PART II

THE C++ LIBRARY

CONTENTS

With each revision of the C++ language, the library has also grown. Indeed, more than two-thirds of the text of the new standard is devoted to the library. Although we cannot cover every library facility in depth, there are core facilities that the library defines that every C++ programmer should be comfortable using. We cover these core facilities in this part.

We'll start by covering the basic IO library facilities in Chapter 8. Beyond using the library to read and write streams associated with the console window, the library defines types that let us read and write named files and do in-memory IO to `strings`.

Central to the library are a number of container classes and a family of generic algorithms that let us write programs that are succinct and efficient. The library worries about bookkeeping details—in particular, taking care of memory management—so that our programs can worry about the actual problems we need to solve.

In Chapter 3 we introduced the `vector` container type. We'll learn more about `vector` in Chapter 9, which will cover the other sequential container types as well. We'll also cover more operations provided by the `string` type. We can think of a `string` as a special kind of container that contains only characters. The `string` type

supports many, but not all, of the container operations.

Chapter 10 introduces the generic algorithms. The algorithms typically operate on a range of elements in a sequential container or other sequence. The algorithms library offers efficient implementations of various classical algorithms, such as sorting and searching, and other common tasks as well. For example, there is a `copy` algorithm, which copies elements from one sequence to another; `find`, which looks for a given element; and so on. The algorithms are generic in two ways: They can be applied to different kinds of sequences, and those sequences may contain elements of most types.

The library also provides several associative containers, which are the topic of Chapter 11. Elements in an associative container are accessed by key. The associative containers share many operations with the sequential containers and also define operations that are specific to the associative containers.

This part concludes with Chapter 12, which looks at language and library facilities for managing dynamic memory. This chapter covers one of the most important new library classes, which are standardized versions of smart pointers. By using smart pointers, we can make code that uses dynamic memory much more robust. This chapter closes with an extended example that uses library facilities introduced throughout Part II.

CHAPTER 8

THE IO LIBRARY

CONTENTS

The C++ language does not deal directly with input and output. Instead, IO is handled by a family of types defined in the standard library. These types support IO to and from devices such as files and console windows. Additional types allow in-memory IO to and from `strings`.

The IO library defines operations to read and write values of the built-in types. In addition, classes, such as `string`, typically define similar IO operations to work on objects of their class type as well.

This chapter introduces the fundamentals of the IO library. Later chapters will cover additional capabilities: Chapter 14 will look at how we can write our own input and output operators, and Chapter 17 will cover how to control formatting and how to perform random access on files.

Our programs have already used many IO library facilities. Indeed, we introduced most of these facilities in § 1.2 (p. 5):

- `istream` (input stream) type, which provides input operations
- `ostream` (output stream) type, which provides output operations
- `cin`, an `istream` object that reads the standard input
- `cout`, an `ostream` object that writes to the standard output
- `cerr`, an `ostream` object, typically used for program error messages, that writes to the standard error
- The `>>` operator, which is used to read input from an `istream` object
- The `<<` operator, which is used to write output to an `ostream` object
- The `getline` function (§ 3.2.2, p. 87), which reads a line of input from a given `istream` into a given `string`

8.1 The IO Classes

The IO types and objects that we've used so far manipulate `char` data. By default these objects are connected to the user's console window. Of course, real programs cannot be limited to doing IO solely to or from a console window. Programs often need to read or write named files. Moreover, it can be convenient to use IO operations to process the characters in a `string`. Applications also may have to read and write languages that require wide-character support.

To support these different kinds of IO processing, the library defines a collection of IO types in addition to the `istream` and `ostream` types that we have already used. These types, which are listed in Table 8.1, are defined in three separate headers: `iostream` defines the basic types used to read from and write to a stream, `fstream` defines the types used to read and write named files, and `sstream` defines the types used to read and write in-memory `string`s.

