.15, and machinery to automatically generate source file dependencies, as shown in Recipe 1.16.
3.
   3. Create a makefile in a directory which is an ancestor of the directories containing all the other makefiles let's call the new makefile the top-level makefile and the others the subordinate makefiles. Declare a default target all whose prerequisite is the directory containing the makefile created in step 2. Declare a rule whose targets consists of the directories containing the subordinate makefiles, and whose command script invokes *make* in each target directory with a target specified as the value of the variable TARGET. Finally, declare targets specifying the dependencies between the default targets of the subordinate makefiles.

For example, to build an executable from the source files listed in Example 1-3 using GCC on Unix, create a makefile as shown in Example 1-23.

Example 1-23. Makefile for hellobeatles.exe using GCC

```
 # Specify the source files, target files, the build directories,
# and the install directory
SOURCES         = hellobeatles.cpp
OUTPUTFILE     = hellobeatles
LIBJOHNPAUL     = libjohnpaul.a
LIBGEORGERINGO = libgeorgeringo.so
JOHNPAULDIR     = ../johnpaul
GEORGERINGODIR     = ../georgeringo
INSTALLDIR     = ../binaries

#
# Add the parent directory as an include path
#
CPPFLAGS      += -I..

#
# Default target
```

Recipe 1.19. Defining a Macro

Problem

You want to define the preprocessor symbol *name*, assigning it either an unspecified value or the value *value*.

Solution

The compiler options for defining a macro from the command line are shown in Table 1-16. Instructions for defining a macro from your IDE are given in Table 1-17. To define a macro using Boost.Build, simply add a property of the form <define>*name*[=*value*] to your target's requirements, as shown in Table 1-15 and Example 1-12.

Table 1-16. Defining a macro from the command line

Toolset	Option
All	*-Dname[=value]*

Table 1-17. Defining a macro from your IDE

IDE	Configuration
Visual C++	From your project's property pages, go to Configuration Properties ⟶ C/C++ ⟶ Preprocessor and enter *name[=value]* under Preprocessor Definitions, using semicolons to separate multiple entries.
CodeWarrior	From the Target Settings Window, go to Language Settings ⟶ C/C++ Preprocessor and enter: `#define name[ = value]` in the area labeled Prefix Text.
C++Builder	From Project Options, go to Directories/Conditionals and enter *name[=value]* under Preprocessor Definitions, using semicolons to separate multiple entries.
Dev-C++	From Project Options, select Parameters and enter: `-D name[ = value]` under C++ Compiler.

Discussion

Recipe 1.20. Specifying a Command-Line Option from Your IDE

Problem

You want to pass a command-line option to your compiler or linker, but it doesn't correspond to any of the project settings available through your IDE.

Solution

Many IDEs provide a way to pass command-line options directly to the compiler or linker. This is summarized in Table 1-18 and Table 1-19.

Table 1-18. Specifying a compiler option from your IDE

IDE	Configuration
Visual C++	From your project's property pages, go to Configuration Properties ⟶ C/C++ ⟶ Command Line and enter the option under Additional options.
CodeWarrior	n/a
C++Builder	n/a
Dev-C++	From Project Options, select Parameters and enter the option under C++ Compiler.

Table 1-19. Specifying a linker option from your IDE

IDE	Configuration
Visual C++	From your project's property pages, go to Configuration Properties ⟶ Linker ⟶ Command Line and enter the option under Additional options.
Metrowerks	n/a
C++Builder	n/a
Dev-C++	From Project Options, select Parameters and enter the option under Linker.

Discussion

Recipe 1.21. Producing a Debug Build

Problem

You want to build a version of your project that will be easy to debug.

Solution

In general, to produce a debug build, you must;

-
- Disable optimizations
-
- Disable expansion of inline function
-
- Enable generation of debugging information

Table 1-20 presents the compiler and linker options to disable optimization and inlining; Table 1-21 presents the compiler and linker options to enable debugging information.

Table 1-20. Disabling optimization and inlining from the command line

Toolset	Optimization	Inlining
GCC	-O0	-fno-inline[12]
Visual C++Intel (Windows)	-Od	-Ob0
Intel (Linux)	-O0	-Ob0
	-opt off	-inline off
Comeau (Unix)	-O0	--no_inlining
Comeau (Windows)	Same as backend, but using a slash (/) instead of a dash (-)	
Borland	-Od	-vi-
Digital Mars	-o+none -S	-C

[12] It's not necessary to specify this option unless -O3 has also been specified.

Table 1-21. Command-line options for enabling debug information

Recipe 1.22. Producing a Release Build

Problem

You want to produce a small, fast executable or dynamic library for distribution to your customers.

Solution

In general, to produce a release build you must

-
- Enable optimizations
-
- Enable the expansion of inline function
-
- Disable the generation of debugging information

Table 1-26 presents the compiler and linker options to enable optimization and inlining. There are no command-line options for disabling the generation of debugging information: when you build from the command line, debugging information is disabled by default. If you use the GCC toolset, however, you can decrease the size of executables and dynamics libraries by specifying the *-s* option to the linker.

Table 1-26. Compiler options to enable optimization and inlining

Toolset	Optimization	Inlining
GCC	-O3	-finline-functions[14]
Visual C++Intel	-O2	-Ob1
Metrowerks	-opt full	-inline auto -inline level=8
Comeau (Unix)	-O3	
Comeau (Windows)	Same as backend, but using a slash (/) instead of a dash (-)	`inlining`
Borland	-O2	-vi
Digital Mars	-o+time	Enabled by default

[14] This option is enabled automatically when -O3 is specified.

Boost.Build provides a simple mechanism for producing a release build: simply add <variant>release to your target's requirements or use the command-line option *variant=release*, which can be abbreviated simply as *release*.

Recipe 1.23. Specifying a Runtime Library Variant

Problem

Your toolset comes with several variants of its runtime support libraries and you want to instruct your compiler and linker which variant to use.

Solution

Runtime libraries supplied with a given toolset can vary depending on whether they are single- or multithreaded, whether they are static or dynamic, and whether or not they were built with debugging information.

If you are using Boost.Build, these three choices can be specified using the threading, runtime-link, and variant features, described in Table 1-15. For example, to specify a statically linked runtime library, add <runtime-link>static to your target's requirements, or use the command-line option *runtime-link=static*. To specify a multithreaded runtime library, add <threading>multi to your target's requirements, or use the command-line option *threading=multi*.

If you are building from the command line, use the compiler and linker options presented in Tables 1-30 through 1-36. The command-line options and library names for debug and release configurations as generally quite similar; in the following tables, the letters in brackets should be supplied only for debug configurations. The names of the dynamic variants of the runtime libraries are provided in parentheses; these libraries must be available at runtime if dynamic linking is selected.

Table 1-30. Compiler options for runtime library selection using Visual C++ or Intel (Windows)

	Static linking	**Dynamic linking**
Single-threaded	*-ML[d]*[15]	n/a
Multithreaded	*-MT[d]*	*-MD[d]*(*msvcrt[d].dll*, *msvcr80[d].dll*)[16]

[15] Beginning with Visual Studio 2005, currently in beta, the options -ML and -MLd have been deprecated, and single-threaded, statically linked runtime libraries are no longer distributed.

[16] Previous versions of Visual C++ used the DLL's *msvcr71.dll*, *msvcr71d.dll*, *msvcr70.dll*, *msvcr70d.dll*, etc.

Table 1-31. Compiler options for runtime library selection using Metrowerks (Windows)

	Static linking	**Dynamic linking**
Single-threaded	*-runtime ss[d]*	n/a
Multithreaded	*-runtime sm[d]*	*-runtime dm[d]*(*MSL_All-DLL90_x86[_D].dll*)

Recipe 1.24. Enforcing Strict Conformance to the C++ Standard

Problem

You want your compiler to accept only programs that conform to the C++ language standard.

Solution

Command-line options for specifying strict conformance to the C++ standard are listed in Table 1-37. Instructions for enforcing strict conformance from your IDE are given in Table 1-38.

Some of the compiler options I introduced in Table 1-6 can be considered conformance options. Examples include options to enable basic language features such as wide-character support, exceptions, and runtime type information. I've omitted these in Table 1-37.

Table 1-37. Enforcing strict conformance from the command line

Toolset	Command-line compiler options
GCC	*-ansi -pedantic-errors*
Visual C++	*-Za*
Intel (Windows)	*-Za -Qms0*
Intel (Linux)	*-strict-ansi*[19]
Metrowerks	*-ansi strict -iso_templates on -msext off*
Comeau (Windows)	*A*
Comeau (Unix)	*strict or -A*
Borland	*-A*[20]
Digital Mars	*-A*

[19] Versions of the Intel compiler for Linux prior to 9.0 used the option -strict_ansi. When using -strict-ansi or -strict_ansi, it may be necessary to enable Intel's standard library, using the option -cxxlib-icc.

[20] With the option -A, some of the standard headers from the STLPort library may fail to compile.

Recipe 1.25. Causing a Source File to Be Linked Automatically Against a Specified Library

Problem

You've written a library that you'd like to distribute as a collection of headers and prebuilt static or dynamic libraries, but you don't want users of your library to have to specify the names of the binaries when they link their applications.

Solution

If you are programming for Windows and using the Visual C++, Intel, Metrowerks, Borland, or Digital Mars toolsets, you can use pragma comment in your library's headers to specify the names, and optionally the full file pathnames, of the prebuilt binaries against which any code that includes the headers should be linked.

For example, suppose you want to distribute the library from Example 1-1 as a static library *libjohnpaul.lib* together with the header *johnpaul.hpp*. Modify the header as shown in Example 1-26.

Example 1-26. Using pragma comment

```
 #ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

#pragma comment(lib, "libjohnpaul")

void johnpaul( );

#endif // JOHNPAUL_HPP_INCLUDED
```

With this change, the Visual C++, Intel, Metrowerks, Borland, and Digital Mars linkers will automatically search for the library *libjohnpaul.lib* when linking code that includes the header *johnpaul.hpp*.

Discussion

In some ways, linking can be a more difficult phase of the build process than compiling. One of the most common problems during linking occurs when the linker finds the wrong version of a library. This is a particular problem on Windows, where runtime librariesand the libraries that depend on themfrequently come in many variants. For this reason, libraries for Windows are often distributed with names mangled to reflect the various build configurations. While this helps to reduce version conflict, it also makes linking harder because you have to specify the correct mangled name to the linker.

For this reason, pragma comment is a very powerful tool. Among other things, it allows you to specify the correct mangled name of a library in a header file, saving the user the trouble of having to understand your name-mangling convention. If, in addition, you design your installation process to copy the binary files to a location automatically searched by the linkersuch as the *lib* subdirectory of the Visual C++, CodeWarrior, or C++Builder root directoriesprogrammers will be able to use your library simply by including your headers.

So far, so good. There's just one problem: pragma comment is not recognized by all compilers. If you wish to write portable code, you should invoke a pragma only after verifying that it is supported by the toolset being used. For example, you could modify *johnpaul.cpp* to read:

```
 #ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

#if defined( MSC VER) || \
```

Recipe 1.26. Using Exported Templates

Problem

You want to build a program that uses exported templates, meaning that it declares templates in headers with the export keyword and places template implementations in *.cpp* files.

Solution

First, compile the *.cpp* files containing the template implementations into object files, passing the compiler the command-line options necessary to enable exported templates. Next, compile and link the *.cpp* files that use the exported templates, passing the compiler and linker the command-line options necessary to enable exported templates as well as the options to specify the directories that contain the template implementations.

The options for enabling exported templates are given in Table 1-39. The options for specifying the location of template implementations are given in Table 1-40. If your toolset does not appear in this table, it likely does not support exported templates.

Table 1-39. Options to enable exported templates

Toolset	**Script**
Comeau (Unix)	*export*, *-A* or *strict*
Comeau (Windows)	*export* or *A*
Intel (Linux)	*-export* or *-strict-ansi*[22]

[22] Versions of the Intel compiler for Linux prior to 9.0 used the option -strict_ansi.

Table 1-40. Option to specify the location of template implementations

Toolset	**Script**
Comeau	*template_directory=<path>*
Intel (Linux)	*-export_dir<path>*

For example, suppose you want to compile the program displayed in Example 1-27. It consists of three files:

-
- The file *plus.hpp* contains the declaration of an exported function template plus( ).
-
- The file *plus.cpp* contains the definition of plus( ).

Chapter 2. Code Organization

PREV NEXT

Introduction

Perhaps one of the reasons C++ has been so popular is its ability to serve small, large, and massive projects well. You can write a few classes for a small prototype or research project, and as the project grows and people are added, C++ will allow you to scale the application into modules that have varying degrees of independence. The trade-off is that you have to make time to do some manual reorganization along the way (adding namespaces, rearranging your header files' physical locations, etc.). Usually this is worth it though, because you can make your application modular and let different people focus only on their logical, functional areas.

The manual labor that you have to invest along the way is inversely proportional to the amount of time you spend designing modularity in the first place. Start with some of the good techniques for modularization, and your code base will scale.

If you don't already use namespaces, you've probably at least heard of them, and very likely you use one already: the std namespace, which is the namespace that contains the standard library. Namespaces are not used as frequently as they ought to be, in my experience, but that's not because they're complicated or using them requires much effort. Recipe 2.3 explains how to modularize code with namespaces.

Many of the recipes in this chapter describe techniques that you apply from within header files. Since there are a number of facilities discussed, each explaining a different part of a header file, I have included Example 2-1 in the introduction, which shows what a typical header file looks like that uses all of the techniques described in this chapter.

Example 2-1. A header file

```
#ifndef MYCLASS_H_  _ // #include guards, Recipe 2.0
#define MYCLASS_H_  _

#include <string>

namespace mylib { // Namespaces, Recipe 2.3

   class AnotherClass; // forward class declarations, Recipe 2.2
   class Logger;

   extern Logger* gpLogger; // External storage declaration, Recipe 2.1

   class MyClass {
   public:
      std::string getVal( ) const;
      // ...
   private:
      static int refCount_;
      std::string val_;
   }

// Inline definitions, Recipe 2.4
inline std::string MyClass::getVal( ) const {
   return(val_);
}

#include "myclass.inl"

} // namespace

#endif // MYCLASS_H_  _
```

Once you have your header file written and out of the way, most of the time you will need an implementation file, too, by which I mean a .cpp file that contains definitions and not just declarations

Recipe 2.1. Making Sure a Header File Gets Included Only Once

Problem

You have a header file that is included by several other files. You want to make sure the preprocessor scans the declarations in the header file no more than once.

Solution

#define a macro in your header file, and include only the contents of the header file if the macro hasn't already been defined. You can use this combination of the #ifndef, #define, and #endif preprocessor directives, as I did in Example 2-1:

```
 #ifndef MYCLASS_H_  _ // #include guards
#define MYCLASS_H_  _

// Put everything here...

#endif // MYCLASS_H_  _
```

When the preprocessor scans this header file, one of the first things it will encounter is the #ifndef directive and the symbol that follows. #ifndef tells the preprocessor to continue processing on the next line only if the symbol MYCLASS_H_ _ is not already defined. If it is already defined, then the preprocessor should skip to the closing #endif. The line following #ifndef defines MYCLASS_H_ _, so if this file is scanned by the preprocessor twice in the same compilation, the second time MYCLASS_H_ _ is defined. By placing all of your code in between the #ifndef and #endif, you ensure that it is only read once during the compilation process.

Discussion

If you don't use this technique, which is called using include guards , you've probably already seen "symbol already defined" compilation errors that result from not taking a protective measure against multiple definitions. C++ does not allow you to define the same symbol more than once, and if you do (on purpose or by accident) you get a compilation error. Include guards prevent such errors, and they are pretty standard practice.

The macros you #define don't have to follow any particular format, but the syntax I used above is common. The idea is to use a symbol that won't conflict with another macro and cause your file to inadvertently be skipped during preprocessing. In practice, you may see other techniques, such as including a header file or module version in the macro, e.g., MYCLASS_H_V301_ _, or maybe even the author's name. It isn't that important how you name it, so long as you are consistent. These macros should only be referenced by the header file they are protecting, and nowhere else.

In some code you may see external include guards, which are the same as the internal include guards I described earlier, except that they appear in the file that is including the header file, not the header file itself:

```
 #ifndef MYCLASS_H_  _
#include "myclass.h"
#endif
```

This short-circuits the inclusion process by not even including the file myclassh.h if the macro MYCLASS_H_ _ is already defined. External include guards were advocated several years ago to improve compile times for large projects, but compilers have improved and they are no longer necessary. Don't use them.

Recipe 2.2. Ensuring You Have Only One Instance of a Variable Across Multiple Source Files

Problem

You need the same variable to be used by different modules in a program, and you can only have one copy of this variable. In other words, a global variable.

Solution

Declare and define the variable in a single implementation file in the usual manner, and use the extern keyword in other implementation files where you require access to that variable at runtime. Often, this means including the extern declarations in a header file that is used by all implementation files that need access to the global variable. Example 2-3 contains a few files that show how the extern keyword can be used to access variables defined in another implementation file.

Example 2-3. Using the extern keyword

```
 // global.h
#ifndef GLOBAL_H_  _ // See Recipe 2.0
#define GLOBAL_H_  _

#include <string>

extern int x;
extern std::string s;

#endif

// global.cpp
#include <string>

int x = 7;
std::string s = "Kangaroo";

// main.cpp
#include <iostream>
#include "global.h"

using namespace std;

int main( ) {

   cout << "x = " << x << endl;
   cout << "s = " << s << endl;
}
```

Discussion

The extern keyword is a way of telling the compiler that the actual storage for a variable is allocated somewhere else. extern tells the linker that the variable it qualifies is somewhere in another object file, and that the linker needs to go find it when creating the final executable or library. If the linker never finds the extern variable you have declared, or it finds more than one of definition for it, you will get a link error.

Example 2-3 isn't terribly exciting, but it illustrates the point well. My two global variables are declared and defined in *global.cpp:*

```
 int x = 7;
std::string s = "Kangaroo";
```

Recipe 2.3. Reducing #includes with Forward Class Declarations

Problem

You have a header file that references classes in other headers, and you need to reduce compilation dependencies (and perhaps time).

Solution

Use forward class declarations where possible to avoid unnecessary compilation dependencies. Example 2-4 gives a short example of a forward class declaration.

Example 2-4. Forward class declaration

```
 // myheader.h
#ifndef MYHEADER_H_  _
#define MYHEADER_H_  _

class A; // No need to include A's header

class B {
   public:
      void f(const A& a);
   // ...
   private:
      A* a_;
};

#endif
```

Somewhere else there is a header and perhaps an implementation file that declares and defines the class A, but from within *myheader.h* I don't care about the details of A: all I need to know is that A is a class.

Discussion

A forward class declaration is a way to ignore details that you don't need to be concerned with. In Example 2-4, *myheader.h* doesn't need to know anything about the class A except that it exists and that it's a class.

Consider what would happen if you #included the header file for A, or, more realistically, the header files for the half-dozen or so classes you would use in a real header file. Now an implementation file (*myheader.cpp*) includes this header, *myheader.h*, because it contains the declarations for everything. So far, so good. If you change one of the header files included by *myheader.h* (or one of the header files included by one of those files), then all of the implementation files that include *myheader.h* will need to be recompiled.

Forward declare your class and these compilation dependencies go away. Using a forward declaration simply creates a name to which everything else in the header file can refer. The linker has the happy task of matching up definitions in the implementation files that use your header.

Sadly, you can't always use forward declarations. The class B in Example 2-4 only uses pointers or references to A, so I can get away with a forward declaration. However, if I use an A member function or variable, or if I have an object of type A--and not just a pointer or reference to onein my definition for the class B, suddenly my forward declaration is insufficient. This is because files including *myheader.h* need to know the size of B, and if A is a member variable of B, then the compiler needs to know A's size

Recipe 2.4. Preventing Name Collisions with Namespaces

Problem

You have names from unrelated modules that are clashing, or you want to avoid such clashes by creating logical groups of code in advance.

Solution

Use namespaces to modularize code. With namespaces, you can group large groups of code in separate files into a single namespace. You can nest namespaces as deeply as necessary to partition a large module into submodules, and consumers of your module can selectively expose the elements in your namespace they want to use. Example 2-5 shows a few of the ways you can use a namespace.

Example 2-5. Using namespaces

```
 // Devices.h
#ifndef DEVICES_H_  _
#define DEVICES_H_  _

#include <string>
#include <list>

namespace hardware {

   class Device {
   public:
      Device( ) : uptime_(0), status_("unknown") {}
      unsigned long getUptime( ) const;
      std::string getStatus( ) const;
      void reset( );
   private:
      unsigned long uptime_;
      std::string status_;
   };

   class DeviceMgr {
   public:
      void getDeviceIds(std::list<std::string>& ids) const;
      Device getDevice(const std::string& id) const;
      // Other stuff...
   };
}

#endif // DEVICES_H_  _

// Devices.cpp
#include "Devices.h"
#include <string>
#include <list>

namespace hardware {

   using std::string;
   using std::list;

   unsigned long Device::getUptime( ) const {
      return(uptime_);
   }
   string Device::getStatus( ) const {
      return(status_);
   }
```

Recipe 2.5. Including an Inline File

Problem

You have a number of membe r functions or standalone functions that you want to make inline, but you don't want to define them all in the class definition (or even after it) in the header file. This way, you keep declaration and implementation separate.

Solution

Create an *.inl* file and #include it at the end of your header file. This is equivalent to putting the function definition at the end of the header file, but this lets you keep declaration and definition separate. Example 2-6 shows how.

Example 2-6. Using an inline file

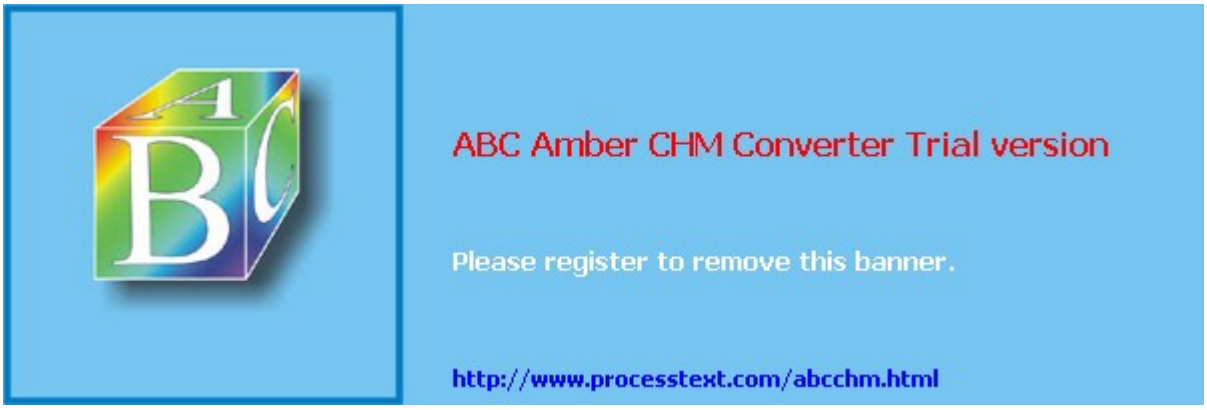
```
 // Value.h
#ifndef VALUE_H_  _
#define VALUE_H_  _

#include <string>

class Value {
public:
   Value (const std::string& val) : val_(val) {}
   std::string getVal( ) const;
private:
   std::string val_;
};

#include "Value.inl"

#endif VALUE_H_  _

// Value.inl
inline std::string Value::getVal( ) const {
   return(val_);
}
```

This solution doesn't require much explanation. #include is replaced with the contents of its argument, so what happens here is that the contents of *Value.inl* are brought into the header file. Any file that includes this header, therefore, has the definition of the inline functions, but you don't have to clutter up your class declaration.

PREV NEXT

Chapter 3. Numbers

PREV NEXT



Introduction

Even if you aren't writing scientific or engineering applications, you will usually have to work with numbers to some degree. This chapter contains solutions to common problems when working with C++'s numeric types.

Several of the recipes contain techniques for converting numbers of various formats (hexadecimal, floating-point, or scientific notation) from numeric types to strings or vice versa. Writing code to make this transformation yourself is cumbersome and tedious, so I present facilities from the standard library or one of the Boost libraries to make the task easier. There are also a few recipes for dealing with only the numeric types: safely converting between them, comparing floating-point numbers within a bounded range, and finding the minimum and maximum values for numeric types.

The recipes in this chapter provide solutions to some general problems you may run into when working with numbers in C++, but it does not attempt to solve problems that are specific to application domains. If you are writing scientific or engineering applications, you should also take a look at Chapter 11, which contains recipes for many common scientific and numerical algorithms.

Recipe 3.1. Converting a String to a Numeric Type

Problem

You have numbers in string format, but you need to convert them to a numeric type, such as an int or float.

Solution

You can do this in one of two ways, with standard library functions or with the lexical_cast class in Boost (written by Kevlin Henney). The standard library functions are cumbersome and unsafe, but they are standard, and in some cases, you need them, so I present them as the first solution. lexical_cast is safer, easier to use, and just more fun, so I present it in the discussion.

The functions strtol, strtod, and strtoul, defined in <cstdlib>, convert a null-terminated character string to a long int, double, or unsigned long. You can use them to convert numeric strings of any base to a numeric type. The code in Example 3-1 demonstrates a function, hex2int, that you can use for converting a hexadecimal string to a long.

Example 3-1. Converting number strings to numbers

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

using namespace std;

long hex2int(const string& hexStr) {
   char *offset;
   if (hexStr.length( ) > 2) {
      if (hexStr[0] == '0' && hexStr[1] == 'x') {
         return strtol(hexStr.c_str( ), &offset, 0);
      }
   }
   return strtol(hexStr.c_str( ), &offset, 16);
}

int main( ) {
   string str1 = "0x12AB";
   cout << hex2int(str1) << endl;
   string str2 = "12AB";
   cout << hex2int(str2) << endl;
   string str3 = "QAFG";
   cout << hex2int(str3) << endl;
}
```

Here's the output from this program:

```
4779
4779
0
```

The first two strings both contain the hexadecimal number 12AB. The first of the two has the 0x prefix, while the second doesn't. The third string doesn't contain a valid hexadecimal number; the function simply returns 0 in that case.

Discussion

Some people might be inclined to write their own function that converts hexadecimal numbers to integers. But why reinvent the wheel? The standard library already provides this functionality. Example

Recipe 3.2. Converting Numbers to Strings

Problem

You have numeric types (int, float) and you need to put the results in a string, perhaps formatted a certain way.

Solution

There are a number of different ways to do this, all with benefits and drawbacks. The first technique I will present uses a stringstream class to store the string data, because it is part of the standard library and easy to use. This approach is presented in Example 3-3. See the discussion for alternative techniques.

Example 3-3. Formatting a number as a string

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>

using namespace std;

int main( ) {

   stringstream ss;

   ss << "There are " << 9 << " apples in my cart.";
   cout << ss.str( ) << endl;  // stringstream::str( ) returns a string
                               // with the contents

   ss.str("");                    // Empty the string
   ss << showbase << hex << 16;  // Show the base in hexadecimal
   cout << "ss = " << ss.str( ) << endl;

   ss.str("");
   ss << 3.14;
   cout << "ss = " << ss.str( ) << endl;
}
```

The output of Example 3-3 looks like this:

```
There are 9 apples in my cart.
ss = 0x10
ss = 3.14
```

Discussion

A stringstream is a convenient way to put data into a string because it lets you use all of the formatting facilities provided by the standard input and output stream classes. In the simplest case in Example 3-3, I just use the left-shift operator (<<) to write a combination of text and numeric data to my string stream:

```
ss << "There are " << 9 << " apples in my cart.";
```

The << operator is overloaded for built-in types to format the input accordingly. When you want to get the string that holds your data, use the str member function:

```
cout << ss.str( ) << endl;
```

There are lots of stream manipulators in <iomanip>, and you can use them to do all sorts of formatting of your numeric data as you put it in the string. I used showbase and hex to format my number as hexadecimal in Example 3-3, but there are lots more. For example, you can set the precision to display more than the default number of digits:

Recipe 3.3. Testing Whether a String Contains a Valid Number

Problem

You have a string and you need to find out if it contains a valid number.

Solution

You can use the Boost lexical_cast function template to test for a valid number. Using this approach, a valid number can include a preceding minus sign, or a preceding plus sign, but not whitespace. I give a few examples of the kinds of formats that work with lexical_cast in Example 3-5.

Example 3-5. Validating a string number

```
 #include <iostream>
#include <boost/lexical_cast.hpp>

using namespace std;
using boost::lexical_cast;
using boost::bad_lexical_cast;

template<typename T>
bool isValid(const string& num) {

   bool res = true;

   try {
      T tmp = lexical_cast<T>(num);
   }
   catch (bad_lexical_cast &e) {
      res = false;
   }

   return(res);
}

void test(const string& s) {

   if (isValid<int>(s))
      cout << s << " is a valid integer." << endl;
   else
      cout << s << " is NOT a valid integer." << endl;

   if (isValid<double>(s))
      cout << s << " is a valid double." << endl;
   else
      cout << s << " is NOT a valid double." << endl;

   if (isValid<float>(s))
      cout << s << " is a valid float." << endl;
   else
      cout << s << " is NOT a valid float." << endl;
}

int main( ) {

   test("12345");
   test("1.23456");
   test("-1.23456");
   test(" - 1.23456");
```

Recipe 3.4. Comparing Floating-Point Numbers with Bounded Accuracy

Problem

You need to compare floating-point values, but you only want tests for equality, greater-than, or less-than to be concerned with a limited number of digits. For example, you want 3.33333 and 3.33333333 to show as being equal when comparing to a precision of .0001.

Solution

Write your own comparison functions that take a parameter that bounds the accuracy of the comparison. Example 3-6 shows the basic technique for such comparison functions.

Example 3-6. Comparing floating-point numbers

```
 #include <iostream>
#include <cmath>    // for fabs( )

using namespace std;

bool doubleEquals(double left, double right, double epsilon) {
   return (fabs(left - right) < epsilon);
}

bool doubleLess(double left, double right, double epsilon,
                bool orequal = false) {
   if (fabs(left - right) < epsilon) {
      // Within epsilon, so considered equal
      return (orequal);
   }
   return (left < right);
}

bool doubleGreater(double left, double right, double epsilon,
                   bool orequal = false) {
   if (fabs(left - right) < epsilon) {
      // Within epsilon, so considered equal
      return (orequal);
   }
   return (left > right);
}

int main( ) {

   double first = 0.33333333;
   double second = 1.0 / 3.0;
   cout << first << endl;
   cout << second << endl;

   // Straight equalify test. Fails when you wouldn't want it to.
   // (boolalpha prints booleans as "true" or "false")
   cout << boolalpha << (first == second) << endl;
   // New equality. Passes as scientific app probably wants.
   cout << doubleEquals(first, second, .0001) << endl;
   // New less-than
   cout << doubleLess(first, second, .0001) << endl;
   // New Greater-than
   cout << doubleGreater(first, second, .0001) << endl;
   // New less-than-or-equal-to
   cout << doubleLess(first, second, .0001, true) << endl;
```

Recipe 3.5. Parsing a String Containing a Number in Scientific Notation

Problem

You have a string containing a number in scientific notation, and you want to store the number's value in a double variable.

Solution

The most direct way to parse a scientific notation number is by using the C++ library's built-in stringstream class declared in <sstream>, as you can see in Example 3-7.

