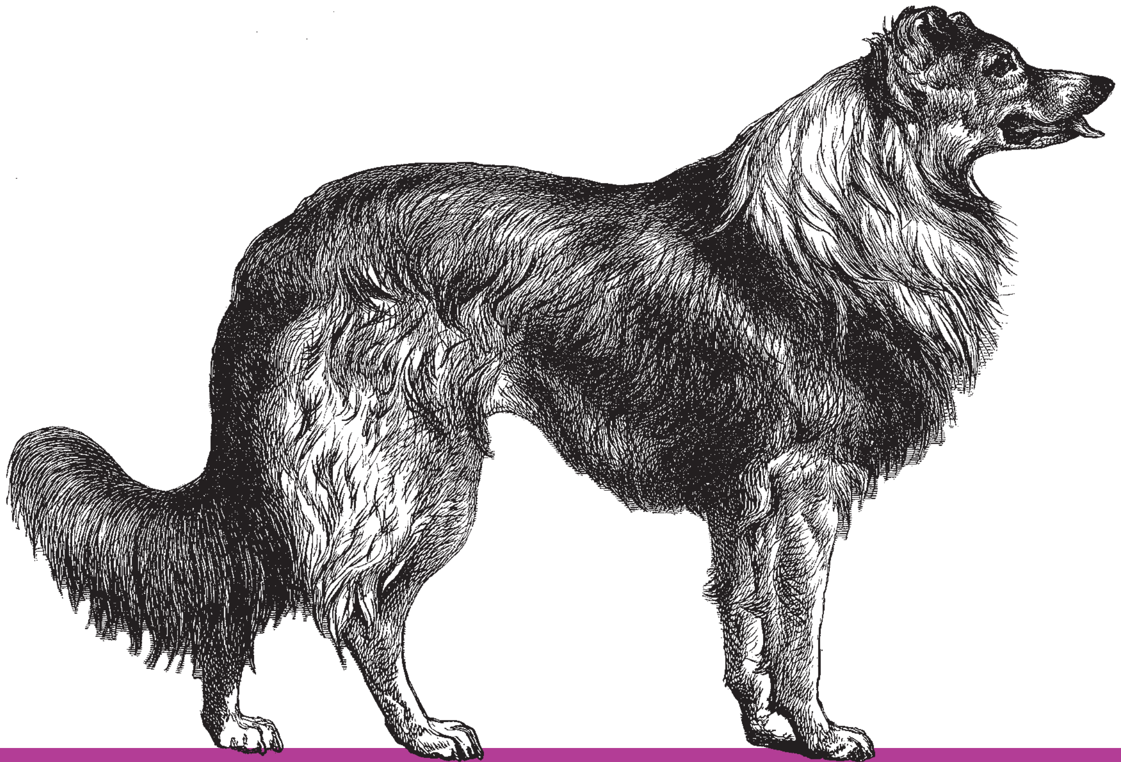
Solutions and Examples for C++ Programmers



C++ Cookbook

D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, & Jeff Cogswell

O'REILLY®

CHAPTER 10
Streams and Files

10.0 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, too—what 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 interaction—you will save yourself lots of time and effort. For more information on the Boost project, see *www.boost.org*.

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 character in the field. If the data you send to the stream is not wide enough to span the entire field width, the right side of it will be padded with the stream's fill character, which is, by default, a single space. You can change the fill character with the `setfill` manipulator, like this:

```
myostr << setfill('.') << "foo";
```

If the value you put in the field is larger than the field width, the entire value is printed and no padding is added.

Table 10-1 contains a summary of manipulators that operate on any kind of value (text, float, integer, etc.). There is a set of manipulators that apply only to floating-point output, and they are described in Recipe 10.2.

Table 10-1. Text manipulators

Manipulator	Description	Sample output

`left`
`right`
	Justify values within the current field width to either the left or right side, and pad the remaining space with the fill character.	Left-justified:
		`apple`
		`bananna`
		`cherry`
		Right-justified (with a field width of 10):
		`     apple`
		`   bananna`
		`    cherry`

`setw(int n)`	Set the width of the field to `n` characters wide.	See earlier example.