Table 8.1: IO Library Types and Headers

Header	Type
`iostream`	`istream`, `wistream` reads from a stream
	`ostream`, `wostream` writes to a stream
	`iostream`, `wiostream` reads and writes a stream
`fstream`	`ifstream`, `wifstream` reads from a file
	`ofstream`, `wofstream` writes to a file
	`fstream`, `wfstream` reads and writes a file
`sstream`	`istringstream`, `wistringstream` reads from a `string`
	`ostringstream`, `wostringstream` writes to a `string`
	`stringstream`, `wstringstream` reads and writes a `string`

To support languages that use wide characters, the library defines a set of types and objects that manipulate `wchar_t` data (§ 2.1.1, p. 32). The names of the wide-character versions begin with a `w`. For example, `wcin`, `wcout`, and `wcerr` are the wide-character objects that correspond to `cin`, `cout`, and `cerr`, respectively. The wide-character types and objects are defined in the same header as the plain `char` types. For example, the `fstream` header defines both the `ifstream` and `wifstream` types.

Relationships among the IO Types

Conceptually, neither the kind of device nor the character size affects the IO operations we want to perform. For example, we'd like to use `>>` to read data regardless of whether we're reading a console window, a disk file, or a `string`. Similarly, we'd like to use that operator regardless of whether the characters we read fit in a `char` or require a `wchar_t`.

The library lets us ignore the differences among these different kinds of streams by using **inheritance**. As with templates (§ 3.3, p. 96), we can use classes related by inheritance without understanding the details of how inheritance works. We'll cover how C++ supports inheritance in Chapter 15 and in § 18.3 (p. 802).

Briefly, inheritance lets us say that a particular class inherits from another class. Ordinarily, we can use an object of an inherited class as if it were an object of the same type as the class from which it inherits.

The types `ifstream` and `istringstream` inherit from `istream`. Thus, we can use objects of type `ifstream` or `istringstream` as if they were `istream` objects. We can use objects of these types in the same ways as we have used `cin`. For example, we can call `getline` on an `ifstream` or `istringstream` object, and we can use the `>>` to read data from an `ifstream` or `istringstream`. Similarly, the types `ofstream` and `ostringstream` inherit from `ostream`. Therefore, we can use objects of these types in the same ways that we have used `cout`.

Everything that we cover in the remainder of this section applies equally to plain streams, file streams, and `string` streams and to the `char` or wide-character stream versions.

8.1.1 No Copy or Assign for IO Objects

As we saw in § 7.1.3 (p. 261), we cannot copy or assign objects of the IO types:

```
ofstream out1, out2;
out1 = out2;               // error: cannot assign stream objects
ofstream print(ofstream); // error: can't initialize the ofstream parameter
out2 = print(out2);        // error: cannot copy stream objects
```

Because we can't copy the IO types, we cannot have a parameter or return type that is one of the stream types (§ 6.2.1, p. 209). Functions that do IO typically pass and return the stream through references. Reading or writing an IO object changes its state, so the reference must not be `const`.

8.1.2 Condition States

Inherent in doing IO is the fact that errors can occur. Some errors are recoverable; others occur deep within the system and are beyond the scope of a program to correct. The IO classes define functions and flags, listed in Table 8.2, that let us access and manipulate the **condition state** of a stream.

As an example of an IO error, consider the following code:

```
int ival;
cin >> ival;
```

If we enter `Boo` on the standard input, the read will fail. The input operator expected to read an `int` but got the character `B` instead. As a result, `cin` will be put in an error state. Similarly, `cin` will be in an error state if we enter an end-of-file.

Once an error has occurred, subsequent IO operations on that stream will fail. We can read from or write to a stream only when it is in a non-error state. Because a stream might be in an error state, code ordinarily should check whether a stream is okay before attempting to use it. The easiest way to determine the state of a stream object is to use that object as a condition:

```
while (cin >> word)
    // ok: read operation successful . . .
```

The `while` condition checks the state of the stream returned from the `>>` expression. If that input operation succeeds, the state remains valid and the condition will succeed.

Interrogating the State of a Stream

Using a stream as a condition tells us only whether the stream is valid. It does not tell us what happened. Sometimes we also need to know why the stream is invalid. For example, what we do after hitting end-of-file is likely to differ from what we'd do if we encounter an error on the IO device.

The IO library defines a machine-dependent integral type named `iostate` that it uses to convey information about the state of a stream. This type is used as a collection of bits, in the same way that we used the `quiz1` variable in § 4.8 (p. 154). The IO classes define four `constexpr` values (§ 2.4.4, p. 65) of type `iostate` that represent particular bit patterns. These values are used to indicate particular kinds of IO conditions. They can be used with the bitwise operators (§ 4.8, p. 152) to test or set multiple flags in one operation.