Example 3-7. Parsing a number in scientific notation

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

using namespace std;

double sciToDub(const string& str) {

   stringstream ss(str);
   double d = 0;
   ss >> d;

   if (ss.fail( )) {
      string s = "Unable to format ";
      s += str;
      s += " as a number!";
      throw (s);
   }

   return (d);
}

int main( ) {

   try {
      cout << sciToDub("1.234e5") << endl;
      cout << sciToDub("6.02e-2") << endl;
      cout << sciToDub("asdf") << endl;
   }
   catch (string& e) {
      cerr << "Whoops: " << e << endl;
   }
}
```

Following is the output from this code:

```
 123400
0.0602
Whoops: Unable to format asdf as a number!
```

Discussion

The stringstream class is, not surprisingly, a string that behaves like a stream. It is declared in <sstring>. If you need to parse a string that contains a number in scientific notation (see also Recipe 3.2), a stringstream will do the job nicely. The standard stream classes already "know" how to parse numbers, so don't waste your time reimplementing this logic if you don't have to.

Recipe 3.6. Converting Between Numeric Types

Problem

You have number of one type and you need to convert it to another, such as an int to a short or a vice versa, but you want to catch any overflow or underflow errors at runtime.

Solution

Use Boost's numeric_cast class template. It performs runtime checks that throw an exception of type bad_numeric_cast if you will overflow or underflow the variable where you are putting a value. Example 3-8 shows you how to do this.

Example 3-8. Safe numeric conversions

```
 #include <iostream>
#include <boost/cast.hpp>

using namespace std;
using boost::numeric_cast;
using boost::bad_numeric_cast;

int main( ) {

   // Integer sizes
   try {
      int i = 32767;
      short s = numeric_cast<short>(i);

      cout << "s = " << s << endl;

      i++; // Now i is out of range (if sizeof(short) is 2)
      s = numeric_cast<short>(i);
   }
   catch (bad_numeric_cast& e) {
      cerr << e.what( ) << endl;
   }

   try {
      int i = 300;
      unsigned int ui = numeric_cast<unsigned int>(i);

      cout << ui << endl; // Fine

      i *= -1;
      ui = numeric_cast<unsigned int>(i); // i is negative!
   }
   catch (bad_numeric_cast& e) {
      cerr << e.what( ) << endl;
   }

   try {
      double d = 3.14;
      int i = numeric_cast<int>(d);

      i = numeric_cast<int>(d); // This shaves off the 0.14!

      cout << i << endl;  // i = 3

   }
   catch (bad_numeric_cast& e) {
      cerr << e.what( ) << endl;
```

Recipe 3.7. Getting the Minimum and Maximum Values for a Numeric Type

Problem

You need to know the largest or smallest representable value for your platform for a numeric type, such as an int or double.

Solution

Use the numeric_limits class template in the <limits> header to get, among other things, the largest and smallest possible values for a numeric type (see Example 3-9).

Example 3-9. Getting numeric limits

```
 #include <iostream>
#include <limits>

using namespace std;

template<typename T>
void showMinMax( ) {
   cout << "min: " << numeric_limits<T>::min( ) << endl;
   cout << "max: " << numeric_limits<T>::max( ) << endl;
   cout << endl;
}

int main( ) {

   cout << "short:" << endl;
   showMinMax<short>( );
   cout << "int:" << endl;
   showMinMax<int>( );
   cout << "long:" << endl;
   showMinMax<long>( );
   cout << "float:" << endl;
   showMinMax<float>( );
   cout << "double:" << endl;
   showMinMax<double>( );
   cout << "long double:" << endl;
   showMinMax<long double>( );
   cout << "unsigned short:" << endl;
   showMinMax<unsigned short>( );
   cout << "unsigned int:" << endl;
   showMinMax<unsigned int>( );
   cout << "unsigned long:" << endl;
   showMinMax<unsigned long>( );
}
```

Here's what I get on Windows XP using Visual C++ 7.1:

```
 short:
min: -32768
max: 32767

int:
min: -2147483648
max: 2147483647

long:
min: -2147483648
max: 2147483647
```

Chapter 4. Strings and Text

Introduction

This chapter contains recipes for working with strings and text files. Most C++ programs, regardless of their application, manipulate strings and text files to some degree. Despite the variety of applications, however, the requirements are often the samefor strings: trimming, padding, searching, splitting, and so on; for text files: wrapping, reformatting, reading delimited files, and more. The recipes that follow provide solutions to many of these common needs that do not have ready-made solutions in the C++ standard library.

The standard library is portable, standardized, and, in general, at least as efficient as homemade solutions, so in the following examples I have preferred it over code from scratch. It contains a rich framework for manipulating and managing strings and text, much of which is in the form of the class templates basic_string (for strings), basic_istream, and basic_ostream (for input and output text streams). Almost all of the techniques in this chapter use or extend these class templates. In cases where they didn't have what I wanted, I turned to another area of the standard library that is full of generic, prebuilt solutions: algorithms and containers.

Everybody uses strings, so chances are that if what you need isn't in the standard library, someone has written it. The Boost String Algorithms library, written by Pavol Droba, fills many of the gaps in the standard library by implementing most of the algorithms that you've had to use at one time or another, and it does it in a portable, efficient way. Check out the Boost project at www.boost.org for more information and documentation of the String Algorithms library. There is some overlap between the String Algorithms library and the solutions I present in this chapter. In most cases, I provide examples of or at least mention Boost algorithms that are related to the solutions presented.

For most examples, I have provided both a nontemplate and a template version. I did this for two reasons. First, most of the areas of the standard library that use character data are class or function templates that are parameterized on the type of character, narrow (char) or wide (wchar_t). By following this model, you will help maximize the compatibility of your software with the standard library. Second, whether you are working with the standard library or not, class and function templates provide an excellent facility for writing generic software. If you do not need templates, however, you can use the nontemplate versions, though I recommend experimenting with templates if you are new to them.

The standard library makes heavy use of templates and uses typedefs to insulate programmers from some of the verbose syntax that templates use. As a result, I use the terms basic_string, string, and wstring interchangeably, since what applies to one usually applies to them all. string and wstring are just typedefs for basic_string<char> and basic_string<wchar_t>.

Finally, you will probably notice that none of the recipes in this chapter use C-style strings, i.e., null-terminated character arrays. The standard library provides such a wealth of efficient and extensible support for C++ strings that to use C-style string functions (which were provided primarily for backward-compatibility anyway) is to forego the flexibility, safety, and generic nature of what you get for free with your compiler: C++ string classes.

Recipe 4.1. Padding a String

Problem

You need to "pad," or fill, a string with a number of occurrences of some character to a certain width. For example, you may want to pad the string "Chapter 1" to 20 characters wide with periods, so that it looks like "Chapter 1.....".

Solution

Use string's insert and append member functions to pad a string with characters on the beginning or end. For example, to pad the end of a string to 20 characters with X's:

```
 std::string s = "foo";
s.append(20 - s.length( ), 'X');
```

To pad the string at the beginning instead:

```
 s.insert(s.begin( ), 20 - s.length( ), 'X');
```

Discussion

The difference in usage between the two functions is insert's first parameter. It is an iterator that points to the character immediately to the right of where the insert should occur. The begin member function returns an iterator pointing to the first element in the string, so in the example, the series of characters is inserted to the left of that. The parameters common to both functions are the number of times to repeat the character and the character itself.

insert and append are actually member functions of the basic_string class template in the <string> header (string is a typedef for basic_string<char> and wstring is a typedef for basic_string<wchar_t>), so they work for strings of narrow or wide characters. Using them as needed, as in the above example, will work fine, but if you are using basic_string member functions from within your own generic utility functions, you should build on the standard library's existing generic design and use a function template. Consider the code in Example 4-1, which defines a generic pad function template that operates on basic_strings.

Example 4-1. A generic pad function template

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

using namespace std;

// The generic approach
template<typename T>
void pad(basic_string<T>& s,
         typename basic_string<T>::size_type n, T c) {
   if (n > s.length( ))
      s.append(n - s.length( ), c);
}

int main( ) {

   string  s  = "Appendix A";
   wstring ws = L"Acknowledgments"; // The "L" indicates that
                                    // this is a wide char
   pad(s, 20, '*');                 // literal
   pad(ws, 20, L'*');

   // cout  << s  << std::endl;  // You shouldn't be able to
   wcout << ws << std::endl;     // run these at the same time
}
```

Recipe 4.2. Trimming a String

Problem

You need to trim some number of characters from the end(s) of a string, usually whitespace.

Solution

Use iterators to identify the portion of the string you want to remove, and the erase member function to remove it. Example 4-2 presents the function rtrim that trims a character from the end of a string.

Example 4-2. Trimming characters from a string

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

// The approach for narrow character strings
void rtrim(std::string& s, char c) {

   if (s.empty( ))
      return;

   std::string::iterator p;
   for (p = s.end( ); p != s.begin( ) && *--p == c;);

   if (*p != c)
      p++;

   s.erase(p, s.end( ));
}

int main( )
{
   std::string s = "zoo";

   rtrim(s, 'o');

   std::cout << s << '\n';
}
```

Discussion

Example 4-2 will do the trick for strings of chars, but it only works for char strings. Just like you saw in Example 4-1, you can take advantage of the generic design of basic_string and use a function template instead. Example 4-3 uses a function template to trim characters from the end of any kind of character string.

Example 4-3. A generic version of rtrim

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

using namespace std;

// The generic approach for trimming single
// characters from a string
template<typename T>
void rtrim(basic_string<T>& s, T c)
{
   if (s.empty( ))
      return;
```

Recipe 4.3. Storing Strings in a Sequence

Problem

You want to store a set of strings in a sequence that looks and feels like an array.

Solution

Use a vector for array-like storage of your strings. Example 4-6 offers a simple example.

Example 4-6. Store strings in a vector

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

using namespace std;

int main( ) {

   vector<string> v;

   string s = "one";
   v.push_back(s);

   s = "two";
   v.push_back(s);

   s = "three";
   v.push_back(s);

   for (int i = 0; i < v.size( ); ++i)
   {
      cout << v[i] << '\n';
   }
}
```

vectors follow array semantics for random access (they also do a lot more), so they are easy and familiar to use. vectors are just one of many sequences in the standard library, however; read on for more of this broad subject.

Discussion

A vector is a dynamically sized sequence of objects that provides array-style operator[] random access. The member function push_back copies its argument via copy constructor, adds that copy as the last item in the vector, and increments its size by one. pop_back does the exact opposite, by removing the last element. Inserting or deleting items from the end of a vector takes amortized constant time, and inserting or deleting from any other location takes linear time. These are the basics of vectors. There is a lot more to them.

In most cases, a vector should be your first choice over a C-style array. First of all, they are dynamically sized, which means they can grow as needed. You don't have to do all sorts of research to figure out an optimal static size, as in the case of C arrays; a vector grows as needed, and it can be resized larger or smaller manually if you need to. Second, vectors offer bounds checking with the at member function (but not with operator[]), so that you can do something if you reference a nonexistent index instead of simply watching your program crash or worse, continuing execution with corrupt data. Look at Example 4-7. It shows how to deal with out-of-bounds indexes.

Recipe 4.4. Getting the Length of a String

Problem

You need the length of a string.

Solution

Use string's length member function:

```
 std::string s = "Raising Arizona";
int i = s.length( );
```

Discussion

Retrieving the length of a string is a trivial task, but it is a good opportunity to discuss the allocation scheme for strings (both wide and narrow character). strings, unlike C-style null-terminated character arrays, are dynamically sized, and grow as needed. Most standard library implementations start with an arbitrary (low) capacity, and grow by doubling the capacity each time it is reached. Knowing how to analyze this growth, if not the exact algorithm, is helpful in diagnosing string performance problems.

The characters in a basic_string are stored in a buffer that is a contiguous chunk of memory with a static size. The buffer a string uses is an arbitrary size initially, and as characters are added to the string, the buffer fills up until its capacity is reached. When this happens, the buffer grows, sort of. Specifically, a new buffer is allocated with a larger size, the characters are copied from the old buffer to the new buffer, and the old buffer is deleted.

You can find out the size of the buffer (not the number of characters it contains, but its maximum size) with the capacity member function. If you want to manually set the capacity to avoid needless buffer copies, use the reserve member function and pass it a numeric argument that indicates the desired buffer size. There is a maximum size on the possible buffer size though, and you can get that by calling max_size. You can use all of these to observe memory growth in your standard library implementation. Take a look at Example 4-9 to see how.

Example 4-9. String length and capacity

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

using namespace std;

int main( ) {

   string s = "";
   string sr = "";

   sr.reserve(9000);

   cout << "s.length    = " << s.length( )    << '\n';
   cout << "s.capacity  = " << s.capacity( ) << '\n';
   cout << "s.max_size  = " << s.max_size( ) << '\n';

   cout << "sr.length   = " << sr.length( )   << '\n';
   cout << "sr.capacity = " << sr.capacity( ) << '\n';
   cout << "sr.max_size = " << sr.max_size( ) << '\n';

   for (int i = 0; i < 10000; ++i) {

      if (s.length( ) == s.capacity( )) {
```

Recipe 4.5. Reversing a String

Problem

You want to reverse a string.

Solution

To reverse a string "in place," without using a temporary string, use the reverse function template in the <algorithm> header:

```
std::reverse(s.begin( ), s.end( ));
```

Discussion

reverse works simply enough: it modifies the range you give it such that it is in the opposite order that it was originally. It takes linear time.

In the event that you want to copy the string to another string, but backward, use reverse iterators, like this:

```
std::string s = "Los Angeles";
std::string rs;

rs.assign(s.rbegin( ), s.rend( ));
```

rbegin and rend return reverse iterators. Reverse iterators behave as though they are looking at the sequence backward. rbegin returns an iterator that points to the last element, and rend returns an iterator that points to one before the first; this is exactly opposite of what begin and end do.

But do you need to reverse the string in the first place? With rbegin and rend, any member functions or algorithms that operate on iterator ranges can be used on the reverse version of the string. And if you want to search through the string, you can use rfind to do what find does but starting from the end of the string and moving backward. For large strings, or large numbers of strings, reversing can be expensive, so avoid it if you can.

Recipe 4.6. Splitting a String

Problem

You want to split a delimited string into multiple strings. For example, you may want to split the string "Name|Address|Phone" into three separate strings, "Name", "Address", and "Phone", with the delimiter removed.

Solution

Use basic_string's find member function to advance from one occurrence of the delimiter to the next, and substr to copy each substring out of the original string. You can use any standard sequence to hold the results; Example 4-10 uses a vector.

Example 4-10. Split a delimited string

```
 #include <string>
#include <vector>
#include <functional>
#include <iostream>

using namespace std;

void split(const string& s, char c,
           vector<string>& v) {
   string::size_type i = 0;
   string::size_type j = s.find(c);

   while (j != string::npos) {
      v.push_back(s.substr(i, j-i));
      i = ++j;
      j = s.find(c, j);

      if (j == string::npos)
         v.push_back(s.substr(i, s.length( )));
   }
}

int main( ) {
   vector<string> v;
   string s = "Account Name|Address 1|Address 2|City";

   split(s, '|', v);

   for (int i = 0; i < v.size( ); ++i) {
      cout << v[i] << '\n';
   }
}
```

Discussion

Making the example above a function template that accepts any kind of character is trivial; just parameterize the character type and change references to string to basic_string<T>:

```
 template<typename T>
void split(const basic_string<T>& s, T c,
           vector<basic_string<T> >& v) {
   basic_string<T>::size_type i = 0;
   basic_string<T>::size_type j = s.find(c);

   while (j != basic_string<T>::npos) {
```

Recipe 4.7. Tokenizing a String

Problem

You need to break a string into pieces using a set of delimiters.

Solution

Use the find_first_of and first_first_not_of member functions on basic_string to iterate through the string and alternately locate the next tokens and non-tokens. Example 4-12 presents a simple StringTokenizer class that does just that.

Example 4-12. A string tokenizer

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

using namespace std;

// String tokenizer class.
class StringTokenizer {

public:

   StringTokenizer(const string& s, const char* delim = NULL) :
      str_(s), count_(-1), begin_(0), end_(0) {

      if (!delim)
         delim_ = " \f\n\r\t\v";  //default to whitespace
      else
         delim_ = delim;

      // Point to the first token
      begin_ = str_.find_first_not_of(delim_);
      end_ = str_.find_first_of(delim_, begin_);
   }

   size_t countTokens( ) {
     if (count_ >= 0) // return if we've already counted
       return(count_);

     string::size_type n = 0;
     string::size_type i = 0;

     for (;;) {
        // advance to the first token
        if ((i = str_.find_first_not_of(delim_, i)) == string::npos)
           break;
        // advance to the next delimiter
        i = str_.find_first_of(delim_, i+1);
        n++;
        if (i == string::npos)
          break;
     }
     return (count_ = n);
   }
   bool hasMoreTokens( ) {return(begin_ != end_);}
   void nextToken(string& s) {
     if (begin_ != string::npos && end_ != string::npos) {
        s = str_.substr(begin_, end_-begin_);
        begin_ = str_.find_first_not_of(delim_, end_);
        end_ = str_.find_first_of(delim_, begin_);
```

Recipe 4.8. Joining a Sequence of Strings

Problem

Given a sequence of strings, such as output from Example 4-10, you want to join them together into a single, long string, perhaps with a delimiter.

Solution

Loop through the sequence and append each string to the output string. You can handle any standard sequence as input; Example 4-13 uses a vector of strings.

Example 4-13. Join a sequence of strings

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

using namespace std;

void join(const vector<string>& v, char c, string& s) {

   s.clear( );

   for (vector<string>::const_iterator p = v.begin( );
        p != v.end( ); ++p) {
      s += *p;
      if (p != v.end( ) - 1)
        s += c;
   }
}

int main( ) {

   vector<string> v;
   vector<string> v2;
   string s;

   v.push_back(string("fee"));
   v.push_back(string("fi"));
   v.push_back(string("foe"));
   v.push_back(string("fum"));

   join(v, '/', s);

   cout << s << '\n';
}
```

Discussion

Example 4-13 has one technique that is slightly different from previous examples. Look at this line:

```
for (vector<string>::const_iterator p = v.begin( );
```

The previous string examples simply used iterators, without the "const" part, but you can't get away with that here because v is declared as a reference to a const object. If you have a const container object, you can only use a const_iterator to access its elements. This is because a plain iterator allows writes to the object it refers to, which, of course, you can't do if your container object is const.

I declared v const for two reasons. First, I know I'm not going to be modifying its contents, so I want the compiler to give me an error if I do. The compiler is much better at spotting that kind of thing than I

Recipe 4.9. Finding Things in Strings

Problem

You want to search a string for something. Maybe it's a single character, another string, or one of (or not of) an unordered set of characters. And, for your own reasons, you have to find it in a particular way, such as the first or last occurrence, or the first or last occurrence relative to a particular index.

Solution

Use one of basic_string's "find" member functions. Almost all start with the word "find," and their name gives you a pretty good idea of what they do. Example 4-15 shows how some of the find member functions work.

Example 4-15. Searching strings

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

int main( ) {
   std::string s = "Charles Darwin";

   std::cout << s.find("ar") << '\n';            // Search from the
                                                 // beginning
   std::cout << s.rfind("ar") << '\n';           // Search from the end

   std::cout << s.find_first_of("swi")           // Find the first of
             << '\n';                            // any of these chars

   std::cout << s.find_first_not_of("Charles")   // Find the first
             << '\n';                            // that's not in this
                                                 // set

   std::cout << s.find_last_of("abg") << '\n';   // Find the first of
                                                 // any of these chars
                                                 // starting from the
                                                 // end

   std::cout << s.find_last_not_of("aDinrw")     // Find the first
             << '\n';                            // that's not in this
                                                 // set, starting from
                                                 // the end
}
```

Each of the find member functions is discussed in more detail in the "Discussion" section.

Discussion

There are six different find member functions for finding things in strings, each of which provides four overloads. The overloads allow for either basic_string or charT* parameters (charT is the character type). Each has a basic_string::size_type parameter pos that lets you specify the index where the search should begin, and there is one overload with a size_type parameter n that allows you only to search based on the first n characters from the set.

It's hard to keep track of all of these member functions, so Table 4-2 gives a quick reference of each function and its parameters.

Table 4-2. Member functions for searching strings

Recipe 4.10. Finding the nth Instance of a Substring

Problem

Given two strings source and pattern, you want to find the nth occurrence of pattern in source.

Solution

Use the find member function to locate successive instances of the substring you are looking for. Example 4-17 contains a simple nthSubstr function.

Example 4-17. Locate the nth version of a substring

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

using namespace std;

int nthSubstr(int n, const string& s,
              const string& p) {
   string::size_type i = s.find(p);     // Find the first occurrence

   int j;
   for (j = 1; j < n && i != string::npos; ++j)
      i = s.find(p, i+1); // Find the next occurrence

   if (j == n)
     return(i);
   else
     return(-1);
}

int main( ) {
   string s = "the wind, the sea, the sky, the trees";
   string p = "the";

   cout << nthSubstr(1, s, p) << '\n';
   cout << nthSubstr(2, s, p) << '\n';
   cout << nthSubstr(5, s, p) << '\n';
}
```

Discussion

There are a couple of improvements you can make to nthSubstr as it is presented in Example 4-17. First, you can make it generic by making it a function template instead of an ordinary function. Second, you can add a parameter to account for substrings that may or may not overlap with themselves. By "overlap," I mean that the beginning of the string matches part of the end of the same string, as in the word "abracadabra," where the last four characters are the same as the first four. Example 4-18 demonstrates this.

Example 4-18. An improved version of nthSubstr

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

using namespace std;

template<typename T>
int nthSubstrg(int n, const basic_string<T>& s,
               const basic_string<T>& p,
               bool repeats = false) {
```

Recipe 4.11. Removing a Substring from a String

Problem

You want to remove a substring from a string.

Solution

Use the find, erase, and length member functions of basic_string:

```
 std::string t = "Banana Republic";
std::string s = "nana";

std::string::size_type i = t.find(s);

if (i != std::string::npos)
   t.erase(i, s.length( ));
```

This will erase s.length( ) elements starting at the index where find found the first occurrence of the substring.

Discussion

There are lots of variations on the theme of finding a substring and removing it. For example, you may want to remove all instances of a substring instead of just one. Or just the last one. Or the seventh one. Each time the steps are the same: find the index of the beginning of the pattern you want to remove, then call erase on that index for the next n characters, where n is the length of the pattern string. See Recipe 4.9 for the different member functions for finding things in strings.

Chances are you also want to make your substring-removal function generic, so you can use it on strings of any kind of character. Example 4-19 offers a generic version that removes all instances of the pattern from a string.

Example 4-19. Remove all substrings from a string (generic version)

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

using namespace std;

template<typename T>
void removeSubstrs(basic_string<T>& s,
                   const basic_string<T>& p) {
   basic_string<T>::size_type n = p.length( );

   for (basic_string<T>::size_type i = s.find(p);
        i != basic_string<T>::npos;
        i = s.find(p))
      s.erase(i, n);
}

int main( ) {
   string s = "One fish, two fish, red fish, blue fish";
   string p = "fish";

   removeSubstrs(s, p);

   cout << s << '\n';
}
```

Recipe 4.12. Converting a String to Lower- or Uppercase

Problem

You have a string that you want to convert to lower- or uppercase.

Solution

Use the toupper and tolower functions in the <cctype> header to convert characters to upper- or lowercase. Example 4-20 shows how to do it using these functions. See the discussion for an alternative.

Example 4-20. Converting a string's case

```
 #include <iostream>
#include <string>
#include <cctype>
#include <cwctype>
#include <stdexcept>

using namespace std;

void toUpper(basic_string<char>& s) {
   for (basic_string<char>::iterator p = s.begin( );
        p != s.end( ); ++p) {
      *p = toupper(*p); // toupper is for char
   }
}

void toUpper(basic_string<wchar_t>& s) {
   for (basic_string<wchar_t>::iterator p = s.begin( );
        p != s.end( ); ++p) {
      *p = towupper(*p); // towupper is for wchar_t
   }
}

void toLower(basic_string<char>& s) {
   for (basic_string<char>::iterator p = s.begin( );
        p != s.end( ); ++p) {
      *p = tolower(*p);
   }
}

void toLower(basic_string<wchar_t>& s) {
   for (basic_string<wchar_t>::iterator p = s.begin( );
        p != s.end( ); ++p) {
      *p = towlower(*p);
   }
}

int main( ) {

   string s = "shazam";
   wstring ws = L"wham";

   toUpper(s);
   toUpper(ws);

   cout << "s =  " << s << endl;
   wcout << "ws = " << ws << endl;

   toLower(s);
   toLower(ws);
```

Recipe 4.13. Doing a Case-Insensitive String Comparison

Problem

You have two strings, and you want to know if they are equal, regardless of the case of the characters. For example, "cat" is not equal to "dog," but "Cat," for your purposes, is equal to "cat," "CAT," or "caT."

Solution

Compare the strings using the equal standard algorithm (defined in <algorithm>), and supply your own comparison function that uses the toupper function in <cctype> (or towupper in <cwctype> for wide characters) to compare the uppercase versions of characters. Example 4-21 offers a generic solution. It also demonstrates the use and flexibility of the STL; see the discussion below for a full explanation.

Example 4-21. Case-insensitive string comparison

```
1   #include <string>
2   #include <iostream>
3   #include <algorithm>
4   #include <cctype>
5   #include <cwctype>
6
7   using namespace std;
8
9   inline bool caseInsCharCompareN(char a, char b) {
10     return(toupper(a) == toupper(b));
11  }
12
13  inline bool caseInsCharCompareW(wchar_t a, wchar_t b) {
14     return(towupper(a) == towupper(b));
15  }
16
17  bool caseInsCompare(const string& s1, const string& s2) {
18     return((s1.size( ) == s2.size( )) &&
19            equal(s1.begin( ), s1.end( ), s2.begin( ),
caseInsCharCompareN));
20  }
21
22  bool caseInsCompare(const wstring& s1, const wstring& s2) {
23     return((s1.size( ) == s2.size( )) &&
24            equal(s1.begin( ), s1.end( ), s2.begin( ),
caseInsCharCompareW));
25  }
26
27  int main( ) {
28     string s1 = "In the BEGINNING...";
29     string s2 = "In the beginning...";
30     wstring ws1 = L"The END";
31     wstring ws2 = L"the endd";
32
33     if (caseInsCompare(s1, s2))
34        cout << "Equal!\n";
35
36     if (caseInsCompare(ws1, ws2))
37        cout << "Equal!\n";
38  }
```

Recipe 4.14. Doing a Case-Insensitive String Search

Problem

You want to find a substring in a string without regard for case.

Solution

Use the standard algorithms transform and search, defined in <algorithm>, along with your own special character comparison functions, similar to the approach presented in. Example 4-22 shows how to do this.

Example 4-22. Case-insensitive string search

```
 #include <string>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <cctype>

using namespace std;

inline bool caseInsCharCompSingle(char a, char b) {
   return(toupper(a) == b);
}

string::const_iterator caseInsFind(string& s, const string& p) {
   string tmp;

   transform(p.begin( ), p.end( ),             // Make the pattern
             back_inserter(tmp),                 // upper-case
             toupper);

   return(search(s.begin( ), s.end( ),         // Return the iter-
                 tmp.begin( ), tmp.end( ),     // ator returned by
                 caseInsCharCompSingle));        // search
}

int main( ) {
   string s = "row, row, row, your boat";
   string p = "YOUR";
   string::const_iterator it = caseInsFind(s, p);

   if (it != s.end( )) {
      cout << "Found it!\n";
   }
}
```

By returning an iterator that refers to the element in the target string where the pattern string starts, you ensure ease of compatibility with other standard algorithms since most of them accept iterator arguments.

Discussion

Example 4-22 demonstrates the usual mode of operation when working with standard algorithms. Create the functions that do the work, then plug them into the most appropriate algorithms as function objects. The charInsCharCompSingle function does the real work here but, unlike Example 4-21, this character comparison function only uppercases the first argument. This is because a little later in caseInsFind, I convert the pattern string to all uppercase before using it to search to avoid having to uppercase each pattern character multiple times.

Recipe 4.15. Converting Between Tabs and Spaces in a Text File

Problem

You have a text file that contains tabs or spaces, and you want to convert from one to the other. For example, you may want to replace all tabs with three spaces, or you may want to do just the opposite and replace occurrences of some number of spaces with a single tab.

Solution

Regardless of whether you are replacing tabs with spaces or spaces with tabs, use the ifstream and ofstream classes in <fstream>. In the first (simpler) case, read data in with an input stream, one character at a time, examine it, and if it's a tab, write some number of spaces to the output stream. Example 4-23 demonstrates how to do this.

Example 4-23. Replacing tabs with spaces

```
 #include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
      return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
     return(EXIT_FAILURE);

   char c;
   while (in.get(c)) {
      if (c == '\t')
         out << "   "; // 3 spaces
      else
         out << c;
   }

   out.close( );

   if (out)
      return(EXIT_SUCCESS);
   else
      return(EXIT_FAILURE);
}
```

If, instead, you need to replace spaces with tabs, see Example 4-24. It contains the function spacesToTabs that reads from an input stream, one character at a time, looking for three consecutive spaces. When it finds three in a row, it writes a tab to the output stream. For all other characters, or for fewer than three spaces, whatever is read from the input stream is written to the output stream.

Example 4-24. Replacing spaces with tabs

```
 #include <iostream>
#include <istream>
#include <ostream>
```

Recipe 4.16. Wrapping Lines in a Text File

Problem

You want to "wrap" text at a specific number of characters in a file. For example, if you want to wrap text at 72 characters, you would insert a new-line character after every 72 characters in the file. If the file contains human-readable text, you probably want to avoid splitting words.

Solution

Write a function that uses input and output streams to read in characters with istream::get(char), do some bookkeeping, and write out characters with ostream::put(char). Example 4-25 shows how to do this for text files that contain human-readable text without splitting words.

Example 4-25. Wrapping text

```
 #include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <cctype>
#include <functional>

using namespace std;

void textWrap(istream& in, ostream& out, size_t width) {

   string tmp;
   char cur = '\0';
   char last = '\0';
   size_t i = 0;

   while (in.get(cur)) {
      if (++i == width) {
         ltrimws(tmp);                  // ltrim as in Recipe
         out << '\n' << tmp;            // 4.1
         i = tmp.length( );
         tmp.clear( );
      } else if (isspace(cur) &&   // This is the end of
                 !isspace(last)) { // a word
         out << tmp;
         tmp.clear( );
      }
      tmp += cur;
      last = cur;
   }
}

int main(int argc, char** argv) {
   if (argc < 3)
      return(EXIT_FAILURE);

   int w = 72;
   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
     return(EXIT_FAILURE);

   if (argc == 4)
     w = atoi(argv[3]);
```

Recipe 4.17. Counting the Number of Characters, Words, and Lines in a Text File

Problem

You have to count the numbers of characters, words, and linesor some other type of text elementin a text file.

Solution

Use an input stream to read the characters in, one at a time, and increment local statistics as you encounter characters, words, and line breaks. Example 4-26 contains the function countStuff, which does exactly that.

Example 4-26. Calculating statistics about a text file

```
 #include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>

using namespace std;

void countStuff(istream& in,
                int& chars,
                int& words,
                int& lines) {

   char cur = '\0';
   char last = '\0';
   chars = words = lines = 0;

   while (in.get(cur)) {
      if (cur == '\n' ||
          (cur == '\f' && last == '\r'))
         lines++;
      else
        chars++;
      if (!std::isalnum(cur) &&   // This is the end of a
          std::isalnum(last))     // word
         words++;
      last = cur;
   }
   if (chars > 0) {               // Adjust word and line
      if (std::isalnum(last))     // counts for special
         words++;                 // case
      lines++;
   }
}

int main(int argc, char** argv) {

   if (argc < 2)
      return(EXIT_FAILURE);

   ifstream in(argv[1]);

   if (!in)
      exit(EXIT_FAILURE);

   int c, w, l;
```

Recipe 4.18. Counting Instances of Each Word in a Text File

Problem

You want to count the number of occurrences of each word in a text file.

Solution

Use operator>>, defined in <string>, to read contiguous chunks of text from the input file, and use a map, defined in <map>, to store each word and its frequency in the file. Example 4-27 demonstrates how to do this.

Example 4-27. Counting word frequencies

```
#include <iostream>
#include <fstream>
#include <map>
#include <string>

typedef std::map<std::string, int> StrIntMap;

void countWords(std::istream& in, StrIntMap& words) {

   std::string s;

   while (in >> s) {
      ++words[s];
   }
}

int main(int argc, char** argv) {

   if (argc < 2)
      return(EXIT_FAILURE);

   std::ifstream in(argv[1]);

   if (!in)
      exit(EXIT_FAILURE);

   StrIntMap w;
   countWords(in, w);

   for (StrIntMap::iterator p = w.begin( );
        p != w.end( ); ++p) {
      std::cout << p->first << " occurred "
                << p->second << " times.\n";
   }
}
```

Discussion

Example 4-27 looks simple enough, but there is more going on than it appears. Most of the subtleties have to do with maps, so let's talk about them first.

If you're not familiar with maps, you should be. A map is a container class template that is part of the STL. It stores key-value pairs in order according to std::less, or your custom comparison function, should

PREV NEXT


ABC Amber CHM Converter Trial version
Please register to remove this banner.
http://www.processtext.com/abcchm.html

Recipe 4.19. Add Margins to a Text File

Problem

Given a text file, you want to add margins to it. In other words, you want to pad either side of each line with some character so that each line is the same width.

Solution

Example 4-28 shows how to add margins to a file using streams, strings, and the getline function template.

Example 4-28. Adding margins to a text file

```
 #include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;
const static char PAD_CHAR = '.';

// addMargins takes two streams and two numbers. The streams are for
// input and output. The first of the two numbers represents the
// left margin width (i.e., the number of spaces to insert at the
// beginning of every line in the file). The second number represents
// the total line width to pad to.
void addMargins(istream& in, ostream& out,
                int left, int right) {

   string tmp;

   while (!in.eof( )) {
      getline(in, tmp, '\n');                    // getline is defined
                                                 // in <string>
      tmp.insert(tmp.begin( ), left, PAD_CHAR);
      rpad(tmp, right, PAD_CHAR);                // rpad from Recipe
                                                 // 4.2
      out << tmp << '\n';
   }
}

int main(int argc, char** argv) {

   if (argc < 3)
      return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
      return(EXIT_FAILURE);

   int left = 8;
   int right = 72;

   if (argc == 5) {
      left = atoi(argv[3]);
      right = atoi(argv[4]);
   }

   addMargins(in, out, left, right);
```

Recipe 4.20. Justify a Text File

Problem

You want to right- or left-justify text.

Solution

Use streams and the standard stream formatting flags right and left that are part of ios_base, defined in <ios>. Example 4-29 shows how they work.

Example 4-29. Justify text

```
 #include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
      return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   int w = 72;
   if (argc == 4)
      w = atoi(argv[3]);

   string tmp;
   out.setf(ios_base::right);    // Tell the stream to
                                 // right-justify
   while (!in.eof( )) {
      out.width(w);                   // Reset width after
      getline(in, tmp, '\n');         // each write
      out << tmp << '\n';
   }
   out.close( );
}
```

This example takes three arguments: an input file, an output file, and the width to right-justify to. You can use an input file like this:

```
 With automatic download of Microsoft's (Nasdaq:
MSFT) enormous SP2 security patch to the Windows
XP operating system set to begin, the industry
still waits to understand its ramifications. Home
users that have their preferences set to receive
operating-system updates as they are made
available by Microsoft may be surprised to learn
that some of the software they already run on
their systems could be disabled by SP2 or may run
very differently.
```

and make it look like this:

```
              With automatic download of Microsoft's (Nasdaq:
            MSFT) enormous SP2 security patch to the Windows
              XP operating system set to begin, the industry
           still waits to understand its ramifications. Home
```

Recipe 4.21. Squeeze Whitespace to Single Spaces in a Text File

Problem

You have a text file with whitespace of varying lengths in it, and you want to reduce every occurrence of a contiguous span of whitespace characters to a single space.

Solution

Use the operator>> function template, defined in <string>, to read in continuous chunks of non-whitespace from a stream into a string. Then use its counterpart, operator<<, to write each of these chunks to an output stream, and append a single character after each one. Example 4-30 gives a short example of this technique.

