# Thinking in C++, 2nd edition, Revision 2





# 2: Iostreams

There's much more you can do with the general I/O problem than just take standard I/O and turn it into a class.

Wouldn't it be nice if you could make all the usual "receptacles" – standard I/O, files and even blocks of memory – look the same, so you need to remember only one interface? That's the idea behind iostreams. They're much easier, safer, and often more efficient than the assorted functions from the Standard C stdio library.

Iostream is usually the first class library that new C++ programmers learn to use. This chapter explores the *use* of iostreams, so they can replace the C I/O functions through the rest of the book. In future chapters, you'll see how to set up your own classes so they're compatible with iostreams.

## Why iostreams?

You may wonder what's wrong with the good old C library. And why not "wrap" the C library in a class and be done with it? Indeed, there are situations when this is the perfect thing to do, when you want to make a C library a bit safer and easier to use. For example, suppose you want to make sure a stdio file is always safely opened and properly closed, without relying on the user to remember to call the **close( )** function:

```
//: C02:FileClass.h
// Stdio files wrapped
#ifndef FILECLAS_H
#define FILECLAS_H
#include <cstdio>

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

In C when you perform file I/O, you work with a naked pointer to a FILE **struct**, but this class wraps around the pointer and guarantees it is properly initialized and cleaned up using the constructor and destructor. The second constructor argument is the file mode, which defaults to "r" for "read."

To fetch the value of the pointer to use in the file I/O functions, you use the **fp( )** access function. Here are the member function definitions:

```
//: C02:FileClass.cpp {O}
// Implementation
#include "FileClass.h"
#include <cstdlib>
using namespace std;

FileClass::FileClass(const char* fname, const char* mode){
  f = fopen(fname, mode);
  if(f == NULL) {
    printf("%s: file not found\n", fname);
    exit(1);
  }
}

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

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

The constructor calls **fopen( )**,as you would normally do, but it also checks to ensure the result isn't zero, which indicates a failure upon opening the file. If there's a failure, the name of the file is printed and **exit( )** is called.

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

```
//: C02:FileClassTest.cpp
//{L} FileClass
// Testing class File
#include "FileClass.h"
#include "../require.h"
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  FileClass f(argv[1]); // Opens and tests
  const int bsize = 100;
  char buf[bsize];
  while(fgets(buf, bsize, f.fp()))
    puts(buf);
} // File automatically closed by destructor
///:~
```

You create the **FileClass** object and use it in normal C file I/O function calls by calling **fp( )**. When you're done with it, just forget about it, and the file is closed by the destructor at the end of the scope.

## True wrapping

Even though the FILE pointer is private, it isn't particularly safe because **fp( )** retrieves it. The only effect seems to be guaranteed initialization and cleanup, so why not make it public, or use a **struct** instead? Notice that while you can get a copy of **f** using **fp( )**, you cannot assign to **f** – that's completely under the control of the class. Of course, after capturing the pointer returned by **fp( )**, the client programmer can still assign to the structure elements, so the safety is in guaranteeing a valid FILE pointer rather than proper contents of the structure.

If you want complete safety, you have to prevent the user from direct access to the FILE pointer. This means some version of all the normal file I/O functions will have to show up as class members, so everything you can do with the C approach is available in the C++ class:

```
//: C02:Fullwrap.h
// Completely hidden file IO
#ifndef FULLWRAP_H
#define FULLWRAP_H

class File {
  std::FILE* f;
  std::FILE* F(); // Produces checked pointer to f
public:
  File(); // Create object but don't open file
  File(const char* path,
       const char* mode = "r");
  ~File();
  int open(const char* path,
           const char* mode = "r");
  int reopen(const char* path,
             const char* mode);
  int getc();
  int ungetc(int c);
  int putc(int c);
  int puts(const char* s);
  char* gets(char* s, int n);
  int printf(const char* format, ...);
  size_t read(void* ptr, size_t size,
              size_t n);
  size_t write(const void* ptr,
               size_t size, size_t n);
  int eof();
  int close();
  int flush();
  int seek(long offset, int whence);
  int getpos(fpos_t* pos);
  int setpos(const fpos_t* pos);
  long tell();
  void rewind();
  void setbuf(char* buf);
  int setvbuf(char* buf, int type, size_t sz);
  int error();
  void clearErr();
};
#endif // FULLWRAP_H ///:~
```

This class contains almost all the file I/O functions from **cstdio**. **vfprintf( )** is missing; it is used to implement the **printf( )** member function.

**File** has the same constructor as in the previous example, and it also has a default constructor. The default constructor is important if you want to create an array of **File** objects or use a **File** object as a member of another class where the initialization doesn't happen in the constructor (but sometime after the enclosing object is created).

The default constructor sets the private **FILE** pointer **f** to zero. But now, before any reference to **f**, its value must be checked to ensure it isn't zero. This is accomplished with the last member function in the class, **F( )**, which is **private** because it is intended to be used only by other member functions. (We don't want to give the user direct access to the **FILE** structure in this class.)[6]

This is not a terrible solution by any means. It's quite functional, and you could imagine making similar classes for standard (console) I/O and for in-core formatting (reading/writing a piece of memory rather than a file or the console).

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

1. Even if you use only a fraction of the functionality of the interpreter, the whole thing gets loaded. So if you say:
   `printf("%c", 'x');`
   you'll get the whole package, including the parts that print out floating-point numbers and strings. There's no option for reducing the amount of space used by the program.
2. Because the interpretation happens at runtime there's a performance overhead you can't get rid of. It's frustrating because all the information is *there* in the format string at compile time, but it's not evaluated until runtime. However, if you could parse the arguments in the format string at compile time you could make hard function calls that have the potential to be much faster than a runtime interpreter (although the **printf( )** family of functions is usually quite well optimized).
3. A worse problem occurs because the evaluation of the format string doesn't happen until runtime: there can be no compile-time error checkin

g. You're probably very familiar with this problem if you've tried to find bugs that came from using the wrong number or type of arguments in a **printf( )** statement. C++ makes a big deal out of compile-time error checking to find errors early and make your life easier. It seems a shame to throw it away for an I/O library, especially because I/O is used a lot.

1. For C++, the most important problem is that the **printf( )** family of functions is not particularly extensibl

e. They're really designed to handle the four basic data types in C (**char**, **int**, **float**, **double** and their variations). You might think that every time you add a new class, you could add an overloaded **printf( )** and **scanf( )** function (and their variants for files and strings) but remember, overloaded functions must have different types in their argument lists and the **printf( )** family hides its type information in the format string and in the variable argument list. For a language like C++, whose goal is to be able to easily add new data types, this is an ungainly restriction.

# Iostreams to the rescue

All these issues make it clear that one of the first standard class libraries for C++ should handle I/O. Because "hello, world" is the first program just about everyone writes in a new language, and because I/O is part of virtually every program, the I/O library in C++ must be particularly easy to use. It also has the much greater challenge that it can never know all the classes it must accommodate, but it must nevertheless be adaptable to use any new class. Thus its constraints required that this first class be a truly inspired design.

This chapter won't look at the details of the design and how to add iostream functionality to your own classes (you'll learn that in a later chapter). First, you need to learn to use iostreams. In addition to gaining a great deal of leverage and clarity in your dealings with I/O and formatting, you'll also see how a really powerful C++ library can work.

## Sneak preview of operator overloading

Before you can use the iostreams library, you must understand one new feature of the language that won't be covered in detail until a later chapter. To use iostreams, you need to know that in C++ all the operators can take on different meanings. In this chapter, we're particularly interested in << and >>. The statement "operators can take on different meanings" deserves some extra insight.

In Chapter XX, you learned how function overloading allows you to use the same function name with different argument lists. Now imagine that when the compiler sees an expression consisting of an argument followed by an operator followed by an argument, it simply calls a function. That is, an operator is simply a function call with a different syntax.

Of course, this is C++, which is very particular about data types. So there must be a previously declared function to match that operator and those particular argument types, or the compiler will not accept the expression.

What most people find immediately disturbing about operator overloading is the thought that maybe everything they know about operators in C is suddenly wrong. This is absolutely false. Here are two of the sacred design goals of C++:

1. A program that compiles in C will compile in C++. The only compilation errors and warnings from the C++ compiler will result from the "holes" in the C language, and fixing these will require only local editing. (Indeed, the complaints by the C++ compiler usually lead you directly to undiscovered bugs in the C program.)
2. The C++ compiler will not secretly change the behavior of a C program by recompiling it under C++.

Keeping these goals in mind will help answer a lot of questions; knowing there are no capricious changes to C when moving to C++ helps make the transition easy. In particular, operators for built-in types won't suddenly start working differently – you cannot change their meaning. Overloaded operators can be created only where new data types are involved. So you can create a new overloaded operator for a new class, but the expression

```
1 << 4;
```

won't suddenly change its meaning, and the illegal code

```
1.414 << 1;
```

won't suddenly start working.

## Inserters and extractors

In the iostreams library, two operators have been overloaded to make the use of iostreams easy. The operator **<<** is often referred to as an *inserter* for iostreams, and the operator **>>** is often referred to as an *extractor*.

A *stream* is an object that formats and holds bytes. You can have an input stream (*istream*) or an output stream (*ostream*). There are different types of istreams and ostreams: *ifstreams* and *ofstreams* for files, *istrstreams* , and *ostrstreams* for **char*** memory (in-core formatting), and *istringstreams* & *ostringstreams* for interfacing with the Standard C++ **string** class. All these stream objects have the same interface, regardless of whether you're working with a file, standard I/O, a piece of memory or a **string** object. The single interface you learn also works for extensions added to support new classes.

If a stream is capable of producing bytes (an istream), you can get information from the stream using an extractor. The extractor produces and formats the type of information that's expected by the destination object. To see an example of this, you can use the **cin** object, which is the iostream equivalent of **stdin** in C, that is, redirectable standard input. This object is pre-defined whenever you include the **iostream.h** header file. (Thus, the iostream library is automatically linked with most compilers.)

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

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

There's an overloaded **operator >>** for every data type you can use as the right-hand argument of **>>** in an iostream statement. (You can also overload your own, which you'll see in a later chapter.)

To find out what you have in the various variables, you can use the **cout** object (corresponding to standard output; there's also a **cerr** object corresponding to standard error) with the inserter **<<**:

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
```

```
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

This is notably tedious, and doesn't seem like much of an improvement over **printf( )**, type checking or no. Fortunately, the overloaded inserters and extractors in iostreams are designed to be chained together into a complex expression that is much easier to write:

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

You'll understand how this can happen in a later chapter, but for now it's sufficient to take the attitude of a class user and just know it works that way.