The `badbit` indicates a system-level failure, such as an unrecoverable read or write error. It is usually not possible to use a stream once `badbit` has been set. The `failbit` is set after a recoverable error, such as reading a character when numeric data was expected. It is often possible to correct such problems and continue using the stream. Reaching end-of-file sets both `eofbit` and `failbit`. The `goodbit`, which is guaranteed to have the value 0, indicates no failures on the stream. If any of `badbit`, `failbit`, or `eofbit` are set, then a condition that evaluates that stream will fail.

The library also defines a set of functions to interrogate the state of these flags. The `good` operation returns `true` if none of the error bits is set. The `bad`, `fail`,

Table 8.2: IO Library Condition State

strm`::iostate`	*strm* is one of the IO types listed in Table 8.1 (p. 310). `iostate` is a machine-dependent integral type that represents the condition state of a stream.
strm`::badbit`	*strm*`::iostate` value used to indicate that a stream is corrupted.
strm`::failbit`	*strm*`::iostate` value used to indicate that an IO operation failed.
strm`::eofbit`	*strm*`::iostate` value used to indicate that a stream hit end-of-file.
strm`::goodbit`	*strm*`::iostate` value used to indicate that a stream is not in an error state. This value is guaranteed to be zero.
`s.eof()`	`true` if `eofbit` in the stream `s` is set.
`s.fail()`	`true` if `failbit` or `badbit` in the stream `s` is set.
`s.bad()`	`true` if `badbit` in the stream `s` is set.
`s.good()`	`true` if the stream `s` is in a valid state.
`s.clear()`	Reset all condition values in the stream `s` to valid state. Returns `void`.
`s.clear(flags)`	Reset the condition of `s` to `flags`. Type of `flags` is *strm*`::iostate`. Returns `void`.
`s.setstate(flags)`	Adds specified condition(s) to `s`. Type of `flags` is *strm*`::iostate`. Returns `void`.
`s.rdstate()`	Returns current condition of `s` as a *strm*`::iostate` value.

and `eof` operations return `true` when the corresponding bit is on. In addition, `fail` returns `true` if `bad` is set. By implication, the right way to determine the overall state of a stream is to use either `good` or `fail`. Indeed, the code that is executed when we use a stream as a condition is equivalent to calling `!fail()`. The `eof` and `bad` operations reveal only whether those specific errors have occurred.

Managing the Condition State

The `rdstate` member returns an `iostate` value that corresponds to the current state of the stream. The `setstate` operation turns on the given condition bit(s) to indicate that a problem occurred. The `clear` member is overloaded (§ 6.4, p. 230): One version takes no arguments and a second version takes a single argument of type `iostate`.

The version of `clear` that takes no arguments turns off all the failure bits. After `clear()`, a call to `good` returns `true`. We might use these members as follows:

```
// remember the current state of cin
auto old_state = cin.rdstate();  // remember the current state of cin
cin.clear();                     // make cin valid
process_input(cin);              // use cin
cin.setstate(old_state);         // now reset cin to its old state
```

The version of `clear` that takes an argument expects an `iostate` value that represents the new state of the stream. To turn off a single condition, we use the `rdstate` member and the bitwise operators to produce the desired new state.

For example, the following turns off `failbit` and `badbit` but leaves `eofbit` untouched:

```
// turns off failbit and badbit but all other bits unchanged
cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);
```

EXERCISES SECTION 8.1.2

Exercise 8.1: Write a function that takes and returns an `istream&`. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid before returning the stream.

Exercise 8.2: Test your function by calling it, passing `cin` as an argument.

Exercise 8.3: What causes the following `while` to terminate?

```
while (cin >> i) /* ... */
```

8.1.3 Managing the Output Buffer

Each output stream manages a buffer, which it uses to hold the data that the program reads and writes. For example, when the following code is executed

```
os << "please enter a value: ";
```

the literal string might be printed immediately, or the operating system might store the data in a buffer to be printed later. Using a buffer allows the operating system to combine several output operations from our program into a single system-level write. Because writing to a device can be time-consuming, letting the operating system combine several output operations into a single write can provide an important performance boost.

