11 Customizing Input and Output

"Keep it simple:
as simple as possible,
but no simpler."

—Albert Einstein

In this chapter, we concentrate on how to adapt the general iostream framework presented in Chapter 10 to specific needs and tastes. This involves a lot of messy details dictated by human sensibilities to what they read and also practical constraints on the uses of files. The final example shows the design of an input stream for which you can specify the set of separators.

11.1 Regularity and irregularity

The iostream library – the input/output part of the ISO C++ standard library – provides a unified and extensible framework for input and output of text. By "text" we mean just about anything that can be represented as a sequence of characters. Thus, when we talk about input and output we can consider the integer **1234** as text because we can write it using the four characters **1**, **2**, **3**, and **4**.

So far, we have treated all input sources as equivalent. Sometimes, that's not enough. For example, files differ from other input sources (such as communications connections) in that we can address individual bytes. Similarly, we worked on the assumption that the type of an object completely determined the layout of its input and output. That's not quite right and wouldn't be sufficient. For example, we often want to specify the number of digits used to represent a floating-point number on output (its precision). This chapter presents a number of ways in which we can tailor input and output to our needs.

As programmers, we prefer regularity; treating all in-memory objects uniformly, treating all input sources equivalently, and imposing a single standard on the way to represent objects entering and exiting the system give the cleanest, simplest, most maintainable, and often the most efficient code. However, our programs exist to serve humans, and humans have strong preferences. Thus, as programmers we must strive for a balance between program complexity and accommodation of users' personal tastes.

11.2 Output formatting

People care a lot about apparently minor details of the output they have to read. For example, to a physicist **1.25** (rounded to two digits after the dot) can be very different from **1.24670477**, and to an accountant **(1.25)** can be legally different from **(1.2467)** and totally different from **1.25** (in financial documents, parentheses are sometimes used to indicate losses, that is, negative values). As programmers, we aim at making our output as clear and as close as possible to the expectations of the "consumers" of our program. Output streams (**ostreams**) provide a variety of ways for formatting the output of built-in types. For user-defined types, it is up to the programmer to define suitable **<<** operations.

There seem to be an infinite number of details, refinements, and options for output and quite a few for input. Examples are the character used for the decimal point (usually dot or comma), the way to output monetary values, a way to represent true as the word **true** (or **vrai** or **sandt**) rather than the number **1** when output, ways to deal with non-ASCII character sets (such as Unicode), and a way to limit the number of characters read into a string. These facilities tend to be uninteresting until you need them, so we'll leave their description to manuals and specialized works such as Langer, *Standard C++ IOStreams and Locales*; Chapters 38 and 39 of *The C++ Programming Language* by Stroustrup; and §22 and §27 of the ISO C++ standard. Here we'll present the most frequently useful features and a few general concepts.

11.2.1 Integer output

Integer values can be output as octal (the base-8 number system), decimal (our usual base-10 number system), and hexadecimal (the base-16 number system). If you don't know about these systems, read §A.2.1.1 before proceeding here. Most output uses decimal. Hexadecimal is popular for outputting hardware-related information. The reason is that a hexadecimal digit exactly represents a 4-bit value. Thus, two hexadecimal digits can be used to present the value of an 8-bit byte, four hexadecimal digits give the value of 2 bytes (that's often a half word), and eight hexadecimal digits can present the value of 4 bytes (that's often the size of a word or a register). When C++'s ancestor C was first designed (in the 1970s), octal was popular for representing bit patterns, but now it's rarely used.

We can specify the output (decimal) value **1234** to be decimal, hexadecimal (often called "hex"), and octal:

```
cout << 1234 << "\t(decimal)\n"
        << hex << 1234 << "\t(hexadecimal)\n"
        << oct << 1234 << "\t(octal)\n";
```

The **'\t'** character is "tab" (short for "tabulation character"). This prints

```
1234  (decimal)
4d2   (hexadecimal)
2322  (octal)
```

The notations **<< hex** and **<< oct** do not output values. Instead, **<< hex** informs the stream that any further integer values should be displayed in hexadecimal and **<< oct** informs the stream that any further integer values should be displayed in octal. For example:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n';  // the octal base is still in effect
```

This produces