### Manipulators

One new element has been added here: a *manipulator* called **endl**. A manipulator acts on the stream itself; in this case it inserts a newline and *flushes* the stream (puts out all pending characters that have been stored in the internal stream buffer but not yet output). You can also just flush the stream:

```
cout << flush;
```

There are additional basic manipulators that will change the number base to **oct** (octal), **dec** (decimal) or **hex** (hexadecimal):

```
cout << hex << "0x" << i << endl;
```

There's a manipulator for extraction that "eats" white space:

```
cin >> ws;
```

and a manipulator called **ends**, which is like **endl**, only for strstreams (covered in a while). These are all the manipulators in **<iostream>**, but there are more in **<iomanip>** you'll see later in the chapter.

## Common usage

Although **cin** and the extractor **>>** provide a nice balance to **cout** and the inserter **<<**, in practice using formatted input routines, especially with standard input, has the same problems you run into with **scanf( )**. If the input produces an unexpected value, the process is skewed, and it's very difficult to recover. In addition, formatted input defaults to whitespace delimiters. So if you collect the above code fragments into a program

```
//: C02:Iosexamp.cpp
// Iostream examples
#include <iostream>
using namespace std;

int main() {
```

```
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

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

  cout << flush;
  cout << hex << "0x" << i << endl;
} ///:~
```

and give it the following input,

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

you'll get the same output as if you give it

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

and the output is, somewhat unexpectedly,

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

Notice that **buf** got only the first word because the input routine looked for a space to delimit the input, which it saw after "this." In addition, if the continuous input string is longer than the storage allocated for **buf**, you'll overrun the buffer.

It seems **cin** and the extractor are provided only for completeness, and this is probably a good way to look at it. In practice, you'll usually want to get your input a line at a time as a sequence of characters and then scan them and perform conversions once they're safely in a buffer. This way you don't have to worry about the input routine choking on unexpected data.

Another thing to consider is the whole concept of a command-line interface. This has made sense in the past when the console was little more than a glass typewriter, but the world is rapidly changing to one where the graphical user interface (GUI) dominates. What is the meaning of console I/O in such a world? It makes much more sense to ignore **cin** altogether other than for very simple examples or tests, and take the following approaches:

1. If your program requires input, read that input from a file – you'll soon see it's remarkably easy to use files with iostreams. Iostreams

for files still works fine with a GUI.

1. Read the input without attempting to convert it. Once the input is someplace where it can't foul things up during conversion, then you can safely scan it.
2. Output is different. If you're using a GUI, **cout** doesn't work and you must send it to a file (which is identical to sending it to **cout**) or use the GUI facilities for data display. Otherwise it often makes sense to send it to **cout**. In both cases, the output formatting functions of iostreams are highly useful.

## Line-oriented input

To grab input a line at a time, you have two choices: the member functions **get( )** and **getline( )**. Both functions take three arguments: a pointer to a character buffer in which to store the result, the size of that buffer (so they don't overrun it), and the terminating character, to know when to stop reading input. The terminating character has a default value of **'\n'**, which is what you'll usually use. Both functions store a zero in the result buffer when they encounter the terminating character in the input.

So what's the difference? Subtle, but important: **get( )** stops when it *sees* the delimiter in the input stream, but it doesn't extract it from the input stream. Thus, if you did another **get( )** using the same delimiter it would immediately return with no fetched input. (Presumably, you either use a different delimiter in the next **get( )** statement or a different input function.) **getline( )**, on the other hand, extracts the delimiter from the input stream, but still doesn't store it in the result buffer.

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

### Overloaded versions of get( )

**get( )** also comes in three other overloaded versions: one with no arguments that returns the next character, using an **int** return value; one that stuffs a character into its **char** argument, using a *reference* (You'll have to jump forward to Chapter XX if you want to understand it right this minute . . . .); and one that stores directly into the underlying buffer structure of another iostream object. That is explored later in the chapter.

### Reading raw bytes

If you know exactly what you're dealing with and want to move the bytes directly into a variable, array, or structure in memory, you can use **read( )**. The first argument is a pointer to the destination memory, and the second is the number of bytes to read. This is especially useful if you've previously stored the information to a file, for example, in binary form using the complementary **write( )** member function for an output stream. You'll see examples of all these functions later.

### Error handling

All the versions of **get( )** and **getline( )** return the input stream from which the characters came *except* for **get( )** with no arguments, which returns the next character or EOF. If you get the input stream object back, you can ask it if it's still OK. In fact, you can ask *any* iostream object if it's OK using the member functions **good( )**, **eof( )**, **fail( )**, and **bad( )**. These return state information based on the **eofbit** (indicates the buffer is at the end of sequence), the **failbit** (indicates some operation has failed because of formatting issues or some other problem that does not affect the buffer) and the **badbit** (indicates something has gone wrong with the buffer).

However, as mentioned earlier, the state of an input stream generally gets corrupted in weird ways only when you're trying to do input to specific types and the type read from the input is inconsistent with what is expected. Then of course you have the problem of what to do with the input stream to correct the problem. If you follow my advice and read input a line at a time or as a big glob of characters (with **read( )**) and don't attempt to use the input formatting functions except in simple cases, then all you're concerned with is whether you're at the end of the input (EOF). Fortunately, testing for this turns out to be simple and can be done inside of conditionals, such as **while(cin)** or **if(cin)**. For now you'll have to accept that when you use an input stream object in this context, the right value is safely, correctly and magically produced to indicate whether the object has reached the end of the input. You can also use the Boolean NOT operator **!**, as in **if(!cin)**, to indicate the stream is *not* OK; that is, you've probably reached the end of input and should quit trying to read the stream.

There are times when the stream becomes not-OK, but you understand this condition and want to go on using it. For example, if you reach the end of an input file, the **eofbit** and **failbit** are set, so a conditional on that stream object will indicate the stream is no longer good. However, you may want to continue using the file, by seeking to an earlier position and reading more data. To correct the condition, simply call the **clear( )** member function.[7]

# File iostreams

Manipulating files with iostreams is much easier and safer than using **cstdio** in C. All you do to open a file is create an object; the constructor does the work. You don't have to explicitly close a file (although you can, using the **close( )** member function) because the destructor will close it when the object goes out of scope.

To create a file that defaults to input, make an **ifstream** object. To create one that defaults to output, make an **ofstream** object.

Here's an example that shows many of the features discussed so far. Note the inclusion of **<fstream>** to declare the file I/O classes; this also includes **<iostream>**.

```
//: C02:Strfile.cpp
// Stream I/O with files
// The difference between get() & getline()
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main() {
```

```
  const int sz = 100; // Buffer size;
  char buf[sz];
  {
    ifstream in("Strfile.cpp"); // Read
    assure(in, "Strfile.cpp"); // Verify open
    ofstream out("Strfile.out"); // Write
    assure(out, "Strfile.out");
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, sz)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("Strfile.out");
  assure(in, "Strfile.out");
  // More convenient line input:
  while(in.getline(buf, sz)) { // Removes \n
    char* cp = buf;
    while(*cp != ':')
      cp++;
    cp += 2; // Past ": "
    cout << cp << endl; // Must still add \n
  }
} ///:~
```

The creation of both the **ifstream** and **ofstream** are followed by an **assure( )** to guarantee the file has been successfully opened. Here again the object, used in a situation where the compiler expects an integral result, produces a value that indicates success or failure. (To do this, an automatic type conversion member function is called. These are discussed in Chapter XX.)

The first **while** loop demonstrates the use of two forms of the **get( )** function. The first gets characters into a buffer and puts a zero terminator in the buffer when either **sz – 1** characters have been read or the third argument (defaulted to **'\n'**) is encountered. **get( )** leaves the terminator character in the input stream, so this terminator must be thrown away via **in.get( )** using the form of **get( )** with no argument, which fetches a single byte and returns it as an **int**. You can also use the **ignore( )** member function, which has two defaulted arguments. The first is the number of characters to throw away, and defaults to one. The second is the character at which the **ignore( )** function quits (after extracting it) and defaults to EOF.

Next you see two output statements that look very similar: one to **cout** and one to the file **out**. Notice the convenience here; you don't need to worry about what kind of object you're dealing with because the formatting statements work the same with all **ostream** objects. The first one echoes the line to standard output, and the second writes the line out to the new file and includes a line number.

To demonstrate **getline( )**, it's interesting to open the file we just created and strip off the line numbers. To ensure the file is properly closed before opening it to read, you have two choices. You can surround the first part of the program in braces to force the **out** object out of scope, thus calling the destructor and closing the file, which is done here. You can also call **close( )** for both files; if you want, you can

even reuse the **in** object by calling the **open( )** member function (you can also create and destroy the object dynamically on the heap as is in Chapter XX).

The second **while** loop shows how **getline( )** removes the terminator character (its third argument, which defaults to **'\n'**) from the input stream when it's encountered. Although **getline( )**, like **get( )**, puts a zero in the buffer, it still doesn't insert the terminating character.

### Open modes

You can control the way a file is opened by changing a default argument. The following table shows the flags that control the mode of the file:

| Flag | Function |
|---|---|
| **ios::in** | Opens an input file. Use this as an open mode for an **ofstream** to prevent truncating an existing file. |
| **ios::out** | Opens an output file. When used for an **ofstream** without **ios::app**, **ios::ate** or **ios::in**, **ios::trunc** is implied. |
| **ios::app** | Opens an output file for appending. |
| **ios::ate** | Opens an existing file (either input or output) and seeks the end. |
| **ios::nocreate** | Opens a file only if it already exists. (Otherwise it fails.) |
| **ios::noreplace** | Opens a file only if it does not exist. (Otherwise it fails.) |
| **ios::trunc** | Opens a file and deletes the old file, if it already exists. |
| **ios::binary** | Opens a file in binary mode. Default is text mode. |

These flags can be combined using a bitwise *or*.

## Iostream buffering

Whenever you create a new class, you should endeavor to hide the details of the underlying implementation as possible from the user of the class. Try to show them only what they need to know and make the rest **private** to avoid confusion. Normally when using iostreams you don't know or care where the bytes are being produced or consumed; indeed, this is different depending on whether you're dealing with standard I/O, files, memory, or some newly created class or device.

There comes a time, however, when it becomes important to be able to send messages to the part of the iostream that produces and consumes bytes. To provide this part with a common interface and still hide its underlying implementation, it is abstracted into its own class, called **streambuf**. Each iostream object contains a pointer to some kind of **streambuf**. (The kind depends on whether it deals with standard I/O, files, memory, etc.) You can access the **streambuf** directly; for example, you can move raw bytes into and out of the **streambuf**, without formatting them through the enclosing iostream. This is accomplished, of course, by calling member functions for the **streambuf** object.