There are several conditions that cause the buffer to be flushed—that is, to be written—to the actual output device or file:

- The program completes normally. All output buffers are flushed as part of the `return` from `main`.
- At some indeterminate time, the buffer can become full, in which case it will be flushed before writing the next value.
- We can flush the buffer explicitly using a manipulator such as `endl` (§ 1.2, p. 7).
- We can use the `unitbuf` manipulator to set the stream's internal state to empty the buffer after each output operation. By default, `unitbuf` is set for `cerr`, so that writes to `cerr` are flushed immediately.
- An output stream might be tied to another stream. In this case, the output stream is flushed whenever the stream to which it is tied is read or written. By default, `cin` and `cerr` are both tied to `cout`. Hence, reading `cin` or writing to `cerr` flushes the buffer in `cout`.

Flushing the Output Buffer

Our programs have already used the `endl` manipulator, which ends the current line and flushes the buffer. There are two other similar manipulators: `flush` and `ends`. `flush` flushes the stream but adds no characters to the output; `ends` inserts a null character into the buffer and then flushes it:

```
cout << "hi!" << endl;   // writes hi and a newline, then flushes the buffer
cout << "hi!" << flush;  // writes hi, then flushes the buffer; adds no data
cout << "hi!" << ends;   // writes hi and a null, then flushes the buffer
```

The `unitbuf` Manipulator

If we want to flush after every output, we can use the `unitbuf` manipulator. This manipulator tells the stream to do a `flush` after every subsequent write. The `nounitbuf` manipulator restores the stream to use normal, system-managed buffer flushing:

```
cout << unitbuf;          // all writes will be flushed immediately
// any output is flushed immediately, no buffering
cout << nounitbuf;        // returns to normal buffering
```

CAUTION: BUFFERS ARE NOT FLUSHED IF THE PROGRAM CRASHES

Output buffers are *not* flushed if the program terminates abnormally. When a program crashes, it is likely that data the program wrote may be sitting in an output buffer waiting to be printed.

When you debug a program that has crashed, it is essential to make sure that any output you *think* should have been written was actually flushed. Countless hours of programmer time have been wasted tracking through code that appeared not to have executed when in fact the buffer had not been flushed and the output was pending when the program crashed.

Tying Input and Output Streams Together

When an input stream is tied to an output stream, any attempt to read the input stream will first flush the buffer associated with the output stream. The library ties `cout` to `cin`, so the statement

```
cin >> ival;
```

causes the buffer associated with `cout` to be flushed.

Interactive systems usually should tie their input stream to their output stream. Doing so means that all output, which might include prompts to the user, will be written before attempting to read the input.

There are two overloaded (§ 6.4, p. 230) versions of `tie`: One version takes no argument and returns a pointer to the output stream, if any, to which this object is currently tied. The function returns the null pointer if the stream is not tied.

The second version of `tie` takes a pointer to an `ostream` and ties itself to that `ostream`. That is, `x.tie(&o)` ties the stream `x` to the output stream `o`.

We can tie either an `istream` or an `ostream` object to another `ostream`:

```
cin.tie(&cout);    // illustration only: the library ties cin and cout for us
// old_tie points to the stream (if any) currently tied to cin
ostream *old_tie = cin.tie(nullptr); // cin is no longer tied
// ties cin and cerr; not a good idea because cin should be tied to cout
cin.tie(&cerr);    // reading cin flushes cerr, not cout
cin.tie(old_tie); // reestablish normal tie between cin and cout
```

To tie a given stream to a new output stream, we pass `tie` a pointer to the new stream. To untie the stream completely, we pass a null pointer. Each stream can be tied to at most one stream at a time. However, multiple streams can tie themselves to the same `ostream`.

8.2 File Input and Output

The `fstream` header defines three types to support file IO: **`ifstream`** to read from a given file, **`ofstream`** to write to a given file, and **`fstream`**, which reads and writes a given file. In § 17.5.3 (p. 763) we'll describe how to use the same file for both input and output.

These types provide the same operations as those we have previously used on the objects `cin` and `cout`. In particular, we can use the IO operators (`<<` and `>>`) to read and write files, we can use `getline` (§ 3.2.2, p. 87) to read an `ifstream`, and the material covered in § 8.1 (p. 310) applies to these types.