Table 10-1. Text manipulators (continued)

Manipulator	Description	Sample output

`setfill(int c)`	Use the character `c` to pad fields that have remaining space.	`cout << setfill('.') << setw(10) << right << "foo"`
		produces:
		`.......foo`

`boolalpha`
`noboolalpha`
	Display Boolean values as the current locale's representation of the words `true` and `false`, instead of `1` and `0`.	`cout << boolalpha << true`
		produces:
		`true`

`endl`	Write a `newline` to the stream and flush the output buffer.	n/a
`ends`	Write a `null` character ('\0') to the stream.	n/a
`flush`	Flush the output buffer.	n/a

Some of the manipulators in Table 10-1 (and Table 10-2 in the next recipe) toggle binary stream flags, and are actually implemented as two manipulators that turn a flag on or off. Take `boolalpha`, for example. If you want Boolean values to be displayed as their written equivalents in the current locale (e.g., "true" and "false"), use the `boolalpha` manipulator. To turn this behavior off, so that `0` and `1` are printed instead, use `noboolalpha`, which is the default.

All manipulators have the behavior that they stay in effect until they are explicitly changed, except for `setw`. In Example 10-1, you can see that it is called before each write, but `left` is used only once. This is because the width is reset to zero after each value is written to the stream with `operator<<`; to keep the same width for each field, I had to call `setw` each time.

The standard manipulators provide a lot of functionality, but they don't do everything. If you want to write your own manipulators, see Recipe 10.2.

As with all other character-based classes in the standard library, manipulators work on streams that use narrow or wide characters. Therefore, you can use them with templates to write formatting utilities that operate on streams of any kind of character. Example 10-2 presents the class template `TableFormatter`, which formats data into equal-width columns and writes it to a stream.

Example 10-2. A generic class for tabular data

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

using namespace std;
```

Example 10-2. A generic class for tabular data (continued)

```
// TableFormatter formats data for output to a stream of characters
// of type T.
template<typename T>
class TableFormatter {

public:
   TableFormatter(basic_ostream<T>& os) : out_(os) {}
  ~TableFormatter() {out_ << flush;}

   template<typename valT>
   void writeTableRow(const vector<valT>& v, int width);
   //...

private:
  basic_ostream<T>& out_;
};

template<typename T>    // refers to class template param list
template<typename valT> // refers to mem fn template param list
void TableFormatter<T>::writeTableRow(const std::vector<valT>& v,
                                      int width) {

   ios_base::fmtflags flags = out_.flags();

   out_.flush();
   out_ << setprecision(2) << fixed;  // Set the precision, in case
                                      // this is floating-point data
   for (vector<valT>::const_iterator p = v.begin();
        p != v.end(); ++p)
     out_ << setw(width) << left << *p;  // Set the width, justify,
                                         // and write the element
   out_ << endl;     // Flush
   out_.setf(flags); // Set the flags back to normal
}

int main() {

   TableFormatter<char> fmt(cout);

   vector<string> vs;

   vs.push_back( "Sunday" );
   vs.push_back( "Monday" );
   vs.push_back( "Tuesday" );

   fmt.writeTableRow(vs, 12);
   fmt.writeTableRow(vs, 12);
   fmt.writeTableRow(vs, 12);

   vector<double> vd;
```

Example 10-2. A generic class for tabular data (continued)

```
   vd.push_back(4.0);
   vd.push_back(3.0);
   vd.push_back(2.0);
   vd.push_back(1.0);

   fmt.writeTableRow(vd, 5);
}
```

The output from Example 10-2 looks like this:

```
 Sunday      Monday      Tuesday
 4.00 3.00 2.00 1.00
```

See Also

Table 10-1, Recipe 10.2

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.
```