Example 4-30. Squeezing whitespace to single spaces

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

using namespace std;

int main(int argc, char** argv) {

   if (argc < 3)
      return(EXIT_FAILURE);

   ifstream in(argv[1]);
   ofstream out(argv[2]);

   if (!in || !out)
      return(EXIT_FAILURE);

   string tmp;

   in >> tmp;           // Grab the first word
   out << tmp;          // Dump it to the output stream

   while (in >> tmp) { // operator>> ignores whitespace, so all I have
      out << ' ';      // to do is add a space and each chunk of non-
      out << tmp;      // whitespace
   }

   out.close( );
}
```

Discussion

This is a simple thing to do if you take advantage of streams and strings. Even if you have to implement a variation of thisfor example, you may want to preserve new linesthe same facilities do the trick. If you want to add new lines, you can use the solution presented in Recipe 4.16 to insert them in the right place.

See Also

Recipe 4.15 and Recipe 4.16

Recipe 4.22. Autocorrect Text as a Buffer Changes

Problem

You have a class that represents some kind of text field or document, and as text is appended to it, you want to correct automatically misspelled words the way Microsoft Word's Autocorrect feature does.

Solution

Using a map, defined in <map>, strings, and a variety of standard library features, you can implement this with relatively little code. Example 4-31 shows how to do it.

Example 4-31. Autocorrect text

```
 #include <iostream>
#include <string>
#include <cctype>
#include <map>

using namespace std;

typedef map<string, string> StrStrMap;

// Class for holding text fields
class TextAutoField {

public:
   TextAutoField(StrStrMap* const p) : pDict_(p) {}
  ~TextAutoField( ) {}

   void append(char c);
   void getText(string& s) {s = buf_;}

private:
   TextAutoField( );
   string buf_;
   StrStrMap* const pDict_;
};

// Append with autocorrect
void TextAutoField::append(char c) {

   if ((isspace(c) || ispunct(c)) &&         // Only do the auto-
       buf_.length( ) > 0 &&                  // correct when ws or
       !isspace(buf_[buf_.length( ) - 1])) {  // punct is entered

      string::size_type i = buf_.find_last_of(" \f\n\r\t\v");

      i = (i == string::npos) ? 0 : ++i;

      string tmp = buf_.substr(i, buf_.length( ) - i);
      StrStrMap::const_iterator p = pDict_->find(tmp);

      if (p != pDict_->end( )) {                  // Found it, so erase
         buf_.erase(i, buf_.length( ) - i);       // and replace
         buf_ += p->second;
      }
   }
   buf_ += c;
}

int main( ) {
```

Recipe 4.23. Reading a Comma-Separated Text File

Problem

You want to read in a text file that is delimited by commas and new lines (or any other pair of delimiters for that matter). Records are delimited by one character, and fields within a record are delimited by another. For example, a comma-separated text file of employee information may look like the following:

```
 Smith, Bill, 5/1/2002, Active
Stanford, John, 4/5/1999, Inactive
```

Such files are usually interim storage for data sets exported from spreadsheets, databases, or other file formats.

Solution

See Example 4-32 for how to do this. If you read the text into strings one contiguous chunk at a time using getline (the function template defined in <string>) you can use the split function I presented in Recipe 4.6 to parse the text and put it in a data structure, in this case, a vector.

Example 4-32. Reading in a delimited file

```
 #include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

void split(const string& s, char c,
           vector<string>& v) {
   int i = 0;
   int j = s.find(c);

   while (j >= 0) {
      v.push_back(s.substr(i, j-i));
      i = ++j;
      j = s.find(c, j);

      if (j < 0) {
         v.push_back(s.substr(i, s.length( )));
      }
   }
}

void loadCSV(istream& in, vector<vector<string>*>& data) {

   vector<string>* p = NULL;
   string tmp;

   while (!in.eof( )) {
      getline(in, tmp, '\n');                    // Grab the next line

      p = new vector<string>( );
      split(tmp, ',', *p);                       // Use split from
                                                 // Recipe 4.7
      data.push_back(p);

      cout << tmp << '\n';
      tmp.clear( );
   }
}
```

Recipe 4.24. Using Regular Expressions to Split a String

Problem

You want to split a string into tokens, but you require more sophisticated searching or flexibility than Recipe 4.7 provides. For example, you may want tokens that are more than one character or can take on many different forms. This often results in code, and causes confusion in consumers of your class or function.

Solution

Use Boost's regex class template. regex enables the use of regular expressions on string and text data. Example 4-33 shows how to use regex to split strings.

Example 4-33. Using Boost's regular expressions

```
 #include <iostream>
#include <string>
#include <boost/regex.hpp>

int main( ) {

   std::string s = "who,lives:in-a,pineapple    under the sea?";

   boost::regex re(",|:|-|\\s+");         // Create the reg exp
   boost::sregex_token_iterator          // Create an iterator using a
     p(s.begin( ), s.end( ), re, -1);   // sequence and that reg exp
   boost::sregex_token_iterator end;     // Create an end-of-reg-exp
                                         // marker
   while (p != end)
      std::cout << *p++ << '\n';
}
```

Discussion

Example 4-33 shows how to use regex to iterate over matches in a regular expression. The following line sets up the regular expression:

```
 boost::regex re(",|:|-|\\s+");
```

What it says, essentially, is that each match of the regular expression is either a comma, or a colon, or a dash, or one or more spaces. The pipe character is the logical operator that ORs each of the delimiters together. The next two lines set up the iterator:

```
 boost::sregex_token_iterator
   p(s.begin( ), s.end( ), re, -1);
boost::sregex_token_iterator end;
```

The iterator p is constructed using the regular expression and an input string. Once that has been built, you can treat p like you would an iterator on a standard library sequence. A sregex_token_iterator constructed with no arguments is a special value that represents the end of a regular expression token sequence, and can therefore be used in a comparison to know when you hit the end.

PREV NEXT

Chapter 5. Dates and Times

PREV NEXT

Introduction

Dates and times are surprisingly vast and complex topics. As a reflection of this fact, the C++ standard library does not provide a proper date type. C++ inherits the structs and functions for date and time manipulation from C, along with a couple of date/time input and output functions that take into account localization. You can find relief, however, in the Boost date_time Library by Jeff Garland, which is possibly the most comprehensive and extensible date and time library for C++ available. I will be using it in several of the recipes. There is an expectation among the C++ community that future date/time extensions to the standard library will be based on the Boost date_time library.

The Boost date_time library includes two separate systems for manipulating dates and times: one for manipulating times and one for manipulating dates using a Gregorian calendar. The recipes will cover both systems.

For more information on dates and times, specifically reading and writing them, please see Chapter 13.

Gregorian Calendar and Leap Years

The Gregorian calendar is the most widely used calendar in the Western world today. The Gregorian calendar was intended to fix a flaw in the Julian calendar. The slow process of adoption of the Gregorian calendar started in 1582.

The Julian calendar dictates that every fourth year is a leap year, but every hundredth year is a non-leap year. The Gregorian calendar introduced a new exception that every 400 years should be a leap year.

Leap years are designed to compensate for the Earth's rotation around the sun being out of synchronization with the length of the day. In other words, dividing the length of a solar year, by the length of a day is an irrational number. The result is that if the calendar is not adjusted we would have seasonal drift, where the equinoxes and solstices (which determine the seasons) would become further out of synchronization with each new year.

Recipe 5.1. Obtaining the Current Date and Time

Problem

You want to retrieve the current date and time from the user's computer, either as a local time or as a Coordinated Universal Time (UTC).

Solution

Call the time function from the <ctime> header, passing a value of 0 as the parameter. The result will be a time_t value. You can use the gmtime function to convert the time_t value to a tm structure representing the current UTC time (a.k.a. Greenwich Mean Time or GMT); or, you can use the localtime function to convert the time_t value to a tm structure representing the local time. The program in Example 5-1 obtains the current date/time, and then converts it to local time and outputs it. Next, the program converts the current date/time to a UTC date/time and outputs that.

Example 5-1. Getting the local and UTC times

```
 #include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

int main( )
{
  // Current date/time based on current system
  time_t now = time(0);

  // Convert now to tm struct for local timezone
  tm* localtm = localtime(&now);
  cout << "The local date and time is: " << asctime(localtm) << endl;

  // Convert now to tm struct for UTC
  tm* gmtm = gmtime(&now);
  if (gmtm != NULL) {
     cout << "The UTC date and time is: " << asctime(gmtm) << endl;
  }
  else {
    cerr << "Failed to get the UTC date and time" << endl;
    return EXIT_FAILURE;
  }
}
```

Discussion

The time function returns a time_t type, which is an implementation-defined arithmetic type for representing a time period (a.k.a. a time interval) with at least a resolution of one second. The largest time interval that can be portably represented using a time_t is 2,147,483,648 seconds, or approximately 68 years.

A call to time(0) returns a time_t representing the time interval from an implementation defined base time (commonly 0:00:00 January 1, 1970) to the current moment.

The Year 2038 Bug

Recipe 5.2. Formatting a Date/Time as a String

Problem

You want to convert a date and/or time to a formatted string.

Solution

You can use the time_put template class from the <locale> header, as shown in Example 5-4.

Example 5-4. Formatting a datetime string

```
 #include <iostream>
#include <cstdlib>
#include <ctime>
#include <cstring>
#include <string>
#include <stdexcept>
#include <iterator>
#include <sstream>

using namespace std;

ostream& formatDateTime(ostream& out, const tm& t, const char* fmt) {
  const time_put<char>& dateWriter = use_facet<time_put<char> >(out.getloc(
));
  int n = strlen(fmt);
  if (dateWriter.put(out, out, ' ', &t, fmt, fmt + n).failed( )) {
    throw runtime_error("failure to format date time");
  }
  return out;
}

string dateTimeToString(const tm& t, const char* format) {
  stringstream s;
  formatDateTime(s, t, format);
  return s.str( );
}

tm now( ) {
  time_t now = time(0);
  return *localtime(&now);
}

int main( )
{
  try {
    string s = dateTimeToString(now( ), "%A %B, %d %Y %I:%M%p");
    cout << s << endl;
    s = dateTimeToString(now( ), "%Y-%m-%d %H:%M:%S");
    cout << s << endl;
  }
  catch(...) {
    cerr << "failed to format date time" << endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Output of the program in Example 5-4 will resemble the following, depending on your local settings:

```
 Sunday July, 24 2005 05:48PM
2005-07-24 17:48:11
```

Recipe 5.3. Performing Date and Time Arithmetic

Problem

You want to know the amount of time elapsed between two date/time points.

Solution

If both date/time points falls between the years of 1970 and 2038, you can use a time_t type and the difftime function from the <ctime> header. Example 5-6 shows how to compute the number of days elapsed between two dates.

Example 5-6. Date and time arithmetic with time_t

```
 #include <ctime>
#include <iostream>
#include <cstdlib>

using namespace std;

time_t dateToTimeT(int month, int day, int year) {
  // january 5, 2000 is passed as (1, 5, 2000)
  tm tmp = tm( );
  tmp.tm_mday = day;
  tmp.tm_mon = month - 1;
  tmp.tm_year = year - 1900;
  return mktime(&tmp);
}

time_t badTime( ) {
  return time_t(-1);
}

time_t now( ) {
  return time(0);
}

int main( ) {
  time_t date1 = dateToTimeT(1,1,2000);
  time_t date2 = dateToTimeT(1,1,2001);

  if ((date1 == badTime( )) || (date2 == badTime( ))) {
    cerr << "unable to create a time_t struct" << endl;
    return EXIT_FAILURE;
  }
  double sec = difftime(date2, date1);
  long days = static_cast<long>(sec / (60 * 60 * 24));
  cout << "the number of days between Jan 1, 2000, and Jan 1, 2001, is ";
  cout << days << endl;
  return EXIT_SUCCESS;
}
```

The program in Example 5-6 should output :

```
 the number of days between Jan 1, 2000, and Jan 1, 2001, is 366
```

Notice that the year 2000 is a leap year because even though it is divisible by 100; it is also divisible by 400, thus it has 366 days.

Discussion

Recipe 5.4. Converting Between Time Zones

Problem

You want to convert the current time from one time zone to another.

Solution

To convert between time zones, use the time zone conversion routines from the Boost date_time library. Example 5-8 shows how to finds the time in Tucson, Arizona given a time in New York City.

Example 5-8. Converting between time zones

```
 #include <iostream>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/local_time_adjustor.hpp>

using namespace std;
using namespace boost::gregorian;
using namespace boost::date_time;
using namespace boost::posix_time;

typedef local_adjustor<ptime, -5, us_dst> EasternTZ;
typedef local_adjustor<ptime, -7, no_dst> ArizonaTZ;

ptime NYtoAZ(ptime nytime) {
  ptime utctime = EasternTZ::local_to_utc(nytime);
  return ArizonaTZ::utc_to_local(utctime);
}

int main( )
{
   // May 1st 2004,
   boost::gregorian::date thedate(2004, 6, 1);
   ptime nytime(thedate, hours(19)); // 7 pm
   ptime aztime = NYtoAZ(nytime);
   cout << "On May 1st, 2004, when it was " << nytime.time_of_day( ).hours( );
   cout << ":00 in New York, it was " << aztime.time_of_day( ).hours( );
   cout << ":00 in Arizona " << endl;
}
```

The program in Example 5-8 outputs the following:

```
 On May 1st, 2004, when it was 19:00 in New York, it was 16:00 in Arizona
```

Discussion

The time zone conversions in Example 5-8 goes through a two-step process. First, I convert the time to UTC, and then convert the UTC time to the second time zone. Note that the time zones in the Boost date_time library are represented as types using the local_adjustor template class. Each type has conversion functions to convert from the given time zone to UTC (the local_to_utc function), and to convert from UTC to the given time zone (the utc_to_local function).

Recipe 5.5. Determining a Day's Number Within a Given Year

Problem

You want to determine a day's number within a given year. For example, January 1 is the first day of each year; February 5 is the 36th day of each year, and so on. But since some years have leap days, after February 28, a given day doesn't necessarily have the same numbering each year.

Solution

The solution to this problem requires the solution to several problems simultaneously. First, you have to know how many days are in each month, which, in turn, means you have to know how to determine whether a year is a leap year. Example 5-9 provides routines for performing these computations.

Example 5-9. Routines for determining a day's number within a given year

```
#include <iostream>

using namespace std;

enum MonthEnum  {
  jan = 0, feb = 1, mar = 2, apr = 3, may = 4, jun = 5,
  jul = 6, aug = 7, sep = 8, oct = 9, nov = 10, dec = 11
};

bool isLeapYear(int y) {
  return (y % 4 == 0) && ((y % 100 != 0) || (y % 400 == 0));
}

const int arrayDaysInMonth[] = {
  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

int n;
int arrayFirstOfMonth[] = {
  n = 0,
  n += arrayDaysInMonth[jan],
  n += arrayDaysInMonth[feb],
  n += arrayDaysInMonth[mar],
  n += arrayDaysInMonth[apr],
  n += arrayDaysInMonth[may],
  n += arrayDaysInMonth[jun],
  n += arrayDaysInMonth[jul],
  n += arrayDaysInMonth[aug],
  n += arrayDaysInMonth[sep],
  n += arrayDaysInMonth[::oct],
  n += arrayDaysInMonth[nov]
};

int daysInMonth(MonthEnum month, int year) {
  if (month == feb) {
    return isLeapYear(year) ? 29 : 28;
  }
  else {
    return arrayDaysInMonth[month];
  }
}

int firstOfMonth(MonthEnum month, int year) {
  return arrayFirstOfMonth[month] + isLeapYear(year);
```

Recipe 5.6. Defining Constrained Value Types

Problem

You want self-validating numerical types to represents numbers with a limited range of valid values such as hours of a day or minutes of an hour.

Solution

When working with dates and times, frequently you will want values that are integers with a limited range of valid values (i.e., 0 to 59 for seconds of a minute, 0 to 23 for hours of a day, 0 to 365 for days of a year). Rather than checking these values every time they are passed to a function, you would probably prefer to have them validated automatically by overloading the assignment operator. Since there are so many of these types, it is preferable to implement a single type that can handle this kind of validation for different numerical ranges. Example 5-10 presents a ConstrainedValue template class implementation that makes it easy to define ranged integers and other constrained value types.

Example 5-10. constrained_value.hpp

```
 #ifndef CONSTRAINED_VALUE_HPP
#define CONSTRAINED_VALUE_HPP

#include <cstdlib>
#include <iostream>

using namespace std;

template<class Policy_T>
struct ConstrainedValue
{
  public:
    // public typedefs
    typedef typename Policy_T policy_type;
    typedef typename Policy_T::value_type value_type;
    typedef ConstrainedValue self;

    // default constructor
    ConstrainedValue( ) : m(Policy_T::default_value) { }
    ConstrainedValue(const self& x) : m(x.m) {  }
    ConstrainedValue(const value_type& x) { Policy_T::assign(m, x); }
    operator value_type( ) const { return m; }

    // uses the policy defined assign function
    void assign(const value_type& x) {
      Policy_T::assign(m, x);
    }

    // assignment operations
    self& operator=(const value_type& x) { assign(x); return *this; }
    self& operator+=(const value_type& x) { assign(m + x); return *this; }
    self& operator-=(const value_type& x) { assign(m - x); return *this; }
    self& operator*=(const value_type& x) { assign(m * x); return *this; }
    self& operator/=(const value_type& x) { assign(m / x); return *this; }
    self& operator%=(const value_type& x) { assign(m % x); return *this; }
    self& operator>>=(int x) { assign(m >> x); return *this; }
    self& operator<<=(int x) { assign(m << x); return *this; }

    // unary operations
    self operator-( ) { return self(-m); }
    self operator+( ) { return self(-m); }
    self operator!( ) { return self(!m); }
```

Chapter 6. Managing Data with Containers

PREV NEXT

Introduction

This chapter describes the data structures in the standard library that you can use to store data. They are generally referred to as containers, since they "contain" objects you add to them. This chapter also describes another sort of container that is not part of the standard library, although it ships with most standard library implementations, namely the hashed container.

The part of the library that comprises the containers is often referred to as the Standard Template Library, or STL, because this is what it was called before it was included in the C++ standard. The STL includes not only the containers that are the subject of this chapter, but iterators and algorithms, which are the two other building blocks of the STL that make it a flexible, generic library. Since this chapter is primarily about the standard containers and not the STL in its entirety, I will refer to containers as the "standard containers" and not "STL containers," as is done in much of the C++ literature. Although I discuss iterators and algorithms as much as necessary here, both are discussed in more detail in Chapter 7.

The C++ standard uses precise terminology to describe its collection of containers. A "container" in the C++ standard library is a data structure that has a well-defined interface described in the standard. For example, any C++ standard library class that calls itself a container must support a member function begin that has no parameters and that returns an iterator referring to the first element in that container. There are a number of required constructors and member functions that define what it is to be a container in C++ terms. There are also optional member functions only some containers implement, usually those that can be implemented efficiently.

The set of all containers is further subdivided into two different kinds of containers: sequence containers and associative containers. A sequence container (usually just called a sequence) stores objects in an order that is specified by the user, and provides a required interface (in addition to container requirements) for accessing and manipulating the elements. Associative containers store their elements in sorted order, and therefore do not permit you to insert elements at a specific location, although you can provide hints when you insert to improve efficiency. Both sequences and associative containers have a required interface they must support, but only sequences have an additional set of operations that are only supported by sequences for which they can be implemented efficiently. These additional sequence operations provide more flexibility and convenience than the required interface.

This sounds a lot like inheritance. A sequence is a container, an associative container is a container, but a container is not a sequence or an associative container. It's not inheritance, though, in the C++ sense, but it is inheritance conceptually. A vector is a sequence, but it is its own, standalone class; it doesn't inherit from a container class or some such thing (standard library implementations are allowed freedom in how they implement vector and other containers, but the standard doesn't mandate that a standard library implementation include a container base class). A great deal of thought went into the design of the containers, and if you would like to read more about it go pick up Matt Austern's Generic Programming and the STL (Addison Wesley).

This chapter has two parts. The first few recipes describe how to use vector, which is a standard sequence, since it is one of the more popular data structures. They describe how to use a vector effectively and efficiently. The rest of the recipes discuss most of the other standard containers that are widely applicable, including the two nonstandard hashed containers I mentioned earlier.

Recipe 6.1. Using vectors Instead of Arrays

Problem

You have to store things (built-in types, objects, pointers, etc.) in a sequence, you require random access to elements, and you can't be confined to a statically sized array.

Solution

Use the standard library's vector class template, which is defined in <vector>; don't use arrays. vector looks and feels like an array, but it has a number of safety and convenience advantages over arrays. Example 6-1 shows a few common vector operations.

Example 6-1. Using common vector member functions

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

using namespace std;

int main( ) {

   vector<int>    intVec;
   vector<string> strVec;

   // Add elements to the "back" of the vector with push_back
   intVec.push_back(3);
   intVec.push_back(9);
   intVec.push_back(6);

   string s = "Army";

   strVec.push_back(s);
   s = "Navy";
   strVec.push_back(s);
   s = "Air Force";
   strVec.push_back(s);

   // You can access them with operator[], just like an array
   for (vector<string>::size_type i = 0; i < intVec.size( ); ++i) {
      cout << "intVec[" << i << "] = " << intVec[i] << '\n';
   }

   // Or you can use iterators
   for (vector<string>::iterator p = strVec.begin( );
        p != strVec.end( ); ++p) {
      cout << *p << '\n';
   }

   // If you need to be safe, use at( ) instead of operator[].  It
   // will throw out_of_range if the index you use is > size( ).
   try {
      intVec.at(300) = 2;
   }
   catch(out_of_range& e) {
      cerr << "out_of_range: " << e.what( ) << endl;
   }
}
```

Discussion

Recipe 6.2. Using vectors Efficiently

Problem

You are using vectors and you have tight space or time requirements and need to reduce or eliminate overhead.

Solution

Understand how a vector is implemented, know the complexity of insertion and deletion member functions, and minimize unnecessary memory churn with the reserve member function. Example 6-2 shows a few of these techniques in action.

Example 6-2. Using a vector efficiently

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

using std::vector;
using std::string;

void f(vector<string>& vec) { // Pass vec by reference (or
                              // pointer, if you have to)
   // ...
}

int main( ) {

   vector<string> vec(500); // Tell the vector that you plan on
                            // putting a certain number of objects
                            // in it at construction
   vector<string> vec2;

   // Fill up vec...
   f(vec);

   vec2.reserve(500);       // Or, after the fact, tell the vector
                            // that you want the buffer to be big
                            // enough to hold this many objects

   // Fill up vec2...
}
```

Discussion

The key to using vectors efficiently lies in knowing how they work. Once you have a good idea of how a vector is usually implemented, the performance hot spots become obvious.

How vectors work

A vector is, essentially, a managed array. More specifically, a vector<T> is a chunk of contiguous memory (i.e., an array) that is large enough to hold n objects of type T, where n is greater than or equal to zero and is less or equal to an implementation-defined maximum size. n usually increases during the lifetime of the container as you add or remove elements, but it doesn't decrease. What makes a vector different from an array is the automatic memory management of that array, the member functions for inserting and retrieving elements, and the member functions that provide metadata about the container, such as the size (number of elements) and capacity (the buffer size), but also the type information: vector<T>::value_type is T's type, vector<T>::pointer is a pointer-to-T type, and so on. These last two

Recipe 6.3. Copying a vector

Problem

You need to copy the contents of one vector into another.

Solution

There are a couple of ways to do this. You can use a copy constructor when you create a vector, or you can use the assign member function. Example 6-3 shows how to do both.

Example 6-3. Copying vector contents

```
 #include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// Util function for printing vector contents
template<typename T>
void vecPrint (const vector<T>& vec) {
   cout << "{";
   for (typename vector<T>::const_iterator p = vec.begin( );
       p != vec.end( ); ++p) {
      cout << "{" << *p << "} ";
   }
   cout << "}" << endl;
}

int main( ) {

   vector<string> vec(5);
   string foo[] = {"My", "way", "or", "the", "highway"};

   vec[0] = "Today";
   vec[1] = "is";
   vec[2] = "a";
   vec[3] = "new";
   vec[4] = "day";

   vector<string> vec2(vec);
   vecPrint(vec2);

   vec.at(0) = "Tomorrow";

   vec2.assign(vec.begin( ), vec.end( )); // Copy each element over
   vecPrint(vec2);                        // with assign

   vec2.assign(&foo[0], &foo[5]); // Assign works for anything that
   vecPrint(vec2);                // behaves like an iterator

   vector<string>::iterator p;

   p = find(vec.begin( ), vec.end( ), "new");

   vec2.assign(vec.begin( ), p); // Copy a subset of the full range
   vecPrint(vec2);               // of vec
}
```


Discussion

Recipe 6.4. Storing Pointers in a vector

Problem

For efficiency or other reasons, you can't store copies of your objects in a vector, but you need to keep track of them somehow.

Solution

Store pointers to your objects in a vector instead of copies of the objects themselves. But if you do, don't forget to delete the objects that are pointed to, because the vector won't do it for you. Example 6-4 shows how to declare and work with vectors of pointers.

Example 6-4. Using vectors of pointers

```
 #include <iostream>
#include <vector>

using namespace std;

static const int NUM_OBJECTS = 10;

class MyClass { /*...*/ };

int main( ) {

   vector<MyClass*> vec;

   MyClass* p = NULL;

   // Load up the vector with MyClass objects
   for (int i = 0; i < NUM_OBJECTS; i++) {
      p = new MyClass( );
      vec.push_back(p);
   }

   // Do something useful with this data, then delete the objects when
   // you're done
   for (vector<MyClass*>::iterator pObj = vec.begin( );
        pObj != vec.end( ); ++pObj) {
      delete *pObj; // Note that this is deleting what pObj points to,
                    // which is a pointer
   }

   vec.clear( ); // Purge the contents so no one tries to delete them
                 // again
}
```

Discussion

You can store pointers in a vector just like you would anything else. Declare a vector of pointers like this:

```
 vector<MyClass*> vec;
```

The important thing to remember is that a vector stores values without regard for what those values represent. It, therefore, doesn't know that it's supposed to delete pointer values when it's destroyed. If you allocate memory, then put pointers to that memory in a vector, you have to delete the memory yourself when you are done with it. Don't be fooled by the term "container" into thinking that somehow when you store a pointer in a vector that it assumes ownership.

Recipe 6.5. Storing Objects in a list

Problem

You need to store items in a sequence, but your requirements don't match up well with a vector. Specifically, you need to be able to efficiently add and remove items in the middle of the sequence, not just at the end.

Solution

Use a list, declared in <list>, to hold your data. lists offer better performance and more flexibility when modifying the sequence at someplace other than the beginning or the end. Example 6-5 shows you how to use a list, and shows off some of its unique operations.

Example 6-5. Using a list

```
 #include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace std;

// A simple functor for printing
template<typename T>
struct printer {
   void operator( )(const T& s) {
      cout << s << '\n';
   }
};

bool inline even(int n) {
   return(n % 2 == 0);
}

printer<string> strPrinter;
printer<int>    intPrinter;

int main( ) {

   list<string> lstOne;
   list<string> lstTwo;

   lstOne.push_back("Red");
   lstOne.push_back("Green");
   lstOne.push_back("Blue");

   lstTwo.push_front("Orange");
   lstTwo.push_front("Yellow");
   lstTwo.push_front("Fuschia");

   for_each(lstOne.begin( ), // Print each element in the list
            lstOne.end( ),   // with a custom functor, print
            strPrinter);

   lstOne.sort( );           // list has a member for sorting
   lstTwo.sort( );

   lstOne.merge(lstTwo);    // Merge the two lists and print
   for_each(lstOne.begin( ), // the results (the lists must be
            lstOne.end( ),   // sorted before merging)
            strPrinter);
```

Recipe 6.6. Mapping strings to Other Things

Problem

You have objects that you need to store in memory, and you want to store them by their string keys. You need to be able to add, delete, and retrieve items quickly (with, at most, logarithmic complexity).

Solution

Use the standard container map, declared in <map>, to map keys (strings) to values (any type that obeys value semantics). Example 6-6 shows how.

Example 6-6. Creating a string map

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

using namespace std;

int main( ) {

   map<string, string> strMap;

   strMap["Monday"]    = "Montag";
   strMap["Tuesday"]   = "Dienstag";
   strMap["Wednesday"] = "Mittwoch";
   strMap["Thursday"]  = "Donnerstag";
   strMap["Friday"]    = "Freitag";
   strMap["Saturday"]  = "Samstag";
   // strMap.insert(make_pair("Sunday", "Sonntag"));
   strMap.insert(pair<string, string>("Sunday", "Sonntag"));

   for (map<string, string>::iterator p = strMap.begin( );
      p != strMap.end( ); ++p ) {
         cout << "English: " << p->first
              << ", German: " << p->second << endl;
   }

   cout << endl;

   strMap.erase(strMap.find("Tuesday"));

   for (map<string, string>::iterator p = strMap.begin( );
      p != strMap.end( ); ++p ) {
         cout << "English: " << p->first
              << ", German: " << p->second << endl;
   }
}
```

Discussion

A map is an associative container that maps keys to values, provides logarithmic complexity for inserting and finding, and constant time for erasing single elements. It is common for developers to use a map to keep track of objects by using a string key. This is what Example 6-6 does; in this case, the mapped type happens to be a string, but it could be nearly anything.

A map is declared like this:

```
map<typename Key,                          // The type of the key
   typename Value,                        // The type of the value
```

Recipe 6.7. Using Hashed Containers

Problem

You are storing keys and values , you need constant-time access to elements, and you don't need the elements to be stored in sorted order.

Solution

Use one of the hashed associated containers, hash_map or hash_set. Be aware, however, that these are not standard containers specified by the C++ Standard, rather they are extensions that most standard library implementations include. Example 6-8 shows how to use a hash_set.

Example 6-8. Storing strings in a hash_set

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

int main( ) {

   hash_set<std::string> hsString;
   string s = "bravo";

   hsString.insert(s);
   s = "alpha";
   hsString.insert(s);
   s = "charlie";
   hsString.insert(s);

   for (hash_set<string>::const_iterator p = hsString.begin( );
        p != hsString.end( ); ++p)
      cout << *p << endl; // Note that these aren't guaranteed
                          // to be in sorted order
}
```

Discussion

Hashed containers are popular data structures in any language, and it is unfortunate that C++ Standard does not require an implementation to supply them. All is not lost, however, if you want to use a hashed container: chances are that the standard library implementation you are using includes hash_map and hash_set, but the fact that they are not standardized means their interfaces may differ from one standard library implementation to the next. I will describe the hashed containers that are provided in the STLPort standard library implementation.

STLPort is a free, portable standard library implementation that has been around for a long time and provides hashed containers. If you are using a different library, the interface may be different, but the general idea is the same.

The main characteristics of hashed containers (called hashed associative containers by much of the C++ literature) are that they provide, in the average case, constant-time location, insertion, and deletion of elements; in the worst case, operations require linear complexity. The trade-off for all of these constant-time operations is that the elements in a hashed container are not stored in order, as they are in a map.

Recipe 6.8. Storing Objects in Sorted Order

Problem

You have to store a set of objects in order, perhaps because you frequently need to access ordered ranges of these objects and you don't want to pay for resorting them each time you do this.

Solution

Use the associative container set, declared in <set>, which stores items in sorted order. It uses the standard less class template, (which invokes operator< on its arguments) by default, or you can supply your own sorting predicate. Example 6-10 shows how to store strings in a set.