In addition to the behavior that they inherit from the `iostream` types, the types defined in `fstream` add members to manage the file associated with the stream. These operations, listed in Table 8.3, can be called on objects of `fstream`, `ifstream`, or `ofstream` but not on the other IO types.

Table 8.3: `fstream`-Specific Operations

fstream `fstrm;`	Creates an unbound file stream. *fstream* is one of the types defined in the `fstream` header.
fstream `fstrm(s);`	Creates an *fstream* and opens the file named `s`. `s` can have type `string` or can be a pointer to a C-style character string (§ 3.5.4, p. 122). These constructors are `explicit` (§ 7.5.4, p. 296). The default file `mode` depends on the type of *fstream*.
fstream `fstrm(s, mode);`	Like the previous constructor, but opens `s` in the given `mode`.
`fstrm.open(s)` `fstrm.open(s, mode)`	Opens the file named by the `s` and binds that file to `fstrm`. `s` can be a `string` or a pointer to a C-style character string. The default file `mode` depends on the type of *fstream*. Returns `void`.
`fstrm.close()`	Closes the file to which `fstrm` is bound. Returns `void`.
`fstrm.is_open()`	Returns a `bool` indicating whether the file associated with `fstrm` was successfully opened and has not been closed.

8.2.1 Using File Stream Objects

When we want to read or write a file, we define a file stream object and associate that object with the file. Each file stream class defines a member function named `open` that does whatever system-specific operations are required to locate the given file and open it for reading or writing as appropriate.

When we create a file stream, we can (optionally) provide a file name. When we supply a file name, `open` is called automatically:

```
ifstream in(ifile); // construct an ifstream and open the given file
ofstream out;       // output file stream that is not associated with any file
```

This code defines `in` as an input stream that is initialized to read from the file named by the `string` argument `ifile`. It defines `out` as an output stream that is not yet associated with a file. With the new standard, file names can be either library `string`s or C-style character arrays (§ 3.5.4, p. 122). Previous versions of the library allowed only C-style character arrays.

C++ 11

Using an `fstream` in Place of an `iostream&`

As we noted in § 8.1 (p. 311), we can use an object of an inherited type in places where an object of the original type is expected. This fact means that functions that are written to take a reference (or pointer) to one of the `iostream` types can be called on behalf of the corresponding `fstream` (or `sstream`) type. That is, if we have a function that takes an `ostream&`, we can call that function passing it an `ofstream` object, and similarly for `istream&` and `ifstream`.

For example, we can use the `read` and `print` functions from § 7.1.3 (p. 261) to read from and write to named files. In this example, we'll assume that the names of the input and output files are passed as arguments to `main` (§ 6.2.5, p. 218):

```
ifstream input(argv[1]);    // open the file of sales transactions
ofstream output(argv[2]);   // open the output file
Sales_data total;           // variable to hold the running sum
if (read(input, total)) {   // read the first transaction
    Sales_data trans;       // variable to hold data for the next transaction
    while(read(input, trans)) {     // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check isbns
            total.combine(trans);  // update the running total
        else {
            print(output, total) << endl; // print the results
            total = trans;          // process the next book
        }
    }
    print(output, total) << endl;   // print the last transaction
} else                              // there was no input
    cerr << "No data?!" << endl;
```

Aside from using named files, this code is nearly identical to the version of the addition program on page 255. The important part is the calls to `read` and to `print`. We can pass our `fstream` objects to these functions even though the parameters to those functions are defined as `istream&` and `ostream&`, respectively.

The `open` and `close` Members

When we define an empty file stream object, we can subsequently associate that object with a file by calling `open`:

```
ifstream in(ifile); //  construct an ifstream and open the given file
ofstream out;       //  output file stream that is not associated with any file
out.open(ifile + ".copy");  //  open the specified file
```

If a call to `open` fails, `failbit` is set (§ 8.1.2, p. 312). Because a call to `open` might fail, it is usually a good idea to verify that the `open` succeeded:

```
if (out)     //  check that the open succeeded
     //  the open succeeded, so we can use the file
```

This condition is similar to those we've used on `cin`. If the `open` fails, this condition will fail and we will not attempt to use `out`.