Currently, the most important thing for you to know is that every iostream object contains a pointer to a **streambuf** object, and the **streambuf** has some member functions you can call if you need to.

To allow you to access the **streambuf**, every iostream object has a member function called **rdbuf( )** that returns the pointer to the object's **streambuf**. This way you can call any member function for the underlying **streambuf**. However, one of the most interesting things you can do with the **streambuf** pointer is to connect it to another iostream object using the **<<** operator. This drains all the bytes from your object into the one on the left-hand side of the **<<**. This means if you want to move all the bytes from one iostream to another, you don't have to go through the tedium (and potential coding errors) of reading them one byte or one line at a time. It's a much more elegant approach.

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

```
//: C02:Stype.cpp
// Type a file to standard output
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Must have a command line
  ifstream in(argv[1]);
  assure(in, argv[1]); // Ensure file exists
  cout << in.rdbuf(); // Outputs entire file
} ///:~
```

After making sure there is an argument on the command line, an **ifstream** is created using this argument. The open will fail if the file doesn't exist, and this failure is caught by the **assert(in)**.

All the work really happens in the statement

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

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

## Using get( ) with a streambuf

There is a form of **get( )** that allows you to write directly into the **streambuf** of another object. The first argument is the destination **streambuf** (whose address is mysteriously taken using a *reference*, discussed in Chapter XX), and the second is the terminating character, which stops the **get( )** function. So yet another way to print a file to standard output is

```
//: C02:Sbufget.cpp
// Get directly into a streambuf
#include "../require.h"
#include <fstream>
#include <iostream>
using namespace std;

int main() {
  ifstream in("Sbufget.cpp");
  assure(in, "Sbufget.cpp");
  while(in.get(*cout.rdbuf()))
```

```
    in.ignore();
} ///:~
```

**rdbuf( )** returns a pointer, so it must be dereferenced to satisfy the function's need to see an object. The **get( )** function, remember, doesn't pull the terminating character from the input stream, so it must be removed using **ignore( )** so **get( )** doesn't just bonk up against the newline forever (which it will, otherwise).

You probably won't need to use a technique like this very often, but it may be useful to know it exists.

# Seeking in iostreams

Each type of iostream has a concept of where its "next" character will come from (if it's an **istream**) or go (if it's an **ostream**). In some situations you may want to move this stream position. You can do it using two models: One uses an absolute location in the stream called the **streampos**; the second works like the Standard C library functions **fseek( )** for a file and moves a given number of bytes from the beginning, end, or current position in the file.

The **streampos** approach requires that you first call a "tell" function: **tellp( )** for an **ostream** or **tellg( )** for an **istream**. (The "p" refers to the "put pointer" and the "g" refers to the "get pointer.") This function returns a **streampos** you can later use in the single-argument version of **seekp( )** for an **ostream** or **seekg( )** for an **istream**, when you want to return to that position in the stream.

The second approach is a relative seek and uses overloaded versions of **seekp( )** and **seekg( )**. The first argument is the number of bytes to move: it may be positive or negative. The second argument is the seek direction:

| | |
|---|---|
| ios::beg | From beginning of stream |
| ios::cur | Current position in stream |
| ios::end | From end of stream |

Here's an example that shows the movement through a file, but remember, you're not limited to seeking within files, as you are with C and **cstdio**. With C++, you can seek in any type of iostream (although the behavior of **cin** & **cout** when seeking is undefined):

```
//: C02:Seeking.cpp
// Seeking in iostreams
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // File must already exist
  in.seekg(0, ios::end); // End of file
  streampos sp = in.tellg(); // Size of file
  cout << "file size = " << sp << endl;
  in.seekg(-sp/10, ios::end);
```

```
  streampos sp2 = in.tellg();
  in.seekg(0, ios::beg); // Start of file
  cout << in.rdbuf(); // Print whole file
  in.seekg(sp2); // Move to streampos
  // Prints the last 1/10th of the file:
  cout << endl << endl << in.rdbuf() << endl;
} ///:~
```

This program picks a file name off the command line and opens it as an **ifstream**. **assert( )** detects an open failure. Because this is a type of **istream**, **seekg( )** is used to position the "get pointer." The first call seeks zero bytes off the end of the file, that is, to the end. Because a **streampos** is a **typedef** for a **long**, calling **tellg( )** at that point also returns the size of the file, which is printed out. Then a seek is performed moving the get pointer 1/10 the size of the file – notice it's a negative seek from the end of the file, so it backs up from the end. If you try to seek positively from the end of the file, the get pointer will just stay at the end. The **streampos** at that point is captured into **sp2**, then a **seekg( )** is performed back to the beginning of the file so the whole thing can be printed out using the **streambuf** pointer produced with **rdbuf( )**. Finally, the overloaded version of **seekg( )** is used with the **streampos sp2** to move to the previous position, and the last portion of the file is printed out.

## Creating read/write files

Now that you know about the **streambuf** and how to seek, you can understand how to create a stream object that will both read and write a file. The following code first creates an **ifstream** with flags that say it's both an input and an output file. The compiler won't let you write to an **ifstream**, however, so you need to create an **ostream** with the underlying stream buffer:

```
ifstream in("filename", ios::in|ios::out);
ostream out(in.rdbuf());
```

You may wonder what happens when you write to one of these objects. Here's an example:

```
//: C02:Iofile.cpp
// Reading & writing one file
#include "../require.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Iofile.cpp");
  assure(in, "Iofile.cpp");
  ofstream out("Iofile.out");
  assure(out, "Iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("Iofile.out", ios::in | ios::out);
  assure(in2, "Iofile.out");
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
```

```
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
} ///:~
```

The first five lines copy the source code for this program into a file called **iofile.out**, and then close the files. This gives us a safe text file to play around with. Then the aforementioned technique is used to create two objects that read and write to the same file. In **cout << in2.rdbuf( )**, you can see the "get" pointer is initialized to the beginning of the file. The "put" pointer, however, is set to the end of the file because "Where does this end up?" appears appended to the file. However, if the put pointer is moved to the beginning with a **seekp( )**, all the inserted text *overwrites* the existing text. Both writes are seen when the get pointer is moved back to the beginning with a **seekg( )**, and the file is printed out. Of course, the file is automatically saved and closed when **out2** goes out of scope and its destructor is called.

# stringstreams

# strstreams

Before there were **stringstream**s, there were the more primitive **strstream**s. Although these are not an official part of Standard C++, they have been around a long time so compilers will no doubt leave in the **strstream** support in perpetuity, to compile legacy code. You should always use **stringstream**s, but it's certainly likely that you'll come across code that uses **strstream**s and at that point this section should come in handy. In addition, this section should make it fairly clear why **stringstream**s have replace **strstream**s.

A **strstream** works directly with memory instead of a file or standard output. It allows you to use the same reading and formatting functions to manipulate bytes in memory. On old computers the memory was referred to as *core* so this type of functionality is often called *in-core formatting*.

The class names for strstreams echo those for file streams. If you want to create a strstream to extract characters from, you create an **istrstream**. If you want to put characters into a strstream, you create an **ostrstream**.

String streams work with memory, so you must deal with the issue of where the memory comes from and where it goes. This isn't terribly complicated, but you must understand it and pay attention (it turned out is was too easy to lose track of this particular issue, thus the birth of **stringstream**s).

## User-allocated storage

The easiest approach to understand is when the user is responsible for allocating the storage. With **istrstream**s this is the only allowed approach. There are two constructors:

```
istrstream::istrstream(char* buf);
istrstream::istrstream(char* buf, int size);
```

The first constructor takes a pointer to a zero-terminated character array; you can extract bytes until the zero. The second constructor additionally requires the size of the array, which doesn't have to be zero-

terminated. You can extract bytes all the way to **buf[size]**, whether or not you encounter a zero along the way.

When you hand an **istrstream** constructor the address of an array, that array must already be filled with the characters you want to extract and presumably format into some other data type. Here's a simple example:[8]

```
//: C02:Istring.cpp
// Input strstreams
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  istrstream s("47 1.414 This is a test");
  int i;
  float f;
  s >> i >> f; // Whitespace-delimited input
  char buf2[100];
  s >> buf2;
  cout << "i = " << i << ", f = " << f;
  cout << " buf2 = " << buf2 << endl;
  cout << s.rdbuf(); // Get the rest...
} ///:~
```

You can see that this is a more flexible and general approach to transforming character strings to typed values than the Standard C Library functions like **atof( )**, **atoi( )**, and so on.

The compiler handles the static storage allocation of the string in

```
  istrstream s("47 1.414 This is a test");
```

You can also hand it a pointer to a zero-terminated string allocated on the stack or the heap.

In **s >> i >> f**, the first number is extracted into **i** and the second into **f**. This isn't "the first whitespace-delimited set of characters" because it depends on the data type it's being extracted into. For example, if the string were instead, "**1.414 47 This is a test**," then **i** would get the value one because the input routine would stop at the decimal point. Then **f** would get **0.414**. This could be useful if you want to break a floating-point number into a whole number and a fraction part. Otherwise it would seem to be an error.

As you may already have guessed, **buf2** doesn't get the rest of the string, just the next whitespace-delimited word. In general, it seems the best place to use the extractor in iostreams is when you know the exact sequence of data in the input stream and you're converting to some type other than a character string. However, if you want to extract the rest of the string all at once and send it to another iostream, you can use **rdbuf( )** as shown.

## Output strstreams

Output strstreams also allow you to provide your own storage; in this case it's the place in memory the bytes are formatted *into*. The appropriate constructor is

```
ostrstream::ostrstream(char*, int, int = ios::out);
```

The first argument is the preallocated buffer where the characters will end up, the second is the size of the buffer, and the third is the mode. If the mode is left as the default, characters are formatted into the starting address of the buffer. If the mode is either **ios::ate** or **ios::app** (same effect), the character buffer is assumed to already contain a zero-terminated string, and any new characters are added starting at the zero terminator.

The second constructor argument is the size of the array and is used by the object to ensure it doesn't overwrite the end of the array. If you fill the array up and try to add more bytes, they won't go in.

An important thing to remember about **ostrstream**s is that the zero terminator you normally need at the end of a character array *is not* inserted for you. When you're ready to zero-terminate the string, use the special manipulator **ends**.