```
1234   4d2   2322
2322   // integers will continue to show as octal until changed
```

Note that the last output is octal; that is, **oct**, **hex**, and **dec** (for decimal) persist ("stick," "are sticky") – they apply to every integer value output until we tell the stream otherwise. Terms such as **hex** and **oct** that are used to change the behavior of a stream are called *manipulators*.

TRY THIS

Output your birth year in decimal, hexadecimal, and octal form. Label each value. Line up your output in columns using the tab character. Now output your age.

Seeing values of a base different from 10 can often be confusing. For example, unless we tell you otherwise, you'll assume that **11** represents the (decimal) number 11, rather than 9 (**11** in octal) or 17 (**11** in hexadecimal). To alleviate such problems, we can ask the **ostream** to show the base of each integer printed. For example:

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec;         // show bases
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

This prints

```
1234  4d2    2322
1234  0x4d2  02322
```

So, decimal numbers have no prefix, octal numbers have the prefix **0**, and hexadecimal values have the prefix **0x** (or **0X**). This is exactly the notation for integer literals in C++ source code. For example:

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

In decimal form, this will print

```
1234  1234  1234
```

As you might have noticed, **showbase** persists, just like **oct** and **hex**. The manipulator **noshowbase** reverses the action of **showbase**, reverting to the default, which shows each number without its base.

In summary, the integer output manipulators are:

Integer output manipulations	
oct	use base-8 (octal) notation
dec	use base-10 (decimal) notation
hex	use base-16 (hexadecimal) notation
showbase	prefix **0** for octal and **0x** for hexadecimal
noshowbase	don't use prefixes

11.2.2 Integer input

By default, **>>** assumes that numbers use the decimal notation, but you can tell it to read hexadecimal or octal integers:

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

If you type in

```
1234  4d2  2322  2322
```

this will print

```
1234  1234  1234  1234
```

Note that this implies that **oct**, **dec**, and **hex** "stick" for input, just as they do for output.

TRY THIS

Complete the code fragment above to make it into a program. Try the suggested input; then type in

```
        1234  1234  1234  1234
```

Explain the results. Try other inputs to see what happens.

You can get **>>** to accept and correctly interpret the **0** and **0x** prefixes. To do that, you "unset" all the defaults. For example:

```
cin.unsetf(ios::dec);   // don't assume decimal (so that 0x can mean hex)
cin.unsetf(ios::oct);   // don't assume octal (so that 12 can mean twelve)
cin.unsetf(ios::hex);   // don't assume hexadecimal (so that 12 can mean twelve)
```

The stream member function **unsetf()** clears the flag (or flags) given as argument. Now, if you write

```
cin >>a >> b >> c >> d;
```

and enter

```
1234   0x4d2   02322    02322
```

you get

```
1234   1234    1234     1234
```

11.2.3 Floating-point output

If you deal directly with hardware, you'll need hexadecimal (or possibly octal) notation. Similarly, if you deal with scientific computation, you must deal with the formatting of floating-point values. They are handled using **iostream** manipulators in a manner very similar to that of integer values. For example:

```
cout << 1234.56789 << "\t\t(defaultfloat)\n"        // \t\t to line up columns
        << fixed << 1234.56789 << "\t(fixed)\n"
        << scientific << 1234.56789 << "\t(scientific)\n";
```

This prints

```
1234.57             (general)
1234.567890         (fixed)
1.234568e+003       (scientific)
```

The manipulators **fixed**, **scientific**, and **defaultfloat** are used to select floating-point formats; **defaultfloat** is the default format (also known as the *general format*). Now, we can write

```
cout << 1234.56789 << '\t'
        << fixed << 1234.56789 << '\t'
        << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';                     // floating format "sticks"
cout << defaultfloat << 1234.56789 << '\t'      // the default format for
                                                // floating-point output
        << fixed << 1234.56789 << '\t'
        << scientific << 1234.56789 << '\n';
```

This prints