Once a file stream has been opened, it remains associated with the specified file. Indeed, calling `open` on a file stream that is already open will fail and set `failbit`. Subsequent attempts to use that file stream will fail. To associate a file stream with a different file, we must first close the existing file. Once the file is closed, we can open a new one:

```
in.close();                 //  close the file
in.open(ifile + "2");       //  open another file
```

If the `open` succeeds, then `open` sets the stream's state so that `good()` is `true`.

Automatic Construction and Destruction

Consider a program whose `main` function takes a list of files it should process (§ 6.2.5, p. 218). Such a program might have a loop like the following:

```
//  for each file passed to the program
for (auto p = argv + 1; p != argv + argc; ++p) {
    ifstream input(*p);    //  create input and open the file
    if (input) {           //  if the file is ok, "process" this file
        process(input);
    } else
        cerr << "couldn't open: " + string(*p);
} //  input goes out of scope and is destroyed on each iteration
```

Each iteration constructs a new `ifstream` object named `input` and opens it to read the given file. As usual, we check that the `open` succeeded. If so, we pass that file to a function that will read and process the input. If not, we print an error message and continue.

Because `input` is defined inside the block that forms the `for` body, it is created and destroyed on each iteration (§ 6.1.1, p. 205). When an `fstream` object goes out of scope, the file it is bound to is automatically closed. On the next iteration, `input` is created anew.

When an `fstream` object is destroyed, `close` is called automatically.

EXERCISES SECTION 8.2.1

Exercise 8.4: Write a function to open a file for input and read its contents into a `vector` of `strings`, storing each line as a separate element in the `vector`.

Exercise 8.5: Rewrite the previous program to store each word in a separate element.

Exercise 8.6: Rewrite the bookstore program from § 7.1.1 (p. 256) to read its transactions from a file. Pass the name of the file as an argument to `main` (§ 6.2.5, p. 218).

8.2.2 File Modes

Each stream has an associated **file mode** that represents how the file may be used. Table 8.4 lists the file modes and their meanings.

Table 8.4: File Modes

`in`	Open for input
`out`	Open for output
`app`	Seek to the end before every write
`ate`	Seek to the end immediately after the open
`trunc`	Truncate the file
`binary`	Do IO operations in binary mode

We can supply a file mode whenever we open a file—either when we call `open` or when we indirectly open the file when we initialize a stream from a file name. The modes that we can specify have the following restrictions:

- `out` may be set only for an `ofstream` or `fstream` object.
- `in` may be set only for an `ifstream` or `fstream` object.
- `trunc` may be set only when `out` is also specified.
- `app` mode may be specified so long as `trunc` is not. If `app` is specified, the file is always opened in output mode, even if `out` was not explicitly specified.
- By default, a file opened in `out` mode is truncated even if we do not specify `trunc`. To preserve the contents of a file opened with `out`, either we must also specify `app`, in which case we can write only at the end of the file, or we must also specify `in`, in which case the file is open for both input and output (§ 17.5.3 (p. 763) will cover using the same file for input and output).
- The `ate` and `binary` modes may be specified on any file stream object type and in combination with any other file modes.

Each file stream type defines a default file mode that is used whenever we do not otherwise specify a mode. Files associated with an `ifstream` are opened in `in` mode; files associated with an `ofstream` are opened in `out` mode; and files associated with an `fstream` are opened with both `in` and `out` modes.

Opening a File in `out` Mode Discards Existing Data

By default, when we open an `ofstream`, the contents of the file are discarded. The only way to prevent an `ostream` from emptying the given file is to specify `app`:

```
// file1 is truncated in each of these cases
ofstream out("file1");    // out and trunc are implicit
ofstream out2("file1", ofstream::out);    // trunc is implicit
ofstream out3("file1", ofstream::out | ofstream::trunc);
// to preserve the file's contents, we must explicitly specify app mode
ofstream app("file2", ofstream::app);    // out is implicit
ofstream app2("file2", ofstream::out | ofstream::app);
```

The only way to preserve the existing data in a file opened by an `ofstream` is to specify `app` or `in` mode explicitly.