Example 6-10. Storing strings in a set

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

using namespace std;

int main( ) {

   set<string> setStr;
   string s = "Bill";

   setStr.insert(s);
   s = "Steve";
   setStr.insert(s);
   s = "Randy";
   setStr.insert(s);
   s = "Howard";
   setStr.insert(s);

   for (set<string>::const_iterator p = setStr.begin( );
        p != setStr.end( ); ++p)
      cout << *p << endl;
}
```

Since the values are stored in sorted order, the output will look like this:

```
 Bill
Howard
Randy
Steve
```

Discussion

A set is an associative container that provides logarithmic complexity insertion and find, and constant-time deletion of elements (once you have found the element you want to delete). sets are unique associative containers, which means that no two elements can be equivalent, though you can use a multiset if you need to store multiple instances of equivalent elements. You can think of a set as a set in the mathematical sense, that is, a collection of items, with the added bonus that order is maintained among the elements.

You can insert and find elements, but, like a list, a set does not allow random access to elements. If you want something in a set, you have to look for it with the find member function, or iterate through the elements using set<T>::iterator or set<T>::const_iterator.

The declaration of a set should look familiar:

Recipe 6.9. Storing Containers in Containers

Problem

You have a number of instances of a standard container (lists, sets, etc.), and you want to keep track of them by storing them in yet another container.

Solution

Store pointers to your containers in a single, master container. For example, you can use a map to store a string key and a pointer to a set as its value. Example 6-12 presents a simple transaction log class that stores its data as a map of string-set pointer pairs.

Example 6-12. Storing set pointers in a map

```
 #include <iostream>
#include <set>
#include <map>
#include <string>

using namespace std;

typedef set<string> SetStr;
typedef map<string, SetStr*> MapStrSetStr;

// Dummy database class
class DBConn {
public:
   void beginTxn( ) {}
   void endTxn( ) {}
   void execSql(string& sql) {}
};

class SimpleTxnLog {

public:
   SimpleTxnLog( ) {}
  ~SimpleTxnLog( ) {purge( );}

   // Add an SQL statement to the list
   void addTxn(const string& id,
               const string& sql) {
      SetStr* pSet = log_[id];      // This creates the entry for
      if (pSet == NULL) {           // this id if it isn't there
         pSet = new SetStr( );
         log_[id] = pSet;
      }
      pSet->insert(sql);
   }

   // Apply the SQL statements to the database, one transaction
   // at a time
   void apply( ) {
      for (MapStrSetStr::iterator p = log_.begin( );
           p != log_.end( ); ++p) {
         conn_->beginTxn( );

         // Remember that a map iterator actually refers to an object
         // of pair<Key,Val>.  The set pointer is stored in p->second.
         for (SetStr::iterator pSql = p->second->begin( );
              pSql != p->second->end( ); ++pSql) {
            string s = *pSql;
```

Chapter 7. Algorithms

PREV NEXT

Introduction

This chapter describes how to work with the standard algorithms and how to use them on the standard containers. These algorithms were originally part of what is often referred to as the Standard Template Library (STL), which is the set of algorithms, iterators, and containers that now belong to the standard library (Chapter 6 contains recipes for working with the standard containers). I will refer to these simply as the standard algorithms, iterators, and containers, but keep in mind that they are the same ones that other authors' refer to as part of the STL. One of the pillars of the standard library is iterators, so the first recipe explains what they are and how to use them. After that, there are a number of recipes that explain how to use and extend the standard algorithms. Finally, if what you need isn't in the standard library, Recipe 7.10 explains how to write your own algorithm.

The recipes presented here are largely biased toward working with the standard containers for two reasons. First, the standard containers are ubiquitous, and it's better to learn the standard than to reinvent the wheel. Second, the algorithms in the standard library implementations provide a good model to follow for interoperability and performance. If you watch how the pros do it in the standard library code, you are likely to learn a few valuable lessons along the way.

All standard algorithms use iterators. Even if you are already familiar with the concept of iterators, which is the subject of the first recipe, take a look at Table 7-1, which contains a list of the conventions I use in the rest of the chapter when listing function declarations for the standard algorithms.

Table 7-1. Iterator category abbreviations

Abbreviation	Meaning
In	Input iterator
Out	Output iterator
Fwd	Forward iterator
Bid	Bidirectional iterator
Rand	Random-access iterator

The standard algorithms also make use of function objects, or functors. A function object is a class that has overridden operator( ) so that it can be called like a function. A functor that returns a bool (and does not maintain state, and is therefore called pure) is called a predicate, and they are another regular feature in the standard algorithms. Generally, a predicate takes one or two arguments: if it takes one argument, it is an unary predicate; and if it takes two, it is called a binary predicate. For the sake of brevity, I use the abbreviations listed in Table 7-2 when listing function declarations.

Table 7-2. Functor types

Type Name	Description
UnPred	An unary predicate. Takes one argument and returns a bool

Recipe 7.1. Iterating Through a Container

Problem

You have a range of iteratorsmost likely from a standard containerand the standard algorithms don't fit your needs, so you need to iterate through them.

Solution

Use an iterator or a const_iterator to access and advance through each of the elements in your container. In the standard library, algorithms and containers communicate using iterators, and one of the very ideas of the standard algorithms is that they insulate you from having to use iterators directly unless you are writing your own algorithm. Even so, you should understand the different kinds of iterators so you can use the standard algorithms and containers effectively. Example 7-1 presents some straightforward uses of iterators.

Example 7-1. Using iterators with containers

```
 #include <iostream>
#include <list>
#include <algorithm>
#include <string>

using namespace std;

static const int ARRAY_SIZE = 5;

template<typename T,
         typename FwdIter>
FwdIter fixOutliersUBound(FwdIter p1,
                          FwdIter p2,
                          const T& oldVal,
                          const T& newVal) {
   for ( ;p1 != p2; ++p1) {
      if (greater<T>(*p1, oldVal)) {
         *p1 = newVal;
      }
   }
}

int main( ) {

   list<string> lstStr;

   lstStr.push_back("Please");
   lstStr.push_back("leave");
   lstStr.push_back("a");
   lstStr.push_back("message");

   // Create an iterator for stepping through the list
   for (list<string>::iterator p = lstStr.begin( );
        p != lstStr.end( ); ++p) {
      cout << *p << endl;
   }

   // Or I can use a reverse_iterator to go from the end
   // to the beginning.  rbegin returns a reverse_iterator
   // to the last element and rend returns a reverse_iterator
   // to one-before-the-first.
   for (list<string>::reverse_iterator p = lstStr.rbegin( );
        p != lstStr.rend( ); ++p) {
```

Recipe 7.2. Removing Objects from a Container

Problem

You want to remove objects from a container.

Solution

Use the container's erase member function to erase a single element or a range of elements, and possibly use one of the standard algorithms to make the job easier. Example 7-2 shows a couple of different ways to remove elements from a sequence.

Example 7-2. Removing elements from a container

```
 #include <iostream>
#include <string>
#include <list>
#include <algorithm>
#include <functional>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

int main( ) {

   list<string> lstStr;

   lstStr.push_back("On");
   lstStr.push_back("a");
   lstStr.push_back("cloudy");
   lstStr.push_back("cloudy");
   lstStr.push_back("day");

   list<string>::iterator p;

   // Find what you want with find
   p = find(lstStr.begin( ), lstStr.end( ), "day");

   p = lstStr.erase(p); // Now p points to the last element

   // Or, to erase all occurrences of something, use remove
   lstStr.erase(remove(lstStr.begin( ), lstStr.end( ), "cloudy"),
                lstStr.end( ));
   printContainer(lstStr); // See 7.10
}
```

Discussion

Use a container's erase member function to remove one or more elements from it. All containers have two overloads of erase: one that takes a single iterator argument that points to the element you want to delete, and another that takes two iterators that represent a range of elements you want deleted. To erase a single element, obtain an iterator referring to that element and pass the iterator to erase, as in Example 7-2:

```
 p = find(lstStr.begin( ), lstStr.end( ), "day");
p = lstStr.erase(p);
```

This will delete the object that p refers to by calling its destructor, and then do any necessary reorganization of the remaining elements in the range. The reorganization that happens depends on the type of container, and therefore the complexity of the operation will vary from one kind of container to another. The signature and behavior also differs slightly when you are using a sequence container versus

Recipe 7.3. Randomly Shuffling Data

Problem

You have a sequence of data, and you need to jumble it into some random order.

Solution

Use the random_shuffle standard algorithm, defined in <algorithm>. random_shuffle takes two random-access iterators, and (optionally) a random-number generation functor, and rearranges the elements in the range at random. Example 7-3 shows how to do this.

Example 7-3. Shuffling a sequence at random

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

int main( ) {

   vector<int> v;
   back_insert_iterator<std::vector<int> > p =
      back_inserter(v);

   for (int i = 0; i < 10; ++i)
     *p = i;

   printContainer(v, true);

   random_shuffle(v.begin( ), v.end( ));

   printContainer(v, true);
}
```

Your output might look like this:

```
-----
0 1 2 3 4 5 6 7 8 9
-----
8 1 9 2 0 5 7 3 4 6
```

Discussion

random_shuffle is intuitive to use. Give it a range, and it will shuffle the range at random. There are two versions, and their prototypes look like this:

```
void random_shuffle(RndIter first, RndIter last);
void random_shuffle(RndIter first, RndIter last, RandFunc& rand);
```

In the first version, the "random" is using an implementation-specific random-number generation function, which should be sufficient for most of your needs. If it isn'tperhaps you want a nonuniform distribution, e.g., Gaussianyou can write your own and supply that instead using the second version.

Your random-number generator must be a functor that a single argument and returns a single value, both of which are convertible to iterator_traits<RndIter>::difference_type. In most cases, an integer will do. For example, here's my knock-off random-number generator:

```
struct RanNumGenFtor {
```

Recipe 7.4. Comparing Ranges

Problem

You have two ranges, and you need to compare them for equality or you need to see which one comes first based on some ordering on the elements.

Solution

Depending on what kind of comparison you want to do, use one of the standard algorithms equal, lexicographical_compare, or mismatch, defined in <algorithm>. Example 7-4 shows several of them in action.

Example 7-4. Different kinds of comparisons

```
 #include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include "utils.h"

using namespace std;
using namespace utils;

int main( ) {

   vector<string> vec1, vec2;

   vec1.push_back("Charles");
   vec1.push_back("in");
   vec1.push_back("Charge");

   vec2.push_back("Charles");
   vec2.push_back("in");
   vec2.push_back("charge");  // Note the small "c"

   if (equal(vec1.begin( ), vec1.end( ), vec2.begin( ))) {
      cout << "The two ranges are equal!" << endl;
   } else {
      cout << "The two ranges are NOT equal!" << endl;
   }

   string s1 = "abcde";
   string s2 = "abcdf";
   string s3 = "abc";

   cout << boolalpha  // Show bools as "true" or "false"
        << lexicographical_compare(s1.begin( ), s1.end( ),
                                   s1.begin( ), s1.end( )) << endl;
   cout << lexicographical_compare(s1.begin( ), s1.end( ),
                                   s2.begin( ), s2.end( )) << endl;
   cout << lexicographical_compare(s2.begin( ), s2.end( ),
                                   s1.begin( ), s1.end( )) << endl;
   cout << lexicographical_compare(s1.begin( ), s1.end( ),
                                   s3.begin( ), s3.end( )) << endl;
   cout << lexicographical_compare(s3.begin( ), s3.end( ),
                                   s1.begin( ), s1.end( )) << endl;

   pair<string::iterator, string::iterator> iters =
      mismatch(s1.begin( ), s1.end( ), s2.begin( ));

   cout << "first mismatch  = " << *(iters.first) << endl;
```

Recipe 7.5. Merging Data

Problem

You have two sorted sequences and you need to merge them.

Solution

Use either the merge or inplace_merge function template. merge merges two sequences and puts the results in a third, and inplace_merge merges two contiguous sequences. Example 7-5 shows how.

Example 7-5. Merging two sequences

```
 #include <iostream>
#include <string>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

int main( ) {

   vector<string> v1, v2, v3;

   v1.push_back("a");
   v1.push_back("c");
   v1.push_back("e");

   v2.push_back("b");
   v2.push_back("d");
   v2.push_back("f");

   v3.reserve(v1.size( ) + v2.size( ) + 1);

   // Use a back_inserter from iterator to avoid having to put
   // a bunch of default objects in the container.  But this doesn't
   // mean you don't have to use reserve!
   merge(v1.begin( ), v1.end( ),
         v2.begin( ), v2.end( ),
         back_inserter<vector<string> >(v3));

   printContainer(v3);

   // Now make a mess
   random_shuffle(v3.begin( ), v3.end( ));
   sort(v3.begin( ), v3.begin( ) + v3.size( ) / 2);
   sort(v3.begin( ) + v3.size( ) / 2, v3.end( ));

   printContainer(v3);

   inplace_merge(v3.begin( ), v3.begin( ) + 3, v3.end( ));

   printContainer(v3);

   // If you are using two lists, though, use list::merge instead.
   // As a general rule, blah blah...
   list<string> lstStr1, lstStr2;

   lstStr1.push_back("Frank");
```

Recipe 7.6. Sorting a Range

Problem

You have a range of elements that you need to sort.

Solution

There are a handful of algorithms you can use for sorting a range. You can do a conventional sort (ascending or descending order) with sort, defined in <algorithm>, or you can use one of the other sorting functions, such as partial_sort . Have a look at Example 7-6 to see how.

Example 7-6. Sorting

```
 #include <iostream>
#include <istream>
#include <string>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

int main( ) {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   // The sort standard algorithm will sort elements in a range.  It
   // requires a random-access iterator, so it works for a vector.
   sort(v.begin( ), v.end( ));
   printContainer(v);

   random_shuffle(v.begin( ), v.end( )); // See 7.2

   string* arr = new string[v.size( )];

   // Copy the elements into the array
   copy(v.begin( ), v.end( ), &arr[0]);

   // Sort works on any kind of range, so long as its arguments
   // behave like random-access iterators.
   sort(&arr[0], &arr[v.size( )]);
   printRange(&arr[0], &arr[v.size( )]);

   // Create a list with the same elements
   list<string> lst(v.begin( ), v.end( ));

   lst.sort( ); // The standalone version of sort won't work; you have
                // to use list::sort.  Note, consequently, that you
                // can't sort only parts of a list.

   printContainer(lst);
}
```

A run of Example 7-6 might look like this:

```
 Enter a series of strings: a z b y c x d w
^Z
```

Recipe 7.7. Partitioning a Range

Problem

You have a range of elements that you need to partition in some well-defined way. For example, you may want all elements less than a particular value moved to the front of the range.

Solution

Use the partition standard algorithm with a predicate functor to move the elements around however you like. See Example 7-7.

Example 7-7. Partitioning a range

```
#include <iostream>
#include <istream>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
#include <iterator>
#include "utils.h" // For printContainer( ): see Recipe 7.10

using namespace std;

int main( ) {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   // Rearrange the elements in v so that those that are less
   // than "foo" occur before the rest.
   vector<string>::iterator p =
      partition(v.begin( ), v.end( ),
                bind2nd(less<string>( ), "foo"));
   printContainer(v);

   cout << "*p = " << *p << endl;
}
```

The output for Example 7-7 would look like the following:

```
Enter a series of strings: a d f j k l
^Z
-----
a d f j k l
*p = j
```

After the partition, the iterator p refers to the first element for which less(*p, "foo") is not true.

Discussion

partition takes the beginning and end of a range and a predicate, and moves all elements for which the predicate is true to the beginning of the range. It returns an iterator to the first element where the predicate is not TRue, or the end of the range if all elements satisfy the predicate. Its declaration looks like this:

```
Bi partition(Bi first, Bi last, Pred pred);
```


pred is a functor that takes one argument and returns TRue or false. There is no default predicate; you

Recipe 7.8. Performing Set Operations on Sequences

Problem

You have sequences that you want to rearrange using set operations like union, difference, or intersection.

Solution

Use the standard library functions built for exactly this purpose: set_union , set_dif-ference , and set_intersection . Each of these performs its respective set operation and places the results in an output range. See how to do this in Example 7-8.

Example 7-8. Using set operations

```
 #include <iostream>
#include <algorithm>
#include <string>
#include <set>
#include <iterator>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

int main( ) {

   cout << "Enter some strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   set<string> s1(start, end);

   cin.clear( );

   cout << "Enter some more strings: ";
   set<string> s2(++start, end);

   set<string> setUnion;
   set<string> setInter;
   set<string> setDiff;

   set_union(s1.begin( ), s1.end( ),
             s2.begin( ), s2.end( ),
             inserter(setUnion, setUnion.begin( )));

   set_difference(s1.begin( ), s1.end( ),
                  s2.begin( ), s2.end( ),
                  inserter(setDiff, setDiff.begin( )));

   set_intersection(s1.begin( ), s1.end( ),
                    s2.begin( ), s2.end( ),
                    inserter(setInter, setInter.begin( )));

   cout << "Union:\n";
   printContainer(setUnion);
   cout << "Difference:\n";
   printContainer(setDiff);
   cout << "Intersection:\n";
   printContainer(setInter);
}
```

The output to this program looks like this (printContainer just prints the contents of a container):

Recipe 7.9. Transforming Elements in a Sequence

Problem

You have a sequence of elements and you have to do something to each one, either in place or as it is copied to another sequence.

Solution

Use the transform or for_each standard algorithms. Both are simple, but allow you to do almost anything you want to the elements in your sequence. See Example 7-9 for an illustration.

Example 7-9. Transforming data

```
 #include <iostream>
#include <istream>
#include <string>
#include <list>
#include <algorithm>
#include <iterator>
#include <cctype>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

// Convert a string to upper case
string strToUpper(const string& s) {
   string tmp;
   for (string::const_iterator p = s.begin( ); p != s.end( ); ++p)
      tmp += toupper(*p);
   return(tmp);
}

string strAppend(const string& s1, const string& s2) {
   return(s1 + s2);
}

int main( ) {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   list<string> lst(start, end), out;

   // Use transform with an unary function...
   transform(lst.begin( ), lst.end( ),
             back_inserter(out),
             strToUpper);

   printContainer(out);

   cin.clear( );

   cout << "Enter another series of strings: ";
   list<string> lst2(++start, end);
   out.clear( );

   // ...or a binary function and another input sequence.
   transform(lst.begin( ), lst.end( ), lst2.begin( ),
             back_inserter(out),
             strAppend);
```

Recipe 7.10. Writing Your Own Algorithm

Problem

You need to execute an algorithm on a range and none of the standard algorithms meets your requirements.

Solution

Write your algorithm as a function template and advertise your iterator requirements with the names of your template parameters. See Example 7-10 for a variation on the copy standard algorithm.

Example 7-10. Writing your own algorithm

```
 #include <iostream>
#include <istream>
#include <iterator>
#include <string>
#include <functional>
#include <vector>
#include <list>
#include "utils.h" // For printContainer( ): see 7.10

using namespace std;

template<typename In, typename Out, typename UnPred>
Out copyIf(In first, In last, Out result, UnPred pred) {
   for ( ;first != last; ++first)
      if (pred(*first))
         *result++ = *first;
   return(result);
}

int main( ) {

   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end; // This creates a "marker"
   vector<string> v(start, end);

   list<string> lst;

   copyIf(v.begin( ), v.end( ), back_inserter<list<string> >(lst),
      bind2nd(less<string>( ), "cookie"));

   printContainer(lst);
}
```

A sample run of Example 7-10 will look something like this:

```
 Enter a series of strings: apple banana danish eclaire
^Z
-----
apple banana
```

You can see that it only copies values less than "cookie" into the destination range.

Discussion

The standard library contains the copy function template, which copies elements from one range to another, but there is no standard version that takes a predicate and conditionally copies each element

Recipe 7.11. Printing a Range to a Stream

Problem

You have a range of elements that you want to print to a stream, most likely cout for debugging.

Solution

Write a function template that takes a range or a container, iterates through each element, and uses the copy algorithm and an ostream_iterator to write each element to a stream. If you want more control over formatting, write your own simple algorithm that iterates through a range and prints each element to the stream. (See Example 7-11.)

Example 7-11. Printing a range to a stream

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;

int main( ) {

   // An input iterator is the opposite of an output iterator: it
   // reads elements from a stream as if it were a container.
   cout << "Enter a series of strings: ";
   istream_iterator<string> start(cin);
   istream_iterator<string> end;
   vector<string> v(start, end);

   // Treat the output stream as a container by using an
   // output_iterator.  It constructs an output iterator where writing
   // to each element is equivalent to writing it to the stream.
   copy(v.begin( ), v.end( ), ostream_iterator<string>(cout, ", "));
}
```

The output for Example 7-11 might look like this:

```
Enter a series of strings: z x y a b c
^Z
z, x, y, a, b, c,
```

Discussion

A stream iterator is an iterator that is based on a stream instead of a range of elements in some container, and stream iterators allow you to treat stream input as an input iterator (read from the dereferenced value and increment the iterator) or an output iterator (just like an input iterator, but you write to its dereferenced value instead of read from it). This makes for concise reading of values (especially strings) from a stream, which is what I have done in a number of other examples in this chapter, and writing values to a stream, which is what I have done in Example 7-11. I know this recipe is about printing a range to a stream, but allow me to stray from the path for a moment to explain input stream iterators since I use them in so many examples in this chapter.

There are three key parts to the istream_iterator in Example 7-11. The first part is creating the istream_iterator that refers to the start of the stream input. I do it like this:

```
istream_iterator<string> start(cin);
```

Chapter 8. Classes

Introduction

This chapter contains solutions to common problems related to working with C++ classes. The recipes are mostly independent, but they are organized into two parts, which each make up about half the chapter. The first half of the chapter contains solutions to common problems you may experience when constructing objects of a class, such as using a function to create objects (which is often called a Factory pattern) or using constructors and destructors to manage resources. The second half contains solutions to problems post-construction, such as determining an object's type at runtime, and miscellaneous implementation techniques, such as how to create an interface with an abstract base class.

Classes are, of course, the central feature of C++ that supports object-oriented programming, and there are lots of different things you can do with classes. This chapter does not contain recipes that explain the basics of classes: virtual functions (polymorphism), inheritance, and encapsulation. I assume you are already familiar with these general object-oriented design principles, whether it's with C++ or another language such as Java or Smalltalk. Rather, the purpose of this chapter is to provide recipes for some of the mechanical difficulties you may run into when implementing object-oriented designs with C++.

Object-oriented design and the related design patterns is a huge subject, and the literature on the subject is vast and comprehensive. I mention only a few design patterns by name in this chapter, and they are the ones for which C++ facilities provide an elegant or perhaps not-so-obvious solution. If you are unfamiliar with the concept of design patterns, I recommend you read Design Patterns by Gamma, et al (Addison Wesley), because it is a useful thing to know in software engineering; however, it is not a prerequisite for this chapter.

PREV NEXT

Recipe 8.1. Initializing Class Member Variables

Problem

You need to initialize member variables that are native types, pointers, or references.

Solution

Use an initializer list to set the initial values for member variables. Example 8-1 shows how you can do this for native types, pointers, and references.

Example 8-1. Initializing class members

```
#include <string>

using namespace std;

class Foo {
public:
   Foo( ) : counter_(0), str_(NULL) {}
   Foo(int c, string* p) :
       counter_(c), str_(p) {}
private:
   int counter_;
   string* str_;
};

int main( ) {

   string s = "bar";
   Foo(2, &s);
}
```

Discussion

You should always initialize native variables, especially if they are class member variables. Class variables, on the other hand, should have a constructor defined that will initialize its state properly, so you do not always have to initialize them. Leaving a native variable in an uninitialized state, where it contains garbage, is asking for trouble. But there are a few different ways to do this in C++, which is what this recipe discusses.

The simplest things to initialize are native types. ints, chars, pointers, and so on are easy to deal with. Consider a simple class and its default constructor:

```
class Foo {
public:
   Foo( ) : counter_(0), str_(NULL) {}
   Foo(int c, string* p) :
       counter_(c), str_(p) {}
private:
   int counter_;
   string* str_;
};
```

Use an initializer list in the constructor to initialize member variables, and avoid doing so in the body of the constructor. This leaves the body of the constructor for any logic that must occur at construction, and makes the member variables' initialization easy to locate. A minor benefit over just assigning member variables in the constructor body, to be sure, but the benefits of using an initializer list becomes more apparent when you have class or reference member variables, or when you are trying to deal with exceptions effectively.

Recipe 8.2. Using a Function to Create Objects (a.k.a. Factory Pattern)

Problem

Instead of creating a heap object with new, you need a function (member or standalone) to do the creation for you so that the type of the object being created is decided dynamically. This sort of behavior is what the Abstract Factory design pattern achieves.

Solution

You have a couple of choices here. You can:

-
- Have the function create an instance of the object on the heap, and return a pointer to the object (or update a pointer that was passed in with the new object's address)
-
- Have the function create and return a temporary object

Example 8-2 shows how to do both of these. The Session class in the example could be any class that you don't want application code to create directly (i.e., with new), but rather you want creation managed by some other class; in this example, the managing class is SessionFactory.

Example 8-2. Functions that create objects

```
#include <iostream>

class Session {};

class SessionFactory {

public:
   Session Create( );
   Session* CreatePtr( );
   void Create(Session*& p);
   // ...
};

// Return a copy of a stack object
Session SessionFactory::Create( ) {
   Session s;
   return(s);
}

// Return a pointer to a heap object
Session* SessionFactory::CreatePtr( ) {
   return(new Session( ));
}

// Update the caller's pointer with the address
// of a new object
void SessionFactory::Create(Session*& p) {
   p = new Session( );
}

static SessionFactory f; // The one factory object

int main( ) {
```

Recipe 8.3. Using Constructors and Destructors to Manage Resources (or RAII)

Problem

For a class that represents some resource, you want to use its constructor to acquire it and the destructor to release it. This technique is often referred to as resource acquisition is initialization (RAII).

Solution

Allocate or acquire the resource in the constructor, and free or release the resource in the destructor. This reduces the amount of code a user of the class must write to deal with exceptions. See Example 8-3 for a simple illustration of this technique.

Example 8-3. Using constructors and destructors

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

using namespace std;

class Socket {
public:
   Socket(const string& hostname) {}
};

class HttpRequest {
public:
   HttpRequest (const string& hostname) :
      sock_(new Socket(hostname)) {}
   void send(string soapMsg) {sock_ << soapMsg;}
  ~HttpRequest ( ) {delete sock_;}
private:
   Socket* sock_;
};

void sendMyData(string soapMsg, string host) {
   HttpRequest req(host);
   req.send(soapMsg);
   // Nothing to do here, because when req goes out of scope
   // everything is cleaned up.
}

int main( ) {
   string s = "xml";
   sendMyData(s, "www.oreilly.com");
}
```

Discussion

The guarantees made by constructors and destructors offer a nice way to let the compiler clean up after you. Typically, you initialize an object and allocate any resources it uses in the constructor, and clean them up in the destructor. This is normal. But programmers have a tendency to use the create-open-use-close sequence of events, where the user of the class is required to do explicit "opening" and "closing" of resources. A file class is a good example.

The usual argument for RAII goes something like this. I could easily have designed my HttpRequest class in Example 8-3 to make the user do a little more work. For example:

```
 class HttpRequest {
```

Recipe 8.4. Automatically Adding New Class Instances to a Container

Problem

You need to store all instances of a class in a single container without requiring the users of the class to do anything special.

Solution

Include in the class a static member that is a container, such as a list, defined in <list>. Add an object's address to the container at construction and remove it upon destruction. Example 8-4 shows how.

Example 8-4. Keeping track of objects

```
 #include <iostream>
#include <list>
#include <algorithm>

using namespace std;

class MyClass {
protected:
   int value_;
public:
   static list<MyClass*> instances_;
   MyClass(int val);
  ~MyClass( );
   static void showList( );
};

list<MyClass*> MyClass::instances_;

MyClass::MyClass(int val) {
   instances_.push_back(this);
   value_ = val;
}

MyClass::~MyClass( ) {
   list<MyClass*>::iterator p =
      find(instances_.begin( ), instances_.end( ), this);
   if (p != instances_.end( ))
      instances_.erase(p);
}

void MyClass::showList( ) {
   for (list<MyClass*>::iterator p = instances_.begin( );
        p != instances_.end( ); ++p)
      cout << (*p)->value_ << endl;
}

int main( ) {
   MyClass a(1);
   MyClass b(10);
   MyClass c(100);
   MyClass::showList( );
}
```

Example 8-4 will create output like this:

```
 1
10
```

Recipe 8.5. Ensuring a Single Copy of a Member Variable

Problem

You have a member variable that you want only one instance of, no matter how many instances of the class are created. This kind of member variable is generally called a static member or a class variable, as opposed to an instance variable, which is one that is instantiated with every object of a class.

Solution

Declare the member variable with the static keyword, then initialize it in a separate source file (not the header file where you declared it) as in Example 8-5.

Example 8-5. Using a static member variable

```
 // Static.h
class OneStatic {
public:
    int getCount( ) {return count;}
    OneStatic( );
protected:
    static int count;
};

// Static.cpp
#include "Static.h"

int OneStatic::count = 0;

OneStatic::OneStatic( ) {
   count++;
}

// StaticMain.cpp
#include <iostream>
#include "static.h"

using namespace std;

int main( ) {
   OneStatic a;
   OneStatic b;
   OneStatic c;

   cout << a.getCount( ) << endl;
   cout << b.getCount( ) << endl;
   cout << c.getCount( ) << endl;
}
```

Discussion

static is C++'s way of allowing only one copy of something. If you declare a member variable static, only one of it will ever be constructed, regardless of the number of objects of that class that are instantiated. Similarly, if you declare a variable static in a function, it is constructed at most once and retains its value from one function call to another. With member variables, you have to do a little extra work to make sure member variables are allocated properly, though. This is why there are three files in Example 8-5.

First, you have to use the static keyword when you declare the variable. This is easy enough: add this keyword in the class header in the header file *Static.h*:

```
protected:
```

Recipe 8.6. Determining an Object's Type at Runtime

Problem

At runtime, you need to interrogate dynamically the type of particular class.

Solution

Use runtime type identification (commonly referred to as RTTI) to query the address of the object for the type of object it points to. Example 8-6 shows how.

Example 8-6. Using runtime type identification

```
 #include <iostream>
#include <typeinfo>

using namespace std;

class Base {};
class Derived : public Base {};

int main( ) {

   Base b, bb;
   Derived d;

   // Use typeid to test type equality
   if (typeid(b) == typeid(d)) { // No
      cout << "b and d are of the same type.\n";
   }
   if (typeid(b) == typeid(bb)) { // Yes
      cout << "b and bb are of the same type.\n";
   }
   if (typeid(d) == typeid(Derived)) { // Yes
      cout << "d is of type Derived.\n";
   }
}
```

Discussion

Example 8-6 shows you how to use the operator typeid to determine and compare the type of an object. typeid takes an expression or a type and returns a reference to an object of type_info or a subclass of it (which is implementation defined). You can use what is returned to test for equality or retrieve a string representation of the type's name. For example, you can compare the types of two objects like this:

```
 if (typeid(b) == typeid(d)) {
```

This will return true if the type_info objects returned by both of these are equal. This is because typeid returns a reference to a static object, so if you call it on two objects that are the same type, you will get two references to the same thing, which is why the equality test returns true.

You can also use typeid with the type itself, as in:

```
 if (typeid(d) == typeid(Derived)) {
```

This allows you to explicitly test for a particular type.

Probably the most common use of typeid is for debugging. To write out the name of the type, use type_info::name, like this:

Recipe 8.7. Determining if One Object's Class Is a Subclass of Another

Problem

You have two objects, and you need to know if their respective classes have a base class/derived class relationship or if they are unrelated.

Solution

Use the dynamic_cast operator to attempt to downcast from one type to another. The result tells you about the class's relationships. Example 8-7 presents some code for doing this.

Example 8-7. Determining class relationships

```
 #include <iostream>
#include <typeinfo>

using namespace std;

class Base {
public:
   virtual ~Base( ) {} // Make this a polymorphic class
};
class Derived : public Base {
public:
   virtual ~Derived( ) {}
};

int main( ) {

   Derived d;

   // Query the type relationship
   if (dynamic_cast<Base*>(&d)) {
      cout << "Derived is a subclass of Base" << endl;
   }
   else {
      cout << "Derived is NOT a subclass of Base" << endl;
   }
}
```

Discussion

Use the dynamic_cast operator to query the relationship between two types. dynamic_cast takes a pointer or reference to a given type and tries to convert it to a pointer or reference of a derived type, i.e., casting down a class hierarchy. If you have a Base* that points to a Derived object, dynamic_cast<Base*>(&d) returns a pointer of type Derived only if d is an object of a type that's derived from Base. If this is not possible (because Derived is not a subclass, directly or indirectly, of Base), the cast fails and NULL is returned if you passed dynamic_cast a pointer to a derived object. If it is a reference, then the standard exception bad_cast is thrown. Also, the base class must be publicly inherited and it must be unambiguous. The result tells you if one class is a descendant of another. Here's what I did in Example 8-7:

```
 if (dynamic_cast<Base*>(&d)) {
```

This returns a non-NULL pointer because d is an object of a class that is a descendant of Base. Use this on any pair of classes to determine their relationship. The only requirement is that the object argument is a polymorphic type, which means that it has at least one virtual function. If it does not, it won't compile.

Recipe 8.8. Giving Each Instance of a Class a Unique Identifier

Problem

You want each object of a class to have a unique identifier.

Solution

Use a static member variable to keep track of the next available identifier to use. In the constructor, assign the next available value to the current object and increment the static member. See Example 8-8 to get an idea of how this works.

Example 8-8. Assigning unique identifiers

```
#include <iostream>

class UniqueID {
protected:
   static int nextID;
public:
   int id;
   UniqueID( );
   UniqueID(const UniqueID& orig);
   UniqueID& operator=(const UniqueID& orig);
};

int UniqueID::nextID = 0;

UniqueID::UniqueID( ) {
   id = ++nextID;
}

UniqueID::UniqueID(const UniqueID& orig) {
   id = orig.id;
}

UniqueID& UniqueID::operator=(const UniqueID& orig) {
   id = orig.id;
   return(*this);
}

int main( ) {
   UniqueID a;
   std::cout << a.id << std::endl;
   UniqueID b;
   std::cout << b.id << std::endl;
   UniqueID c;
   std::cout << c.id << std::endl;
}
```

Discussion

Use a static variable to keep track of the next identifier to use. In Example 8-8, I used a static int, but you can use anything as the unique identifier, so long as you have a function that can generate the unique values.

In this case, the identifiers are not reused until you reach the maximum size of an int. Once you delete an object, that object's unique value is gone until the program restarts or the identifier value maxes out and flips over. This uniqueness throughout the program can have some interesting advantages. For example, if

Recipe 8.9. Creating a Singleton Class

Problem

You have a class that must only ever be instantiated once, and you need to provide a way for clients to access that class in such a way that the same, single object is returned each time. This is commonly referred to as a singleton pattern, or a singleton class.

Solution

Create a static member that is a pointer to the current class, restrict the use of constructors to create the class by making them private, and provide a public static member function that clients can use to access the single, static instance. Example 8-9 demonstrates how to do this.

Example 8-9. Creating a singleton class

```
#include <iostream>

using namespace std;

class Singleton {
public:
   // This is how clients can access the single instance
   static Singleton* getInstance( );

   void setValue(int val) {value_ = val;}
   int  getValue( )           {return(value_);}

protected:
   int value_;

private:
   static Singleton* inst_;   // The one, single instance
   Singleton( ) : value_(0) {} // private constructor
   Singleton(const Singleton&);
   Singleton& operator=(const Singleton&);
};

// Define the static Singleton pointer
Singleton* Singleton::inst_ = NULL;

Singleton* Singleton::getInstance( ) {
   if (inst_ == NULL) {
      inst_ = new Singleton( );
   }
   return(inst_);
}

int main( ) {

   Singleton* p1 = Singleton::getInstance( );

   p1->setValue(10);

   Singleton* p2 = Singleton::getInstance( );

   cout << "Value = " << p2->getValue( ) << '\n';
}
```

Discussion

Recipe 8.10. Creating an Interface with an Abstract Base Class

Problem

You need to define an interface that subclasses will implement, but the concept that the interface defines is just an abstraction, and is not something that should be instantiated itself.

Solution

Create an abstract class that defines the interface by declaring at least one of its functions as pure virtual. Subclass this abstract class by clients who will use different implementations to fulfill the same interface guarantees. Example 8-10 shows how you might define an abstract class for reading a configuration file.