```
1234.57     1234.567890     1.234568e+003
1.234568e+003                               // scientific manipulator "sticks"
1234.57     1234.567890     1.234568e+003
```

In summary, the basic floating-point output-formatting manipulators are:

Floating-point formats	
fixed	use fixed-point notation
scientific	use mantissa and exponent notation; the mantissa is always in the [1:10) range; that is, there is a single nonzero digit before the decimal point
defaultfloat	choose **fixed** or **scientific** to give the numerically most accurate representation, within the precision of **defaultfloat**

11.2.4 Precision

By default, a floating-point value is printed using six total digits using the **defaultfloat** format. The most appropriate format is chosen and the number is rounded to give the best approximation that can be printed using only six digits (the default precision for the **defaultfloat** format). For example:

1234.567 prints as **1234.57**

1.2345678 prints as **1.23457**

The rounding rule is the usual 4/5 rule: 0 to 4 round down (toward zero) and 5 to 9 round up (away from zero). Note that floating-point formatting applies only to floating-point numbers, so

1234567 prints as **1234567** (because it's an integer)

1234567.0 prints as **1.23457e+006**

In the latter case, the **ostream** determines that **1234567.0** cannot be printed using the **fixed** format using only six digits and switches to **scientific** format to preserve the most accurate representation. Basically the **defaultfloat** format chooses between **scientific** and **fixed** formats to present the user with the most accurate representation of a floating-point value within the precision of the **general** format, which defaults to six total digits.

TRY THIS

Write some code to print the number **1234567.89** three times, first using **defaultfloat**, then **fixed**, then **scientific**. Which output form presents the user with the most accurate representation? Explain why.

A programmer can set the precision using the manipulator **setprecision()**. For example:

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(5)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(8)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
```

This prints (note the rounding)

```
1234.57      1234.567890      1.234568e+003
1234.6 1234.56789    1.23457e+003
1234.5679    1234.56789000    1.23456789e+003
```

The precision is defined as:

Floating-point precision	
defaultfloat	precision is the total number of digits
scientific	precision is the number of digits after the decimal point
fixed	precision is the number of digits after the decimal point

Use the default (**defaultfloat** format with precision 6) unless there is a reason not to. The usual reason not to is "Because we need greater accuracy of the output."

11.2.5 Fields

Using **scientific** and **fixed** formats, a programmer can control exactly how much space a value takes up on output. That's clearly useful for printing tables, etc. The equivalent mechanism for integer values is called *fields*. You can specify exactly how many character positions an integer value or string value will occupy using the "set field width" manipulator **setw()**. For example:

```
cout << 123456                          // no field used
     <<'|'<< setw(4) << 123456 << '|'   // 123456 doesn't fit in a 4-char field
     << setw(8) << 123456 << '|'        // set field width to 8
     << 123456 << "|\n";                // field sizes don't stick
```

This prints

```
123456|123456|  123456|123456|
```

Note first the two spaces before the third occurrence of **123456**. That's what we would expect for a six-digit number in an eight-character field. However, **123456** did not get truncated to fit into a four-character field. Why not? **|1234|** or **|3456|** might be considered plausible outputs for the four-character field. However, that would have completely changed the value printed without any warning to the poor reader that something had gone wrong. The **ostream** doesn't do that; instead it breaks the output format. Bad formatting is almost always preferable to "bad output data." In the most common uses of fields (such as printing out a table), the "overflow" is visually very noticeable, so that it can be corrected.

Fields can also be used for floating-point numbers and strings. For example:

```
cout << 12345 <<'|'<< setw(4) << 12345 << '|'
     << setw(8) << 12345 << '|' << 12345 << "|\n";
cout << 1234.5 <<'|'<< setw(4) << 1234.5 << '|'
     << setw(8) << 1234.5 << '|' << 1234.5 << "|\n";
```

```
cout << "asdfg" <<'|'<< setw(4) << "asdfg" << '|'
        << setw(8) << "asdfg" << '|' << "asdfg" << "|\n";
```

This prints

```
12345|12345|    12345|12345|
1234.5|1234.5|  1234.5|1234.5|
asdfg|asdfg|    asdfg|asdfg|
```

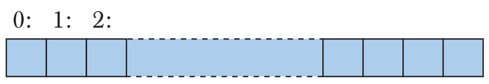
Note that the field width "doesn't stick." In all three cases, the first and the last values are printed in the default "as many characters as it takes" format. In other words, unless you set the field width immediately before an output operation, the notion of "field" is not used.

TRY THIS

Make a simple table including the last name, first name, telephone number, and email address for yourself and at least five of your friends. Experiment with different field widths until you are satisfied that the table is well presented.

11.3 File opening and positioning

As seen from C++, a file is an abstraction of what the operating system provides. As described in §10.3, a file is simply a sequence of bytes numbered from 0 upward:



The question is how we access those bytes. Using **iostreams**, this is largely determined when we open a file and associate a stream with it. The properties of a stream determine what operations we can perform after opening the file, and their meaning. The simplest example of this is that if we open an **istream** for a file, we can read from the file, whereas if we open a file with an **ostream**, we can write to it.