File Mode Is Determined Each Time `open` Is Called

The file mode of a given stream may change each time a file is opened.

```
ofstream out;     // no file mode is set
out.open("scratchpad"); // mode implicitly out and trunc
out.close();      // close out so we can use it for a different file
out.open("precious", ofstream::app);   // mode is out and app
out.close();
```

The first call to `open` does not specify an output mode explicitly; this file is implicitly opened in `out` mode. As usual, `out` implies `trunc`. Therefore, the file named `scratchpad` in the current directory will be truncated. When we open the file named `precious`, we ask for append mode. Any data in the file remains, and all writes are done at the end of the file.

Any time `open` is called, the file mode is set, either explicitly or implicitly. Whenever a mode is not specified, the default value is used.

EXERCISES SECTION 8.2.2

Exercise 8.7: Revise the bookstore program from the previous section to write its output to a file. Pass the name of that file as a second argument to `main`.

Exercise 8.8: Revise the program from the previous exercise to append its output to its given file. Run the program on the same output file at least twice to ensure that the data are preserved.

8.3 string Streams

The sstream header defines three types to support in-memory IO; these types read from or write to a string as if the string were an IO stream.

The **istringstream** type reads a string, **ostringstream** writes a string, and **stringstream** reads and writes the string. Like the fstream types, the types defined in sstream inherit from the types we have used from the iostream header. In addition to the operations they inherit, the types defined in sstream add members to manage the string associated with the stream. These operations are listed in Table 8.5. They may be called on stringstream objects but not on the other IO types.

Note that although fstream and sstream share the interface to iostream, they have no other interrelationship. In particular, we cannot use open and close on a stringstream, nor can we use str on an fstream.

Table 8.5: stringstream-Specific Operations

sstream strm;	strm is an unbound stringstream. *sstream* is one of the types defined in the sstream header.
sstream strm(s);	strm is an *sstream* that holds a copy of the string s. This constructor is explicit (§ 7.5.4, p. 296).
strm.str()	Returns a copy of the string that strm holds.
strm.str(s)	Copies the string s into strm. Returns void.

8.3.1 Using an istringstream

An istringstream is often used when we have some work to do on an entire line, and other work to do with individual words within a line.

As one example, assume we have a file that lists people and their associated phone numbers. Some people have only one number, but others have several—a home phone, work phone, cell number, and so on. Our input file might look like the following:

```
morgan 2015552368 8625550123
drew 9735550130
lee 6095550132 2015550175 8005550000
```

Each record in this file starts with a name, which is followed by one or more phone numbers. We'll start by defining a simple class to represent our input data:

```
// members are public by default; see § 7.2 (p. 268)
struct PersonInfo {
    string name;
    vector<string> phones;
};
```

Objects of type PersonInfo will have one member that represents the person's name and a vector holding a varying number of associated phone numbers.

Our program will read the data file and build up a `vector` of `PersonInfo`. Each element in the `vector` will correspond to one record in the file. We'll process the input in a loop that reads a record and then extracts the name and phone numbers for each person:

```
string line, word;  // will hold a line and word from input, respectively
vector<PersonInfo> people;  // will hold all the records from the input
// read the input a line at a time until cin hits end-of-file (or another error)
while (getline(cin, line)) {
    PersonInfo info;       // create an object to hold this record's data
    istringstream record(line); // bind record to the line we just read
    record >> info.name;  // read the name
    while (record >> word)       // read the phone numbers
        info.phones.push_back(word);  // and store them
    people.push_back(info); // append this record to people
}
```

Here we use `getline` to read an entire record from the standard input. If the call to `getline` succeeds, then `line` holds a record from the input file. Inside the `while` we define a local `PersonInfo` object to hold data from the current record.

Next we bind an `istringstream` to the line that we just read. We can now use the input operator on that `istringstream` to read each element in the current record. We first read the name followed by a `while` loop that will read the phone numbers for that person.

The inner `while` ends when we've read all the data in `line`. This loop works analogously to others we've written to read `cin`. The difference is that this loop reads data from a `string` rather than from the standard input. When the `string` has been completely read, "end-of-file" is signaled and the next input operation on `record` will fail.

We end the outer `while` loop by appending the `PersonInfo` we just processed to the `vector`. The outer `while` continues until we hit end-of-file on `cin`.

EXERCISES SECTION 8.3.1

Exercise 8.9: Use the function you wrote for the first exercise in § 8.1.2 (p. 314) to print the contents of an `istringstream` object.