Example 10-3. Formatting pi (continued)

```
   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 such that only a set number of digits are displayed at one time. So, by default, pi would be displayed as `3.14286`, and pi times 100 would display `314.286`.

Fixed
: In this format, the number of digits displayed to the *right* of the decimal point is fixed, while the number of those displayed to the left is not. In this case, again with a default precision of six, pi would be displayed as `3.142857`, and pi times 100 would be `314.285714`. In both cases, the number of digits displayed to the right of the decimal point is six while the total number of digits can grow indefinitely.

Scientific
: The value is shown as a single digit, followed by a decimal point, followed by a number of digits determined by the precision setting, followed by the letter "e" and the power of ten to raise the preceding value to. In this case, pi times 1,000 would display as `3.142857e+003`.

Table 10-2 shows all manipulators that affect floating-point output (and sometimes numeric output in general). See Table 10-1 for general manipulators you can use together with the floating-point manipulators.

Table 10-2. Floating-point and numeric manipulators

Manipulator	Description	Sample output
`Fixed`	Show floating-point values with a fixed number of digits to the right of the decimal point.	With a default precision of six digits: `pi = 3.142857`
`scientific`	Show floating-point values using scientific notation, which means a decimal number and an exponent multiplier.	pi * 1000, with a default precision of six digits: `pi = 3.142857e+003`

`setprecision`	Control the number of digits displayed in the output. (See further explanation later.)	Pi in the default format, with a precision of `3`:
		`pi = 3.14`
		In fixed format:
		`pi = 3.143`
		In scientific format:
		`pi = 3.143e+000`

`showpos` `noshowpos`	Show a plus sign in front of positive numbers. This works for any kind of number, decimal or integer.	`+3.14`

`showpoint`
`noshowpoint`
	Show the decimal, even if there are only zeros after it. This works only for floating-point values, and not for integers.	The following line, with a precision of `2`:
		`cout << showpoint << 2.0`
		will display like this:
		`2.00`

`showbase` `noshowbase`	Show the base for the number: decimal (none), octal (leading zero), or hexadecimal (leading 0x). See the next entry.	Decimal: `32` Octal: `040` Hexadecimal: `0x20`

`dec`
`oct`
`hex`
	Set the base for the numbers to be displayed to decimal, octal, or hexadecimal. The base is not shown by default; use `showbase` to display the base.	See previous entry.

`Uppercase` `nouppercase`	Display values in uppercase.	This sets the case for numeric output, such as 0X for hexadecimal numbers and E for numbers in scientific notation.

In all three formats, all manipulators have the same effects except `setprecision`. In the default mode, "precision" refers to the number of digits on both sides of the decimal point. For example, to display pi in the default format with a precision of `2`, do this:

```
cout << "pi = " << setprecision(2)
     << pi << '\n';
```

Your output will look like this:

```
pi = 3.1
```

By comparison, consider if you want to display pi in fixed-point format instead:

```
cout << "pi = " << fixed
     << setprecision(2)
     << pi << '\n';
```

Now the output will look like this:

```
pi = 3.14
```

This is because, in fixed-point format, the precision refers to the number of digits to the *right* of the decimal point. If we multiply pi by 1,000 in the same format, the number of digits to the right of the decimal remains unchanged:

```
cout << "pi = " << fixed
     << setprecision(2)
     << pi * 1000 << '\n';
```

produces:

```
pi = 3142.86
```

This is nice, because you can set your precision, set your field width with `setw`, right-justify your output with `right` (see Recipe 10.1), and your decimal points will all be lined up vertically.

Since a manipulator is just a convenient way of setting a format flag on the stream, remember that the settings stick around until you undo them or until the stream is destroyed. Save the format flags (see Example 10-3) before you start making changes, and restore them when you are done

See Also

Recipe 10.3

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;
```

Example 10-4. A simple stream manipulator (continued)

```
// 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 output floating-point data in the default format.

The compiler knows what to do with your new function because the standard library already defines an operator for `basic_ostream` (`basic_ostream` is the name of the class template that `ostream` and `wostream` are instantiations of) like this:

```
basic_ostream<charT,traits>& operator<<
(basic_ostream<charT,traits>& (* pf)(basic_ostream<charT,traits>&))
```

In this example, `pf` is a pointer to a function that takes a `basic_ostream` reference argument and returns a `basic_ostream` reference. This operator just calls your function with the current stream as an argument.

Writing manipulators that take arguments is more complicated. To understand why, consider how a manipulator without arguments works. When you use a manipulator like this:

```
myostream << myManip << "foo";
```

You use it without parenthesis, so that it actually resolves to the address of your manipulator function. `operator<<` is what actually calls the manipulator function, and it passes in the stream so the manipulator can do its work.

For the sake of comparison, say you have a manipulator that takes a numeric argument, so that, ideally, you would use it like this:

```
myostream << myFancyManip(17) << "apple";
```

How is this going to work? If you assume `myFancyManip` is a function that takes an integer argument, then there is a problem: How do you pass the stream to the function without including in the parameters and using it explicitly? Here's what you might do:

```
myostream << myFancyManip(17, myostream) << "apple";
```

But this is ugly and redundant. One of the conveniences of a manipulator is the ability to just add it in line with a bunch of `operator<<`s and to read and use it easily.

The solution is to send the compiler on a detour. Instead of `operator<<` just invoking your manipulator function on the stream, you need to introduce an ephemeral object that returns something `operator<<` can use. Here's how.

First, you need to define a temporary class to do the work. For the sake of simplicity, say you want to write a manipulator called `setWidth` that does the same thing as `setw`. The temporary structure you need to build should look something like this:

```
class WidthSetter {

public:
   WidthSetter (int n) : width_(n) {}
   void operator( )(ostream& os) const {os.width(width_);}
private:
   int width_;
};
```

The function of this class is simple. Construct it with an integer argument, and when `operator( )` is invoked with a stream argument, set the width on the stream to the value that the object was initialized with. The point of this behavior is that `WidthSetter` will

be constructed by one function and used by another. Your manipulator function is what will construct it, and it should look like this:

```
WidthSetter setWidth(int n) {
   return(WidthSetter(n));   // Return the initialized object
}
```

All this does is return a `WidthSetter` object that was initialized with the integer value. This is the manipulator that you will use in line with `operator<<`s, like this:

```
myostream << setWidth(20) << "banana";
```

But this alone is not enough, because if `setWidth` just returns a `WidthSetter` object, `operator<<` won't know what to do with it. You have to overload `operator<<` so it knows how to handle a `WidthSetter`:

```
ostream& operator<<(ostream& os, const WidthSetter& ws) {
   ws(os);     // Pass the stream to the ws object
   return(os); // to do the real work
}
```

That solves the problem, but in a nongeneric way. You don't want to have to write a `WidthSetter`-style class for every argument-accepting manipulator you write (maybe you do, but never mind that), so a better approach is to use templates and function pointers to make a nice, generic infrastructure on which you can base any number of manipulators. Example 10-5 provides the `ManipInfra` class and a version of `operator<<` that uses template arguments to deal with the different kinds of characters a stream may handle and the different kinds of arguments a stream manipulator might use.

Example 10-5. Manipulator infrastructure

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

using namespace std;

// ManipInfra is a small, intermediary class that serves as a utility
// for custom manipulators with arguments. Call its constructor with a
// function pointer and a value from your main manipulator function.
// The function pointer should be a helper function that does the
// actual work. See examples below.
template<typename T, typename C>
class ManipInfra {

public:
   ManipInfra (basic_ostream<C>& (*pFun)
               (basic_ostream<C>&, T), T val)
     : manipFun_(pFun), val_(val) {}
   void operator()(basic_ostream<C>& os) const
      {manipFun_(os, val_);}  // Invoke the function pointer with the
private:                      // stream and value
   T val_;
```

Example 10-5. Manipulator infrastructure (continued)

```
   basic_ostream<C>& (*manipFun_)
      (basic_ostream<C>&, T);
};

template<typename T, typename C>
basic_ostream<C>& operator<<(basic_ostream<C>& os,
                             const ManipInfra<T, C>& manip) {
   manip(os);
   return(os);
}

// Helper function that is ultimately called by the ManipInfra class
ostream& setTheWidth(ostream& os, int n) {
   os.width(n);
   return(os);
}

// Manipulator function itself.  This is what is used by client
// code
ManipInfra<int, char> setWidth(int n) {
   return(ManipInfra<int, char>(setTheWidth, n));
}

// Another helper that takes a char argument
ostream& setTheFillChar(ostream& os, char c) {
   os.fill(c);
   return(os);
}

ManipInfra<char, char> setFill(char c) {
   return(ManipInfra<char, char>(setTheFillChar, c));
}

int main( ) {

  cout << setFill('-')
       << setWidth(10) << right << "Proust\n";
}
```

If the sequence of events is still hazy, I suggest running Example 10-5 in the debugger. Once you see it in action, it will make perfect sense.

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) {
```

Example 10-6. Writing objects to a stream (continued)

```
   out << emp.firstName_ << endl;
   out << emp.lastName_ << endl;
   if (emp.empr_)
     out << *emp.empr_ << endl;

   return(out);
}

int main( ) {

   Employee emp;
   string first = "William";
   string last = "Shatner";
   Employer empr;
   string name = "Enterprise";
   empr.setName(name);

   emp.setFirstName(first);
   emp.setLastName(last);
   emp.setEmployer(empr);

   cout << emp;  // Write to the stream
}
```

Discussion

The first thing you need to do is declare `operator<<` as a `friend` of the class you want to write to a stream. You should use `operator<<` instead of a member function like `writeToStream(ostream& os)` because the left-shift operator is the convention for writing everything else in the standard library to a stream. You need to declare it as a `friend` because, in most cases, you want to write private members to the stream, and non-`friend` functions can't access them.

After that, define the version of `operator<<` that operates on an `ostream` or `wostream` (defined in `<ostream>`) and your class that you have already declared as a `friend`. This is where you have to decide which data members should be written to the stream. Typically, you will want to write all data members to the stream, as I did in Example 10-6:

```
out << emp.firstName_ << endl;
out << emp.lastName_ << endl;
```

In Example 10-6, I wrote the object pointed to by `empr_` by invoking `operator<<` on it:

```
if (emp.empr_)
   out << *emp.empr_ << endl;
```

I can do this because `empr_` points to an object of the `Employer` class, and, like `Employee`, I have defined `operator<<` for it.

When you are done writing your class's members to the stream, your `operator<<` must return the stream it was passed. You need to do this whenever you overload `operator<<`, so it can be used in succession, like this:

```
cout << "Here's my object: " << myObj << '\n';
```

The approach I give is simple, and when you want to be able to write a class to an output stream for human consumption, it will work just fine, but that's only part of the story. If you are writing an object to a stream, it's usually for one of two reasons. Either that stream goes somewhere that will be read by a person (`cout`, console window, a log file, etc.), or the stream is a temporary or persistent storage medium (a `stringstream`, a network connection, a file, etc.), and you plan on reassembling the object from that stream in the future. If you need to recreate the object from a stream (the subject of Recipe 10.5), you need to think carefully about your class relationships.

Implementing serialization for anything other than trivial classes is hard work. If your class references (via pointer or reference) other classes—as most nontrivial classes do—you have to accommodate the potential for circular references in a meaningful way when writing out objects, and you have to reconstruct references correctly when reading them in. If you have to build something from scratch, then you'll have to handle these design considerations, but if you can use an external library, you should try the Boost Serialization library, which provides a framework for serializing objects in a portable way.

See Also

Recipe 10.5

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>
```

Example 10-7. Reading data into an object from a stream (continued)

```
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";

   emp.setFirstName(first);
   emp.setLastName(last);

   ofstream out("tmp\\emp.txt");

   if (!out) {
      cerr << "Unable to open output file.\n";
      exit(EXIT_FAILURE);
   }
```

Example 10-7. Reading data into an object from a stream (continued)

```
   out << emp;  // Write the Emp to the file
   out.close( );

   ifstream in("tmp\\emp.txt");

   if (!in) {
      cerr << "Unable to open input file.\n";
      exit(EXIT_FAILURE);
   }

   Employee emp2;

   in >> emp2;  // Read the file into an empty object
   in.close( );

   cout << emp2;
}
```

Discussion

The steps for making a class readable from a stream are nearly identical to, but the opposite of, those for writing an object to a stream. If you have not already read Recipe 10.4, you should do so for Example 10-7 to make sense.

First, you have to declare an `operator>>` as a `friend` of your target class, but, in this case, you want it to use an `istream` instead of an `ostream`. Then define `operator>>` (instead of `operator<<`) to read values from the stream directly into each of your class's member variables. When you are done reading in data, return the input stream.

See Also

Recipe 10.4

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>
```

Example 10-8. Obtaining file information (continued)

```
#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 that hold data about a file, and second there is a system call (function) of the same name, which gets information about whatever file you specify and populates a `stat struct` with it. A *system call* is a function that provides some service from the OS. A number of system calls are part of Standard C, and many of

them are standardized across the different versions of Unix. The stat struct looks like this (from Kernigan and Richie's *The C Programming Language* [Prentice Hall]):

```
struct stat {
   dev_t   st_dev;    /* device of inode */
   ino_t   st_ino;    /* inode number */
   short   st_mode;   /* mode bits */
   short   st_nlink;  /* number of links to file */
   short   st_uid;    /* owner's user id */
   short   st_gid;    /* owner's group id */
   dev_t   st_rdev;   /* for special files */
   off_t   st_size;   /* file size in characters */
   time_t  st_atime;  /* time last accessed */
   time_t  st_mtime;  /* time last modified */
   time_t  st_ctime;  /* time inode last changed */
};
```

The meaning of each of stat's members depends on the OS. For example, st_uid and st_gid mean nothing on Windows systems; whereas on Unix systems, they actually contain the user and group ids of the file's owner. Take a look at your OS documentation to see which values are supported and how to interpret them.

Example 10-8 shows how to display some of the portable members of stat. st_mode contains a bit mask describing the type of file. You can use it to determine if the file is a directory or not. st_size is the file size in bytes. The three time_t members are timestamps of the access, modification, and creation times of the files.

The remaining members contain operating-system specific information. Consider st_dev: on Windows systems, it contains the device number (drive) as an offset from ASCII letter A (which is why I add an 'A' to it in the example—this gives you the drive letter). But that won't give you the same results on Unix; pass the value returned to the ustat system call to obtain the filesystem name.

If you need more information about a file, the best thing to do is to do some investigating in your OS's documentation. The standard C system calls are Unix-centric, so they are usually more useful on Unix systems (and have a number of other system calls that can be used in conjunction with them). If you are not using Unix, chances are there are proprietary libraries that ship with your OS's development environment that provide more detailed information.