Example 8-10. Using an abstract base class

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

using namespace std;

class AbstractConfigFile {

public:
   virtual ~AbstractConfigFile( ) {}

   virtual void getKey(const string& header,
                       const string& key,
                             string& val) const = 0;
   virtual void exists(const string& header,
                       const string& key,
                             string& val) const = 0;
};

class TXTConfigFile : public AbstractConfigFile {

public:
            TXTConfigFile( ) : in_(NULL) {}
            TXTConfigFile(istream& in) : in_(&in) {}
   virtual ~TXTConfigFile( ) {}

   virtual void getKey(const string& header,
                       const string& key,
                             string& val) const {}
   virtual void exists(const string& header,
                       const string& key,
                             string& val) const {}

protected:
   istream* in_;
};

class MyAppClass {
public:
   MyAppClass( ) : config_(NULL) {}
  ~MyAppClass( ) {}
   void setConfigObj(const AbstractConfigFile* p) {config_ = p;}
   void myMethod( );

private:
```

Recipe 8.11. Writing a Class Template

Problem

You have a class whose members need to be different types in different situations, and using conventional polymorphic behavior is cumbersome or redundant. In other words, as the class designer, you want a class user to be able to choose the types of various parts of your class when he instantiates it, rather than setting them all in the original definition of the class.

Solution

Use a class template to parameterize types that can be used to declare class members (and much more). That is, write your class with placeholders for types; thus, leaving it to the user of the class template to choose which types to use. See Example 8-12 for an example of a tree node that can point to any type.

Example 8-12. Writing a class template

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

using namespace std;

template<typename T>
class TreeNode {
public:
   TreeNode(const T& val) : val_(val), left_(NULL), right_(NULL) {}
  ~TreeNode( ) {
      delete left_;
      delete right_;
   }

   const T& getVal( ) const {return(val_);}
   void setVal(const T& val) {val_ = val;}
   void addChild(TreeNode<T>* p) {
      const T& other = p->getVal( );
      if (other > val_)
         if (right_)
            right_->addChild(p);
         else
            right_ = p;
      else
         if (left_)
            left_->addChild(p);
         else
            left_ = p;
   }
   const TreeNode<T>* getLeft( ) {return(left_);}
   const TreeNode<T>* getRight( ) {return(right_);}

private:
   T val_;
   TreeNode<T>* left_;
   TreeNode<T>* right_;
};

int main( ) {

   TreeNode<string> node1("frank");
   TreeNode<string> node2("larry");
   TreeNode<string> node3("bill");
```

Recipe 8.12. Writing a Member Function Template

Problem

You have a single member function that needs to take a parameter that can be of any type, and you can't or don't want to be constrained to a particular type or category of types (by using a base class pointer parameter).

Solution

Use a member function template and declare a template parameter for the type of object the function parameter is supposed to have. See Example 8-13 for a short example.

Example 8-13. Using a member function template

```
 class ObjectManager {
public:
   template<typename T>
   T* gimmeAnObject( );

   template<typename T>
   void gimmeAnObject(T*& p);
};

template<typename T>
T* ObjectManager::gimmeAnObject( ) {
   return(new T);
}

template<typename T>
void ObjectManager::gimmeAnObject(T*& p) {
   p = new T;
}

class X { /* ... */ };
class Y { /* ... */ };

int main( ) {
   ObjectManager om;

   X* p1 = om.gimmeAnObject<X>( ); // You have to specify the template
   Y* p2 = om.gimmeAnObject<Y>( ); // parameter

   om.gimmeAnObject(p1);  // Not here, though, since the compiler can
   om.gimmeAnObject(p2);  // deduce T from the arguments
}
```

Discussion

When talking about function or class templates, the words parameter and argument have some ambiguity. There are two kinds of each: template and function. Template parameters are the parameters in the angle brackets, e.g., T in Example 8-13, and function parameters are parameters in the conventional sense.

Consider the ObjectManager class in Example 8-13. It is a simplistic version of the Factory pattern discussed in Recipe 8.2, so I have defined the member function gimmeAnObject as something that creates new objects that client code would use instead of calling new directly. I can do this by either returning a pointer to a new object or by modifying a pointer passed in by the client code. Let's take a look at each approach.

Recipe 8.13. Overloading the Increment and Decrement Operators

Problem

You have a class where the familiar increment and decrement operators make sense, and you want to overload operator++ and operator-- to make incrementing and decrementing objects of your class easy and intuitive to users.

Solution

Overload the prefix and postfix forms of ++ and -- to do what you want. Example 8-14 shows the conventional technique for overloading the increment and decrement operators.

Example 8-14. Overloading increment and decrement

```
#include <iostream>

using namespace std;

class Score {
public:
   Score( ) : score_(0) {}
   Score(int i) : score_(i) {}

   Score& operator++( ) { // prefix
      ++score_;
      return(*this);
   }
   const Score operator++(int) { // postfix
      Score tmp(*this);
      ++(*this); // Take advantage of the prefix operator
      return(tmp);
   }
   Score& operator--( ) {
      --score_;
      return(*this);
   }
   const Score operator--(int x) {
      Score tmp(*this);
      --(*this);
      return(tmp);
   }
   int getScore( ) const {return(score_);}

private:
   int score_;
};

int main( ) {
   Score player1(50);

   player1++;
   ++player1; // score_ = 52
   cout << "Score = " << player1.getScore( ) << '\n';
   (--player1)--; // score_ = 50
   cout << "Score = " << player1.getScore( ) << '\n';
}
```

Discussion

Recipe 8.14. Overloading Arithmetic and Assignment Operators for Intuitive Class Behavior

Problem

You have a class for which some of C++'s unary or binary operators make sense, and you want users of your class to be able to use them when working with objects of your class. For example, if you have a class named Balance that contains, essentially, a floating-point value (i.e., an account balance), it would be convenient if you could use Balance objects with some standard C++ operators, like this:

```
Balance checking(50.0), savings(100.0);

checking += 12.0;
Balance total = checking + savings;
```

Solution

Overload the operators you want to use as member functions and standalone functions to allow arguments of various types for which the given operator makes sense, as in Example 8-15.

Example 8-15. Overloading unary and binary operators

```
#include <iostream>

using namespace std;

class Balance {
   // These have to see private data
   friend const Balance operator+(const Balance& lhs, const Balance& rhs);
   friend const Balance operator+(double lhs, const Balance& rhs);
   friend const Balance operator+(const Balance& lhs, double rhs);

public:
   Balance( ) : val_(0.0) {}
   Balance(double val) : val_(val) {}
  ~Balance( ) {}

   // Unary operators
   Balance& operator+=(const Balance& other) {
      val_ += other.val_;
      return(*this);
   }
   Balance& operator+=(double other) {
      val_ += other;
      return(*this);
   }

   double getVal( ) const {return(val_);}

private:
   double val_;
};

// Binary operators
const Balance operator+(const Balance& lhs, const Balance& rhs) {
   Balance tmp(lhs.val_ + rhs.val_);
   return(tmp);
}

const Balance operator+(double lhs, const Balance& rhs) {
   Balance tmp(lhs + rhs.val_);
   return(tmp);
```

Recipe 8.15. Calling a Superclass Virtual Function

Problem

You need to invoke a function on a superclass of a particular class, but it is overridden in subclasses, so the usual syntax of p->method( ) won't give you the results you are after.

Solution

Qualify the name of the member function you want to call with the target base class; for example, if you have two classes. (See Example 8-16.)

Example 8-16. Calling a specific version of a virtual function

```
#include <iostream>

using namespace std;

class Base {
public:
   virtual void foo( ) {cout << "Base::foo( )" << endl;}
};

class Derived : public Base {
public:
   virtual void foo( ) {cout << "Derived::foo( )" << endl;}
};

int main( ) {
   Derived* p = new Derived( );

   p->foo( );       // Calls the derived version
   p->Base::foo( ); // Calls the base version
}
```

Discussion

Making a regular practice of overriding C++'s polymorphic facilities is not a good idea, but there are times when you have to do it. As with so many techniques in C++, it is largely a matter of syntax. When you want to call a specific base class's version of a virtual function, just qualify it with the name of the class you are after, as I did in Example 8-16:

```
p->Base::foo( );
```

This will call the version of foo defined for Base, and not the one defined for whatever subclass of Base p points to.

Chapter 9. Exceptions and Safety

PREV NEXT

Introduction

This chapter contains recipes for using C++'s exception-handling features. C++ has strong support for exception handling, and by employing a few techniques you can write code that handles exceptional circumstances effectively and is easy to debug.

The first recipe describes C++'s semantics for throwing and catching exceptions, then it explains how to write a class to represent exceptions. This is a good starting point if you have little or no experience with exceptions. It also describes the standard exception classes that are defined in <stdexcept> and <exception>.

The rest of the recipes illustrate techniques for using exceptions optimally, and they define several key terms along the way. Just throwing an exception when something unexpected happens, or catching an exception only to print an error message and abort does not make for good software. To use C++'s exception-handling facilities effectively, you have to write code that doesn't leak resources if an exception is thrown, and that otherwise has well-defined behavior when an exception is thrown. These are known as the basic and strong exception-safety guarantees. I describe techniques you can use that allow you to make these guarantees for constructors and various member functions.

Recipe 9.1. Creating an Exception Class

Problem

You want to create your own exception class for tHRowing and catching.

Solution

You can throw or catch any C++ type that lives up to some simple requirements, namely that it has a valid copy constructor and destructor. Exceptions are a complicated subject though, so there are a number of things to consider when designing a class to represent exceptional circumstances. Example 9-1 shows what a simple exception class might look like.

Example 9-1. A simple exception class

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

using namespace std;

class Exception {

public:
   Exception(const string& msg) : msg_(msg) {}
  ~Exception( ) {}

   string getMessage( ) const {return(msg_);}
private:
   string msg_;
};

void f( ) {
   throw(Exception("Mr. Sulu"));
}

int main( ) {

   try {
      f( );
   }
   catch(Exception& e) {
      cout << "You threw an exception: " << e.getMessage( ) << endl;
   }
}
```

Discussion

C++ supports exceptions with three keywords: try, catch, and tHRow. The syntax looks like this:

```
 try {
   // Something that may call "throw", e.g.
   throw(Exception("Uh-oh"));
}
catch(Exception& e) {
   // Do something useful with e
}
```

An exception in C++ (Java and C# are similar) is a way to put a message in a bottle at some point in a program, abandon ship, and hope that someone is looking for your message somewhere down the call stack. It is an alternative to other, simpler techniques, such as returning an error code or message. The semantics of using exceptions (e.g., "trying" something, "throwing" an exception, and subsequently

Recipe 9.2. Making a Constructor Exception-Safe

Problem

Your constructor needs to uphold basic and strong exception-safety guarantees. See the discussion that follows for the definitions of "basic" and "strong" guarantees.

Solution

Use try and catch in the constructor to clean up properly if an exception is thrown during construction. Example 9-2 presents examples of the simple Device and Broker classes. Broker constructs two Device objects on the heap, but needs to be able to properly clean them up if an exception is thrown during construction.

Example 9-2. An exception-safe constructor

```
 #include <iostream>
#include <stdexcept>

using namespace std;

class Device {
public:
   Device(int devno) {
      if (devno == 2)
         throw runtime_error("Big problem");
   }
  ~Device( ) {}
};

class Broker {

public:
   Broker (int devno1, int devno2) :
      dev1_(NULL), dev2_(NULL) {
      try {
         dev1_ = new Device(devno1);  // Enclose the creation of heap
         dev2_ = new Device(devno2);  // objects in a try block...
      }
      catch (...) {
         delete dev1_;                // ...clean up and rethrow if
         throw;                       // something goes wrong.
      }
   }
  ~Broker( ) {
      delete dev1_;
      delete dev2_;
   }

private:
   Broker( );
   Device* dev1_;
   Device* dev2_;
};

int main( ) {

   try {
      Broker b(1, 2);
   }
   catch(exception& e) {
      cerr << "Exception: " << e.what( ) << endl;
```

Recipe 9.3. Making an Initializer List Exception-Safe

Problem

You have to initialize your data members in the constructor's initializer list, and, therefore, cannot use the approach described in Recipe 9.2.

Solution

Use a special syntax for try and catch that catches exceptions thrown in the initializer list. Example 9-3 shows how.

Example 9-3. Handling exceptions in an initializer

```
 #include <iostream>
#include <stdexcept>

using namespace std;

// Some device
class Device {
public:
   Device(int devno) {
      if (devno == 2)
         throw runtime_error("Big problem");
   }
  ~Device( ) {}
private:
   Device( );
};

class Broker {

public:
   Broker (int devno1, int devno2)
      try : dev1_(Device(devno1)),   // Create these in the initializer
            dev2_(Device(devno2)) {} // list.
      catch (...) {
         throw; // Log the message or translate the error here (see
                // the discussion)
      }
  ~Broker( ) {}

private:
   Broker( );
   Device dev1_;
   Device dev2_;
};

int main( ) {

   try {
      Broker b(1, 2);
   }
   catch(exception& e) {
      cerr << "Exception: " << e.what( ) << endl;
   }
}
```

Discussion

Recipe 9.4. Making Member Functions Exception-Safe

Problem

You are writing a member function and you need it to uphold the basic and strong exception-safety guarantees, namely that it won't leak resources and it won't leave the object in an invalid state if an exception is thrown.

Solution

Be aware of what operations can throw exceptions and do them first, usually in a try/catch block. Once the code that can throw exceptions is done executing, then you can update the object state. Example 9-4 offers one way to make a member function exception-safe.

Example 9-4. An exception-safe member function

```
class Message {

public:
   Message(int bufSize = DEFAULT_BUF_SIZE) :
      bufSize_(bufSize),
      initBufSize_(bufSize),
      msgSize_(0),
      buf_(NULL) {
      buf_ = new char[bufSize];
   }

  ~Message( ) {
      delete[] buf_;
   }

   // Append character data
   void appendData(int len, const char* data) {
      if (msgSize_+len > MAX_SIZE) {
         throw out_of_range("Data size exceeds maximum size.");
      }
      if (msgSize_+len > bufSize_) {

         int newBufSize = bufSize_;
         while ((newBufSize *= 2) < msgSize_+len);

         char* p = new char[newBufSize];     // Allocate memory
                                             // for new buffer

         copy(buf_, buf_+msgSize_, p);       // Copy old data
         copy(data, data+len, p+msgSize_);   // Copy new data

         msgSize_ += len;
         bufSize_ = newBufSize;

         delete[] buf_;  // Get rid of old buffer and point to new
         buf_ = p;
      }
      else {
         copy(data, data+len, buf_+msgSize_);
         msgSize_ += len;
      }
   }

   // Copy the data out to the caller's buffer
   int getData(int maxLen, char* data) {
      if (maxLen < msgSize_) {
```

Recipe 9.5. Safely Copying an Object

Problem

You need the basic class copy operationscopy construction and assignmentto be exception-safe.

Solution

Employ the tactics discussed in Recipe 9.4 by doing everything that might throw first, then changing the object state with operations that can't throw only after the hazardous work is complete. Example 9-6 presents the Message class again, this time with the assignment operator and copy constructor defined.

Example 9-6. Exception-safe assignment and copy construction

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

const static int DEFAULT_BUF_SIZE = 3;
const static int MAX_SIZE         = 4096;

class Message {

public:
   Message(int bufSize = DEFAULT_BUF_SIZE) :
      bufSize_(bufSize),
      initBufSize_(bufSize),
      msgSize_(0),
      key_("") {
      buf_ = new char[bufSize]; // Note: now this is in the body
   }

  ~Message( ) {
      delete[] buf_;
   }

   // Exception-safe copy ctor
   Message(const Message& orig) :
      bufSize_(orig.bufSize_),
      initBufSize_(orig.initBufSize_),
      msgSize_(orig.msgSize_),
      key_(orig.key_) { // This can throw...

      buf_ = new char[orig.bufSize_];  // ...so can this
      copy(orig.buf_, orig.buf_+msgSize_, buf_); // This can't
   }

   // Exception-safe assignment, using the copy ctor
   Message& operator=(const Message& rhs) {

      Message tmp(rhs);   // Copy construct a temporary
      swapInternals(tmp); // Swap members with it
      return(*this);      // When we leave, tmp is destroyed, taking
                          // the original data with it
   }

   const char* data( ) {
      return(buf_);
   }

private:
   void swapInternals(Message& msg) {
      // Since key_ is not a built-in data type it can throw,
```

Chapter 10. Streams and Files

Introduction

Streams are one of the most powerful (and complicated) components of the C++ standard library. Using them for plain, unformatted input and output is generally straightforward, but changing the format to suit your needs with standard manipulators, or writing your own manipulators, is not. Therefore, the first few recipes describe different ways to format stream output. The two after that describe how to write objects of a class to a stream or read them from one.

Then the recipes shift from reading and writing file content to operating on the files themselves (and directories). If your program uses files, especially if it's a daemon or server-side process, you will probably create files and directories, clean them up, rename them, and so on. There are a number of recipes that explain how to do these unglamorous, but necessary, tasks in C++.

The last third of the recipes demonstrate how to manipulate file and pathnames themselves using many of the standard string member functions. Standard strings contain an abundance of functions for inspecting and manipulating their contents, and if you have to parse path and filenames they come in handy. If what you need is not discussed in these recipes, take a look at Chapter 7, toowhat you're after might be described there.

File manipulation requires direct interaction with the operating system (OS), and there are often subtle differences (and occasionally glaring incompatibilities) between OSs. Many of the typical file and directory manipulation needs are part of the standard C system calls, and work the same or similarly on different systems. Where there are differences between OSs' versions of libraries, I note it in the recipes.

As I have discussed in previous chapters, Boost is an open source project that has generated a number of high-quality, portable libraries. But since this is a book about C++ and not the Boost project, I have preferred standard C++ solutions whenever possible. In many cases, however, (most notably Recipe 10.12) there isn't a Standard C++ solution, so I have used the Boost Filesystem library written by Beman Dawes, which provides a portable filesystem interface, to give a portable solution. Take a look at the Boost Filesystem library if you have to do portable filesystem interactionyou will save yourself lots of time and effort. For more information on the Boost project, see www.boost.org.

Recipe 10.1. Lining Up Text Output

Problem

You need to line up your text output vertically. For example, if you are exporting tabular data, you may want it to look like this:

```
 Jim            Willcox         Mesa           AZ
Bill           Johnson         San Mateo      CA
Robert         Robertson       Fort Collins   CO
```

You will probably also want to be able to right- or left-justify the text.

Solution

Use ostream or wostream, for narrow or wide characters, defined in <ostream>, and the standard stream manipulators to set the field width and justify the text. Example 10-1 shows how.

Example 10-1. Lining up text output

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

using namespace std;

int main( ) {

   ios_base::fmtflags flags = cout.flags( );
   string first, last, citystate;
   int width = 20;

   first = "Richard";
   last  = "Stevens";
   citystate = "Tucson, AZ";

   cout << left                       // Left-justify in each field
        << setw(width) << first       // Then, repeatedly set the width
        << setw(width) << last        // and write some data
        << setw(width) << citystate << endl;

   cout.flags(flags);
}
```

The output looks like this:

```
 Richard             Stevens             Tucson, AZ
```

Discussion

A manipulator is a function that operates on a stream. Manipulators are applied to a stream with operator<<. The stream's format (input or output) is controlled by a set of flags and settings on the ultimate base stream class, ios_base. Manipulators exist to provide convenient shorthand for adjusting these flags and settings without having to explicitly set them via setf or flags, which is cumbersome to write and ugly to read. The best way to format stream output is to use manipulators.

Example 10-1 uses two manipulators to line up text output into columns. The manipulator setw sets the field width, and left left-justifies the value within that field (the counterpart to left is, not surprisingly, right). A "field" is just another way of saying that you want the output to be padded on one side or the other to make sure that the value you write is the only thing printed in that field. If, as in Example 10-1, you left-justify a value, then set the field width, the next thing you write to the stream will begin with the first

Recipe 10.2. Formatting Floating-Point Output

Problem

You need to present floating-point output in a well-defined format, either for the sake of precision (scientific versus fixed-point notation) or simply to line up decimal points vertically for easier reading.

Solution

Use the standard manipulators provided in <iomanip> and <ios> to control the format of floating-point values that are written to the stream. There are too many combinations of ways to cover here, but Example 10-3 offers a few different ways to display the value of pi.

Example 10-3. Formatting pi

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

using namespace std;

int main( ) {

   ios_base::fmtflags flags =  // Save old flags
      cout.flags( );

   double pi = 3.14285714;

   cout << "pi = " << setprecision(5)  // Normal (default) mode; only
        << pi << '\n';                 // show 5 digits, including both
                                       // sides of decimal point.

   cout << "pi = " << fixed            // Fixed-point mode;
        << showpos                     // show a "+" for positive nums,
        << setprecision(3)             // show 3 digits to the *right*
        << pi << '\n';                 // of the decimal.

   cout << "pi = " << scientific       // Scientific mode;
        << noshowpos                   // don't show plus sign anymore
        << pi * 1000 << '\n';

   cout.flags(flags);  // Set the flags to the way they were
}
```

This will produce the following output:

```
 pi = 3.1429
pi = +3.143
pi = 3.143e+003
```

Discussion

Manipulators that specifically manipulate floating-point output divide into two categories. There are those that set the format, which, for the purposes of this recipe, set the general appearance of floating-point and integer values, and there are those that fine-tune the display of each format. The formats are as follows:

Normal (the default)

In this format, the number of digits displayed is fixed (with a default of six) and the decimal is displayed

Recipe 10.3. Writing Your Own Stream Manipulators

Problem

You need a stream manipulator that does something the standard ones can't. Or, you want to have a single manipulator set several flags on the stream instead of calling a set of manipulators each time you want a particular format.

Solution

To write a manipulator that doesn't take an argument (à la left), write a function that takes an ios_base parameter and sets stream flags on it. If you need a manipulator that takes an argument, see the discussion a little later. Example 10-4 shows how to write a manipulator that doesn't take an argument.

Example 10-4. A simple stream manipulator

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

using namespace std;

// make floating-point output look normal
inline ios_base& floatnormal(ios_base& io) {
   io.setf(0, ios_base::floatfield);
   return(io);
}
int main( ) {

   ios_base::fmtflags flags =  // Save old flags
      cout.flags( );

   double pi = 22.0/7.0;

   cout << "pi = " << scientific   // Scientific mode
        << pi * 1000 << '\n';

   cout << "pi = " << floatnormal
        << pi << '\n';

   cout.flags(flags);
}
```

Discussion

There are two kinds of manipulators: those that accept arguments and those that don't. Manipulators that take no arguments are easy to write. All you have to do is write a function that accepts a stream parameter, does something to it (sets a flag or changes a setting), and returns it. Writing a manipulator that takes one or more arguments is more complicated because you need to create additional classes and functions that operate behind the scenes. Since argument-less manipulators are simple, let's start with those.

After reading Recipe 10.1, you may have realized that there are three floating-point formats and only two manipulators for choosing the format. The default format doesn't have a manipulator; you have to set a flag on the stream to get back to the default format, like this:

```
 myiostr.setf(0, ios_base::floatfield);
```

But for consistency and convenience, you may want to add your own manipulator that does the same thing. That's what Example 10.4 does. The floatnormal manipulator sets the appropriate stream flag to

Recipe 10.4. Making a Class Writable to a Stream

Problem

You have to write a class to an output stream, either for human readability or persistent storage, i.e., serialization.

Solution

Overload operator<< to write the appropriate data members to the stream. Example 10-6 shows how.

Example 10-6. Writing objects to a stream

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

using namespace std;

class Employer {
   friend ostream& operator<<              // This has to be a friend
     (ostream& out, const Employer& empr); // so it can access non-
public:                                    // public members
   Employer( ) {}
  ~Employer( ) {}

   void setName(const string& name) {name_ = name;}
private:
   string name_;
};

class Employee {
   friend ostream& operator<<
     (ostream& out, const Employee& obj);
public:
   Employee( ) : empr_(NULL) {}
  ~Employee( ) {if (empr_) delete empr_;}

   void setFirstName(const string& name) {firstName_ = name;}
   void setLastName(const string& name) {lastName_ = name;}
   void setEmployer(Employer& empr) {empr_ = &empr;}
   const Employer* getEmployer( ) const {return(empr_);}

private:
   string firstName_;
   string lastName_;
   Employer* empr_;
};

// Allow us to send Employer objects to an ostream...
ostream& operator<<(ostream& out, const Employer& empr) {

   out << empr.name_ << endl;

   return(out);
}

// Allow us to send Employee objects to an ostream...
ostream& operator<<(ostream& out, const Employee& emp) {

   out << emp.firstName_ << endl;
   out << emp.lastName_ << endl;
   if (emp.empr_)
```

Recipe 10.5. Making a Class Readable from a Stream

Problem

You have written an object of some class to a stream, and now you need to read that data from the stream and use it to initialize an object of the same class.

Solution

Use operator>> to read data from the stream into your class to populate its data members, which is simply the reverse of what Example 10-6 does. See Example 10-7 for an implementation.

Example 10-7. Reading data into an object from a stream

```
 #include <iostream>
#include <istream>
#include <fstream>
#include <string>

using namespace std;

class Employee {
   friend ostream& operator<<              // These have to be friends
      (ostream& out, const Employee& emp); // so they can access
   friend istream& operator>>              // nonpublic members
      (istream& in, Employee& emp);

public:
   Employee( ) {}
  ~Employee( ) {}

   void setFirstName(const string& name) {firstName_ = name;}
   void setLastName(const string& name) {lastName_ = name;}

private:
   string firstName_;
   string lastName_;
};

// Send an Employee object to an ostream...
ostream& operator<<(ostream& out, const Employee& emp) {

   out << emp.firstName_ << endl;
   out << emp.lastName_ << endl;

   return(out);
}

// Read an Employee object from a stream
istream& operator>>(istream& in, Employee& emp) {

   in >> emp.firstName_;
   in >> emp.lastName_;

   return(in);
}

int main( ) {

   Employee emp;
   string first = "William";
   string last = "Shatner";
```

Recipe 10.6. Getting Information About a File

Problem

You want information about a file, such as its size, device, last modification time, etc.

Solution

Use the C system call stat in <sys/stat.h>. See Example 10-8 for a typical use of stat that prints out a few file attributes.

Example 10-8. Obtaining file information

```
 #include <iostream>
#include <ctime>
#include <sys/types.h>
#include <sys/stat.h>
#include <cerrno>
#include <cstring>

int main(int argc, char** argv )
{
   struct stat fileInfo;

   if (argc < 2) {
      std::cout << "Usage: fileinfo <file name>\n";
      return(EXIT_FAILURE);
   }

   if (stat(argv[1], &fileInfo) != 0) {  // Use stat( ) to get the info
      std::cerr << "Error: " << strerror(errno) << '\n';
      return(EXIT_FAILURE);
   }

   std::cout << "Type:         : ";
   if ((fileInfo.st_mode & S_IFMT) == S_IFDIR) { // From sys/types.h
      std::cout << "Directory\n";
   } else {
      std::cout << "File\n";
   }

   std::cout << "Size          : " <<
      fileInfo.st_size << '\n';              // Size in bytes
   std::cout << "Device        : " <<
      (char)(fileInfo.st_dev + 'A') << '\n';  // Device number
   std::cout << "Created       : " <<
      std::ctime(&fileInfo.st_ctime);        // Creation time
   std::cout << "Modified      : " <<
      std::ctime(&fileInfo.st_mtime);        // Last mod time
}
```

Discussion

The C++ standard library supports manipulation of file content with streams, but it has no built-in support for reading or altering the metadata the OS maintains about a file, such as its size, ownership, permissions, various timestamps, and other information. However, standard C contains a number of standard system call libraries that you can use to get this kind of information about a file, and that's what Example 10-8 uses.

There are two parts to obtaining file information. First, there is a struct named stat that contains members

Recipe 10.7. Copying a File

Problem

You need to copy one file to another in a portable manner, i.e., without using OS-specific APIs.

Solution

Use C++ file streams in <fstream> to copy data from one stream to another. Example 10-9 gives an example of a buffered stream copy.

Example 10-9. Copying a file

```
 #include <iostream>
#include <fstream>

const static int BUF_SIZE = 4096;

using std::ios_base;

int main(int argc, char** argv) {

   std::ifstream in(argv[1],
      ios_base::in | ios_base::binary);  // Use binary mode so we can
   std::ofstream out(argv[2],            // handle all kinds of file
      ios_base::out | ios_base::binary); // content.

   // Make sure the streams opened okay...

   char buf[BUF_SIZE];

   do {
      in.read(&buf[0], BUF_SIZE);      // Read at most n bytes into
      out.write(&buf[0], in.gcount( )); // buf, then write the buf to
   } while (in.gcount( ) > 0);          // the output.

   // Check streams for problems...

   in.close( );
   out.close( );
}
```

Discussion

Copying a file may appear to be a simple matter of reading from one stream and writing to another. But the C++ streams library is large, and there are a number of different ways to do the reading and the writing, so you should know a little about the library to avoid costly performance mistakes.

Example 10-9 runs fast because it buffers input and output. The read and write functions operate on entire buffers at a timeinstead of a character-at-a-time copy loopby reading from the input stream to the buffer and writing from the buffer to the output stream in chunks. They also do not do any kind of formatting on the data like the left- and right-shift operators, which keeps things fast. Additionally, since the streams are in binary mode, EOF characters can be read and written without incident. Depending on your hardware, OS, and so on, you will get different results for different buffer sizes. Experiment to find the best parameters for your system.

But there's more to it than this. All C++ streams already buffer data when reading or writing, so Example 10-9 is actually doing double buffering. The input stream has its own internal stream buffer that holds characters that have been read from the source but not extracted with read, operator<<, getc, or any

Recipe 10.8. Deleting or Renaming a File

Problem

You have to remove or rename a file, and you want to do it portably, i.e., without using OS-specific APIs.

Solution

The Standard C functions remove and rename, in <cstdio>, will do this. See Example 10-11 for a brief demonstration of them.

Example 10-11. Removing a file

```
#include <iostream>
#include <cstdio>
#include <cerrno>

using namespace std;

int main(int argc, char** argv) {

   if (argc != 2) {
      cerr << "You must supply a file name to remove." << endl;
      return(EXIT_FAILURE);
   }

   if (remove(argv[1]) == -1) {  // remove( ) returns -1 on error
      cerr << "Error: " << strerror(errno) << endl;
      return(EXIT_FAILURE);
   }
   else {
      cout << "File '" << argv[1] << "' removed." << endl;
   }
}
```

Discussion

These system calls are easy to use: just call one or the other with the filename you want to delete or rename. If something goes wrong, the return value is non-zero and errno is set to the appropriate error number. You can use strerror or perror (both declared in <cstdio>) to print out the implementation-defined error message.

To rename a file, you can replace the remove call in Example 10-11 with the following code:

```
if (rename(argv[1], argv[2])) {
   cerr << "Error: " << strerror(errno) << endl;
   return(EXIT_FAILURE);
}
```

Recipe 10.9. Creating a Temporary Filename and File

Problem

You have to store some stuff on disk temporarily, and you don't want to have to write a routine that generates a unique name yourself.

Solution

Use either the tmpfile or tmpnam functions, declared in <cstdio>. tmpfile returns a FILE* that is already opened for writing, and tmpnam generates a unique filename that you can open yourself. Example 10-13 shows how to use tmpfile.

Example 10-13. Creating a temporary file

```
 #include <iostream>
#include <cstdio>

int main( ) {

   FILE* pf = NULL;
   char buf[256];

   pf = tmpfile( );  // Create and open a temp file

   if (pf) {
      fputs("This is a temp file", pf);  // Write some data to it
   }

   fseek(pf, 5, SEEK_SET);  // Reset the file position
   fgets(buf, 255, pf);     // Read a string from it
   fclose(pf);

   std::cout << buf << '\n';
}
```

Discussion

There are two ways to create a temporary file; Example 10-13 shows the first way. The function tmpfile is declared in <cstdio>, takes no parameters, and returns a FILE* if successful, NULL if not. The FILE* is the same type you can use with the C input/output functions fread, fwrite, fgets, fputs, etc. tmpfile opens the temporary file in "wb+" mode, which means you can write to it or read from it in binary mode (i.e., the characters are not interpreted as they are read). When your program terminates normally, the temporary file created by tmpfile is automatically deleted.

This may or may not work for you depending on your requirements. You will notice that tmpfile does not give you a filenamehow do you pass the file to another program? You can't; you'll have to use a similar function instead: tmpnam.

tmpnam doesn't actually create a temporary file, it just creates a unique file name that you can use to go open a file using that name yourself. tmpnam takes a single char* parameter and returns a char*. You can pass in a pointer to a char buffer (that has to be at least as big as the macro L_tmpnam, also defined in <cstdio>), where tmpnam will copy the temporary name, and it will return a pointer to the same buffer. If you pass in NULL, tmpfile will return a pointer to a static buffer that contains the filename, which means that subsequent calls to tmpnam will overwrite it. (See Example 10-14.)

Example 10-14. Creating a temporary filename

```
#include <iostream>
```

Recipe 10.10. Creating a Directory

Problem

You have to create a directory, and you want to do it portably, i.e., without using OS-specific APIs.

Solution

On most platforms, you will be able to use the mkdir system call that is shipped with most compilers as part of the C headers. It takes on different forms in different OSs, but regardless, you can use it to create a new directory. There is no standard C++, portable way to create a directory. Check out Example 10-15 to see how.

Example 10-15. Creating a directory

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

int main(int argc, char** argv) {

   if (argc < 2) {
      std::cerr << "Usage: " << argv[0] << " [new dir name]\n";
      return(EXIT_FAILURE);
   }

   if (mkdir(argv[1]) == -1) {  // Create the directory
      std::cerr << "Error: " << strerror(errno);
      return(EXIT_FAILURE);
   }
}
```

Discussion

The system call for creating directories differs somewhat from one OS to another, but don't let that stop you from using it anyway. Variations of mkdir are supported on most systems, so creating a directory is just a matter of knowing which header to include and what the function's signature looks like.

Example 10-15 works on Windows, but not Unix. On Windows, mkdir is declared in <direct.h>. It takes one parameter (the directory name), returns -1 if there is an error, and sets errno to the corresponding error number. You can get the implementation-defined error text by calling strerror or perror.

On Unix, mkdir is declared in <sys/stat.h>, and its signature is slightly different. The error semantics are just like Windows, but there is a second parameter that specifies the permissions to apply to the new directory. Instead, you must specify the permissions using the traditional chmod format (see the chmod man page for specifics), e.g., 0777 means owner, group, and others all have read, write, and execute permissions. Thus, you might call it like this on Unix:

```
 #include <iostream>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char** argv) {

   if (argc < 2) {
      std::cerr << "Usage: " << argv[0] << " [new dir name]\n";
      return(EXIT_FAILURE);
   }

   if (mkdir(argv[1], 0777) == -1) {  // Create the directory
```

Recipe 10.11. Removing a Directory

Problem

You need to remove a directory, and you want to do it portably, i.e., without using OS-specific APIs.

Solution

On most platforms, you will be able to use the rmdir system call that is shipped with most compilers as part of the C headers. There is no standard C++, portable way to remove a directory. rmdir takes on different forms in different OSs, but regardless, you can use it to remove a directory. See Example 10-17 for a short program that removes a directory.

Example 10-17. Removing a directory

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

using namespace std;

int main(int argc, char** argv) {

   if (argc < 2) {
      cerr << "Usage: " << argv[0] << " [dir name]" << endl;
      return(EXIT_FAILURE);
   }

   if (rmdir(argv[1]) == -1) {  // Remove the directory
      cerr << "Error: " << strerror(errno) << endl;;
      return(EXIT_FAILURE);
   }
}
```

Discussion

The signature of rmdir is the same on most OSs, but the header file where it is declared is not. On Windows, it is declared in <direct.h>, and on Unix, it is declared in <unistd.h>. It takes one parameter (the directory name), returns -1 if there is an error, and sets errno to the corresponding error number. You can get the implementation-defined error text by calling strerror or perror.