Exercise 8.10: Write a program to store each line from a file in a `vector<string>`. Now use an `istringstream` to read each element from the `vector` a word at a time.

Exercise 8.11: The program in this section defined its `istringstream` object inside the outer `while` loop. What changes would you need to make if `record` were defined outside that loop? Rewrite the program, moving the definition of `record` outside the `while`, and see whether you thought of all the changes that are needed.

Exercise 8.12: Why didn't we use in-class initializers in `PersonInfo`?

8.3.2 Using ostringstreams

An ostringstream is useful when we need to build up our output a little at a time but do not want to print the output until later. For example, we might want to validate and reformat the phone numbers we read in the previous example. If all the numbers are valid, we want to print a new file containing the reformatted numbers. If a person has any invalid numbers, we won't put them in the new file. Instead, we'll write an error message containing the person's name and a list of their invalid numbers.

Because we don't want to include any data for a person with an invalid number, we can't produce the output until we've seen and validated all their numbers. We can, however, "write" the output to an in-memory ostringstream:

```
for (const auto &entry : people) {     // for each entry in people
    ostringstream formatted, badNums; // objects created on each loop
    for (const auto &nums : entry.phones) {  // for each number
        if (!valid(nums)) {
            badNums << " " << nums;  // string in badNums
        } else
            // "writes" to formatted's string
            formatted << " " << format(nums);
    }
    if (badNums.str().empty())       // there were no bad numbers
        os << entry.name << " "      // print the name
           << formatted.str() << endl; // and reformatted numbers
    else                    // otherwise, print the name and bad numbers
        cerr << "input error: " << entry.name
             << " invalid number(s) " << badNums.str() << endl;
}
```

In this program, we've assumed two functions, valid and format, that validate and reformat phone numbers, respectively. The interesting part of the program is the use of the string streams formatted and badNums. We use the normal output operator (<<) to write to these objects. But, these "writes" are really string manipulations. They add characters to the strings inside formatted and badNums, respectively.

EXERCISES SECTION 8.3.2

Exercise 8.13: Rewrite the phone number program from this section to read from a named file rather than from cin.

Exercise 8.14: Why did we declare entry and nums as const auto &?

Chapter Summary

C++ uses library classes to handle stream-oriented input and output:

- The `iostream` classes handle IO to console
- The `fstream` classes handle IO to named files
- The `stringstream` classes do IO to in-memory `strings`

The `fstream` and `stringstream` classes are related by inheritance to the `iostream` classes. The input classes inherit from `istream` and the output classes from `ostream`. Thus, operations that can be performed on an `istream` object can also be performed on either an `ifstream` or an `istringstream`. Similarly for the output classes, which inherit from `ostream`.

Each IO object maintains a set of condition states that indicate whether IO can be done through this object. If an error is encountered—such as hitting end-of-file on an input stream—then the object's state will be such that no further input can be done until the error is rectified. The library provides a set of functions to set and test these states.

Defined Terms

condition state Flags and associated functions usable by any of the stream classes that indicate whether a given stream is usable.

file mode Flags defined by the `fstream` classes that are specified when opening a file and control how a file can be used.

file stream Stream object that reads or writes a named file. In addition to the normal `iostream` operations, file streams also define `open` and `close` members. The `open` member takes a `string` or a C-style character string that names the file to open and an optional open mode argument. The `close` member closes the file to which the stream is attached. It must be called before another file can be `open`ed.

fstream File stream that reads and writes to the same file. By default `fstreams` are opened with `in` and `out` mode set.

ifstream File stream that reads an input file. By default `ifstreams` are opened with `in` mode set.

inheritance Programming feature that lets a type inherit the interface of another type. The `ifstream` and `istringstream` classes inherit from `istream` and the `ofstream` and `ostringstream` classes inherit from `ostream`. Chapter 15 covers inheritance.

istringstream String stream that reads a given `string`.

ofstream File stream that writes to an output file. By default, `ofstreams` are opened with `out` mode set.

ostringstream String stream that writes to a given `string`.

string stream Stream object that reads or writes a `string`. In addition to the normal `iostream` operations, string streams define an overloaded member named `str`. Calling `str` with no arguments returns the `string` to which the string stream is attached. Calling it with a `string` attaches the string stream to a copy of that `string`.

stringstream String stream that reads and writes to a given `string`.