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 time—instead of a character-at-a-time copy loop—by 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 other member functions, and the output stream has a buffer that holds output that has been written to the stream but not the destination (in the case of an `ofstream`, it's a file but it could be a string, a network connection, or who-knows-what). Therefore, the best thing to do is to let the buffers exchange data directly. You can do this with `operator<<`, which behaves differently than usual when used with stream buffers. For example, instead of the `do`/`while` loop in Example 10-9, use this:

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

Don't place this statement in the body of the loop, replace the loop with this single line. It looks a little odd, since, typically, `operator<<` says, "take the righthand side and send it to the lefthand stream," but bear with me and it will make sense. `rdbuf` returns the buffer from the input stream, and the implementation of `operator<<` that takes a stream buffer as a righthand argument reads a character at a time from the input buffer and writes it to the output buffer. When the input buffer is emptied, it knows it has to refill itself with data from the real source, and `operator<<` is none the wiser.

Example 10-9 shows how to copy the *contents* of a file yourself, but your OS is responsible for managing the filesystem, which encompasses copying them, so why not let the OS do the work? In most cases, the answer to this question is that a direct call to the OS API is, of course, not portable. Boost's Filesystem library masks a lot of the OS-specific APIs for you by providing the function `copy_file`, which makes different OS calls based on the platform it was compiled for. Example 10-10 contains a short program that copies a file from one location to another.

Example 10-10. Copying a file with Boost

```
#include <iostream>
#include <string>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

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

   // Parameter checking...

   try {
      // Turn the args into absolute paths using native formatting
      path src = complete(path(argv[1], native));
      path dst = complete(path(argv[2], native));
```

Example 10-10. Copying a file with Boost (continued)

```
      copy_file(src, dst);
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }

   return(EXIT_SUCCESS);
}
```

This a small program, but there are a few key parts that need explaining because other recipes in this chapter use the Boost Filesystem library. To begin with, the central component of the Boost Filesystem library is the `path` class, which represents, in an OS-independent way, a path to a file or directory. You can create a `path` using either a portable or OS-native string. In Example 10-10, I create a `path` out of the program arguments (that I then pass to `complete`, which I discuss in a moment):

```
path src = complete(path(argv[1], native));
```

The first argument is the text of the path, e.g., `"tmp\\foo.txt"` and the second argument is the name of a function that accepts a `string` argument and returns a `boolean` that validates that a path is valid according to certain rules. The `native` function means to use the OS's native format for validation. I used it in Example 10-10 because the arguments are passed in from the command line where they are presumably typed in by a human user, who will probably use the native OS format when specifying files. There are a number of functions that you can use to validate file and directory names, all of which are self-explanatory: `portable_posix_name`, `windows_name`, `portable_name`, `portable_directory_name`, `portable_file_name`, and `no_check`. See the documentation for specifics.

`complete` composes an absolute path using the current working directory and the relative path you pass it. Thus, I can do this to create an absolute path to the source file:

```
path src = complete(path("tmp\\foo.txt", native));
```

`complete` handles the case where the first argument is already an absolute filename by using the value given rather than trying to merge it with the current working directory. In other words, the following code invoked from a current directory of `"c:\myprograms"` ignores the current working directory since the path given is already complete:

```
path src = complete(path("c:\\windows\\garbage.txt", native));
```

Many of the Boost Filesystem functions will throw an exception if a precondition is not met. The documentation has all the details, but a good example is with the `copy_file` function itself. A file must exist before it can be copied, so if the source file does not exist, the operation cannot succeed; therefore, `copy_file` will throw an exception.

Catch the exception as I did in Example 10-10 and you will get an error message that explains the problem.

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);
}
```

The Boost Filesystem library also provides the ability to remove or rename a file. Example 10-12 shows a short program for removing a file (or directory, but see the discussion after the example).

Example 10-12. Removing a file with Boost

```
#include <iostream>
#include <string>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

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

   // Do parameter checking...

   try {
      path p = complete(path(argv[1], native));
      remove(p);
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }
   return(EXIT_SUCCESS);
}
```

The important part of Example 10-12 is the remove function. Call it with a valid path argument that refers to a file or an empty directory, and it will be removed. For an explanation of the path class and complete function, both of which are part of the Boost Filesystem library, take a look at the discussion in Recipe 10.7. See Recipe 10.11 for an example of how to remove a directory and all the files it contains.

Renaming a file or directory is similar. Replace the code in the try block in Example 10-12 with this code:

```
path src = complete(path(argv[1], native));
path dst = complete(path(argv[2], native));
rename(src, dst);
```

This will rename src to dst, so long as each is a valid path. src and dst don't have to have a common base directory, and in that respect, the rename function logically moves a file or directory to a new base directory, so long as dst exists.

See Also

Recipe 10.7

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 filename—how 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>
#include <fstream>
#include <cstdio>
#include <string>

int main( ) {

   char* pFileName = NULL;

   pFileName = tmpnam(NULL);
   // Right here is where another program may get the same temp
   // filename.

   if (!pFileName) {
      std::cerr << "Couldn't create temp file name.\n";
      return(EXIT_FAILURE);
   }

   std::cout << "The temp file name is: " << pFileName << '\n';

   std::ofstream of(pFileName);

   if (of) {
      of << "Here is some temp data.";
      of.close( );
   }

   std::ifstream ifs(pFileName);
   std::string s;

   if (ifs) {
      ifs >> s;
```

Example 10-14. Creating a temporary filename (continued)

```
      std::cout << "Just read in \"" << s << "\"\n";
      ifs.close();
   }
}
```

But there is something important you should know about tmpnam. It has a race condition whereby more than one process may generate the same filename if one calls tmpname and the other calls tmpname before the first process opens the file. This is bad for two reasons. First, a malicious program can do this to intercept the data in a temporary file, and second, an unsuspecting program can get the same filename and simply corrupt or delete data.

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
      std::cerr << "Error: " << strerror(errno);
      return(EXIT_FAILURE);
   }
}
```

If you want portability, and don't want to write all the `#ifdef`s yourself, you should consider using the Boost Filesystem library. You can create a directory using the `create_directory` function, as shown in Example 10-16, which contains a short program that creates a directory.

Example 10-16. Creating a directory with Boost

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

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

   // Parameter checking...
```

Example 10-16. Creating a directory with Boost (continued)

```
   try {
      path p = complete(path(argv[1], native));
      create_directory(p);
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }

  return(EXIT_SUCCESS);
}
```

The `create_directory` function creates a directory identified by the `path` argument you give it. If that directory already exists, a `filesystem_error` exception is thrown (derived from the standard exception class). For an explanation of the `path` class and `complete` function, both of which are part of the Boost Filesystem library, take a look at the discussion in Recipe 10.7. See Recipe 10.11 for an example of how to remove a directory and all the files it contains. If, on the other hand, portability is not a concern, consult your OS's proprietary filesystem API, which will most likely offer more flexibility.

See Also

Recipe 10.12

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) {
```

Example 10-17. Removing a directory (continued)

```
   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>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

void removeRecurse(const path& p) {

   // First, remove the contents of the directory
   directory_iterator end;
   for (directory_iterator it(p);
        it != end; ++it) {
```

Example 10-18. Removing a directory with Boost (continued)

```
      if (is_directory(*it)) {
         removeRecurse(*it);
      }
      else {
         remove(*it);
      }
   }
   // Then, remove the directory itself
   remove(p);
}

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

   if (argc != 2) {
      cerr << "Usage: " << argv[0] << " [dir name]\n";
      return(EXIT_FAILURE);
   }

   path thePath = system_complete(path(argv[1], native));

   if (!exists(thePath)) {
      cerr << "Error: the directory " << thePath.string( )
           << " does not exist.\n";
      return(EXIT_FAILURE);
   }

   try {
      removeRecurse(thePath);
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
      return(EXIT_FAILURE);
   }
   return(EXIT_SUCCESS);
}
```

The code that iterates through the directory contents requires some explanation, and that is the subject of Recipe 10.12.

The Boost Filesystem library is handy, but remember that Boost is not a formal standard and therefore is not guaranteed to run everywhere. If you examine the source code for the Boost Filesystem library, you will notice that essentially, it compiles native OS calls based on the target platform. If portability is not a concern, consult your OS's proprietary filesystem API, which will most likely offer more flexibility.

See Also

Recipe 10.12

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)";
```

Example 10-19. Reading a directory (continued)

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

   return(EXIT_SUCCESS);
}
```

Discussion

Like creating or deleting directories (see Recipes 10.10 and 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 provides many more—see the other recipes in this chapter or the Boost Filesystem web page at *www.boost.com* for more information.

Example 10-19 is a simple directory listing program (like `ls` on Unix or `dir` on MS-DOS). First, it builds an absolute pathname out of the argument passed to the program, like this:

```
path fullPath = complete(path(argv[1], native));
```

The data type of a path is called, appropriately, `path`. This is the type that the filesystem routines operate on, and is easily convertible to a string by calling `path::string`. Once the path has been assembled, the program checks its existence (with `exists`), then checks to see if it is a directory with another utility function, `is_directory`. If it is, then everything is in good shape and it can proceed to the real work of listing the directory contents.

There is a class called `directory_iterator` in filesystem that uses standard iterator semantics, like the standard containers, to allow you to use an iterator like you would a pointer to a directory element. Unlike standard containers, however, there is no `end` member function you can call on a directory that represents one-past-the-last-element (i.e., `vector<T>::end`). Instead, if you create a `directory_iterator` with the default constructor, it represents an end marker that you can use for comparison to determine when you are done. So do this:

```
directory_iterator end;
```

and then you can create an iterator from your path, and compare it to end, like this:

```
for (directory_iterator it(fullPath);
     it != end; ++it) {
   // do whatever you want to *it
   std::cout << it->leaf();
}
```

The `leaf` member function returns a string representing the element referred to by a path, and not the full path itself, which is what you get if you call the `string` member function.

If you have to write something that is portable, but for some reason you cannot use Boost, take a look at the Boost code itself. It contains `#ifdefs` that deal with (for the most part) Windows versus Posix OS interface environments and path particulars, such as drive letters versus device names.

See Also

Recipes 10.10 and 10.11

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

Recipes 4.9 and 10.12

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));
   }
```

Example 10-21. Extracting a filename from a path (continued)

```
   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 pathname—which may be a file or directory name—easy 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) {

   // Parameter checking...

   try {
      path p = complete(path(argv[1], native));
      cout << p.leaf( ) << " is a "
           << (is_directory(p) ? "directory" : "file") << endl;
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }

  return(EXIT_SUCCESS);
}
```

See the discussion in Recipe 10.7 for more information about the `path` class.

See Also

Recipe 10.15

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) {

   // Parameter checking...

   try {
      path p = complete(path(argv[1], native));
      cout << p.branch_path( ).string( ) << endl;
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }
  return(EXIT_SUCCESS);
}
```

Sample output from Example 10-24 looks like this:

```
D:\src\ccb\c10>bin\GetPathBoost.exe c:\windows\system32\1033
c:/windows/system32
```

See Also

Recipes 10.13 and 10.14

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

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.)

Example 10-27. Combining paths with Boost

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

Example 10-27. Combining paths with Boost (continued)

```
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/fstream.hpp>

using namespace std;
using namespace boost::filesystem;

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

   // Parameter checking...

   try {
      // Compose a path from the two args
      path p1 = complete(path(argv[2], native),
                         path(argv[1], native));
      cout << p1.string( ) << endl;

      // Create a path with a base of the current dir
      path p2 = system_complete(path(argv[2], native));
      cout << p2.string( ) << endl;
   }
   catch (exception& e) {
      cerr << e.what( ) << endl;
   }

  return(EXIT_SUCCESS);
}
```

The output of the program in Example 10-27 might look like this:

```
D:\src\ccb\c10>bin\MakePathBoost.exe d:\temp some\other\dir
d:/temp/some/other/dir
D:/src/ccb/c10/some/other/dir
```

or:

```
D:\src\ccb\c10>bin\MakePathBoost.exe d:\temp c:\WINDOWS\system32
c:/WINDOWS/system32
c:/WINDOWS/system32
```

What you can see here is that `complete` and `system_complete` merge paths when possible, or return the absolute path when merging paths makes no sense. For example, in the first output, the first argument given to the program is an absolute directory and the second is a relative directory. `complete` merges them together and produces a single, absolute path. The first argument to `complete` is the relative path, and the second is the absolute path, and if the first argument is already an absolute path, the second argument is ignored. That's why in the second output you can see that the argument `"d:\temp"` is ignored since the second argument I give is already an absolute path.

`system_complete` only takes a single argument (the relative path in this case) and appends it to the current working directory to produce another absolute path. Again,

if you give it a path that is already absolute, it ignores the current working directory and simply returns the absolute path you gave it.

These paths are not reconciled with the filesystem though. You have to explicitly test to see if a `path` object represents a valid filesystem path. For example, to check if one of these paths exists, you can use the `exists` function on a path:

```
path p1 = complete(path(argv[2], native),
                   path(argv[1], native));
if (exists(p1)) {
   // ...
```

There are many more functions you can use to get information about a path: `is_directory`, `is_empty`, `file_size`, `last_write_time`, and so on. See the Boost Filesystem library documentation at *www.boost.org* for more information.

See Also

Recipe 10.7