11.3.1 File open modes

You can open a file in one of several modes. By default, an **ifstream** opens its file for reading and an **ofstream** opens its file for writing. That takes care of most common needs. However, you can choose between several alternatives:

File stream open modes	
ios_base::app	append (i.e., add to the end of the file)
ios_base::ate	"at end" (open and seek to end)
ios_base::binary	binary mode — beware of system-specific behavior
ios_base::in	for reading
ios_base::out	for writing
ios_base::trunc	truncate file to 0 length

A file mode is optionally specified after the name of the file. For example:

```
ofstream of1 {name1};                    // defaults to ios_base::out
ifstream if1 {name2};                    // defaults to ios_base::in

ofstream ofs {name, ios_base::app};      // ofstreams by default include
                                         // io_base::out
fstream fs {"myfile", ios_base::in|ios_base::out};     // both in and out
```

The **|** in that last example is the "bitwise or" operator (§A.5.5) that can be used to combine modes as shown. The **app** option is popular for writing log files where you always add to the end.

In each case, the exact effect of opening a file may depend on the operating system, and if an operating system cannot honor a request to open a file in a certain way, the result will be a stream that is not in the **good()** state:

```
if (!fs)     // oops: we couldn't open that file that way
```

The most common reason for a failure to open a file for reading is that the file doesn't exist (at least not with the name we used):

```
ifstream ifs {"redungs"};
if (!ifs)    // error: can't open "readings" for reading
```

In this case, we guess that a spelling error might be the problem.

Note that typically, an operating system will create a new file if you try to open a nonexistent file for output, but (fortunately) not if you try to open a nonexistent file for input:

```
ofstream ofs {"no-such-file"};           // create new file called no-such-file
ifstream ifs {"no-file-of-this-name"};   // error: ifs will not be good()
```

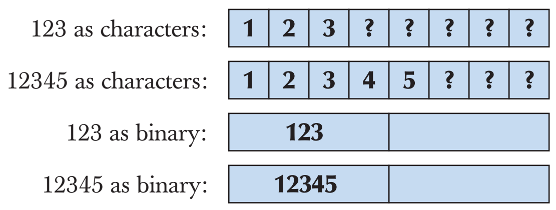
Try not to be clever with file open modes. Operating systems don't handle "unusual" mode consistently. When you can, stick to reading from files opened as **istreams** and writing to files opened as **ostreams**.

11.3.2 Binary files

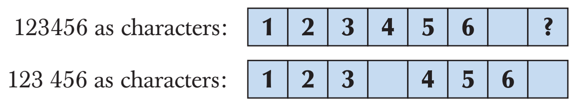
In memory, we can represent the number 123 as an integer value or as a string value. For example:

```
int n = 123;
string s = "123";
```

In the first case, **123** is stored as a (binary) number in an amount of memory that is the same as for all other **int**s (4 bytes, that is, 32 bits, on a PC). Had we chosen the value **12345** instead, the same 4 bytes would have been used. In the second case, **123** is stored as a string of three characters. Had we chosen the string value **"12345"** it would have used five characters (plus the fixed overhead for managing a **string**). We could illustrate this like this (using the ordinary decimal and character representation, rather than the binary representation actually used within the computer):


123 as characters: 1 2 3 ? ? ? ? ?
12345 as characters: 1 2 3 4 5 ? ? ?
123 as binary: 123
12345 as binary: 12345

When we use a character representation, we must use some character to represent the end of a number in memory, just as we do on paper: 123456 is one number and 123 456 are two numbers. On "paper," we use the space character to represent the end of the number. In memory, we could do the same:


123456 as characters: 1 2 3 4 5 6 ?
123 456 as characters: 1 2 3 4 5 6

The distinction between storing fixed-size binary representation (e.g., an **int**) and variable-size character string representation (e.g., a **string**) also occurs in files. By default, **iostreams** deal with character representations; that is, an **istream** reads a sequence of characters and turns it into an object of the desired type. An **ostream** takes an object of a specified type and transforms it into a sequence of characters which it writes out. However, it is possible to request **istream** and **ostream** to simply copy bytes to and from files. That's called *binary I/O* and is requested by opening a file with the mode **ios_base::binary.** Here is an example that reads and writes binary files of integers. The key lines that specifically deal with "binary" are explained below:

```
int main()
{
        // open an istream for binary input from a file:
        cout << "Please enter input file name\n";
        string iname;
        cin >> iname;
        ifstream ifs {iname,ios_base::binary};          // note: stream mode
                // binary tells the stream not to try anything clever with the bytes
        if (!ifs) error("can't open input file ",iname);

        // open an ostream for binary output to a file:
        cout << "Please enter output file name\n";
        string oname;
        cin >> oname;
        ofstream ofs {oname,ios_base::binary};          // note: stream mode
                // binary tells the stream not to try anything clever with the bytes
        if (!ofs) error("can't open output file ",oname);

        vector<int> v;

        // read from binary file:
        for(int x; ifs.read(as_bytes(x),sizeof(int)); )  // note: reading bytes
                v.push_back(x);

        // . . . do something with v . . .

        // write to binary file:
        for(int x : v)
                ofs.write(as_bytes(x),sizeof(int));      // note: writing bytes
        return 0;
}
```

We open the files using **ios_base::binary** as the stream mode:

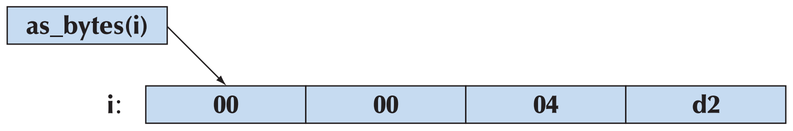
```
ifstream ifs {iname, ios_base::binary};

ofstream ofs {oname, ios_base::binary};
```

In both cases, we chose the trickier, but often more compact, binary representation. When we move from character-oriented I/O to binary I/O, we give up our usual >> and << operators. Those operators specifically turn values into character sequences using the default conventions (e.g., the string "asdf" turns into the characters a, s, d, f and the integer 123 turns into the characters 1, 2, 3). If we wanted that, we wouldn't need to say binary – the default would suffice. We use binary only if we (or someone else) thought that we somehow could do better than the default. We use binary to tell the stream not to try anything clever with the bytes.

What "cleverness" might we do to an int? The obvious is to store a 4-byte int in 4 bytes; that is, we can look at the representation of the int in memory (a sequence of 4 bytes) and transfer those bytes to the file. Later, we can read those bytes back the same way and reassemble the int:

```
ifs.read(as_bytes(i),sizeof(int))          // note: reading bytes
ofs.write(as_bytes(v[i]),sizeof(int))      // note: writing bytes
```

The ostream write() and the istream read() both take an address (supplied here by as_bytes()) and a number of bytes (characters) which we obtained by using the operator sizeof. That address should refer to the first byte of memory holding the value we want to read or write. For example, if we had an int with the value 1234, we would get the 4 bytes (using hexadecimal notation) 00, 00, 04, d2:



The as_bytes() function is needed to get the address of the first byte of an object's representation. It can – using language facilities yet to be explained (§17.8 and §19.3) – be defined like this:

```
template<class T>
char* as_bytes(T& i)                    // treat a T as a sequence of bytes
{
        void* addr = &i;                // get the address of the first byte
                                        // of memory used to store the object
        return static_cast<char*>(addr);      // treat that memory as bytes
}
```

The (unsafe) type conversion using static_cast is necessary to get to the "raw bytes" of a variable. The notion of addresses will be explored in some detail in Chapters 17 and 18. Here, we just show how to treat any object in memory as a sequence of bytes for the use of read() and write().

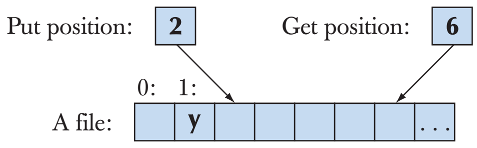
This binary I/O is messy, somewhat complicated, and error-prone. However, as programmers we don't always have the freedom to choose file formats, so occasionally we must use binary I/O simply because that's the format someone chose for the files we need to read or write. Alternatively, there may be a good logical reason for choosing a non-character representation. A typical example is an image or a sound file, for which there is no reasonable character representation: a photograph or a piece of music is basically just a bag of bits.

The character I/O provided by default by the iostream library is portable, human readable, and reasonably supported by the type system. Use it when you have a choice and don't mess with binary I/O unless you really have to.

11.3.3 Positioning in files

Whenever you can, just read and write files from the beginning to the end. That's the easiest and least error-prone way. Many times, when you feel that you have to make a change to a file, the better solution is to produce a new file containing the change.