Once you've created an **ostrstream** you can insert anything you want, and it will magically end up formatted in the memory buffer. Here's an example:

```
//: C02:Ostring.cpp
// Output strstreams
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  const int sz = 100;
  cout << "type an int, a float and a string:";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Throw away white space
  char buf[sz];
  cin.getline(buf, sz); // Get rest of the line
  // (cin.rdbuf() would be awkward)
  ostrstream os(buf, sz, ios::app);
  os << endl;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << ends;
  cout << buf;
  cout << os.rdbuf(); // Same effect
  cout << os.rdbuf(); // NOT the same effect
} ///:~
```

This is similar to the previous example in fetching the **int** and **float**. You might think the logical way to get the rest of the line is to use **rdbuf( )**; this works, but it's awkward because all the input including newlines is collected until the user presses control-Z (control-D on Unix) to indicate the end of the input. The approach shown, using **getline( )**, gets the input until the user presses the carriage return. This input is fetched into **buf**, which is subsequently used to construct the **ostrstream os**. If the third argument **ios::app** weren't supplied, the constructor would default to writing at the beginning of **buf**, overwriting the line that was just collected. However, the "append" flag causes it to put the rest of the formatted information at the end of the string.

You can see that, like the other output streams, you can use the ordinary formatting tools for sending bytes to the **ostrstream**. The only difference is that you're responsible for inserting the zero at the end with **ends**. Note that **endl** inserts a newline in the strstream, but no zero.

Now the information is formatted in **buf**, and you can send it out directly with **cout << buf**. However, it's also possible to send the information out with **os.rdbuf( )**. When you do this, the get pointer inside the **streambuf** is moved forward as the characters are output. For this reason, if you say **cout << os.rdbuf( )** a second time, nothing happens – the get pointer is already at the end.

## Automatic storage allocation

Output strstreams (but *not* **istrstream**s) give you a second option for memory allocation: they can do it themselves. All you do is create an **ostrstream** with no constructor arguments:

```
ostrstream a;
```

Now **a** takes care of all its own storage allocation on the heap. You can put as many bytes into **a** as you want, and if it runs out of storage, it will allocate more, moving the block of memory, if necessary.

This is a very nice solution if you don't know how much space you'll need, because it's completely flexible. And if you simply format data into the strstream and then hand its **streambuf** off to another iostream, things work perfectly:

```
a << "hello, world. i = " << i << endl << ends;
cout << a.rdbuf();
```

This is the best of all possible solutions. But what happens if you want the physical address of the memory that **a**'s characters have been formatted into? It's readily available – you simply call the **str( )** member function:

```
char* cp = a.str();
```

There's a problem now. What if you want to put more characters into **a**? It would be OK if you knew **a** had already allocated enough storage for all the characters you want to give it, but that's not true. Generally, **a** will run out of storage when you give it more characters, and ordinarily it would try to allocate more storage on the heap. This would usually require moving the block of memory. But the stream objects has just handed you the address of its memory block, so it can't very well move that block, because you're expecting it to be at a particular location.

The way an **ostrstream** handles this problem is by "freezing" itself. As long as you don't use **str( )** to ask for the internal **char***, you can add as many characters as you want to the **ostrstream**. It will allocate all the necessary storage from the heap, and when the object goes out of scope, that heap storage is automatically released.

However, if you call **str( )**, the **ostrstream** becomes "frozen." You can't add any more characters to it. Rather, you aren't *supposed* to – implementations are not required to detect the error. Adding characters to a frozen **ostrstream** results in undefined behavior. In addition, the **ostrstream** is no longer responsible for cleaning up the storage. You took over that responsibility when you asked for the **char*** with **str( )**.

To prevent a memory leak, the storage must be cleaned up somehow. There are two approaches. The more common one is to directly release the memory when you're done. To understand this, you need a sneak preview of two new keywords in C++: **new** and **delete**. As you'll see in Chapter XX, these do quite a bit, but for now you can think of them as replacements for **malloc( )** and **free( )** in C. The operator **new** returns a chunk of memory, and **delete** frees it. It's important to know about them here because virtually all memory allocation in C++ is performed with **new**, and this is also true with **ostrstream**. If it's allocated with **new**, it must be released with **delete**, so if you have an **ostrstream a** and you get the **char*** using **str( )**, the typical way to clean up the storage is

```
delete []a.str();
```

This satisfies most needs, but there's a second, much less common way to release the storage: You can unfreeze the **ostrstream**. You do this by calling **freeze( )**, which is a member function of the **ostrstream**'s **streambuf**. **freeze( )** has a default argument of one, which freezes the stream, but an argument of zero will unfreeze it:

```
a.rdbuf()->freeze(0);
```

Now the storage is deallocated when **a** goes out of scope and its destructor is called. In addition, you can add more bytes to **a**. However, this may cause the storage to move, so you better not use any pointer you previously got by calling **str( )** – it won't be reliable after adding more characters.

The following example tests the ability to add more characters after a stream has been unfrozen:

```
//: C02:Walrus.cpp
// Freezing a strstream
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  ostrstream s;
  s << "'The time has come', the walrus said,";
  s << ends;
  cout << s.str() << endl; // String is frozen
  // s is frozen; destructor won't delete
  // the streambuf storage on the heap
  s.seekp(-1, ios::cur); // Back up before NULL
  s.rdbuf()->freeze(0); // Unfreeze it
  // Now destructor releases memory, and
  // you can add more characters (but you
  // better not use the previous str() value)
  s << " 'To speak of many things'" << ends;
  cout << s.rdbuf();
} ///:~
```

After putting the first string into **s**, an **ends** is added so the string can be printed using the **char*** produced by **str( )**. At that point, **s** is frozen. We want to add more characters to **s**, but for it to have any effect, the put pointer must be backed up one so the next character is placed on top of the zero inserted by **ends**. (Otherwise the string would be printed only up to the original zero.) This is accomplished with **seekp( )**. Then **s** is unfrozen by fetching the underlying **streambuf** pointer using **rdbuf( )** and calling **freeze(0)**. At this point **s** is like it was before calling **str( )**: We can add more characters, and cleanup will occur automatically, with the destructor.

It is *possible* to unfreeze an **ostrstream** and continue adding characters, but it is not common practice. Normally, if you want to add more characters once you've gotten the **char*** of a **ostrstream**, you create a new one, pour the old stream into the new one using **rdbuf( )** and continue adding new characters to the new **ostrstream**.

### Proving movement

If you're still not convinced you should be responsible for the storage of a **ostrstream** if you call **str( )**, here's an example that demonstrates the storage location is moved, therefore the old pointer returned by **str( )** is invalid:

```
//: C02:Strmove.cpp
// ostrstream memory movement
#include <iostream>
#include <strstream>
using namespace std;

int main() {
  ostrstream s;
  s << "hi";
  char* old = s.str(); // Freezes s
  s.rdbuf()->freeze(0); // Unfreeze
  for(int i = 0; i < 100; i++)
    s << "howdy"; // Should force reallocation
  cout << "old = " << (void*)old << endl;
  cout << "new = " << (void*)s.str(); // Freezes
  delete s.str(); // Release storage
} ///:~
```

After inserting a string to **s** and capturing the **char*** with **str( )**, the string is unfrozen and enough new bytes are inserted to virtually assure the memory is reallocated and most likely moved. After printing out the old and new **char*** values, the storage is explicitly released with **delete** because the second call to **str( )** froze the string again.

To print out addresses instead of the strings they point to, you must cast the **char*** to a **void***. The operator **<<** for **char*** prints out the string it is pointing to, while the operator **<<** for **void*** prints out the hex representation of the pointer.

It's interesting to note that if you don't insert a string to **s** before calling **str( )**, the result is zero. This means no storage is allocated until the first time you try to insert bytes to the **ostrstream**.

### A better way

Again, remember that this section was only left in to support legacy code. You should always use **string** and **stringstream** rather than character arrays and **strstream**. The former is much safer and easier to use and will help ensure your projects get finished faster.

## Output stream formatting

The whole goal of this effort, and all these different types of iostreams, is to allow you to easily move and translate bytes from one place to another. It certainly wouldn't be very useful if you couldn't do all

the formatting with the **printf( )** family of functions. In this section, you'll learn all the output formatting functions that are available for iostreams, so you can get your bytes the way you want them.

The formatting functions in iostreams can be somewhat confusing at first because there's often more than one way to control the formatting: through both member functions and manipulators. To further confuse things, there is a generic member function to set state flags to control formatting, such as left- or right-justification, whether to use uppercase letters for hex notation, whether to always use a decimal point for floating-point values, and so on. On the other hand, there are specific member functions to set and read values for the fill character, the field width, and the precision.

In an attempt to clarify all this, the internal formatting data of an iostream is examined first, along with the member functions that can modify that data. (Everything can be controlled through the member functions.) The manipulators are covered separately.

## Internal formatting data

The class **ios** (which you can see in the header file **<iostream>**) contains data members to store all the formatting data pertaining to that stream. Some of this data has a range of values and is stored in variables: the floating-point precision, the output field width, and the character used to pad the output (normally a space). The rest of the formatting is determined by flags, which are usually combined to save space and are referred to collectively as the *format flags*. You can find out the value of the format flags with the **ios::flags( )** member function, which takes no arguments and returns a **long** (**typedef**ed to **fmtflags**) that contains the current format flags. All the rest of the functions make changes to the format flags and return the previous value of the format flags.

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

The first function forces *all* the flags to change, which you do sometimes. More often, you change one flag at a time using the remaining three functions.

The use of **setf( )** can seem more confusing: To know which overloaded version to use, you must know what type of flag you're changing. There are two types of flags: ones that are simply on or off, and ones that work in a group with other flags. The on/off flags are the simplest to understand because you turn them on with **setf(fmtflags)** and off with **unsetf(fmtflags)**. These flags are

| **on/off flag** | **effect** |
|---|---|
| ios::skipws | Skip white space. (For input; this is the default.) |
| ios::showbase | Indicate the numeric base (dec, oct, or hex) when printing an integral value. The format used can be read by the C++ compiler. |
| ios::showpoint | Show decimal point and trailing zeros for floating-point values. |
| ios::uppercase | Display uppercase A-F for hexadecimal values and E for scientific values. |
| ios::showpos | Show plus sign (+) for positive values. |
| ios::unitbuf | "Unit buffering." The stream is flushed after each insertion. |

| | |
|---|---|
| ios::stdio | Synchronizes the stream with the C standard I/O system. |

For example, to show the plus sign for **cout**, you say **cout.setf(ios::showpos)**. To stop showing the plus sign, you say **cout.unsetf(ios::showpos)**.

The last two flags deserve some explanation. You turn on unit buffering when you want to make sure each character is output as soon as it is inserted into an output stream. You could also use unbuffered output, but unit buffering provides better performance.

The **ios::stdio** flag is used when you have a program that uses both iostreams and the C standard I/O library (not unlikely if you're using C libraries). If you discover your iostream output and **printf( )** output are occurring in the wrong order, try setting this flag.