If the target directory is not empty rmdir will return an error. To list the contents of a directory, to enumerate them for deletion, etc., see Recipe 10.12.

If you want portability, and don't want to write a bunch of #ifdefs around the various OS-specific directory functions, you should consider using the Boost Filesystem library. The Boost Filesystem library uses the concept of a path to refer to a directory or file, and paths can be removed with a single function, remove.

The function removeRecurse in Example 10-18 recursively removes a directory and all of its contents. The most important part is the remove function (which is boost::filesystem::remove, not a standard library function). It takes a path argument, and removes it if it is a file or an empty directory, but it doesn't remove a directory that contains files.

Example 10-18. Removing a directory with Boost

```
 #include <iostream>
#include <string>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>
```

Recipe 10.12. Reading the Contents of a Directory

Problem

You need to read the contents of a directory, most likely to do something to each file or subdirectory that's in it.

Solution

To write something portable, use the Boost Filesystem library's classes and functions. It provides a number of handy utilities for manipulating files, such as a portable path representation, directory iterators, and numerous functions for renaming, deleting, and copying files, and so on. Example 10-19 demonstrates how to use a few of these facilities.

Example 10-19. Reading a directory

```
 #include <iostream>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace boost::filesystem;

int main(int argc, char** argv) {

   if (argc < 2) {
      std::cerr << "Usage: " << argv[0] << " [dir name]\n";
      return(EXIT_FAILURE);
   }

   path fullPath =    // Create the full, absolute path name
     system_complete(path(argv[1], native));

   if (!exists(fullPath)) {
      std::cerr << "Error: the directory " << fullPath.string( )
                << " does not exist.\n";
      return(EXIT_FAILURE);
   }

   if (!is_directory(fullPath)) {
      std::cout << fullPath.string( ) << " is not a directory!\n";
      return(EXIT_SUCCESS);
   }

   directory_iterator end;
   for (directory_iterator it(fullPath);
        it != end; ++it) {               // Iterate through each
                                         // element in the dir,
      std::cout << it->leaf( );           // almost as you would
      if (is_directory(*it))             // an STL container
         std::cout << " (dir)";
      std::cout << '\n';
   }

   return(EXIT_SUCCESS);
}
```

Discussion

Like creating or deleting directories (see Recipe 10.10 and Recipe 10.11), there is no standard, portable way to read the contents of a directory. To make your C++ life easier, the Filesystem library in the Boost project provides a set of portable routines for operating on files and directories. It also

Recipe 10.13. Extracting a File Extension from a String

Problem

Given a filename or a complete path, you need to retrieve the file extension, which is the part of a filename that follows the last period. For example, in the filenames *src.cpp*, *Window.class*, and *Resume.doc*, the file extensions are *.cpp*, *.class*, and *.doc*.

Solution

Convert the file and/or pathname to a string, use the rfind member function to locate the last period, and return everything after that. Example 10-20 shows how to do this.

Example 10-20. Getting a file extension from a filename

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

using std::string;

string getFileExt(const string& s) {

   size_t i = s.rfind('.', s.length( ));
   if (i != string::npos) {
      return(s.substr(i+1, s.length( ) - i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The extension is \"" << getFileExt(path) << "\"\n";
}
```

Discussion

To get an extension from a filename, you just need to find out where the last dot "." is and take everything to the right of that. The standard string class, defined in <string> contains functions for doing both of these things: rfind and substr.

rfind will search backward for whatever you sent it (a char in this case) as the first argument, starting at the index specified by the second argument, and return the index where it was found. If the pattern wasn't found, rfind will return string::npos. substr also takes two arguments. The first is the index of the first element to copy, and the second is the number of characters to copy.

The standard string class contains a number of member functions for finding things. See Recipe 4.9 for a longer discussion of string searching.

See Also

Recipe 4.9 and Recipe 10.12

Recipe 10.14. Extracting a Filename from a Full Path

Problem

You have the full path of a filename, e.g., *d:\apps\src\foo.c*, and you need to get the filename, *foo.c*.

Solution

Employ the same technique as the previous recipe and use rfind and substr to find and get what you want from the full pathname. Example 10-21 shows how.

Example 10-21. Extracting a filename from a path

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

using std::string;

string getFileName(const string& s) {

   char sep = '/';

#ifdef _WIN32
   sep = '\\';
#endif

   size_t i = s.rfind(sep, s.length( ));
   if (i != string::npos) {
      return(s.substr(i+1, s.length( ) - i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The file name is \"" << getFileName(path) << "\"\n";
}
```

Discussion

See the previous recipe for details on how rfind and substr work. The only thing noteworthy about Example 10-21 is that, as you probably are already aware, Windows has a path separator that is a backslash instead of a forward-slash, so I added an #ifdef to conditionally set the path separator.

The path class in the Boost Filesystem library makes getting the last part of a full pathnamewhich may be a file or directory nameeasy with the path::leaf member function. Example 10-22 shows a simple program that uses it to print out whether a path refers to a file or directory.

Example 10-22. Getting a filename from a path

```
 #include <iostream>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {
```

Recipe 10.15. Extracting a Path from a Full Path and Filename

Problem

You have the full path of a filename, e.g., d:\apps\src\foo.c, and you need to get the pathname, d:\apps\src.

Solution

Use the same technique as the previous two recipes by invoking rfind and substr to find and get what you want from the full pathname. See Example 10-23 for a short sample program.

Example 10-23. Get the path from a full path and filename

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

using std::string;

string getPathName(const string& s) {

   char sep = '/';

#ifdef _WIN32
   sep = '\\';
#endif

   size_t i = s.rfind(sep, s.length( ));
   if (i != string::npos) {
      return(s.substr(0, i));
   }

   return("");
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "The path name is \"" << getPathName(path) << "\"\n";
}
```

Discussion

Example 10-23 is trivial, especially if you've already looked at the previous few recipes, so there is no more to explain. However, as with many of the other recipes, the Boost Filesystem library provides a way to extract everything but the last part of the filename with its branch_path function. Example 10-24 shows how to use it.

Example 10-24. Getting the base path

```
 #include <iostream>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>

using namespace std;
using namespace boost::filesystem;

int main(int argc, char** argv) {
```

Recipe 10.16. Replacing a File Extension

Problem

Given a filename, or a path and filename, you want to replace the file's extension. For example, if you are given thesis.tex, you want to convert it to thesis.txt.

Solution

Use string's rfind and replace member functions to find the extension and replace it. Example 10-25 shows you how to do this.

Example 10-25. Replacing a file extension

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

using std::string;

void replaceExt(string& s, const string& newExt) {

   string::size_type i = s.rfind('.', s.length( ));

   if (i != string::npos) {
      s.replace(i+1, newExt.length( ), newExt);
   }
}

int main(int argc, char** argv) {

   string path = argv[1];

   replaceExt(path, "foobar");
   std::cout << "The new name is \"" << path << "\"\n";
}
```

Discussion

This solution is similar to the ones in the preceding recipes, but in this case I used replace to replace a portion of the string with a new string. replace has three parameters. The first parameter is the index where the replace should begin, and the second is the number of characters to delete from the destination string. The third parameter is the value that will be used to replace the deleted portion of the string.

See Also

Recipe 4.9

Recipe 10.17. Combining Two Paths into a Single Path

Problem

You have two paths and you have to combine them into a single path. You may have something like /usr/home/ryan as a first path, and utils/compilers as the second, and wish to get /usr/home/ryan/utils/compilers, without having to worry whether or not the first path ends with a path separator.

Solution

Treat the paths as strings and use the append operator, operator+=, to compose a full path out of partial paths. See Example 10-26.

Example 10-26. Combining paths

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

using std::string;

string pathAppend(const string& p1, const string& p2) {

   char sep = '/';
   string tmp = p1;

#ifdef _WIN32
  sep = '\\';
#endif

  if (p1[p1.length( )] != sep) { // Need to add a
     tmp += sep;                 // path separator
     return(tmp + p2);
  }
  else
     return(p1 + p2);
}

int main(int argc, char** argv) {

   string path = argv[1];

   std::cout << "Appending somedir\\anotherdir is \""
             << pathAppend(path, "somedir\\anotherdir") << "\"\n";
}
```

Discussion

The code in Example 10-26 uses strings that represent paths, but there's no additional checking on the path class for validity and the paths used are only as portable as the values they contain. If, for example, these paths are retrieved from the user, you don't know if they're using the right OS-specific format, or if they contain illegal characters.

For many other recipes in this chapter I have included examples that use the Boost Filesystem library, and when working with paths, this approach has lots of benefits. As I discussed in Recipe 10.7, the Boost Filesystem library contains a path class that is a portable representation of a path to a file or directory. The operations in the Filesystem library mostly work with path objects, and as such, the path class can handle path composition from an absolute base and a relative path. (See Example 10-27.)

Chapter 11. Science and Mathematics

Introduction

C++ is a language well suited for scientific and mathematical programming, due to its flexibility, expressivity, and efficiency. One of the biggest advantages of C++ for numerical processing code is that it can help you avoid redundancy.

Historically, numerical code in many programming languages would repeat algorithms over and over for different kinds of numerical types (e.g., short, long, single, double, custom numerical types, etc.). C++ provides a solution to this problem of redundancy through templates. Templates enable you to write algorithms independantly of the data representation, a technique known commonly as generic programming.

C++ is not without its shortcomings with regards to numerical processing code. The biggest drawback with C++in contrast to specialized mathematical and scientific programming languagesis that the standard library is limited in terms of support of algorithms and data-types relevant to numerical programming. The biggest oversights in the standard library are arguably the lack of matrix types and arbitrary precision integers.

In this chapter, I will provide you with solutions to common numerical programming problems and demonstrate how to use generic programming techniques to write numerical code effectively. Where appropriate, I will recommend widely used open-source libraries with commercially friendly licenses and a proven track record. This chapter introduces the basic techniques of generic programming gradually from recipe to recipe.

Many programmers using C++ still distrust templates and generic programming due to their apparent complexity. When templates were first introduced into the language they were neither well implemented nor well understood by programmers and compiler implementers alike. As a result, many programmers, including yours truly, avoided generic programming in C++ for several years while the technology matured.

Today, generic programming is widely accepted as a powerful and useful programming paradigm, and is supported by the most popular programming languages. Furthermore, C++ compiler technology has improved by leaps and bounds, and modern compilers deal with templates in a much more standardized and efficient manner. As a result, modern C++ is a particularly powerful language for scientific and numerical applications.

Recipe 11.1. Computing the Number of Elements in a Container

Problem

You want to find the number of elements in a container.

Solution

You can compute the number of elements in a container by using the size member function or the distance function from the <algorithm> header as in Example 11-1.

Example 11-1. Computing the Number of Elements in a Container

```
 #include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main( ) {
  vector<int> v;
  v.push_back(0);
  v.push_back(1);
  v.push_back(2);
  cout << v.size( ) << endl;
  cout << distance(v.begin( ), v.end( )) << endl;
}
```

The program in Example 11-1 produces the following output:

```
 3
3
```

Discussion

The size member function, which returns the number of elements in a standard container, is the best solution in cases where the container object is accessible. I also demonstrated distance in Example 11-1, because when writing generic code it is common to work with only a pair of iterators. When working with iterators, you often don't have access to the type of the container or to its member functions.

The distance function, like most STL algorithms, is actually a template function. Since the type of the template argument can be deduced automatically by the compiler from the function arguments, you don't have to explicitly pass it as a template parameter. You can, of course, write out the template parameter explicitly if you want to, as follows:

```
   cout << distance<vector<int>::iterator>(v.begin( ), v.end( )) << endl;
```

The distance function performance depends on the kind of iterator used. It takes constant time if the input iterator is a random-access iterator; otherwise, it operates in linear time. (Iterator concepts are explained in Recipe 7.1.)

See Also

Recipe 15.1

Recipe 11.2. Finding the Greatest or Least Value in a Container

Problem

You want to find the maximum or minimum value in a container.

Solution

Example 11-2 shows how to find the minimum and maximum elements in a container by using the functions max_element and min_element found in the <algorithm> header. These functions return iterators that point to the first occurence of an element with the largest or smallest value, respectively.

Example 11-2. Finding the minimum or maximum element from a container

```
 #include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int getMaxInt(vector<int>& v) {
  return *max_element(v.begin( ), v.end( ));
}

int getMinInt(vector<int>& v) {
  return *min_element(v.begin( ), v.end( ));
}

int main( ) {
  vector<int> v;
  for (int i=10; i < 20; ++i) v.push_back(i);
  cout << "min integer = " << getMinInt(v) << endl;
  cout << "max integer = " << getMaxInt(v) << endl;
}
```

The program in Example 11-2 produces the following output:

```
 min integer = 10
max integer = 19
```

Discussion

You may have noticed the dereferencing of the return value from the calls to min_element and max_element. This is because these functions return iterators and not actual values, so the results have to be dereferenced. You may find it a minor inconvenience to have to dereference the return type, but it avoids unnecssarily copying the return value. This can be especially significant when the return value has expensive copy semantics (e.g., large strings).

The generic algorithms provided by the standard library are obviously quite useful, but it is more important for you to be able to write your own generic functions for getting the minimum and maximum value from a container. For instance, let's say that you want a single function which returns the minimum and maximum values by modifying reference parameters instead of returning them in a pair or some other structure. This is demonstrated in Example 11-3.

Example 11-3. Generic function for returning the minimum and maximum value

```
 #include <algorithm>
#include <vector>
#include <iostream>
```

Recipe 11.3. Computing the Sum and Mean of Elements in a Container

Problem

You want to compute the sum and mean of elements in a container of numbers.

Solution

You can use the accumulate function from the <numeric> header to compute the sum, and then divide by the size to get the mean. Example 11-5 demonstrates this using a vector.

Example 11-5. Computing the sum and mean of a container

```
 #include <numeric>
#include <iostream>
#include <vector>

using namespace std;

int main( ) {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  int sum = accumulate(v.begin( ), v.end( ), 0);
  double mean = double(sum) / v.size( );
  cout << "sum = " << sum << endl;
  cout << "count = " << v.size( ) << endl;
  cout << "mean = " << mean << endl;
}
```

The program in Example 11-5 produces the following output:

```
 sum = 10
count = 4
mean = 2.5
```

Discussion

The accumulate function generally provides the most efficient and simplest method to find the sum of all the elements in a container.

Even though this recipe has a relatively simple solution, writing your own generic function to compute a mean is not so easy. Example 11-6 shows one way to write such a generic function:

Example 11-6. A generic function to compute the mean

```
 template<class Iter_T>
double computeMean(Iter_T first, Iter_T last) {
  return static_cast<double>(accumulate(first, last, 0.0))
    / distance(first, last);
}
```

The computeMean function in Example 11-6 is sufficient for most purposes but it has one restriction: it doesn't work with input iterators such as istream_iterator.

Recipe 11.4. Filtering Values Outside a Given Range

Problem

You want to ignore values from a sequence that fall above or below a given range.

Solution

Use the remove_copy_if function found in the <algorithm>, as shown in Example 11-8.

Example 11-8. Removing elements from a sequence below a value

```
 #include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

using namespace std;

struct OutOfRange
{
  OutOfRange(int min, int max)
    : min_(min), max_(max)
  { }
  bool operator( )(int x) {
    return (x < min_) || (x > max_);
  }
  int min_;
  int max_;
};

int main( )
{
  vector<int> v;
  v.push_back(6);
  v.push_back(12);
  v.push_back(18);
  v.push_back(24);
  v.push_back(30);
  remove_copy_if(v.begin( ), v.end( ),
    ostream_iterator<int>(cout, "\n"), OutOfRange(10,25));
}
```

The program in Example 11-8 produces the following output:

```
 12
18
24
```

Discussion

The remove_copy_if function copies the elements from one container to another container (or output iterator), ignoring any elements that satisfy a predicate that you provide (it probably would have been more accurate if the function was named copy_ignore_if). The function, however, does not change the size of the target container. If, as is often the case, the number of elements copied by remove_copy_if is fewer than the size of the target container, you will have to shrink the target container by calling the erase member function.

The function remove_copy_if requires a unary predicate (a functor that takes one argument and returns a boolean value) that returns true when an element should not be copied. In Example 11-8 the predicate is the function object OutOfRange. The OutOfRange constructor takes a lower and upper range, and

Recipe 11.5. Computing Variance, Standard Deviation, and Other Statistical Functions

Problem

You want to compute one or more of the common statistics such as variance, standard deviation, skew, and kurtosis of a sequence of numbers.

Solution

You can use the accumulate function from the <numeric> header to compute many meaningful statistical functions beyond simply the sum by passing custom function objects. Example 11-9 shows how to compute several important statistical functions, using accumulate.

Example 11-9. Statistical functions

```
 #include <numeric>
#include <cmath>
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>

using namespace std;

template<int N, class T>
T nthPower(T x) {
  T ret = x;
  for (int i=1; i < N; ++i) {
    ret *= x;
  }
  return ret;
}

template<class T, int N>
struct SumDiffNthPower {
  SumDiffNthPower(T x) : mean_(x) { };
  T operator( )(T sum, T current) {
    return sum + nthPower<N>(current - mean_);
  }
  T mean_;
};

template<class T, int N, class Iter_T>
T nthMoment(Iter_T first, Iter_T last, T mean)  {
  size_t cnt = distance(first, last);
  return accumulate(first, last, T( ), SumDiffNthPower<T, N>(mean)) / cnt;
}

template<class T, class Iter_T>
T computeVariance(Iter_T first, Iter_T last, T mean) {
  return nthMoment<T, 2>(first, last, mean);
}

template<class T, class Iter_T>
T computeStdDev(Iter_T first, Iter_T last, T mean) {
  return sqrt(computeVariance(first, last, mean));
}

template<class T, class Iter_T>
T computeSkew(Iter_T begin, Iter_T end, T mean) {
```

Recipe 11.6. Generating Random Numbers

Problem

You want to generate some random floating-point numbers in the interval of [0.0, 1.0) with a uniform distribution.

Solution

The C++ standard provides the C runtime function rand in the <cstdlib> header that returns a random number in the range of 0 to RAND_MAX inclusive. The RAND_MAX macro represents the highest value returnable by the rand function. A demonstration of using rand to generate random floating-point numbers is shown in Example 11-11.

Example 11-11. Generating random numbers using rand

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

double doubleRand( ) {
  return double(rand( )) / (double(RAND_MAX) + 1.0);
}

int main( ) {
  srand(static_cast<unsigned int>(clock( )));
  cout << "expect 5 numbers within the interval [0.0, 1.0)" << endl;
  for (int i=0; i < 5; i++) {
    cout << doubleRand( ) << "\n";
  }
  cout << endl;
}
```

The program in Example 11-11 should produce output similar to:

```
expect 5 numbers within the interval [0.0, 1.0)
0.010437
0.740997
0.34906
0.369293
0.544373
```

Discussion

To be precise, random number generation functions, including rand, return pseudo-random numbers as opposed to truly random numbers, so whenever I say random, I actually mean pseudo-random.

Before using the rand function you need to seed (i.e., initialize) the random number generator with a call to srand. This assures that subsequent calls to rand won't produce the same sequence of numbers each time the program is run. The simplest way to seed the random number generator is to pass the result from a call to clock from the <ctime> header as an unsigned int. Reseeding a random number generator causes number generation to be less random.

The rand function is limited in many ways. To begin with, it only generates integers, and only does so using a uniform distribution. Furthermore, the specific random number generation algorithm used is implementation specific and, thus, random number sequences are not reproducible from system to system given the same seed. This is a problem for certain kinds of applications, as well as when testing and

Recipe 11.7. Initializing a Container with Random Numbers

Problem

You want to fill an arbitrary container with random numbers.

Solution

You can use either the generate or generate_n functions from the <algorithm> header with a functor that returns random numbers. See Example 11-13 for an example of how to do this.

Example 11-13. Initializing containers with random numbers

```
 #include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
#include <cstdlib>

using namespace std;

struct RndIntGen
{
  RndIntGen(int l, int h)
    : low(l), high(h)
  { }
  int operator( )( ) const {
    return low + (rand( ) % ((high - low) + 1));
  }
private:
  int low;
  int high;
};

int main( ) {
  srand(static_cast<unsigned int>(clock( )));
  vector<int> v(5);
  generate(v.begin( ), v.end( ), RndIntGen(1, 6));
  copy(v.begin( ), v.end( ), ostream_iterator<int>(cout, "\n"));
}
```

The program in Example 11-13 should produce output similar to:

```
 3
1
2
6
4
```

Discussion

The standard C++ library provides the functions generate and generate_n specifically for filling containers with the result of a generator function. These functions accept a nullary functor (a function pointer or function object with no arguments) whose result is assigned to contiguous values in the container. Sample implementations of the generate and generate_n functions are shown in Example 11-14.

Example 11-14. Sample implementations of generate and generate_n

```
 template<class Iter_T, class Fxn_T>
void generate(Iter_T first, Iter_T last, Fxn_T f) {
  while (first != last) *first++ = f( );
```

Recipe 11.8. Representing a Dynamically Sized Numerical Vector

Problem

You want a type for manipulating numerical vectors with dynamic size.

Solution

You can use the valarray template from the <valarray> header. Example 11-15 shows how you can use the valarray template.

Example 11-15. Using valarray

```
 #include <valarray>
#include <iostream>

using namespace std;

int main( ) {
  valarray<int> v(3);
  v[0] = 1; v[1] = 2; v[2] = 3;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v = v + v;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
  v /= 2;
  cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
}
```

The program in Example 11-15 will output the following:

```
 1, 2, 3
2, 4, 6
1, 2, 3
```

Discussion

Despite its name, vector is not intended to be used as a numerical vector; rather, the valarray template is. The valarray is designed so that C++ implementations, especially those on high-performance machines, can apply specialized vector optimizations to it. The other big advantage of valarray is that it provides numerous overloaded operators specifically for working with numerical vectors. These operators provide such functionality as vector addition and scalar multiplication.

The valarray template can also be used with the standard algorithms like a C-style array. See Example 11-16 to see how you can create iterators to the beginning of, and one past the end of, a valarray.

Example 11-16. Getting iterators to valarray

```
 template<class T>
T* valarray_begin(valarray<T>& x) {
  return &x[0];
}

template<class T>
T* valarray_end(valarray<T>& x) {
  return valarray_begin(x) + x.size( );
}
```

Even though it appears somewhat academic, you should not try to create an end iterator for a valarray by writing &x[x.size( )]. If this works, it is only by accident since indexing a valarray past the last valid index

Recipe 11.9. Representing a Fixed-Size Numerical Vector

Problem

You want an efficient representation for manipulating constant-sized numerical vectors

Solution

On many common software architectures, it is more efficient to use a custom vector implementation than a valarray when the size is known at compile time. Example 11-17 provides a sample implementation of a fixed-size vector template called a kvector.

Example 11-17. kvector.hpp

```
 #include <algorithm>
#include <cassert>

template<class Value_T, unsigned int N>
class kvector
{
public:
  // public fields
  Value_T m[N];

  // public typedefs
  typedef Value_T value_type;
  typedef Value_T* iterator;
  typedef const Value_T* const_iterator;
  typedef Value_T& reference;
  typedef const Value_T& const_reference;
  typedef size_t size_type;

  // shorthand for referring to kvector
  typedef kvector self;

  // member functions
  template<typename Iter_T>
  void copy(Iter_T first, Iter_T last) { copy(first, last, begin( )); }
  iterator begin( ) { return m; }
  iterator end( ) { return m + N; }
  const_iterator begin( ) const { return m; }
  const_iterator end( ) const { return m + N; }
  reference operator[](size_type n) { return m[n]; }
  const_reference operator[](size_type n) const { return m[n]; }
  static size_type size( ) { return N; }

  // vector operations
  self& operator+=(const self& x) {
      for (int i=0; i<N; ++i) m[i] += x.m[i]; return *this;
  }
  self& operator-=(const self& x) {
      for (int i=0; i<N; ++i) m[i] -= x.m[i]; return *this;
  }

  // scalar operations
  self& operator=(value_type x) {
    std::fill(begin( ), end( ), x); return *this;
  }
  self& operator+=(value_type x) {
    for (int i=0; i<N; ++i) m[i] += x; return *this;
  }
  self& operator-=(value_type x) {
```

Recipe 11.10. Computing a Dot Product

Problem

You have two containers of numbers that are the same length and you want to compute their dot product.

Solution

Example 11-19 shows how you can compute a dot product using the inner_product function from the <numeric> header.

Example 11-19. Computing the dot product

```
#include <numeric>
#include <iostream>
#include <vector>

using namespace std;

int main( ) {
  int v1[] = { 1, 2, 3 };
  int v2[] = { 4, 6, 8 };
  cout << "the dot product of (1,2,3) and (4,6,8) is ";
  cout << inner_product(v1, v1 + 3, v2, 0) << endl;
}
```

The program in Example 11-19 produces the following output:

```
the dot product of (1,2,3) and (4,6,8) is 40
```

Discussion

The dot product is a form of inner product known as the Euclidean Inner Product. The inner_product function is declared as follows:

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init);

template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, Binop2 op2);
```

The first form of inner_product sums the result of multiplying corresponding elements from two containers. The second form of the inner_product function allows you to supply your own pairwise operation and accumulation function. See Example 11-20 to see a sample implementation demonstrating how inner_product works.

Example 11-20. Sample implementation of inner_product( )

```
template<class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, Binop2 op2) {
  while (first != last) {
    BinOp(init, BinOp2(*first++, *first2++));
  }
  return init;
}
```

Because of its flexible implementation, you can use inner_product for many more purposes than just computing a dot product (e.g., you can use it to compute the distance between two vectors or compute the norm of a vector).

Recipe 11.11. Computing the Norm of a Vector

Problem

You want to find the norm (i.e., the length) of a numerical vector.

Solution

You can use the inner_product function from the <numeric> header to multiply a vector with itself as shown in Example 11-21.

Example 11-21. Computing the norm of a vector

```
#include <numeric>
#include <vector>
#include <cmath>
#include <iostream>

using namespace std;

template<typename Iter_T>
long double vectorNorm(Iter_T first, Iter_T last) {
  return sqrt(inner_product(first, last, first, 0.0L));
}

int main( ) {
  int v[] = { 3, 4 };
  cout << "The length of the vector (3,4) is ";
  cout << vectorNorm(v, v + 2) << endl;
}
```

The program in Example 11-21 produces the following output:

```
The length of the vector (3,4) is 5
```

Discussion

Example 11-21 uses the inner_product function from the <numeric> header to find the dot product of the numerical vector with itself. The square root of this is known as the vector norm or the length of a vector.

Rather than deduce the result type in the vectorNorm function, I chose to return a long double to lose as little data as possible. If a vector is a series of integers, it is unlikely that in a real example, that the distance can be meaningfully represented as an integer as well.

Recipe 11.12. Computing the Distance Between Two Vectors

Problem

You want to find the Euclidean distance between two vectors.

Solution

The Euclidean distance between two vectors is defined as the square root of the sum of squares of differences between corresponding elements. This can be computed as shown in Example 11-22.

Example 11-22. Finding the distance between two vectors

```
 #include <cmath>
#include <iostream>

using namespace std;

template<class Iter_T, class Iter2_T>
double vectorDistance(Iter_T first, Iter_T last, Iter2_T first2) {
  double ret = 0.0;
  while (first != last) {
    double dist = (*first++) - (*first2++);
    ret += dist * dist;
  }
  return ret > 0.0 ? sqrt(ret) : 0.0;
}

int main( ) {
  int v1[] = { 1, 5 };
  int v2[] = { 4, 9 };
  cout << "distance between vectors (1,5) and (4,9) is ";
  cout << vectorDistance(v1, v1 + 2, v2) << endl;
}
```

The program in Example 11-22 produces the following output:

```
 distance between vectors (1,5) and (4,9) is 5
```

Discussion

Example 11-22 is a straightforward recipe that shows how to write a simple generic function in the style of the STL. To compute the vector distances, I could have instead used the inner_product function I chose not to use a functor, because it was more complex than was strictly needed. Example 11-23 shows how you can compute vector distance using a functor and the inner_product function from the <numeric> header.

Example 11-23. Computing the distance between vectors using inner_product

```
 #include <numeric>
#include <cmath>
#include <iostream>
#include <functional>

using namespace std;

template<class Value_T>
struct DiffSquared {
  Value_T operator( )(Value_T x, Value_T y) const {
    return (x - y) * (x - y);
  }
};
```

Recipe 11.13. Implementing a Stride Iterator

Problem

You have a contiguous series of numbers and you want to iterate through the elements *n* at a time.

Solution

Example 11-24 presents a stride iterator class as a separate header file.

Example 11-24. stride_iter.hpp

```
 #ifndef STRIDE_ITER_HPP
#define STRIDE_ITER_HPP

#include <iterator>
#include <cassert>

template<class Iter_T>
class stride_iter
{
public:
  // public typedefs
  typedef typename std::iterator_traits<Iter_T>::value_type value_type;
  typedef typename std::iterator_traits<Iter_T>::reference reference;
  typedef typename std::iterator_traits<Iter_T>::difference_type
    difference_type;
  typedef typename std::iterator_traits<Iter_T>::pointer pointer;
  typedef std::random_access_iterator_tag iterator_category;
  typedef stride_iter self;

  // constructors
  stride_iter( ) : m(NULL), step(0) { };
  stride_iter(const self& x) : m(x.m), step(x.step) { }
  stride_iter(Iter_T x, difference_type n) : m(x), step(n) { }

  // operators
  self& operator++( ) { m += step; return *this; }
  self operator++(int) { self tmp = *this; m += step; return tmp; }
  self& operator+=(difference_type x) { m += x * step; return *this; }
  self& operator--( ) { m -= step; return *this; }
  self operator--(int) { self tmp = *this; m -= step; return tmp; }
  self& operator-=(difference_type x) { m -= x * step; return *this; }
  reference operator[](difference_type n) { return m[n * step]; }
  reference operator*( ) { return *m; }

  // friend operators
  friend bool operator==(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m == y.m;
  }
  friend bool operator!=(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m != y.m;
  }
  friend bool operator<(const self& x, const self& y) {
    assert(x.step == y.step);
    return x.m < y.m;
  }
  friend difference_type operator-(const self& x, const self& y) {
    assert(x.step == y.step);
    return (x.m - y.m) / x.step;
```

Recipe 11.14. Implementing a Dynamically Sized Matrix

Problem

You need to store and represent Matricies of numbers where the dimensions (number of rows and columns) are not known at compile time.

Solution

Example 11-28 provides a general purpose and efficient implementation of a dynamically sized matrix class using the stride iterator from Recipe 11.12 and a valarray.

Example 11-28. matrix.hpp

```
#ifndef MATRIX_HPP
#define MATRIX_HPP

#include "stride_iter.hpp" // see Recipe 11.12

#include <valarray>
#include <numeric>
#include <algorithm>

template<class Value_T>
class matrix
{
public:
  // public typedefs
  typedef Value_T value_type;
  typedef matrix self;
  typedef value_type* iterator;
  typedef const value_type* const_iterator;
  typedef Value_T* row_type;
  typedef stride_iter<value_type*> col_type;
  typedef const value_type* const_row_type;
  typedef stride_iter<const value_type*> const_col_type;

  // constructors
  matrix( ) : nrows(0), ncols(0), m( ) { }
  matrix(int r, int c) : nrows(r), ncols(c), m(r * c) { }
  matrix(const self& x) : m(x.m), nrows(x.nrows), ncols(x.ncols) { }

  template<typename T>
  explicit matrix(const valarray<T>& x)
  : m(x.size( ) + 1), nrows(x.size( )), ncols(1)
  {
    for (int i=0; i<x.size( ); ++i) m[i] = x[i];
  }

  // allow construction from matricies of other types
  template<typename T>
  explicit matrix(const matrix<T>& x)
  : m(x.size( ) + 1), nrows(x.nrows), ncols(x.ncols)
  {
    copy(x.begin( ), x.end( ), m.begin( ));
  }

  // public functions
  int rows( ) const { return nrows; }
  int cols( ) const { return ncols; }
  int size( ) const { return nrows * ncols; }
```

Recipe 11.15. Implementing a Constant-Sized Matrix

Problem

You want an efficient matrix implementation where the dimensions (i.e., number of rows and columns) are constants known at compile time.

Solution

When the dimensions of a matrix are known at compile time, the compiler can more easily optimize an implementation that accepts the row and columns as template parameters as shown in Example 11-30.

Example 11-30. kmatrix.hpp

```
 #ifndef KMATRIX_HPP
#define KMATRIX_HPP

#include "kvector.hpp"
#include "kstride_iter.hpp"

template<class Value_T, int Rows_N, int Cols_N>
class kmatrix
{
public:
  // public typedefs
  typedef Value_T value_type;
  typedef kmatrix self;
  typedef Value_T* iterator;
  typedef const Value_T* const_iterator;
  typedef kstride_iter<Value_T*, 1> row_type;
  typedef kstride_iter<Value_T*, Cols_N> col_type;
  typedef kstride_iter<const Value_T*, 1> const_row_type;
  typedef kstride_iter<const Value_T*, Cols_N> const_col_type;