However, if you must, you can use positioning to select a specific place in a file for reading or writing. Basically, every file that is open for reading has a "read/get position" and every file that is open for writing has a "write/put position":



This can be used like this:

```
fstream fs {name};       // open for input and output
if (!fs) error("can't open ",name);

fs.seekg(5);          // move reading position (g for "get") to 5 (the 6th character)
char ch;
fs>>ch;               // read and increment reading position
cout << "character[5] is " << ch << ' (' << int(ch) << ")\n";

fs.seekp(1);          // move writing position (p for "put") to 1
fs<<'y';              // write and increment writing position
```

Note that **seekg()** and **seekp()** increment their respective positions, so the figure represents the state of the program *after* execution.

Please be careful: there is next to no run-time error checking when you use positioning. In particular, it is undefined what happens if you try to seek (using **seekg()** or **seekp()**) beyond the end of a file, and operating systems really do differ in what happens then.

11.4 String streams

You can use a **string** as the source of an **istream** or the target for an **ostream**. An **istream** that reads from a **string** is called an **istringstream** and an **ostream** that stores characters written to it in a **string** is called an **ostringstream**. For example, an **istringstream** is useful for extracting numeric values from a **string**:

```
double str_to_double(string s)
        // if possible, convert characters in s to floating-point value
{
        istringstream is {s};           // make a stream so that we can read from s
        double d;
        is >> d;
        if (!is) error("double format error: ",s);
        return d;
}

double d1 = str_to_double("12.4");                    // testing
double d2 = str_to_double("1.34e–3");
double d3 = str_to_double("twelve point three");      // will call error()
```

If we try to read beyond the end of an **istringstream**'s string, the **istringstream** will go into **eof()** state. This means that we can use "the usual input loop" for an **istringstream**; an **istringstream** really is a kind of **istream**.

Conversely, an **ostringstream** can be useful for formatting output for a system that requires a simple **string** argument, such as a GUI system (see §16.5). For example:

```
void my_code(string label, Temperature temp)
{
        // . . .
        ostringstream os;       // stream for composing a message
        os << setw(8) << label << ": "
                << fixed << setprecision(5) << temp.temp << temp.unit;
        someobject.display(Point(100,100), os.str().c_str());
        // . . .
}
```

The **str()** member function of **ostringstream** returns the **string** composed by output operations to an **ostringstream**. The **c_str()** is a member function of **string** that returns a C-style string as required by many system interfaces.

The **stringstream**s are generally used when we want to separate actual I/O from processing. For example, a **string** argument for **str_to_double()** will usually originate in a file (e.g., a web log) or from a keyboard. Similarly, the message we composed in **my_code()** will eventually end up written to an area of a screen. For example, in §11.7, we use a **stringstream** to filter undesirable characters out of our input. Thus, **stringstream**s can be seen as a mechanism for tailoring I/O to special needs and tastes.

A simple use of an **ostringstream** is to construct strings by concatenation. For example:

```
int seq_no = get_next_number();             // get the number of a log file
ostringstream name;
name << "myfile" << seq_no << ".log";       // e.g., myfile17.log
ofstream logfile{name.str()};               // e.g., open myfile17.log
```

Usually, we initialize an **istringstream** with a **string** and then read the characters from that **string** using input operations. Conversely, we typically initialize an **ostringstream** to the empty **string** and then fill it using output operations. There is a more direct way of accessing characters in a **stringstream** that is sometimes useful: **ss.str()** returns a copy of **ss**'s string, and **ss.str(s)** sets **ss**'s string to a copy of **s**. §11.7 shows an example where **ss.str(s)** is essential.

11.5 Line-oriented input

A **>>** operator reads into objects of a given type according to that type's standard format. For example, when reading into an **int**, **>>** will read until it encounters something that's not a digit, and when reading into a **string**, **>>** will read until it encounters whitespace. The standard library **istream** library also provides facilities for reading individual characters and whole lines. Consider:

```
string name;
cin >> name;                     // input: Dennis Ritchie
cout << name << '\n';            // output: Dennis
```

What if we wanted to read everything on that line at once and decide how to format it later? That could be done using the function **getline()**. For example:

```
string name;
getline(cin,name);               // input: Dennis Ritchie
cout << name << '\n';            // output: Dennis Ritchie
```

Now we have the whole line. Why would we want that? A good answer would be "Because we want to do something that can't be done by **>>**." Often, the answer is a poor one: "Because the user typed a whole line." If that's the best you can think of, stick to **>>**, because once you have the line entered, you usually have to parse it somehow. For example:

```
string first_name;
string second_name;
stringstream ss {name};
ss>>first_name;                  // input Dennis
ss>>second_name;                 // input Ritchie
```

Reading directly into **first_name** and **second_name** would have been simpler.