## Format fields

The second type of formatting flags work in a group. You can have only one of these flags on at a time, like the buttons on old car radios – you push one in, the rest pop out. Unfortunately this doesn't happen automatically, and you have to pay attention to what flags you're setting so you don't accidentally call the wrong **setf( )** function. For example, there's a flag for each of the number bases: hexadecimal, decimal, and octal. Collectively, these flags are referred to as the **ios::basefield**. If the **ios::dec** flag is set and you call **setf(ios::hex)**, you'll set the **ios::hex** flag, but you *won't* clear the **ios::dec** bit, resulting in undefined behavior. The proper thing to do is call the second form of **setf( )** like this: **setf(ios::hex, ios::basefield)**. This function first clears all the bits in the **ios::basefield**, *then* sets **ios::hex**. Thus, this form of **setf( )** ensures that the other flags in the group "pop out" whenever you set one. Of course, the **hex( )** manipulator does all this for you, automatically, so you don't have to concern yourself with the internal details of the implementation of this class or to even *care* that it's a set of binary flags. Later you'll see there are manipulators to provide equivalent functionality in all the places you would use **setf( )**.

Here are the flag groups and their effects:

| **ios::basefield** | **effect** |
|---|---|
| ios::dec | Format integral values in base 10 (decimal) (default radix). |
| ios::hex | Format integral values in base 16 (hexadecimal). |
| ios::oct | Format integral values in base 8 (octal). |

| **ios::floatfield** | **effect** |
|---|---|
| ios::scientific | Display floating-point numbers in scientific format. Precision field indicates number of digits after the decimal point. |
| ios::fixed | Display floating-point numbers in fixed format. Precision field indicates number of digits after the decimal point. |
| "automatic" (Neither bit is set.) | Precision field indicates the total number of significant digits. |

| **ios::adjustfield** | **effect** |
|---|---|
| ios::left | Left-align values; pad on the right with the fill character. |

| | |
|---|---|
| ios::right | Right-align values. Pad on the left with the fill character. This is the default alignment. |
| ios::internal | Add fill characters after any leading sign or base indicator, but before the value. |

### Width, fill and precision

The internal variables that control the width of the output field, the fill character used when the data doesn't fill the output field, and the precision for printing floating-point numbers are read and written by member functions of the same name.

| **function** | **effect** |
|---|---|
| int ios::width( ) | Reads the current width. (Default is 0.) Used for both insertion and extraction. |
| int ios::width(int n) | Sets the width, returns the previous width. |
| int ios::fill( ) | Reads the current fill character. (Default is space.) |
| int ios::fill(int n) | Sets the fill character, returns the previous fill character. |
| int ios::precision( ) | Reads current floating-point precision. (Default is 6.) |
| int ios::precision(int n) | Sets floating-point precision, returns previous precision. See **ios::floatfield** table for the meaning of "precision." |

The fill and precision values are fairly straightforward, but width requires some explanation. When the width is zero, inserting a value will produce the minimum number of characters necessary to represent that value. A positive width means that inserting a value will produce at least as many characters as the width; if the value has less than width characters, the fill character is used to pad the field. However, the value will never be truncated, so if you try to print 123 with a width of two, you'll still get 123. The field width specifies a *minimum* number of characters; there's no way to specify a maximum number.

The width is also distinctly different because it's reset to zero by each inserter or extractor that could be influenced by its value. It's really not a state variable, but an implicit argument to the inserters and extractors. If you want to have a constant width, you have to call **width( )** after each insertion or extraction.

## An exhaustive example

To make sure you know how to call all the functions previously discussed, here's an example that calls them all:

```
//: C02:Format.cpp
// Formatting functions
#include <fstream>
using namespace std;
#define D(A) T << #A << endl; A
ofstream T("format.out");

int main() {
  D(int i = 47;)
  D(float f = 2300114.414159;)
  char* s = "Is there any more?";
```

```
  D(T.setf(ios::unitbuf);)
//  D(T.setf(ios::stdio);)  // SOMETHING MAY HAVE CHANGED

  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase);)
  D(T.setf(ios::showpos);)
  D(T << i << endl;) // Default to dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "fill char: " << T.fill() << endl;)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T << i << endl;) // Without width(10)

  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "prec = " << T.precision() << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "prec = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;

  D(T.unsetf(ios::showpoint);)
  D(T.unsetf(ios::unitbuf);)
//  D(T.unsetf(ios::stdio);) // SOMETHING MAY HAVE CHANGED
} ///:~
```

This example uses a trick to create a trace file so you can monitor what's happening. The macro **D(a)** uses the preprocessor "stringizing" to turn **a** into a string to print out. Then it reiterates **a** so the statement takes effect. The macro sends all the information out to a file called **T**, which is the trace file. The output is

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::stdio);
T.setf(ios::showbase);
T.setf(ios::uppercase);
T.setf(ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
+0X2F
T.unsetf(ios::uppercase);
T.setf(ios::oct, ios::basefield);
T << i << endl;
+057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300115e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.setf(0, ios::floatfield);
T << f << endl;
2.300115e+06
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000020000000000
T.setf(ios::scientific, ios::floatfield);
```

```
T << endl << f << endl;

2.30011450000000020000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.setf(0, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
T.unsetf(ios::showpoint);
T.unsetf(ios::unitbuf);
T.unsetf(ios::stdio);
```

Studying this output should clarify your understanding of the iostream formatting member functions.

# Formatting manipulators

As you can see from the previous example, calling the member functions can get a bit tedious. To make things easier to read and write, a set of manipulators is supplied to duplicate the actions provided by the member functions.

Manipulators with no arguments are provided in **<iostream>**. These include **dec**, **oct**, and **hex** , which perform the same action as, respectively, **setf(ios::dec, ios::basefield)**, **setf(ios::oct, ios::basefield)**, and **setf(ios::hex, ios::basefield)**, albeit more succinctly. **<iostream>**[9] also includes **ws**, **endl**, **ends**, and **flush** and the additional set shown here:

| **manipulator** | **effect** |
|---|---|
| showbase<br>noshowbase | Indicate the numeric base (dec, oct, or hex) when printing an integral value. The format used can be read by the C++ compiler. |
| showpos<br>noshowpos | Show plus sign (+) for positive values |
| uppercase<br>nouppercase | Display uppercase A-F for hexadecimal values, and E for scientific values |
| showpoint<br>noshowpoint | Show decimal point and trailing zeros for floating-point values. |
| skipws<br>noskipws | Skip white space on input. |
| left<br>right<br>internal | Left-align, pad on right.<br>Right-align, pad on left.<br>Fill between leading sign or base indicator and value. |
| scientific<br>fixed | Use scientific notation<br>**setprecision( )** or **ios::precision( )** sets number of places after the decimal point. |

## Manipulators with arguments

If you are using manipulators with arguments, you must also include the header file <**iomanip**>. This contains code to solve the general problem of creating manipulators with arguments. In addition, it has six predefined manipulators:

| **manipulator** | **effect** |
|---|---|
| setiosflags (fmtflags n) | Sets only the format flags specified by n. Setting remains in effect until the next change, like **ios::setf( )**. |
| resetiosflags(fmtflags n) | Clears only the format flags specified by n. Setting remains in effect until the next change, like **ios::unsetf( )**. |
| setbase(base n) | Changes base to n, where n is 10, 8, or 16. (Anything else results in 0.) If n is zero, output is base 10, but input uses the C conventions: 10 is 10, 010 is 8, and 0xf is 15. You might as well use **dec**, **oct**, and **hex** for output. |
| setfill(char n) | Changes the fill character to n, like **ios::fill( )**. |
| setprecision(int n) | Changes the precision to n, like **ios::precision( )**. |
| setw(int n) | Changes the field width to n, like **ios::width( )**. |

If you're using a lot of inserters, you can see how this can clean things up. As an example, here's the previous program rewritten to use the manipulators. (The macro has been removed to make it easier to read.)

```
//: C02:Manips.cpp
// Format.cpp using manipulators
#include <fstream>
#include <iomanip>
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  trc << setiosflags(
         ios::unitbuf /*| ios::stdio */ /// ?????
         | ios::showbase | ios::uppercase
         | ios::showpos);
  trc << i << endl; // Default to dec
  trc << hex << i << endl;
  trc << resetiosflags(ios::uppercase)
    << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
    << dec << setfill('0');
  trc << "fill char: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
```

```
  trc.setf(ios::internal, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc << i << endl; // Without setw(10)

  trc << resetiosflags(ios::showpos)
    << setiosflags(ios::showpoint)
    << "prec = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc.setf(0, ios::floatfield); // Automatic
  trc << f << endl;
  trc << setprecision(20);
  trc << "prec = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc.setf(0, ios::floatfield); // Automatic
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;

  trc << resetiosflags(
         ios::showpoint | ios::unitbuf
         // | ios::stdio // ????????
 );
} ///:~
```

You can see that a lot of the multiple statements have been condensed into a single chained insertion. Note the calls to **setiosflags( )** and **resetiosflags( )**, where the flags have been bitwise-ORed together. This could also have been done with **setf( )** and **unsetf( )** in the previous example.

# Creating manipulators

(Note: This section contains some material that will not be introduced until later chapters.) Sometimes you'd like to create your own manipulators, and it turns out to be remarkably simple. A zero-argument manipulator like **endl** is simply a function that takes as its argument an **ostream** reference (references are a different way to pass arguments, discussed in Chapter XX). The declaration for **endl** is

```
ostream& endl(ostream&);
```

Now, when you say:

```
cout << "howdy" << endl;
```

the **endl** produces the *address* of that function. So the compiler says "is there a function I can call that takes the address of a function as its argument?" There is a pre-defined function in **Iostream.h** to do

this; it's called an *applicator*. The applicator calls the function, passing it the **ostream** object as an argument.

You don't need to know how the applicator works to create your own manipulator; you only need to know the applicator exists. Here's an example that creates a manipulator called **nl** that emits a newline *without* flushing the stream:

```
//: C02:nl.cpp
// Creating a manipulator
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
} ///:~
```

The expression

```
os << '\n';
```

calls a function that returns **os**, which is what is returned from **nl**.[10]

People often argue that the **nl** approach shown above is preferable to using **endl** because the latter always flushes the output stream, which may incur a performance penalty.

## Effectors

As you've seen, zero-argument manipulators are quite easy to create. But what if you want to create a manipulator that takes arguments? The iostream library has a rather convoluted and confusing way to do this, but Jerry Schwarz, the creator of the iostream library, suggests[11] a scheme he calls *effectors*. An effector is a simple class whose constructor performs the desired operation, along with an overloaded **operator<<** that works with the class. Here's an example with two effectors. The first outputs a truncated character string, and the second prints a number in binary (the process of defining an overloaded **operator<<** will not be discussed until Chapter XX):