  // public constants
  static const int nRows = Rows_N;
  static const int nCols = Cols_N;

  // constructors
  kmatrix( ) { m = Value_T( ); }
  kmatrix(const self& x) { m = x.m; }
  explicit kmatrix(Value_T& x) { m = x.m; }

  // public functions
  static int rows( ) { return Rows_N; }
  static int cols( ) { return Cols_N; }
  row_type row(int n) { return row_type(begin( ) + (n * Cols_N)); }
  col_type col(int n) { return col_type(begin( ) + n); }
  const_row_type row(int n) const {
    return const_row_type(begin( ) + (n * Cols_N));
  }
  const_col_type col(int n) const {
    return const_col_type(begin( ) + n);
  }
  iterator begin( ) { return m.begin( ); }
  iterator end( ) { return m.begin( ) + size( ); }
  const_iterator begin( ) const { return m; }
  const_iterator end( ) const { return m + size( ); }
  static int size( ) { return Rows_N * Cols_N; }

  // operators
  row_type operator[](int n) { return row(n); }
```

Recipe 11.16. Multiplying Matricies

Problem

You want to perform efficient multiplication of two matricies.

Solution

Example 11-32 shows an implementation of matrix multiplication that can be used with both the dynamic- or fixed-size matrix implementations. This algorithm technically produces the result of the equation A=A+B*C, which is, perhaps surprisingly, an equation more efficiently computed than A=B*C.

Example 11-32. Matrix multiplication

```
 #include "matrix.hpp" // recipe 11.13
#include "kmatrix.hpp" // recipe 11.14
#include <iostream>
#include <cassert>

using namespace std;

template<class M1, class M2, class M3>
void matrixMultiply(const M1& m1, const M2& m2, M3& m3)
{
  assert(m1.cols( ) == m2.rows( ));
  assert(m1.rows( ) == m3.rows( ));
  assert(m2.cols( ) == m3.cols( ));
  for (int i=m1.rows( )-1; i >= 0; --i) {
    for (int j=m2.cols( )-1; j >= 0; --j) {
      for (int k = m1.cols( )-1; k >= 0; --k) {
        m3[i][j] += m1[i][k] * m2[k][j];
      }
    }
  }
}

int main( )
{
  matrix<int> m1(2, 1);
  matrix<int> m2(1, 2);
  kmatrix<int, 2, 2> m3;
  m3 = 0;
  m1[0][0] = 1; m1[1][0] = 2;
  m2[0][0] = 3; m2[0][1] = 4;
  matrixMultiply(m1, m2, m3);
  cout << "(" << m3[0][0] << ", " << m3[0][1] << ")" << endl;
  cout << "(" << m3[1][0] << ", " << m3[1][1] << ")" << endl;
}
```

Example 11-32 produces the following output:

```
 (3, 4)
(6, 8)
```

Discussion

When multiplying two matricies, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix has the number of rows of the first matrix and the number of columns of the second matrix. I assure that these conditions are true during debug builds by using the assert macro found in the <cassert> header.

Recipe 11.17. Computing the Fast Fourier Transform

Problem

You want to compute the Discrete Fourier Transform (DFT) efficiently using the Fast Fourier Transform (FFT) algorithm.

Solution

The code in Example 11-33 provides a basic implementation of the FFT.

Example 11-33. FFT implementation

```
 #include <iostream>
#include <complex>
#include <cmath>
#include <iterator>

using namespace std;

unsigned int bitReverse(unsigned int x, int log2n) {
  int n = 0;
  int mask = 0x1;
  for (int i=0; i < log2n; i++) {
    n <<= 1;
    n |= (x & 1);
    x >>= 1;
  }
  return n;
}

const double PI = 3.1415926536;

template<class Iter_T>
void fft(Iter_T a, Iter_T b, int log2n)
{
  typedef typename iterator_traits<Iter_T>::value_type complex;
  const complex J(0, 1);
  int n = 1 << log2n;
  for (unsigned int i=0; i < n; ++i) {
    b[bitReverse(i, log2n)] = a[i];
  }
  for (int s = 1; s <= log2n; ++s) {
    int m = 1 << s;
    int m2 = m >> 1;
    complex w(1, 0);
    complex wm = exp(-J * (PI / m2));
    for (int j=0; j < m2; ++j) {
      for (int k=j; k < n; k += m) {
        complex t = w * b[k + m2];
        complex u = b[k];
        b[k] = u + t;
        b[k + m2] = u - t;
      }
      w *= wm;
    }
  }
}

int main( ) {
  typedef complex<double> cx;
  cx a[] = { cx(0,0), cx(1,1), cx(3,3), cx(4,4),
```

Recipe 11.18. Working with Polar Coordinates

Problem

You want to represent and manipulate polar coordinates.

Solution

The complex template from the <complex> header provides functions for conversion to and from polar coordinates. Example 11-34 shows how you can use the complex template class to represent and manipulate polar coordinates.

Example 11-34. Using complex template class to represent polar coordinates

```
 #include <complex>
#include <iostream>

using namespace std;

int main( )  {
  double rho = 3.0; // magnitude
  double theta = 3.141592 / 2; // angle
  complex<double> coord = polar(rho, theta);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
  coord += polar(4.0, 0.0);
  cout << "rho = " << abs(coord) << ", theta = " << arg(coord) << endl;
}
```

Example 11-34 produces the following output:

```
 rho = 3, theta = 1.5708
rho = 5, theta = 0.643501
```

Discussion

There is a natural relationship between polar coordinates and complex numbers. Even though the two are somewhat interchangeable, it is generally not a good idea to use the same type to represent different concepts. Since using the complex template to represent polar coordinates is inelegant, I have provided a polar coordinate class that is more natural to use in Example 11-35.

Example 11-35. A polar coordinate class

```
 #include <complex>
#include <iostream>

using namespace std;

template<class T>
struct BasicPolar
{
  public:
    typedef BasicPolar self;

    // constructors
    BasicPolar( ) : m( ) {  }
    BasicPolar(const self& x) : m(x.m) {  }
    BasicPolar(const T& rho, const T& theta) : m(polar(rho, theta)) { }

    // assignment operations
    self operator-( ) { return Polar(-m); }
    self& operator+=(const self& x) { m += x.m; return *this; }
    self& operator-=(const self& x) { m -= x.m; return *this; }
```

Recipe 11.19. Performing Arithmetic on Bitsets

Problem

You want to perform basic arithmetic and comparison operations on a set of bits as if it were a binary representation of an unsigned integer number.

Solution

The functions in Example 11-36 provide functions that allow arithmetic and comparison of bitset class template from the <bitset> header as if it represents an unsigned integer.

Example 11-36. bitset_arithmetic.hpp

```
 #include <stdexcept>
#include <bitset>

bool fullAdder(bool b1, bool b2, bool& carry) {
  bool sum = (b1 ^ b2) ^ carry;
  carry = (b1 && b2) || (b1 && carry) || (b2 && carry);
  return sum;
}

bool fullSubtractor(bool b1, bool b2, bool& borrow) {
  bool diff;
  if (borrow) {
    diff = !(b1 ^ b2);
    borrow = !b1 || (b1 && b2);
  }
  else {
    diff = b1 ^ b2;
    borrow = !b1 && b2;
  }
  return diff;
}

template<unsigned int N>
bool bitsetLtEq(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return false;
    if (!x[i] && y[i]) return true;
  }
  return true;
}

template<unsigned int N>
bool bitsetLt(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return false;
    if (!x[i] && y[i]) return true;
  }
  return false;
}

template<unsigned int N>
bool bitsetGtEq(const std::bitset<N>& x, const std::bitset<N>& y)
{
  for (int i=N-1; i >= 0; i--) {
    if (x[i] && !y[i]) return true;
    if (!x[i] && y[i]) return false;
```

Recipe 11.20. Representing Large Fixed-Width Integers

Problem

You need to perform arithmetic of numbers larger than can be represented by a long int.

Solution

The BigInt template in Example 11-38 uses the bitset from the <bitset> header to allow you to represent unsigned integers using a fixed number of bits specified as a template parameter.

Example 11-38. big_int.hpp

```
 #ifndef BIG_INT_HPP
#define BIG_INT_HPP

#include <bitset>

#include "bitset_arithmetic.hpp" // Recipe 11.20

template<unsigned int N>
class BigInt
{
  typedef BigInt self;
public:
  BigInt( ) : bits( ) { }
  BigInt(const self& x) : bits(x.bits) { }
  BigInt(unsigned long x) {
    int n = 0;
    while (x) {
      bits[n++] = x & 0x1;
      x >>= 1;
    }
  }
  explicit BigInt(const std::bitset<N>& x) : bits(x) { }

  // public functions
  bool operator[](int n) const { return bits[n]; }
  unsigned long toUlong( ) const { return bits.to_ulong( ); }

  // operators
  self& operator<<=(unsigned int n) {
    bits <<= n;
    return *this;
  }
  self& operator>>=(unsigned int n) {
    bits >>= n;
    return *this;
  }
  self operator++(int) {
    self i = *this;
    operator++( );
    return i;
  }
  self operator--(int) {
    self i = *this;
    operator--( );
    return i;
  }
  self& operator++( ) {
    bool carry = false;
    bits[0] = fullAdder(bits[0], 1, carry);
```

Recipe 11.21. Implementing Fixed-Point Numbers

Problem

You want to perform computations on real numbers using a fixed-point representation of a real number rather than using a floating-point type.

Solution

Example 11-40 provides the implementation of a fixed-point real number, where the number of places to the right of the binary point is a template parameter. For instance basic_fixed_real<10> has 10 binary digits to the right of the binary point, allowing it to represent numbers up to a precision of 1/1,024.

Example 11-40. Representing real numbers using a fixed-point implementation

```
#include <iostream>

using namespace std;

template<int E>
struct BasicFixedReal
{
  typedef BasicFixedReal self;
  static const int factor = 1 << (E - 1);
  BasicFixedReal( ) : m(0) { }
  BasicFixedReal(double d) : m(static_cast<int>(d * factor)) { }
  self& operator+=(const self& x) { m += x.m; return *this; }
  self& operator-=(const self& x) { m -= x.m; return *this; }
  self& operator*=(const self& x) { m *= x.m; m >>= E; return *this; }
  self& operator/=(const self& x) { m /= x.m; m *= factor; return *this; }
  self& operator*=(int x) { m *= x; return *this; }
  self& operator/=(int x) { m /= x; return *this; }
  self operator-( ) { return self(-m); }
  double toDouble( ) const { return double(m) / factor;  }

  // friend functions
  friend self operator+(self x, const self& y) { return x += y; }
  friend self operator-(self x, const self& y) { return x -= y; }
  friend self operator*(self x, const self& y) { return x *= y; }
  friend self operator/(self x, const self& y) { return x /= y; }

  // comparison operators
  friend bool operator==(const self& x, const self& y) { return x.m == y.m; }
  friend bool operator!=(const self& x, const self& y) { return x.m != y.m; }
  friend bool operator>(const self& x, const self& y) { return x.m > y.m; }
  friend bool operator<(const self& x, const self& y) { return x.m < y.m; }
  friend bool operator>=(const self& x, const self& y) { return x.m >= y.m; }
  friend bool operator<=(const self& x, const self& y) { return x.m <= y.m; }
private:
  int m;
};

typedef BasicFixedReal<10> FixedReal;

int main( ) {
  FixedReal x(0);
  for (int i=0; i < 100; ++i) {
    x += FixedReal(0.0625);
  }
  cout << x.toDouble( ) << endl;
}
```

Chapter 12. Multithreading

PREV NEXT

Introduction

This chapter describes how to write multithreaded programs in C++ using the Boost Threads library written by William Kempf. Boost is a set of open source, peer-reviewed, portable, high-performance libraries ranging from simple data structures to a complex parsing framework. The Boost Threads library is a framework for multithreading. For more information on Boost, see www.boost.org.

Standard C++ contains no native support for multithreading, so it is not possible to write portable multithreaded code the same way you would write portable code that uses other standard library classes like string, vector, list, and so on. The Boost Threads library goes a long way toward making a standard, portable multithreading library though, and it is designed to minimize many common multithreading headaches.

Unlike the standard library or third-party libraries, however, using a multithreading library is not as easy as unzipping it into a directory, adding your #includes, and coding away. For all but trivial multithreaded applications, you must design carefully using proven patterns and known tactics to avoid bugs that are otherwise virtually guaranteed to happen. In a typical, single-threaded application, it is easy to find common programming errors: off-by-one loops, dereferencing a null or deleted pointer, loss of precision on floating-point conversions, and so on. Multithreaded programs are different. Not only is it tedious to keep track of what several threads are doing in your debugger, but multithreaded programs are nondeterministic, meaning that bugs may only show up under rare or complicated circumstances.

It is for this reason that this chapter should not be your introduction to multithreaded programming. If you have already done some programming with threads, but not with C++ or the Boost Threads library, this chapter will get you on your way. But describing the fundamentals of multithreaded programming is beyond the scope of this book. If you have never done any multithreaded programming before, then you may want to read an introductory book on multithreading, though such titles are scant because most programmers don't use threads (though they probably ought to).

Much of the Boost documentation and some of the following recipes discuss the classes using the concept/model idea. A concept is an abstract description of something, usually a class, and its behavior, without any assumptions about its implementation. Typically, this description includes construction and destruction behavior, and each of the methods, including their preconditions, parameters, and postconditions. For example, the concept of a Mutex is something that can be locked and unlocked by one thread at a time. A model is a concrete manifestation of a concept, such as the mutex class in the Boost Threads library. A refinement on a concept is a specialization of it, such as a ReadWriteMutex, which is a Mutex with some additional behavior.

Finally, threads are doing one of three things: working, waiting for something, or ready to go but not waiting for anything or doing any work. These states are called run, wait, and ready. These are the terms I will use in the following recipes.

Recipe 12.1. Creating a Thread

Problem

You want to create a thread to perform some task while the main thread continues its work.

Solution

Create an object of the class thread, and pass it a functor that does the work. The creation of the thread object will instantiate an operating system thread that begins executing at operator( ) on your functor (or the beginning of the function if you passed in a function pointer instead). Example 12-1 shows you how.

Example 12-1. Creating a thread

```
 #include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/xtime.hpp>

struct MyThreadFunc {
   void operator( )( ) {
      // Do something long-running...
   }
} threadFun;

int main( ) {

   boost::thread myThread(threadFun); // Create a thread that starts
                                      // running threadFun

   boost::thread::yield( ); // Give up the main thread's timeslice
                            // so the child thread can get some work
                            // done.

   // Go do some other work...

   myThread.join( ); // The current (i.e., main) thread will wait
                     // for myThread to finish before it returns

}
```

Discussion

Creating a thread is deceptively simple. All you have to do is create a thread object on the stack or the heap, and pass it a functor that tells it where it can begin working. For this discussion, a "thread" is actually two things. First, it's an object of the class thread, which is a C++ object in the conventional sense. When I am referring to this object, I will say "thread object." Then there is the thread of execution, which is an operating system thread that is represented by the tHRead object. When I say "thread" (not in fixed-width font), I mean the operating system thread.

Let's get right to the code in the example. The tHRead constructor takes a functor (or function pointer) that takes no arguments and returns void. Look at this line from Example 12-1:

```
 boost::thread myThread(threadFun);
```

This creates the myTHRead object on the stack, which represents a new operating system thread that begins executing tHReadFun. At that point, the code in threadFun and the code in main are, at least in theory, running in parallel. They may not exactly be running in parallel, of course, because your machine may have only one processor, in which case this is impossible (recent processor architectures have made this not quite true, but I'll ignore dual-core processors and the like for now). If you have only one

Recipe 12.2. Making a Resource Thread-Safe

Problem

You are using multiple threads in a program and you need to ensure a resource is not modified by more than one thread at a time. In general, this process is called making the resource thread-safe, or serializing access to it.

Solution

Use the class mutex, defined in *boost/thread/mutex.hpp*, to synchronize access among threads. Example 12-2 shows a simple use of a mutex object to control concurrent access to a queue.

Example 12-2. Making a class thread-safe

```
 #include <iostream>
#include <boost/thread/thread.hpp>
#include <string>

// A simple queue class; don't do this, use std::queue
template<typename T>
class Queue {
public:
   Queue( ) {}
  ~Queue( ) {}

   void enqueue(const T& x) {
      // Lock the mutex for this queue
      boost::mutex::scoped_lock lock(mutex_);
      list_.push_back(x);
      // A scoped_lock is automatically destroyed (and thus unlocked)
      // when it goes out of scope
   }

   T dequeue( ) {
      boost::mutex::scoped_lock lock(mutex_);

      if (list_.empty( ))
         throw "empty!";     // This leaves the current scope, so the
      T tmp = list_.front( ); // lock is released
      list_.pop_front( );
      return(tmp);
   } // Again: when scope ends, mutex_ is unlocked

private:
   std::list<T> list_;
   boost::mutex mutex_;
};

Queue<std::string> queueOfStrings;

void sendSomething( ) {
   std::string s;
   for (int i = 0; i < 10; ++i) {
      queueOfStrings.enqueue("Cyrus");
   }
}

void recvSomething( ) {
   std::string s;

   for (int i = 0; i < 10; ++i) {
```

Recipe 12.3. Notifying One Thread from Another

Problem

You are using a pattern where one thread (or group of threads) does something and it needs to let another thread (or group of threads) know about it. You may have a master thread that is handing out work to slave threads, or you may use one group of threads to populate a queue and another to remove the data from it and do something useful.

Solution

Use mutex and condition objects, declared in *boost/thread/mutex.hpp* and *boost/thread/condition.hpp*. You can create a condition for each situation you want threads to wait for, and notify any waiting threads on the condition. Example 12-4 shows how to use signaling in a master/slave threading model.

Example 12-4. Signaling between threads

```
 #include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/condition.hpp>
#include <boost/thread/mutex.hpp>
#include <list>
#include <string>

class Request { /*...*/ };

// A simple job queue class; don't do this, use std::queue
template<typename T>
class JobQueue {
public:
   JobQueue( ) {}
  ~JobQueue( ) {}

   void submitJob(const T& x) {
      boost::mutex::scoped_lock lock(mutex_);
      list_.push_back(x);
      workToBeDone_.notify_one( );
   }

   T getJob( ) {
      boost::mutex::scoped_lock lock(mutex_);

      workToBeDone_.wait(lock); // Wait until this condition is
                                // satisfied, then lock the mutex
      T tmp = list_.front( );
      list_.pop_front( );
      return(tmp);
   }

private:
   std::list<T> list_;
   boost::mutex mutex_;
   boost::condition workToBeDone_;
};

JobQueue<Request> myJobQueue;

void boss( ) {
   for (;;) {
      // Get the request from somewhere
```

Recipe 12.4. Initializing Shared Resources Once

Problem

You have a number of threads that are using a resource that must only be initialized once.

Solution

Either initialize the resource before the threads are started, or if you can't, use the call_once function defined in <boost/thread/once.hpp> and the type once_flag. Example 12-5 shows how to use call_once.

Example 12-5. Initializing something once

```
 #include <iostream>
#include <boost/thread/thread.hpp>
#include <boost/thread/once.hpp>

// Some sort of connection class that should only be initialized once
struct Conn {
   static void init( ) {++i_;}
   static boost::once_flag init_;
   static int i_;
   // ...
};

int Conn::i_ = 0;
boost::once_flag Conn::init_ = BOOST_ONCE_INIT;

void worker( ) {
   boost::call_once(Conn::init, Conn::init_);
   // Do the real work...
}

Conn c;  // You probably don't want to use a global, so see the
         // next Recipe

int main( ) {

   boost::thread_group grp;

   for (int i = 0; i < 100; ++i)
      grp.create_thread(worker);

   grp.join_all( );

   std::cout << c.i_ << '\n'; // c.i_ = 1
}
```

Discussion

A shared resource has to be initialized somewhere, and you may want the first thread to use it to do the initializing. A variable of type once_flag (whose exact type is platform-dependent) and the call_once function can keep multiple threads from re-initializing the same object. You have to do two things.

First, initialize your once_flag variable to the macro BOOST_ONCE_INIT . This is a platform-dependent value. In Example 12-5, the class Conn represents some sort of connection (database, socket, hardware, etc.) that I only want initialized once even though multiple threads may try to initialize it. This sort of thing comes up often when you want to load a library dynamically, perhaps one specified in an application config file. The once_flag is a static class variable because I only want one initialization, no matter how many instances of the class there may be. So, I give the flag a starting value

Recipe 12.5. Passing an Argument to a Thread Function

Problem

You have to pass an argument to your thread function, but the thread creation facilities in the Boost Threads library only accept functors that take no arguments.

Solution

Create a functor adapter that takes your parameters and returns a functor that takes no parameters. You can use the functor adapter where you would have otherwise put the thread functor. Take a look at Example 12-6 to see how this is done.

Example 12-6. Passing an argument to a thread function

```
 #include <iostream>
#include <string>
#include <functional>
#include <boost/thread/thread.hpp>

// A typedef to make the declarations below easier to read
typedef void (*WorkerFunPtr)(const std::string&);

template<typename FunT,    // The type of the function being called
         typename ParamT> // The type of its parameter
struct Adapter {
   Adapter(FunT f, ParamT& p) : // Construct this adapter and set the
      f_(f), p_(&p) {}          // members to the function and its arg

   void operator( )( ) { // This just calls the function with its arg
      f_(*p_);
   }
private:
   FunT    f_;
   ParamT* p_;  // Use the parameter's address to avoid extra copying
};

void worker(const std::string& s) {
   std::cout << s << '\n';
}

int main( ) {

   std::string s1 = "This is the first thread!";
   std::string s2 = "This is the second thread!";

   boost::thread thr1(Adapter<WorkerFunPtr, std::string>(worker, s1));
   boost::thread thr2(Adapter<WorkerFunPtr, std::string>(worker, s2));

   thr1.join( );
   thr2.join( );
}
```

Discussion

The fundamental problem you need to solve here is not specific to threading or Boost, but a general problem when you have to pass a functor with one signature to something that requires a different signature. The solution is to create an adapter.

The syntax can get a little messy, but essentially what Example 12-6 does is create a temporary functor

Chapter 13. Internationalization

PREV NEXT

Introduction

This chapter describes solutions to some common requirements when internationalizing C++ programs. Making software work in different locales (usually referred to as localization) usually requires solving two problems: formatting user-visible strings such that they obey local conventions (such as those for date, time, money, and numbers), and reconciling data in different character sets. This chapter deals mostly with the first issue, and only briefly with the second, because there is little standardized support for different character sets since most aspects of it are largely implementation dependent.

Most software will also run in countries other than the one where it was written. To support this practical reality, the C++ standard library has several facilities for writing code that will run in different countries. The design of these facilities, however, is different than many other standard library facilities such as strings, file input and output, containers, algorithms, and so forth. For example, the class that is used to represent a locale is locale, and is provided in the <locale> header. locale provides facilities for writing to and reading from streams using locale-specific formatting, and for getting information about a locale, such as the currency symbol or the date format. The standard only requires that a single locale be provided though, and that is the "C" or classic locale. The classic locale uses ANSI C conventions: American English conventions and 7-bit ASCII character encoding. It is up to the implementation whether it will provide locale instances for the various languages and regions.

There are three fundamental parts to the <locale> header. First, there is the locale class. It encapsulates all aspects of behavior for a locale that C++ supports, and it is your entry point to the different kinds of locale information you need to do locale-aware formatting. Second, the most granular part of a locale, and the concrete classes you will be working with, are called facets. An example of a facet is a class such as time_put for writing a date to a stream. Third, each facet belongs to a category, which is a way of grouping related facets together. Examples of categories are numeric, time, and monetary (the time_put facet I mentioned a moment ago belongs to the time category). I mention categories briefly in this chapter, but they only really come in handy when you are doing some more sophisticated stuff with locales, so I don't cover their use in depth here.

Every C++ program has at least one locale, referred to as the global locale (it is often implemented as a global static object). By default, it is the classic "C" locale unless you change it to something else. One of the locale constructors allows you to instantiate the user's preferred locale, although an implementation is free to define exactly what a user's "preferred" locale is.

In most cases, you will only use locales when writing to or reading from streams. This is the main focus of this chapter.

Recipe 13.1. Hardcoding a Unicode String

Problem

You have to hardcode a Unicode, i.e., wide-character, string in a source file.

Solution

Do this by hardcoding the string with a prefix of L and typing the character into your source editor as you would any other string, or use the hexadecimal number that represents the Unicode character you're after. Example 13-1 shows how to do it both ways.

Example 13-1. Hardcoding a Unicode string

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

using namespace std;

int main( ) {

   // Create some strings with Unicode characters
   wstring ws1 = L"Infinity: \u221E";
   wstring ws2 = L"Euro: _";

   wchar_t w[] = L"Infinity: \u221E";

   wofstream out("tmp\\unicode.txt");
   out << ws2 << endl;
   wcout << ws2 << endl;
}
```

Discussion

Hardcoding a Unicode string is mostly a matter of deciding how you want to enter the string in your source editor. C++ provides a wide-character type, wchar_t, which can store Unicode strings. The exact implementation of wchar_t is implementation defined, but it is often UTF-32. The class wstring, defined in <string>, is a sequence of wchar_ts, just like the string class is a sequence of chars. (Strictly speaking, of course, wstring is a typedef for basic_string<wchar_t>).

The easiest way to enter Unicode characters is to use the L prefix to a string literal, as in Example 13-1:

```
 wstring ws1 = L"Infinity: \u2210";  // Use the code itself
wstring ws2 = L"Euro: _";           // Or just type it in
```

Now, you can write these wide-character strings to a wide-character stream, like this:

```
 wcout << ws1 << endl; // wcout is the wide char version of cout
```

This goes for files, too:

```
 wofstream out("tmp\\unicode.txt");
out << ws2 << endl;
```

The trickiest part of dealing with different character encodings isn't embedding the right characters in your source files, it's knowing what kind of character data you are getting back from a database, HTTP request, user input, and so on, and this is beyond the realm of the C++ standard. The C++ standard does not require a particular encoding, rather that the character encoding used by your operating system to store source files can be anything, as long as it supports at least the 96 characters used by the C++ language. For characters that are not part of this character set, called the basic source character set, the

Recipe 13.2. Writing and Reading Numbers

Problem

You need to write a number to a stream in a formatted way that obeys local conventions, which are different depending on where you are.

Solution

Imbue the stream you are writing to with the current locale and then write the numbers to it, as in Example 13-2, or you can set the global locale and then create a stream. The latter approach is explained in the discussion.

Example 13-2. Writing numbers using localized formatting

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

using namespace std;

// There is a global locale in the background that is set up by the
// runtime environment. It is the "C" locale by default. You can
// replace it with locale::global(const locale&).
int main( ) {
   locale loc(""); // Create a copy of the user's locale
   cout << "Locale name = " << loc.name( ) << endl;

   cout.imbue(loc); // Tell cout to use the formatting of
                    // the user's locale

   cout << "pi in locale " << cout.getloc( ).name( ) << " is "
        << 3.14 << endl;
}
```

Discussion

Example 13-2 shows how to use the user's locale to format a floating-point number. Doing so requires two steps, creating an instance of the locale class and then associating, or imbuing, the stream with it.

To begin with, Example 13-2 creates loc, which is a copy of the user's locale. You have to do this using locale's constructor with an empty string (and not the default constructor), like this:

```
locale loc("");
```

The difference is subtle but important, and I'll come back to it in a moment. Creating a locale object in this way creates a copy of the "user's locale," which is something that is implementation defined. This means that if the machine has been configured to use American English, locale::name( ) will return a locale string such as "en_US", "English_United States.1252", "english-american", and so on. The actual string is implementation defined, and the only one required to work by the C++ standard is "C".

By comparison, locale's default constructor returns a copy of the current global locale. There is a single, global locale object for every C++ program that is run (probably implemented as a static variable somewhere in the runtime libraryexactly how this is done is implementation defined). By default, it is the C locale, and you can replace it with locale::global(locale& loc). When streams are created, they use the global locale at the time of creation, which means that cin, cout, cerr, wcin, wcout, and wcerr use the C locale, so you have to change them explicitly if you want the formatting to obey a certain locale's conventions.

Recipe 13.3. Writing and Reading Dates and Times

Problem

You need to display or read dates and times using local formatting conventions.

Solution

Use the time_t type and tm struct from <ctime>, and the date and time facets provided in <locale>, to write and read dates and times (facets are described in the discussion in a moment). See Example 13-4 for a sample.

Example 13-4. Writing and reading dates

```
 #include <iostream>
#include <ctime>
#include <locale>
#include <sstream>
#include <iterator>

using namespace std;

void translateDate(istream& in, ostream& out) {

   // Create a date reader
   const time_get<char>& dateReader =
     use_facet<time_get<char> >(in.getloc( ));

   // Create a state object, which the facets will use to tell
   // us if there was a problem.
   ios_base::iostate state = 0;

   // End marker
   istreambuf_iterator<char> end;

   tm t; // Time struct (from <ctime>)

   // Now that all that's out of the way, read in the date from
   // the input stream and put it in a time struct.
   dateReader.get_date(in, end, in, state, &t);

   // Now the date is in a tm struct. Print it to the out stream
   // using its locale. Make sure you only print out what you
   // know is valid in t.
   if (state == 0 || state == ios_base::eofbit) {
      // The read succeeded.
      const time_put<char>& dateWriter =
        use_facet<time_put<char> >(out.getloc( ));

      char fmt[] = "%x";

      if (dateWriter.put(out, out, out.fill( ),
                         &t, &fmt[0], &fmt[2]).failed( ))
        cerr << "Unable to write to output stream.\n";
   } else {
      cerr << "Unable to read cin!\n";
   }
}

int main( ) {

   cin.imbue(locale("english"));
```

Recipe 13.4. Writing and Reading Currency

Problem

You need to write or read a formatted currency value to or from a stream.

Solution

Use the money_put and money_get facets to write and read currency, as shown in Example 13-6.

Example 13-6. Writing and reading currency

```
 #include <iostream>
#include <locale>
#include <string>
#include <sstream>

using namespace std;

long double readMoney(istream& in, bool intl = false) {

   long double val;

   // Create a reader facet
   const money_get<char>& moneyReader =
     use_facet<money_get<char> >(in.getloc( ));

   // End marker
   istreambuf_iterator<char> end;

   // State variable for detecting errors
   ios_base::iostate state = 0;

   moneyReader.get(in, end, intl, in, state, val);

   // failbit will be set if something went wrong
   if (state != 0 && !(state & ios_base::eofbit))
      throw "Couldn't read money!\n";

   return(val);
}

void writeMoney(ostream& out, long double val, bool intl = false) {

   // Create a writer facet
   const money_put<char>& moneyWriter =
     use_facet<money_put<char> >(out.getloc( ));

   // Write to the stream. Call failed( ) (the return value is an
   // ostreambuf_iterator) to see if anything went wrong.
   if (moneyWriter.put(out, intl, out, out.fill( ), val).failed( ))
      throw "Couldn't write money!\n";
}

int main( ) {

   long double val = 0;
   float exchangeRate = 0.775434f;  // Dollars to Euros
   locale locEn("english");
   locale locFr("french");

   cout << "Dollars: ";
```

Recipe 13.5. Sorting Localized Strings

Problem

You have a sequence of strings that contain non-ASCII characters, and you need to sort according to local convention.

Solution

The locale class has built-in support for comparing characters in a given locale by overriding operator. You can use an instance of the locale class as your comparison functor when you call any standard function that takes a functor for comparison. (See Example 13-8.)

Example 13-8. Locale-specific sorting

```
 #include <iostream>
#include <locale>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

bool localeLessThan (const string& s1, const string& s2) {

   const collate<char>& col =
     use_facet<collate<char> >(locale( )); // Use the global locale

   const char* pb1 = s1.data( );
   const char* pb2 = s2.data( );

   return (col.compare(pb1, pb1 + s1.size( ),
                       pb2, pb2 + s2.size( )) < 0);
}

int main( ) {

   // Create two strings, one with a German character
   string s1 = "diät";
   string s2 = "dich";

   vector<string> v;
   v.push_back(s1);
   v.push_back(s2);

   // Sort without giving a locale, which will sort according to the
   // current global locale's rules.
   sort(v.begin( ), v.end( ));
   for (vector<string>::const_iterator p = v.begin( );
        p != v.end( ); ++p)
      cout << *p << endl;

   // Set the global locale to German, and then sort
   locale::global(locale("german"));
   sort(v.begin( ), v.end( ), localeLessThan);
   for (vector<string>::const_iterator p = v.begin( );
        p != v.end( ); ++p)
      cout << *p << endl;
}
```

The first sort follows ASCII sorting convention, and therefore the output looks like this:

PREV NEXT

Chapter 14. XML

PREV NEXT

Introduction

XML is important in many areas, including information storage and retrieval, publishing, and network communication; in this chapter, you'll learn to work with XML in C++. Because this book is about C++ rather than XML, I'll assume you already have some experience with the various XML-related technologies I discuss, including SAX, DOM, XML Schema, XPath, and XSLT. Don't worry if you're not an expert in all of these areas; the recipes in this chapter are more or less independent of each other, so you should be able to skip some of the recipes and still understand the rest. In any case, each recipe provides a quick explanation of the XML concepts and tools it uses.

If you come from another programming language, such as Java, you may expect to find the tools for XML processing in C++ to be included in the C++ standard library. Unfortunately, XML was in its infancy when the C++ standard was approved, and while there's strong interest in adding XML processing to a future version of the C++ standard library, for now you will have to rely on the collection of excellent third-party XML libraries available in C++.

Before you start reading recipes, you may want download and install the libraries I'll be covering in this chapter. Table 14-1 shows the homepage of each library; Table 14-2 shows the features of each library and the recipes that use the library. The table doesn't show each library's exact level of conformance to the various XML specifications and recommendations because this information is likely to change in the near future.

Table 14-1. C++ libraries for XML

Library name	Homepage
TinyXml	www.grinninglizard.com/tinyxml
Xerxes	xml.apache.org/xerces-c
Xalan	xml.apache.org/xalan-c
Pathan 1	software.decisionsoft.com/pathanIntro.html
Boost.Serialization	www.boost.org/libs/serialization

Table 14-2. How each library is used

Library name	Features	Recipes
TinyXml	DOM (nonstandard)	Recipe 14.1
Xerxes	SAX2, DOM, XML Schema	Recipe 14.2-Recipe 14.8
Xalan	XSLT, XPath	Recipe 14.7-Recipe 14.8
Pathan	XPath	Recipe 14.8

Recipe 14.1. Parsing a Simple XML Document

Problem

You have a collection of data stored in an XML document. You want to parse the document and turn the data it contains into a collection of C++ objects. Your XML document is small enough to fit into memory and doesn't use an internal Document Type Definition (DTD) or XML Namespaces.

Solution

Use the TinyXml library. First, define an object of type TiXmlDocument and call its LoadFile() method, passing the pathname of your XML document as its argument. If LoadFile( ) returns true, your document has been successfully parsed. If parsing was successful, call the RootElement() method to obtain a pointer to an object of type TiXmlElement representing the document root. This object has a hierarchical structure that reflects the structure of your XML document; by traversing this structure, you can extract information about the document and use this information to create a collection of C++ objects.

For example, suppose you have an XML document *animals.xml* representing a collection of circus animals, as shown in Example 14-1. The document root is named animalList and has a number of child animal elements each representing an animal owned by the Feldman Family Circus. Suppose you also have a C++ class named Animal, and you want to construct a std::vector of Animals corresponding to the animals listed in the document.

Example 14-1. An XML document representing a list of circus animals

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals -->

<animalList>
    <animal>
        <name>Herby</name>
        <species>elephant</species>
        <dateOfBirth>1992-04-23</dateOfBirth>
        <veterinarian name="Dr. Hal Brown" phone="(801)595-9627"/>
        <trainer name="Bob Fisk" phone="(801)881-2260"/>
    </animal>
    <animal>
        <name>Sheldon</name>
        <species>parrot</species>
        <dateOfBirth>1998-09-30</dateOfBirth>
        <veterinarian name="Dr. Kevin Wilson" phone="(801)466-6498"/>
        <trainer name="Eli Wendel" phone="(801)929-2506"/>
    </animal>
    <animal>
        <name>Dippy</name>
        <species>penguin</species>
        <dateOfBirth>2001-06-08</dateOfBirth>
        <veterinarian name="Dr. Barbara Swayne" phone="(801)459-7746"/>
        <trainer name="Ben Waxman" phone="(801)882-3549"/>
    </animal>
</animalList>
```

Example 14-2 shows how the definition of the class Animal might look. Animal has five data members corresponding to an animal's name, species, date of birth, veterinarian, and trainer. An animal's name and species are represented as std::strings, its date of birth is represented as a boost::gregorian::date from Boost.Date_Time, and its veterinarian and trainer are represented as instances of the class Contact, also defined in Example 14-2. Example 14-3 shows how to use TinyXml to parse the document *animals.xml* traverse the parsed document, and populate a std::vector of Animals using data extracted from the

Recipe 14.2. Working with Xerces Strings

Problem

You want to be able to handle the wide-character strings used by the Xerces library safely and easily. In particular, you want to be able to store strings returned by Xerces functions as well as to convert between Xerces strings and C++ standard library strings.