```
//: C02:Effector.txt
// (Should be "cpp" but I can't get it to compile with
// My windows compilers, so making it a txt file will
// keep it out of the makefile for the time being)
// Jerry Schwarz's "effectors"
#include<iostream>
#include <cstdlib>
#include <string>
#include <climits> // ULONG_MAX
using namespace std;

// Put out a portion of a string:
class Fixw {
```

```
  string str;
public:
  Fixw(const string& s, int width)
    : str(s, 0, width) {}
  friend ostream&
  operator<<(ostream& os, Fixw& fw) {
    return os << fw.str;
  }
};

typedef unsigned long ulong;

// Print a number in binary:
class Bin {
  ulong n;
public:
  Bin(ulong nn) { n = nn; }
  friend ostream& operator<<(ostream&, Bin&);
};

ostream& operator<<(ostream& os, Bin& b) {
  ulong bit = ~(ULONG_MAX >> 1); // Top bit set
  while(bit) {
    os << (b.n & bit ? '1' : '0');
    bit >>= 1;
  }
  return os;
}

int main() {
  char* string =
    "Things that make us happy, make us wise";
  for(int i = 1; i <= strlen(string); i++)
    cout << Fixw(string, i) << endl;
  ulong x = 0xCAFEBABEUL;
  ulong y = 0x76543210UL;
  cout << "x in binary: " << Bin(x) << endl;
  cout << "y in binary: " << Bin(y) << endl;
} ///:~
```

The constructor for **Fixw** creates a shortened copy of its **char*** argument, and the destructor releases the memory created for this copy. The overloaded **operator<<** takes the contents of its second argument, the **Fixw** object, and inserts it into the first argument, the **ostream**, then returns the **ostream** so it can be used in a chained expression. When you use **Fixw** in an expression like this:

```
cout << Fixw(string, i) << endl;
```

a *temporary object* is created by the call to the **Fixw** constructor, and that temporary is passed to **operator<<**. The effect is that of a manipulator with arguments.

The **Bin** effector relies on the fact that shifting an unsigned number to the right shifts zeros into the high bits. ULONG_MAX (the largest **unsigned long** value, from the standard include file **<climits>** ) is used to produce a value with the high bit set, and this value is moved across the number in question (by shifting it), masking each bit.

Initially the problem with this technique was that once you created a class called **Fixw** for **char*** or **Bin** for **unsigned long**, no one else could create a different **Fixw** or **Bin** class for their type. However, with *namespaces* (covered in Chapter XX), this problem is eliminated.

# Iostream examples

In this section you'll see some examples of what you can do with all the information you've learned in this chapter. Although many tools exist to manipulate bytes (stream editors like **sed** and **awk** from Unix are perhaps the most well known, but a text editor also fits this category), they generally have some limitations. **sed** and **awk** can be slow and can only handle lines in a forward sequence, and text editors usually require human interaction, or at least learning a proprietary macro language. The programs you write with iostreams have none of these limitations: They're fast, portable, and flexible. It's a very useful tool to have in your kit.

## Code generation

The first examples concern the generation of programs that, coincidentally, fit the format used in this book. This provides a little extra speed and consistency when developing code. The first program creates a file to hold **main( )** (assuming it takes no command-line arguments and uses the iostream library):

```
//: C02:Makemain.cpp
// Create a shell main() file
#include "../require.h"
#include <fstream>
#include <strstream>
#include <cstring>
#include <cctype>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ofstream mainfile(argv[1]);
  assure(mainfile, argv[1]);
  istrstream name(argv[1]);
  ostrstream CAPname;
  char c;
  while(name.get(c))
    CAPname << char(toupper(c));
  CAPname << ends;
  mainfile << "//" << ": " << CAPname.rdbuf()
    << " -- " << endl
    << "#include <iostream>" << endl
    << endl
    << "main() {" << endl << endl
    << "}" << endl;
} ///:~
```

The argument on the command line is used to create an **istrstream**, so the characters can be extracted one at a time and converted to upper case with the Standard C library macro **toupper( )**. This returns an **int** so it must be explicitly cast to a **char**. This name is used in the headline, followed by the remainder of the generated file.

## Maintaining class library source

The second example performs a more complex and useful task. Generally, when you create a class you think in library terms, and make a header file **Name.h** for the class declaration and a file where the member functions are implemented, called **Name.cpp**. These files have certain requirements: a particular coding standard (the program shown here will use the coding format for this book), and in the header file the declarations are generally surrounded by some preprocessor statements to prevent multiple declarations of classes. (Multiple declarations confuse the compiler – it doesn't know which one you want to use. They could be different, so it throws up its hands and gives an error message.)

This example allows you to create a new header-implementation pair of files, or to modify an existing pair. If the files already exist, it checks and potentially modifies the files, but if they don't exist, it creates them using the proper format.

[[ This should be changed to use **string** instead of <cstring> ]]

```
//: C02:Cppcheck.cpp
// Configures .h & .cpp files
// To conform to style standard.
// Tests existing files for conformance
#include "../require.h"
#include <fstream>
#include <strstream>
#include <cstring>
#include <cctype>
using namespace std;

int main(int argc, char* argv[]) {
  const int sz = 40;  // Buffer sizes
  const int bsz = 100;
  requireArgs(argc, 1); // File set name
  enum bufs { base, header, implement,
    Hline1, guard1, guard2, guard3,
    CPPline1, include, bufnum };
  char b[bufnum][sz];
  ostrstream osarray[] = {
    ostrstream(b[base], sz),
    ostrstream(b[header], sz),
    ostrstream(b[implement], sz),
    ostrstream(b[Hline1], sz),
    ostrstream(b[guard1], sz),
    ostrstream(b[guard2], sz),
    ostrstream(b[guard3], sz),
    ostrstream(b[CPPline1], sz),
    ostrstream(b[include], sz),
  };
  osarray[base] << argv[1] << ends;
  // Find any '.' in the string using the
  // Standard C library function strchr():
  char* period = strchr(b[base], '.');
  if(period) *period = 0; // Strip extension
  // Force to upper case:
  for(int i = 0; b[base][i]; i++)
    b[base][i] = toupper(b[base][i]);
  // Create file names and internal lines:
  osarray[header] << b[base] << ".h" << ends;
```

```
osarray[implement] << b[base] << ".cpp" << ends;
osarray[Hline1] << "//" << ": " << b[header]
  << " -- " << ends;
osarray[guard1] << "#ifndef " << b[base]
                << "_H" << ends;
osarray[guard2] << "#define " << b[base]
                << "_H" << ends;
osarray[guard3] << "#endif // " << b[base]
                << "_H" << ends;
osarray[CPPline1] << "//" << ": "
                  << b[implement]
                  << " -- " << ends;
osarray[include] << "#include \""
                 << b[header] << "\"" <<ends;
// First, try to open existing files:
ifstream existh(b[header]),
         existcpp(b[implement]);
if(!existh) { // Doesn't exist; create it
  ofstream newheader(b[header]);
  assure(newheader, b[header]);
  newheader << b[Hline1] << endl
    << b[guard1] << endl
    << b[guard2] << endl << endl
    << b[guard3] << endl;
}
if(!existcpp) { // Create cpp file
  ofstream newcpp(b[implement]);
  assure(newcpp, b[implement]);
  newcpp << b[CPPline1] << endl
    << b[include] << endl;
}
if(existh) { // Already exists; verify it
  strstream hfile; // Write & read
  ostrstream newheader; // Write
  hfile << existh.rdbuf() << ends;
  // Check that first line conforms:
  char buf[bsz];
  if(hfile.getline(buf, bsz)) {
    if(!strstr(buf, "//" ":") ||
       !strstr(buf, b[header]))
      newheader << b[Hline1] << endl;
  }
  // Ensure guard lines are in header:
  if(!strstr(hfile.str(), b[guard1]) ||
     !strstr(hfile.str(), b[guard2]) ||
     !strstr(hfile.str(), b[guard3])) {
     newheader << b[guard1] << endl
       << b[guard2] << endl
       << buf
       << hfile.rdbuf() << endl
       << b[guard3] << endl << ends;
  } else
    newheader << buf
      << hfile.rdbuf() << ends;
  // If there were changes, overwrite file:
  if(strcmp(hfile.str(),newheader.str())!=0){
    existh.close();
    ofstream newH(b[header]);
    assure(newH, b[header]);
```

```
      newH << "//@//" << endl // Change marker
        << newheader.rdbuf();
    }
    delete hfile.str();
    delete newheader.str();
  }
  if(existcpp) { // Already exists; verify it
    strstream cppfile;
    ostrstream newcpp;
    cppfile << existcpp.rdbuf() << ends;
    char buf[bsz];
    // Check that first line conforms:
    if(cppfile.getline(buf, bsz))
      if(!strstr(buf, "//" ":") ||
         !strstr(buf, b[implement]))
        newcpp << b[CPPline1] << endl;
    // Ensure header is included:
    if(!strstr(cppfile.str(), b[include]))
      newcpp << b[include] << endl;
    // Put in the rest of the file:
    newcpp << buf << endl; // First line read
    newcpp << cppfile.rdbuf() << ends;
    // If there were changes, overwrite file:
    if(strcmp(cppfile.str(),newcpp.str())!=0){
      existcpp.close();
      ofstream newCPP(b[implement]);
      assure(newCPP, b[implement]);
      newCPP << "//@//" << endl // Change marker
        << newcpp.rdbuf();
    }
    delete cppfile.str();
    delete newcpp.str();
  }
} ///:~
```

This example requires a lot of string formatting in many different buffers. Rather than creating a lot of individually named buffers and **ostrstream** objects, a single set of names is created in the **enum bufs**. Then two arrays are created: an array of character buffers and an array of **ostrstream** objects built from those character buffers. Note that in the definition for the two-dimensional array of **char** buffers **b**, the number of **char** arrays is determined by **bufnum**, the last enumerator in **bufs**. When you create an enumeration, the compiler assigns integral values to all the **enum** labels starting at zero, so the sole purpose of **bufnum** is to be a counter for the number of enumerators in **buf**. The length of each string in **b** is **sz**.

The names in the enumeration are **base**, the capitalized base file name without extension; **header**, the header file name; **implement**, the implementation file (**cpp**) name; **Hline1**, the skeleton first line of the header file; **guard1**, **guard2**, and **guard3**, the "guard" lines in the header file (to prevent multiple inclusion); **CPPline1**, the skeleton first line of the **cpp** file; and **include**, the line in the **cpp** file that includes the header file.

**osarray** is an array of **ostrstream** objects created using aggregate initialization and automatic counting. Of course, this is the form of the **ostrstream** constructor that takes two arguments (the buffer address and buffer size), so the constructor calls must be formed accordingly inside the aggregate initializer list. Using the **bufs** enumerators, the appropriate array element of **b** is tied to the corresponding **osarray** object. Once the array is created, the objects in the array can be selected using

the enumerators, and the effect is to fill the corresponding **b** element. You can see how each string is built in the lines following the **ostrstream** array definition.

Once the strings have been created, the program attempts to open existing versions of both the header and **cpp** file as **ifstream**s. If you test the object using the operator '**!**' and the file doesn't exist, the test will fail. If the header or implementation file doesn't exist, it is created using the appropriate lines of text built earlier.

If the files *do* exist, then they are verified to ensure the proper format is followed. In both cases, a **strstream** is created and the whole file is read in; then the first line is read and checked to make sure it follows the format by seeing if it contains both a "//**:**" and the name of the file. This is accomplished with the Standard C library function **strstr( )**. If the first line doesn't conform, the one created earlier is inserted into an **ostrstream** that has been created to hold the edited file.

In the header file, the whole file is searched (again using **strstr( )**) to ensure it contains the three "guard" lines; if not, they are inserted. The implementation file is checked for the existence of the line that includes the header file (although the compiler effectively guarantees its existence).

In both cases, the original file (in its **strstream**) and the edited file (in the **ostrstream**) are compared to see if there are any changes. If there are, the existing file is closed, and a new **ofstream** object is created to overwrite it. The **ostrstream** is output to the file after a special change marker is added at the beginning, so you can use a text search program to rapidly find any files that need reviewing to make additional changes.

## Detecting compiler errors

All the code in this book is designed to compile as shown without errors. Any line of code that should generate a compile-time error is commented out with the special comment sequence "//!". The following program will remove these special comments and append a numbered comment to the line, so that when you run your compiler it should generate error messages and you should see all the numbers appear when you compile all the files. It also appends the modified line to a special file so you can easily locate any lines that don't generate errors:

```
//: C02:Showerr.cpp
// Un-comment error generators
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
#include <cctype>
#include <cstring>
using namespace std;
char* marker = "//!";

char* usage =
"usage: showerr filename chapnum\n"
"where filename is a C++ source file\n"
"and chapnum is the chapter name it's in.\n"
"Finds lines commented with //! and removes\n"
"comment, appending //(#) where # is unique\n"
"across all files, so you can determine\n"
"if your compiler finds the error.\n"
```

```
"showerr /r\n"
"resets the unique counter.";

// File containing error number counter:
char* errnum = "../errnum.txt";
// File containing error lines:
char* errfile = "../errlines.txt";
ofstream errlines(errfile,ios::app);

int main(int argc, char* argv[]) {
  requireArgs(argc, 2, usage);
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(errnum); // Delete files
        remove(errfile);
        return 0;
      default:
        cerr << usage << endl;
        return 1;
    }
  }
  char* chapter = argv[2];
  strstream edited; // Edited file
  int counter = 0;
  {
    ifstream infile(argv[1]);
    assure(infile, argv[1]);
    ifstream count(errnum);
    assure(count, errnum);
    if(count) count >> counter;
    int linecount = 0;
    const int sz = 255;
    char buf[sz];
    while(infile.getline(buf, sz)) {
      linecount++;
      // Eat white space:
      int i = 0;
      while(isspace(buf[i]))
        i++;
      // Find marker at start of line:
      if(strstr(&buf[i], marker) == &buf[i]) {
        // Erase marker:
        memset(&buf[i], ' ', strlen(marker));
        // Append counter & error info:
        ostrstream out(buf, sz, ios::ate);
        out << "//(" << ++counter << ") "
            << "Chapter " << chapter
            << " File: " << argv[1]
            << " Line " << linecount << endl
            << ends;
          edited << buf;
        errlines << buf; // Append error file
      } else
        edited << buf << "\n"; // Just copy
    }
  } // Closes files
```

```
  ofstream outfile(argv[1]); // Overwrites
  assure(outfile, argv[1]);
  outfile << edited.rdbuf();
  ofstream count(errnum); // Overwrites
  assure(count, errnum);
  count << counter; // Save new counter
} ///:~
```

The marker can be replaced with one of your choice.

Each file is read a line at a time, and each line is searched for the marker appearing at the head of the line; the line is modified and put into the error line list and into the **strstream edited**. When the whole file is processed, it is closed (by reaching the end of a scope), reopened as an output file and **edited** is poured into the file. Also notice the counter is saved in an external file, so the next time this program is invoked it continues to sequence the counter.

## A simple datalogger

This example shows an approach you might take to log data to disk and later retrieve it for processing. The example is meant to produce a temperature-depth profile of the ocean at various points. To hold the data, a class is used:

```
//: C02:DataLogger.h
// Datalogger record layout
#ifndef DATALOG_H
#define DATALOG_H
#include <ctime>
#include <iostream>

class DataPoint {
  std::tm time; // Time & day
  static const int bsz = 10;
  // Ascii degrees (*) minutes (') seconds ("):
  char latitude[bsz], longitude[bsz];
  double depth, temperature;
public:
  std::tm getTime();
  void setTime(std::tm t);
  const char* getLatitude();
  void setLatitude(const char* l);
  const char* getLongitude();
  void setLongitude(const char* l);
  double getDepth();
  void setDepth(double d);
  double getTemperature();
  void setTemperature(double t);
  void print(std::ostream& os);
};
#endif // DATALOG_H ///:~
```

The access functions provide controlled reading and writing to each of the data members. The **print( )** function formats the **DataPoint** in a readable form onto an **ostream** object (the argument to **print( )**). Here's the definition file:

```
//: C02:Datalog.cpp {O}
```

```
// Datapoint member functions
#include "DataLogger.h"
#include <iomanip>
#include <cstring>
using namespace std;

tm DataPoint::getTime() { return time; }

void DataPoint::setTime(tm t) { time = t; }

const char* DataPoint::getLatitude() {
  return latitude;
}

void DataPoint::setLatitude(const char* l) {
  latitude[bsz - 1] = 0;
  strncpy(latitude, l, bsz - 1);
}

const char* DataPoint::getLongitude() {
  return longitude;
}

void DataPoint::setLongitude(const char* l) {
  longitude[bsz - 1] = 0;
  strncpy(longitude, l, bsz - 1);
}

double DataPoint::getDepth() { return depth; }

void DataPoint::setDepth(double d) { depth = d; }

double DataPoint::getTemperature() {
  return temperature;
}

void DataPoint::setTemperature(double t) {
  temperature = t;
}

void DataPoint::print(ostream& os) {
  os.setf(ios::fixed, ios::floatfield);
  os.precision(4);
  os.fill('0'); // Pad on left with '0'
  os << setw(2) << getTime().tm_mon << '\\'
     << setw(2) << getTime().tm_mday << '\\'
     << setw(2) << getTime().tm_year << ' '
     << setw(2) << getTime().tm_hour << ':'
     << setw(2) << getTime().tm_min << ':'
     << setw(2) << getTime().tm_sec;
  os.fill(' '); // Pad on left with ' '
  os << " Lat:" << setw(9) << getLatitude()
     << ", Long:" << setw(9) << getLongitude()
     << ", depth:" << setw(9) << getDepth()
     << ", temp:" << setw(9) << getTemperature()
     << endl;
} ///:~
```

In **print( )**, the call to **setf( )** causes the floating-point output to be fixed-precision, and **precision( )** sets the number of decimal places to four.

The default is to right-justify the data within the field. The time information consists of two digits each for the hours, minutes and seconds, so the width is set to two with **setw( )** in each case. (Remember that any changes to the field width affect only the next output operation, so **setw( )** must be given for each output.) But first, to put a zero in the left position if the value is less than 10, the fill character is set to '0'. Afterwards, it is set back to a space.

The latitude and longitude are zero-terminated character fields, which hold the information as degrees (here, '*' denotes degrees), minutes ('), and seconds("). You can certainly devise a more efficient storage layout for latitude and longitude if you desire.

## Generating test data

Here's a program that creates a file of test data in binary form (using **write( )**) and a second file in ASCII form using **DataPoint::print( )**. You can also print it out to the screen but it's easier to inspect in file form.

```
//: C02:Datagen.cpp
//{L} Datalog
// Test data generator
#include "DataLogger.h"
#include "../require.h"
#include <fstream>
#include <cstdlib>
#include <cstring>
using namespace std;

int main() {
  ofstream data("data.txt");
  assure(data, "data.txt");
  ofstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  time_t timer;
  // Seed random number generator:
  srand(time(&timer));
  for(int i = 0; i < 100; i++) {
    DataPoint d;
    // Convert date/time to a structure:
    d.setTime(*localtime(&timer));
    timer += 55; // Reading each 55 seconds
    d.setLatitude("45*20'31\"");
    d.setLongitude("22*34'18\"");
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += span
style='color:blue'>double(1) / fraction;
    d.setDepth(newdepth);
    double newtemp = 150 + rand()%200; // Kelvin
    fraction = rand() % 100 + 1;
    newtemp += (double)1 / fraction;
    d.setTemperature(newtemp);
    d.print(data);
```

```
    bindata.write((unsigned char*)&d,
                  sizeof(d));
  }
} ///:~
```

The file DATA.TXT is created in the ordinary way as an ASCII file, but DATA.BIN has the flag **ios::binary** to tell the constructor to set it up as a binary file.

The Standard C library function **time( )**, when called with a zero argument, returns the current time as a **time_t** value, which is the number of seconds elapsed since 00:00:00 GMT, January 1 1970 (the dawning of the age of Aquarius?). The current time is the most convenient way to seed the random number generator with the Standard C library function **srand( )**, as is done here.

Sometimes a more convenient way to store the time is as a **tm** structure, which has all the elements of the time and date broken up into their constituent parts as follows:

```
struct tm {
  int tm_sec; // 0-59 seconds
  int tm_min; // 0-59 minutes
  int tm_hour; // 0-23 hours
  int tm_mday; // Day of month
  int tm_mon; // 0-11 months
  int tm_year; // Calendar year
  int tm_wday; // Sunday == 0, etc.
  int tm_yday; // 0-365 day of year
  int tm_isdst; // Daylight savings?
};
```

To convert from the time in seconds to the local time in the **tm** format, you use the Standard C library **localtime( )** function, which takes the number of seconds and returns a pointer to the resulting **tm**. This **tm**, however, is a **static** structure inside the **localtime( )** function, which is rewritten every time **localtime( )** is called. To copy the contents into the **tm struct** inside **DataPoint**, you might think you must copy each element individually. However, all you must do is a structure assignment, and the compiler will take care of the rest. This means the right-hand side must be a structure, not a pointer, so the result of **localtime( )** is dereferenced. The desired result is achieved with

```
d.setTime(*localtime(&timer));
```

After this, the **timer** is incremented by 55 seconds to give an interesting interval between readings.

The latitude and longitude used are fixed values to indicate a set of readings at a single location. Both the depth and the temperature are generated with the Standard C library **rand( )** function, which returns a pseudorandom number between zero and the constant RAND_MAX. To put this in a desired range, use the modulus operator **%** and the upper end of the range. These numbers are integral; to add a fractional part, a second call to **rand( )** is made, and the value is inverted after adding one (to prevent divide-by-zero errors).

In effect, the DATA.BIN file is being used as a container for the data in the program, even though the container exists on disk and not in RAM. To send the data out to the disk in binary form, **write( )** is used. The first argument is the starting address of the source block – notice it must be cast to an **unsigned char*** because that's what the function expects. The second argument is the number of bytes

to write, which is the size of the **DataPoint** object. Because no pointers are contained in **DataPoint**, there is no problem in writing the object to disk. If the object is more sophisticated, you must implement a scheme for *serialization* . (Most vendor class libraries have some sort of serialization structure built into them.)

## Verifying & viewing the data

To check the validity of the data stored in binary format, it is read from the disk and put in text form in DATA2.TXT, so that file can be compared to DATA.TXT for verification. In the following program, you can see how simple this data recovery is. After the test file is created, the records are read at the command of the user.

```
//: C02:Datascan.cpp
//{L} Datalog
// Verify and view logged data
#include "DataLogger.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <strstream>
#include <iomanip>
using namespace std;

int main() {
  ifstream bindata("data.bin", ios::binary);
  assure(bindata, "data.bin");
  // Create comparison file to verify data.txt:
  ofstream verify("data2.txt");
  assure(verify, "data2.txt");
  DataPoint d;
  while(bindata.read(
    (unsigned char*)&d, sizeof d))
    d.print(verify);
  bindata.clear(); // Reset state to "good"
  // Display user-selected records:
  int recnum = 0;
  // Left-align everything:
  cout.setf(ios::left, ios::adjustfield);
  // Fixed precision of 4 decimal places:
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(4);
  for(;;) {
    bindata.seekg(recnum* sizeof d, ios::beg);
    cout << "record " << recnum << endl;
    if(bindata.read(
      (unsigned char*)&d, sizeof d)) {
      cout << asctime(&(d.getTime()));
      cout << setw(11) << "Latitude"
           << setw(11) << "Longitude"
           << setw(10) << "Depth"
           << setw(12) << "Temperature"
           << endl;
      // Put a line after the description:
      cout << setfill('-') << setw(43) << '-'
           << setfill(' ') << endl;
      cout << setw(11) << d.getLatitude()
           << setw(11) << d.getLongitude()
```

```
            << setw(10) << d.getDepth()
            << setw(12) << d.getTemperature()
            << endl;
      } else {
        cout << "invalid record number" << endl;
        bindata.clear(); // Reset state to "good"
      }
      cout << endl
        << "enter record number, x to quit:";
      char buf[10];
      cin.getline(buf, 10);
      if(buf[0] == 'x') break;
      istrstream input(buf, 10);
      input >> recnum;
    }
  } ///:~
```

The **ifstream bindata** is created from DATA.BIN as a binary file, with the **ios::nocreate** flag on to cause the **assert( )** to fail if the file doesn't exist. The **read( )** statement reads a single record and places it directly into the **DataPoint d**. (Again, if **DataPoint** contained pointers this would result in meaningless pointer values.) This **read( )** action will set **bindata**'s **failbit** when the end of the file is reached, which will cause the **while** statement to fail. At this point, however, you can't move the get pointer back and read more records because the state of the stream won't allow further reads. So the **clear( )** function is called to reset the **failbit**.

Once the record is read in from disk, you can do anything you want with it, such as perform calculations or make graphs. Here, it is displayed to further exercise your knowledge of iostream formatting.

The rest of the program displays a record number (represented by **recnum**) selected by the user. As before, the precision is fixed at four decimal places, but this time everything is left justified.

The formatting of this output looks different from before:

```
record 0
Tue Nov 16 18:15:49 1993
Latitude   Longitude  Depth     Temperature
-----------------------------------------
45*20'31"  22*34'18"  186.0172  269.0167
```

To make sure the labels and the data columns line up, the labels are put in the same width fields as the columns, using **setw( )**. The line in between is generated by setting the fill character to '-', the width to the desired line width, and outputting a single '-'.

If the **read( )** fails, you'll end up in the **else** part, which tells the user the record number was invalid. Then, because the **failbit** was set, it must be reset with a call to **clear( )** so the next **read( )** is successful (assuming it's in the right range).

Of course, you can also open the binary data file for writing as well as reading. This way you can retrieve the records, modify them, and write them back to the same location, thus creating a flat-file database management system. In my very first programming job, I also had to create a flat-file DBMS – but using BASIC on an Apple II. It took months, while this took minutes. Of course, it might make

more sense to use a packaged DBMS now, but with C++ and iostreams you can still do all the low-level operations that are necessary in a lab.

## Counting editor

Often you have some editing task where you must go through and sequentially number something, but all the other text is duplicated. I encountered this problem when pasting digital photos into a Web page – I got the formatting just right, then duplicated it, then had the problem of incrementing the photo number for each one. So I replaced the photo number with XXX, dup licated that, and wrote the following program to find and replace the "XXX" with an incremented count. Notice the formatting, so the value will be "001," "002," etc.:

```
//: C02:NumberPhotos.cpp
// Find the marker "XXX" and replace it with an
// incrementing number whereever it appears. Used
// to help format a web page with photos in it
#include "../require.h"
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 2);
  ifstream in(argv[1]);
  assure(in, argv[1]);
  ofstream out(argv[2]);
  assure(out, argv[2]);
  string line;
  int counter = 1;
  while(getline(in, line)) {
    int xxx = line.find("XXX");
    if(xxx != string::npos) {
      ostringstream cntr;
      cntr << setfill('0') << setw(3) << counter++;
      line.replace(xxx, 3, cntr.str());
    }
    out << line << endl;
  }
} ///:~
```

## Breaking up big files

This program was created to break up big files into smaller ones, in particular so they could be more easily downloaded from an Internet server (since hangups sometimes occur, this allows someone to download a file a piece at a time and then re-assemble it at the client end). You'll note that the program also creates a reassembly batch file for DOS (where it is messier), whereas under Linux/Unix you simply say something like "**cat *piece* > destination.file**".

This program reads the entire file into memory, which of course relies on having a 32-bit operating system with virtual memory for big files. It then pieces it out in chunks to the smaller files, generating the names as it goes. Of course, you can come up with a possibly more reasonable strategy that reads a chunk, creates a file, reads another chunk, etc.

Note that this program can be run on the server, so you only have to download the big file once and then break it up once it's on the server.

```
//: C02:Breakup.cpp
// Breaks a file up into smaller files for
// easier downloads
#include "../require.h"
#include <iostream>
#include <fstream>
#include <iomanip>
#include <strstream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1);
  ifstream in(argv[1], ios::binary);
  assure(in, argv[1]);
  in.seekg(0, ios::end); // End of file
  long fileSize = in.tellg(); // Size of file
  cout << "file size = " << fileSize << endl;
  in.seekg(0, ios::beg); // Start of file
  char* fbuf = new char[fileSize];
  require(fbuf != 0);
  in.read(fbuf, fileSize);
  in.close();
  string infile(argv[1]);
  int dot = infile.find('.');
  while(dot != string::npos) {
    infile.replace(dot, 1, "-");
    dot = infile.find('.');
  }
  string batchName(
    "DOSAssemble" + infile + ".bat");
  ofstream batchFile(batchName.c_str());
  batchFile << "copy /b ";
  int filecount = 0;
  const int sbufsz = 128;
  char sbuf[sbufsz];
  const long pieceSize = 1000L * 100L;
  long byteCounter = 0;
  while(byteCounter < fileSize) {
    ostrstream name(sbuf, sbufsz);
    name << argv[1] << "-part" << setfill('0')
      << setw(2) << filecount++ << ends;
    cout << "creating " << sbuf << endl;
    if(filecount > 1)
      batchFile << "+";
    batchFile << sbuf;
    ofstream out(sbuf, ios::out | ios::binary);
    assure(out, sbuf);
    long byteq;
    if(byteCounter + pieceSize < fileSize)
```

```
      byteq = pieceSize;
    else
      byteq = fileSize - byteCounter;
    out.write(fbuf + byteCounter, byteq);
    cout << "wrote " << byteq << " bytes, ";
    byteCounter += byteq;
    out.close();
    cout << "ByteCounter = " << byteCounter
      << ", fileSize = " << fileSize << endl;
  }
  batchFile << " " << argv[1] << endl;
} ///:~
```

# Summary

This chapter has given you a fairly thorough introduction to the iostream class library. In all likelihood, it is all you need to create programs using iostreams. (In later chapters you'll see simple examples of adding iostream functionality to your own classes.) However, you should be aware that there are some additional features in iostreams that are not used often, but which you can discover by looking at the iostream header files and by reading your compiler's documentation on iostreams.

# Exercises

1. Open a file by creating an **ifstream** object called **in**. Make an **ostrstream** object called **os**, and read the entire contents into the **ostrstream** using the **rdbuf( )** member function. Get the address of **os**'s **char*** with the **str( )** function, and capitalize every character in the file using the Standard C **toupper( )** macro. Write the result out to a new file, and **delete** the memory allocated by **os**.
2. Create a program that opens a file (the first argument on the command line) and searches it for any one of a set of words (the remaining arguments on the command line). Read the input a line at a time, and print out the lines (with line numbers) that match.
3. Write a program that adds a copyright notice to the beginning of all source-code files. This is a small modification to exercise 1.
4. Use your favorite text-searching program (**grep**, for example) to output the names (only) of all the files that contain a particular pattern. Redirect the output into a file. Write a program that uses the contents of that file to generate a batch file that invokes your editor on each of the files found by the search program.

---

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

[7] Newer implementations of iostreams will still support this style of handling errors, but in some cases will also throw exceptions.

[8] Note the name has been truncated to handle the DOS limitation on file names. You may need to adjust the header file name if your system supports longer file names (or simply copy the header file).

[9] These only appear in the revised library; you won't find them in older implementations of iostreams.

[10] Before putting **nl** into a header file, you should make it an **inline** function (see Chapter 7).

[11] In a private conversation.