Solution

You can store wide-character strings returned by Xerces library functions using the template std::basic_string specialized for the Xerces wide-character type XMLCh:

```
typedef std::basic_string<XMLCh> XercesString;
```

To translate between Xerces strings and narrow-character strings, use the overloaded static method TRanscode( ) from the class xercesc::XMLString, defined in the header *xercesc/util/XMLString.hpp*. Example 14-4 defines two overloaded utility functions, toNative and fromNative, that use transcode to translate from narrow-character strings to Xerces strings and vice versa. Each function has two variants, one that takes a C-style string and one that takes a C++ standard library string. These utility functions are all you'll need to convert between Xerces string and narrow-character strings; once you define them, you'll never need to call transcode directly.

Example 14-4. The header xerces_strings.hpp, for converting between Xerces strings and narrow-character strings

```
#ifndef XERCES_STRINGS_HPP_INCLUDED
#define XERCES_STRINGS_HPP_INCLUDED

#include <string>
#include <boost/scoped_array.hpp>
#include <xercesc/util/XMLString.hpp>

typedef std::basic_string<XMLCh> XercesString;

// Converts from a narrow-character string to a wide-character string.
inline XercesString fromNative(const char* str)
{
    boost::scoped_array<XMLCh> ptr(xercesc::XMLString::transcode(str));
    return XercesString(ptr.get( ));
}

// Converts from a narrow-character string to a wide-charactr string.
inline XercesString fromNative(const std::string& str)
{
    return fromNative(str.c_str( ));
}

// Converts from a wide-character string to a narrow-character string.
inline std::string toNative(const XMLCh* str)
{
    boost::scoped_array<char> ptr(xercesc::XMLString::transcode(str));
    return std::string(ptr.get( ));
}

// Converts from a wide-character string to a narrow-character string.
inline std::string toNative(const XercesString& str)
{
    return toNative(str.c_str( ));
}
```

Recipe 14.3. Parsing a Complex XML Document

Problem

You have a collection of data stored in an XML document that uses an internal DTD or XML Namespaces. You want to parse the document and turn the data it contains into a collection of C++ objects.

Solution

Use Xerces's implementation of the SAX2 API (the Simple API for XML, Version 2.0). First, derive a class from xercesc::ContentHandler; this class will receive notifications about the structure and content of your XML document as it is being parsed. Next, if you like, derive a class from xercesc::ErrorHandler to receive warnings and error notifications. Construct a parser of type xercesc::SAX2XMLReader, register instances of your handler classes using the parser's setContentHandler( ) and setErrorHandler() methods. Finally, invoke the parser's parse( ) method, passing the file pathname of your document as its argument.

For example, suppose you want to parse the XML document *animals.xml* from Example 14-1 and construct a std::vector of Animals representing the animals listed in the document. (See Example 14-2 for the definition of the class Animal.) In Example 14-3, I showed how to do this using TinyXml. To make the problem more challenging, let's add namespaces to the document, as shown in Example 14-5.

Example 14-5. List of circus animals, using XML Namespaces

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Feldman Family Circus Animals with Namespaces -->

<ffc:animalList xmlns:ffc="http://www.feldman-family-circus.com">
    <ffc:animal>
        <ffc:name>Herby</ffc:name>
        <ffc:species>elephant</ffc:species>
        <ffc:dateOfBirth>1992-04-23</ffc:dateOfBirth>
        <ffc:veterinarian name="Dr. Hal Brown" phone="(801)595-9627"/>
        <ffc:trainer name="Bob Fisk" phone="(801)881-2260"/>
    </ffc:animal>

    <!-- etc. -->

</ffc:animalList>
```

To parse this document with SAX2, define a ContentHandler, as shown in Example 14-6, and an ErrorHandler, as shown in Example 14-7. Then construct a SAX2XMLReader, register your handlers, and run the parser. This is illustrated in Example 14-8.

Example 14-6. A SAX2 ContentHandler for parsing the document animals.xml

```
#include <stdexcept>                        // runtime_error
#include <vector>
#include <xercesc/sax2/Attributes.hpp>
#include <xercesc/sax2/DefaultHandler.hpp> // Contains no-op
                                           // implementations of
                                           // the various handlers
#include "xerces_strings.hpp"              // Example 14-4
#include "animal.hpp"

using namespace std;
using namespace xercesc;

// Returns an instance of Contact based
```

Recipe 14.4. Manipulating an XML Document

Problem

You want to represent an XML document as a C++ object so that you can manipulate its elements, attributes, text, DTD, processing instructions, and comments.

Solution

Use Xerces's implementation of the W3C DOM. First, use the class xercesc::DOMImplementationRegistry to obtain an instance of xercesc::DOMImplementation, then use the DOMImplementation to create an instance of the parser xercesc::DOMBuilder. Next, register an instance of xercesc::DOMErrorHandler to receive notifications of parsing errors, and invoke the parser's parseURI( ) method with your XML document's URI or file pathname as its argument. If the parse is successful, parseURI will return a pointer to a DOMDocument representing the XML document. You can then use the functions defined by the W3C DOM specification to inspect and manipulate the document.

When you are done manipulating the document, you can save it to a file by obtaining a DOMWriter from the DOMImplementation and calling its writeNode( ) method with a pointer to the DOMDocument as its argument.

Example 14-10 shows how to use DOM to parse the document *animals.xml* from Example 14-1, locate and remove the node corresponding to Herby the elephant, and save the modified document.

Example 14-10. Using DOM to load, modify, and then save an XML document

```
 #include <exception>
#include <iostream>     // cout
#include <xercesc/dom/DOM.hpp>
#include <xercesc/framework/LocalFileFormatTarget.hpp>
#include <xercesc/sax/SAXException.hpp>
#include <xercesc/util/PlatformUtils.hpp>
#include "animal.hpp"
#include "xerces_strings.hpp"

using namespace std;
using namespace xercesc;

/*
 * Define XercesInitializer as in Example 14-8
 */

// RAII utility that releases a resource when it goes out of scope.
template<typename T>
class DOMPtr {
public:
    DOMPtr(T* t) : t_(t) { }
    ~DOMPtr( ) { t_->release( ); }
    T* operator->( ) const { return t_; }
private:
    // prohibit copying and assigning
    DOMPtr(const DOMPtr&);
    DOMPtr& operator=(const DOMPtr&);
    T* t_;
};

// Reports errors encountered while parsing using a DOMBuilder.
class CircusErrorHandler : public DOMErrorHandler {
public:
```

Recipe 14.5. Validating an XML Document with a DTD

Problem

You want to verify that an XML document is valid according to a DTD.

Solution

Use the Xerces library with either the SAX2 (Simple API for XML) or the DOM parser.

To validate an XML document using SAX2, obtain a SAX2XMLReader, as in Example 14-8. Next, enable DTD validation by calling the parser's setFeature( ) method with the arguments xercesc::XMLUni::fgSAX2CoreValidation and true. Finally, register an ErrorHandler to receive notifications of DTD violations and call the parser's parse() method with your XML document's name as its argument.

To validate an XML document using DOM, first construct an instance of XercesDOMParser. Next, enable DTD validation by calling the parser's setValidationScheme( ) method with the argument xercesc::XercesDOMParser::Val_Always. Finally, register an ErrorHandler to receive notifications of DTD violations and call the parser's parse( ) method with your XML document's name as its argument.

Here I'm using the class XercesDOMParser, an XML parser that has been part of Xerces since before the DOM Level 3 DOMBuilder interface was introduced. Using a XercesDOMParser makes the example a bit simpler, but you can use a DOMBuilder instead if you like. See Discussion and Recipe 14.4.

For example, suppose you modify the XML document *animals.xml* from Example 14-1 to contain a reference to an external DTD, as illustrated in Examples Example 14-11 and Example 14-12. The code to validate this document using the SAX2 API is presented in Example 14-13; the code to validate it using the DOM parser is presented in Example 14-14.

Example 14-11. DTD animals.dtd for the file animals.xml

```
<!-- DTD for Feldman Family Circus Animals -->

<!ELEMENT animalList (animal+)>
<!ELEMENT animal ( name, species, dateOfBirth,
                   veterinarian, trainer ) >
<!ELEMENT name (#PCDATA)>
<!ELEMENT species (#PCDATA)>
<!ELEMENT dateOfBirth (#PCDATA)>
<!ELEMENT veterinarian EMPTY>
<!ELEMENT trainer EMPTY>
<!ATTLIST veterinarian
    name  CDATA #REQUIRED
    phone CDATA #REQUIRED
>
<!ATTLIST trainer
    name  CDATA #REQUIRED
    phone CDATA #REQUIRED
>
```

Example 14-12. The file animals.xml, modified to contain a DTD

```
<?xml version="1.0" encoding="UTF-8"?>
```

Recipe 14.6. Validating an XML Document with a Schema

Problem

You want to verify that an XML document is valid according to a schema, as specified in the XML Schema 1.0 recommendation.

Solution

Use the Xerces library with either the SAX2 or the DOM parser.

Validating an XML document against a schema using the SAX2 API is exactly the same as validating a document that contains a DTD, assuming the schema is contained in or referenced from the target document. If you want to validate an XML document against an external schema, you must call the parser's setProperty( ) method to enable external schema validation. The first argument to setProperty( ) should be XMLUni::fgXercesSchemaExternalSchemaLocation or XMLUni::fgXercesSche-maExternalNoNameSpaceSchemaLocation, depending on whether the schema has a target namespace. The second argument should be the location of the schema, expressed as a const XMLCh*. Make sure to cast the second argument to void*, as explained in Recipe 14.5.

Validating an XML document against a schema using the XercesDOMParser is similar to validating a document against a DTD, assuming the schema is contained in or referenced from the target document. The only difference is that schema and namespace support must be explicitly enabled, as shown in Example 14-15.

Example 14-15. Enabling schema validation with a XercesDOMParser

```
 XercesDOMParser parser;
parser.setValidationScheme(XercesDOMParser::Val_Always);
parser.setDoSchema(true);
parser.setDoNamespaces(true);
```

If you want to validate an XML document against an external schema with a target namespace, call the parser's setExternalSchemaLocation( ) method with your schema's location as its argument. If you want to validate an XML document against an external schema that has no target namespace, call the parser's setExternalNoNamespaceSchema-Location( ) instead.

Similarly, to validate an XML document against a schema using a DOMBuilder, enable its validation feature as follows:

```
 DOMBuilder* parser = ...;
parser->setFeature(XMLUni::fgDOMNamespaces, true);
parser->setFeature(XMLUni::fgDOMValidation, true);
parser->setFeature(XMLUni::fgXercesSchema, true);
```

To validate against an external schema using DOMBuilder, set the property XMLUni::fgXercesSchemaExternalSchemaLocation or XMLUni::fgXercesSchemaExternalNoName-SpaceSchemaLocation to the location of the schema.

For example, suppose you want to validate the document *animals.xml* from Example 14-1 using the schema in Example 14-16. One way to do this is to add a reference to the schema to *animals.xml*, as shown in Example 14-17. You can then validate the document with the SAX2 API, as shown in Example 14-13, or using DOM, as shown in Example 14-14, with the modification indicated in Example 14-15.

Example 14-16. Schema animals.xsd for the file animals.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

Recipe 14.7. Transforming an XML Document with XSLT

Problem

You want to transform an XML document using an XSLT stylesheet.

Solution

Use the Xalan library. First, construct an instance of the XSTL engine xalanc::XalanTransformer. Next, construct two instances of xalanc::XSLTInputSourceone to represent the document to be transformed and the other to represent your stylesheetand an instance of xalanc::XSLTResultTarget to represent the document to be generated by the transformation. Finally, call the XSLT engine's transform( ) method, passing the two XSLTInputSources and the XSLTResultTarget as arguments.

For example, suppose you want to be able to view the list of circus animals from Example 14-1 with your web browser. An easy way to do this is with XSLT. Example 14-19 shows an XSLT stylesheet that takes an XML document like *animals.xml* as input and generates an HTML document containing a table with one data row per animal listing the animal's name, species, date of birth, veterinarian, and trainer. Example 14-20 shows how to use the Xalan library to apply this stylesheet to the document *animals.xml*. The HTML generated by the program in Example 14-20 is shown in Example 14-21, reformatted for readability.

Example 14-19. Stylesheet for animals.xml

```
<?xml version="1.0" encoding="utf-8"?>

<!-- Stylesheet for Feldman Family Circus Animals -->

<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <html>
        <head>
            <title>Feldman Family Circus Animals</title>
        </head>
        <body>
            <h1>Feldman Family Circus Animals</h1>
            <table cellpadding="3" border="1">
                <tr>
                    <th>Name</th>
                    <th>Species</th>
                    <th>Date of Birth</th>
                    <th>Veterinarian</th>
                    <th>Trainer</th>
                </tr>
                <xsl:apply-templates match="animal">
                </xsl:apply-templates>
            </table>
        </body>
        </html>
    </xsl:template>
    <xsl:template match="animal">
        <tr>
            <td><xsl:value-of select="name"/></td>
            <td><xsl:value-of select="species"/></td>
            <td><xsl:value-of select="dateOfBirth"/></td>
            <xsl:apply-templates select="veterinarian"/>
            <xsl:apply-templates select="trainer"/>
        </tr>
```

Recipe 14.8. Evaluating an XPath Expression

Problem

You want to extract information from a parsed XML document by evaluating an XPath expression.

Solution

Use the Xalan library. First, parse the XML document to obtain a pointer to a xalanc::XalanDocument. This can be done by using instances of XalanSourceTreeInit, XalanSourceTreeDOMSupport, and XalanSourceTreeParserLiaisoneach defined in the namespace xalanclike so:

```
 #include <xercesc/framework/LocalFileInputSource.hpp>
#include <xalanc/XalanSourceTree/XalanSourceTreeDOMSupport.hpp>
#include <xalanc/XalanSourceTree/XalanSourceTreeInit.hpp>
#include <xalanc/XalanSourceTree/XalanSourceTreeParserLiaison.hpp>
...
int main( )
{
     ...
     // Initialize the XalanSourceTree subsystem
     XalanSourceTreeInit           init;
     XalanSourceTreeDOMSupport     support;

     // Interface to the parser
     XalanSourceTreeParserLiaison  liaison(support);

     // Hook DOMSupport to ParserLiaison
     support.setParserLiaison(&liaison);
     LocalFileInputSource          src(document-location);
     XalanDocument*                doc = liason.ParseXMLStream(doc);
    ...
}
```

Alternatively, you can use the Xerces DOM parser to obtain a pointer to a DOMDocument, as in Example 14-14, and then use instances of XercesDOMSupport, XercesParserLiaison, and XercesDOMWrapperParsedSource each defined in namespace xalanc to obtain a pointer to a XalanDocument corresponding to the DOMDocument:

```
 #include <xercesc/dom/DOM.hpp>
#include <xalanc/XalanTransformer/XercesDOMWrapperParsedSource.hpp>
#include <xalanc/XercesParserLiaison/XercesParserLiaison.hpp>
#include <xalanc/XercesParserLiaison/XercesDOMSupport.hpp>
...
int main( ) {
     ...
    DOMDocument*                  doc = ... ;
    XercesDOMSupport              support;
    XercesParserLiaison           liaison(support);
    XercesDOMWrapperParsedSource  src(doc, liaison, support);
    XalanDocument*                xalanDoc = src.getDocument( );
    ...
}
```

Next, obtain a pointer to the node that serves as the context node when evaluating the XPath expression. You can do this by using XalanDocument's DOM interface. Construct an XPathEvaluator to evaluate the XPath expression and a XalanDocumentPrefixResolver to resolve namespace prefixes in the XML document. Finally, call the XPathEvaluator's evaluate( ) method, passing the DOMSupport, the context node, the XPath expression, and the PrefixResolver as arguments. The result of evaluating the expression is returned as an object of type XObjectPtr; the operations you can perform on this object depend on its XPath data type, which you can query using the getType( ) method.

Recipe 14.9. Using XML to Save and Restore a Collection of Objects

Problem

You want to be able to save a collection of C++ objects to an XML document and read it back into memory later.

Solution

Use the Boost Serialization library. This library allows you to save and restore objects using classes called *archives* . To make use of this library, you must first make each of your classes *serializable*, which just means that instances of the class can be written to an archive, or *serialized*, and read back into memory, or *deserialized*. Then, at runtime, you can save your objects to an XML archive using the << operator and restore them using the >> operator.

To make a class serializable, add a member function template serialize with the following signature:

```
template<typename Archive>
void serialize(Archive& ar, const unsigned int version);
```

The implementation of serialize should write each data member of the class to the specified archive as a name-value pair, using the & operator. For example, if you want to serialize and deserialize instances of the class Contact from Example 14-2, add a member function serialize, as shown in Example 14-25.

Example 14-25. Adding support for serialization to the class Contact from Example 14-2

```
#include <boost/serialization/nvp.hpp> // "name-value pair"

class Contact {
...
private:
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        // Write (or read) each data-member as a name-value pair
        using boost::serialization::make_nvp;
        ar & make_nvp("name", name_);
        ar & make_nvp("phone", phone_);
    }
    ...
};
```

Similarly, you can make the class Animal from Example 14-2 serializable, as shown in Example 14-26.

Example 14-26. Adding support for serialization to the class Animal from Example 14-2

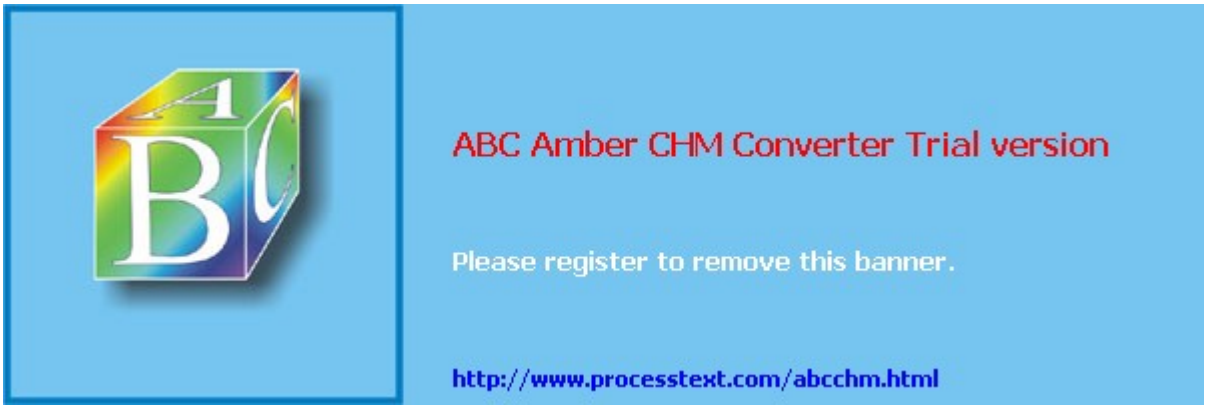
```
...
// Include serialization support for boost::gregorian::date
#include <boost/date_time/gregorian/greg_serialize.hpp>
...
class Contact {
...
private:
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize(Archive& ar, const unsigned int version)
    {
        // Write (or read) each data-member as a name-value pair
        using boost::serialization::make_nvp;
```

Chapter 15. Miscellaneous

Introduction

This chapter describes a few facets of C++ that don't neatly fit into any of the other chapters: function and member pointers, const variables and member functions, and standalone (i.e., nonmember) operators and a few other topics.

PREV NEXT



Recipe 15.1. Using Function Pointers for Callbacks

Problem

You plan to call some function func1, and at runtime you need it to invoke another function func2. For one reason or another, however, you cannot simply hardcode the name of func2 within func1. func2 may not be known definitively at compile time, or perhaps func1 belongs to a third-party API that you can't change and recompile. In either case, you need a callback function.

Solution

In the case of the functions above, declare func1 to take a pointer to a function, and pass it the address of func2 at runtime. Use a typedef to make the messy syntax easier to read and debug. Example 15-1 shows how to implement a callback function with a function pointer.

Example 15-1. A callback function

```
#include <iostream>

// An example of a callback function
bool updateProgress(int pct) {

   std::cout << pct << "% complete...\n";
   return(true);
}

// A typedef to make for easier reading
typedef bool (*FuncPtrBoolInt)(int);

// A function that runs for a while
void longOperation(FuncPtrBoolInt f) {

   for (long l = 0; l < 100000000; l++)
      if (l % 10000000 == 0)
         f(l / 1000000);
}

int main( ) {

   longOperation(updateProgress); // ok
}
```

Discussion

In a situation such as that shown in Example 15-1, a function pointer is a good idea if updateProgress and longOperation shouldn't know anything about each other. For example, a function that updates the progress by displaying it to the usereither in a user interface (UI) dialog box, in a console window, or somewhere elsedoes not care about the context in which it is invoked. Similarly, the longOperation function may be part of some data loading API that doesn't care whether it's invoked from a graphical UI, a console window, or by a background process.

The first thing you will want to do is determine what the signature of the function is you plan to call and create a typedef for it. typedef is your friend when it comes to function pointers, because their syntax is ugly. Consider how you would declare a function pointer variable f that contains the address of a function that takes a single integer argument and returns a boolean. It would look like this:

```
bool (*f)(int);  // f is the variable name
```

One could argue, convincingly, that this is no big deal and that I'm just a whiner. But what if you want a

Recipe 15.2. Using Pointers to Class Members

Problem

You need to refer to a data member or a member function with its address.

Solution

Use the class name and the scope operator (::) with an asterisk to correctly qualify the name. Example 15-2 shows how.

Example 15-2. Obtaining a pointer to a member

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

class MyClass {

public:
   MyClass( ) : ival_(0), sval_("foo") {}
  ~MyClass( ) {}

   void incr( ) {++ival_;}
   void decr( ) {ival_--;}

private:
   std::string sval_;
   int ival_;
};

int main( ) {

   MyClass obj;

   int         MyClass::* mpi = &MyClass::ival_;  // Data member
   std::string MyClass::* mps = &MyClass::sval_;  // pointers

   void (MyClass::*mpf)( ); // A pointer to a member function that
                            // takes no params and returns void
   void (*pf)( );           // A normal function pointer

   int* pi = &obj.ival_;    // int pointer referring to int member--no
                            // problem.

   mpf = &MyClass::incr;    // A pointer to a member function. You can't
                            // write this value to a stream. Look at it
                            // in your debugger to see what its
                            // representation looks like.

   pf = &MyClass::incr;     // Error: &MyClass::incr is not an instance
                            // of a function

   std::cout << "mpi = " << mpi << '\n';
   std::cout << "mps = " << mps << '\n';
   std::cout << "pi =  " << pi << '\n';
   std::cout << "*pi = " << *pi << '\n';

   obj.*mpi = 5;
   obj.*mps = "bar";

   (obj.*mpf)( ); // now obj.ival_ is 6
```

Recipe 15.3. Ensuring That a Function Doesn't Modify an Argument

Problem

You are writing a function, and you need to guarantee that its arguments will not be modified when it is invoked.

Solution

Declare your arguments with the keyword const to prevent your function from changing the arguments. See Example 15-3 for a short sample.

Example 15-3. Guaranteeing unmodified arguments

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

void concat(const std::string& s1, // These are declared const, so they
            const std::string& s2, // cannot be changed
            std::string& out) {
   out = s1 + s2;
}

int main( ) {

   std::string s1 = "Cabo ";
   std::string s2 = "Wabo";
   std::string s3;

   concat(s1, s2, s3);

   std::cout << "s1 = " << s1 << '\n';
   std::cout << "s2 = " << s2 << '\n';
   std::cout << "s3 = " << s3 << '\n';
}
```

Discussion

Example 15-3 demonstrates a straightforward use of const. There are a couple of good reasons for declaring your function parameters const when you don't plan on changing them. First, you communicate your intent to human readers of your code. By declaring a parameter as const, what you are saying, essentially, is that the const parameters are for input. This lets consumers of your function, code with the assumption that the values will not change. Second, it tells the compiler to disallow any modifying operations, in the event you do so by accident. Consider an unsafe version of concat from Example 15-3:

```
void concatUnsafe(std::string& s1,
                  std::string& s2,
                  std::string& out) {
   out = s1 += s2; // Whoops, wrote to s1
}
```

Despite my fastidious coding habits, I have made a silly mistake and typed += when I meant to type +. As a result, when concatUnsafe is called, it will modify the arguments out and s1, which may come as surprise to the userwho would expect a concatenation function to modify one of the source strings?

const to the rescue. Create a new function concatSafe, declare the variables const as in Example 15-3, and it won't compile:

```
void concatSafe(const std::string& s1,
```

Recipe 15.4. Ensuring That a Member Function Doesn't Modify Its Object

Problem

You need to invoke member functions on a const object, but your compiler is complaining that it can't convert the type of object you are operating from const *type* to *type*.

Solution

Place the const keyword to the right of the member function declaration in both the class declaration and definition. Example 15-4 shows how to do this.

Example 15-4. Declaring a member function const

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

class RecordSet {
public:
   bool getFieldVal(int i, std::string& s) const;
   // ...
};

bool RecordSet::getFieldVal(int i, std::string& s) const {
   // In here, you can't modify any nonmutable data
   // members (see discussion)
}

void displayRecords(const RecordSet& rs) {
   // Here, you can only invoke const member functions
   // on rs
}
```

Discussion

Adding a trailing const to a member declaration and its definition forces the compiler to look more carefully at what that member's body is doing to the object. const member functions are not allowed to invoke any nonconst operation on data members. If one does, compilation fails. For example, if, in RecordSet::getFieldVal, I updated a counter member, it wouldn't compile (assume that getFieldCount_ is a member variable of RecordSet):

```
bool RecordSet::getFieldVal(int i, std::string& s) const {
   ++getFieldCount_;  // Error: const member function can't modify
                      // a member variable
   // ...
}
```

It can also help catch more subtle errors, similar to how const works in its variable-qualifier role (see Recipe 15.3). Consider this silly typo:

```
bool RecordSet::getFieldVal(int i, std::string& s) const {

   fieldArray_[i] = s; // Oops, I meant the other way around
   // ...
}
```

Once again, the compiler will abort and give you an error because you are trying to change a member variable, and that's not allowed in const member functions. Well, with one exception.

Recipe 15.5. Writing an Operator That Isn't a Member Function

Problem

You have to write a binary operator, and you can't or don't want to make it a class member function.

Solution

Use the operator keyword, a temporary variable, and a copy constructor to do most of the work, and return the temporary object. Example 15-5 presents a simple string concatenation operator for a custom String class.

Example 15-5. Concatenation with a nonmember operator

```
 #include <iostream>
#include <cstring>

class String {  // Assume the String class declaration
                // has at least everything shown here
public:
   String( );
   String(const char* p);
   String(const String& orig);
  ~String( ) {delete buf_;}

   String& append(const String& s);
   size_t length( ) const;
   const char* data( ) const;
   String& operator=(const String& orig);

   // ...
};

String operator+(const String& lhs, const String& rhs) {

   String tmp(lhs); // Copy construct a temp object
   tmp.append(rhs); // Use a member function to do the real work

   return(tmp);     // Return the temporary
}

int main( ) {

   String s1("banana ");
   String s2("rancher");
   String s3, s4, s5, s6;

   s3 = s1 + s2;           // Works fine, no surprises
   s4 = s1 + "rama";       // Constructs "rama" automatically using
                           // the constructor String(const char*)
   s5 = "ham " + s2;       // Hey cool, it even does it backward
   s6 = s1 + "rama " + s2;

   std::cout << "s3 = " << s3.data( ) << '\n';
   std::cout << "s4 = " << s4.data( ) << '\n';
   std::cout << "s5 = " << s5.data( ) << '\n';
   std::cout << "s6 = " << s6.data( ) << '\n';
}
```

Discussion

Recipe 15.6. Initializing a Sequence with Comma-Separated Values

Problem

You want to initialize a sequence with a comma-delimited set of values, like you can with a built-in array.

Solution

You can use a comma-initialization syntax on standard sequences (such as vector and list) by defining a helper class and overloading the comma operator for it as demonstrated in Example 15-6.

Example 15-6. Utilities for comma initialization of standard sequences

```
 #include <vector>
#include <iostream>
#include <iterator>
#include <algorithm>

using namespace std;

template<class Seq_T>
struct comma_helper
{
  typedef typename Seq_T::value_type value_type;
  explicit comma_helper(Seq_T& x) : m(x) { }
  comma_helper& operator=(const value_type& x) {
    m.clear( );
    return operator+=(x);
  }
  comma_helper& operator+=(const value_type& x) {
    m.push_back(x);
    return *this;
  }
  Seq_T& m;
};

template<typename Seq_T>
comma_helper<Seq_T>
initialize(Seq_T& x) {
  return comma_helper<Seq_T>(x);
}

template<class Seq_T, class Scalar_T>
comma_helper<Seq_T>&
operator,(comma_helper<Seq_T>& h, Scalar_T x) {
  h += x;
  return h;
}

int main( ) {
  vector v;
  int a = 2;
  int b = 5;
  initialize(v) = 0, 1, 1, a, 3, b, 8, 13;
  cout << v[3] << endl; // outputs 2
  system("pause");
  return EXIT_SUCCESS;
}
```

Discussion

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of C++ Cookbook is a collie. The name refers to a type of sheepherding dog that originated in the highlands of Scotland and Britain in the 1600s. One variety of sheep in the Scottish Highlands had dark markings around its legs and face and was called the "Colley" sheep, a name derived from the Older Scots word for "coal." The modern version of the collie, lighter and more thick-boned than its Scottish ancestors, was bred in England in the late 19th century. Today, collies are primarily house pets, though they are still used as farm dogs in the United States.

There are two distinct breeds of collie: rough-coated collies were used to guard sheep, and the smooth-coated variety drove the livestock to market. Both are limber, streamlined dogs with a pronounced snout and pointed ears. They are 22-26 inches tall and weigh 50-75 pounds. Their fur is usually white with a second color that can vary from yellowish-white to brownish-red to coal-black.

Famous collies include Lassie, of course; Lyndon Johnson's pet Blanco; and Laddie from The Simpsons.

Matt Hutchinson was the production editor for C++ Cookbook . Octal Publishing, Inc. provided production services. Darren Kelly, Adam Witwer, and Claire Cloutier provided quality control.

Karen Montgomery designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from Cassell's Natural History. Karen Montgomery produced the cover layout with Adobe InDesign CS using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Keith Fahlgren to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano, Jessamyn Read, and Lesley Borash using Macromedia FreeHand MX and Adobe Photoshop CS. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Matt Hutchinson.

The online edition of this book was created by the Digital Books production group (John Chodacki, Ken Douglass, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.

PREV NEXT

Index

PREV NEXT

PREV NEXT

Index

PREV NEXT

Index

Index

Index

Index

Index

Index

Index

Index

PREV NEXT

Index

Index

PREV NEXT

PREV NEXT

Index

PREV NEXT

Index

Index

Index

Index

Index

Index

PREV NEXT

Index

Index

Index

Index

Index

Index

Index

PREV NEXT

PREV NEXT

Index

PREV NEXT

Index