One common reason for wanting to read a whole line is that the definition of whitespace isn't always appropriate. Sometimes, we want to consider a newline as different from other whitespace characters. For example, a text communication with a game might consider a line a sentence, rather than relying on conventional punctuation:

```
go left until you see a picture on the wall to your right
remove the picture and open the door behind it. take the bag from there
```

In that case, we'd first read a whole line and then extract individual words from that.

```
string command;
getline(cin,command);            // read the line

stringstream ss {command};
vector<string> words;
for (string s; ss>>s; )
        words.push_back(s);      // extract the individual words
```

On the other hand, had we had a choice, we would most likely have preferred to rely on some proper punctuation rather than a line break.

11.6 Character classification

Usually, we read integers, floating-point numbers, words, etc. as defined by format conventions. However, we can – and sometimes must – go down a level of abstraction and read individual characters. That's more work, but when we read individual characters, we have full control over what we are doing. Consider tokenizing an expression (§7.8.2). For example, we want **1+4*x<=y/z*5** to be separated into the eleven tokens

```
1 + 4 * x <= y / z * 5
```

We could use **>>** to read the numbers, but trying to read the identifiers as strings would cause **x<=y** to be read as one string (since **<** and **=** are not whitespace characters) and **z*** to be read as one string (since ***** isn't a whitespace character either). Instead, we could write

```
for (char ch; cin.get(ch); ) {
        if (isspace(ch)) {      // if ch is whitespace
                // do nothing (i.e., skip whitespace)
        }
        if (isdigit(ch)) {
                // read a number
        }
        else if (isalpha(ch)) {
                // read an identifier
        }
        else {
                // deal with operators
        }
}
```

The **istream::get()** function reads a single character into its argument. It does not skip whitespace. Like **>>**, **get()** returns a reference to its **istream** so that we can test its state.

When we read individual characters, we usually want to classify them: Is this character a digit? Is this character uppercase? And so forth. There is a set of standard library functions for that:

Character classification	
isspace(c)	Is **c** whitespace (**' '**, **'\t'**, **'\n'**, etc.)?
isalpha(c)	Is **c** a letter (**'a'**.. **'z'**, **'A'**.. **'Z'**) (note: not **'_'**)?
isdigit(c)	Is **c** a decimal digit (**'0'**.. **'9'**)?
isxdigit(c)	Is **c** a hexadecimal digit (decimal digit or **'a'**.. **'f'** or **'A'**.. **'F'**)?
isupper(c)	Is **c** an uppercase letter?
islower(c)	Is **c** a lowercase letter?
isalnum(c)	Is **c** a letter or a decimal digit?
iscntrl(c)	Is **c** a control character (ASCII 0..31 and 127)?

Character classification (*continued*)	
ispunct(c)	Is **c** not a letter, digit, whitespace, or invisible control character?
isprint(c)	Is **c** printable (ASCII **' '**.. **'~'**)?
isgraph(c)	Is **isalpha(c)** or **isdigit(c)** or **ispunct(c)** (note: not space)?

Note that the classifications can be combined using the "or" operator (**||**). For example, **isalnum(c)** means **isalpha(c)||isdigit(c)**; that is, "Is **c** either a letter or a digit?"

In addition, the standard library provides two useful functions for getting rid of case differences:

Character case	
toupper(c)	**c** or **c**'s uppercase equivalent
tolower(c)	**c** or **c**'s lowercase equivalent

These are useful when you want to ignore case differences. For example, in input from a user **Right**, **right**, and **rigHT** most likely mean the same thing (**rigHT** most likely being the result of an unfortunate hit on the Caps Lock key). After applying **tolower()** to each character in each of those strings, we get **right** for each. We can do that for an arbitrary **string**:

```
void tolower(string& s)        // put s into lower case
{
        for (char& x : s) x = tolower(x);
}
```

We use pass-by-reference (§8.5.5) to actually change the **string**. Had we wanted to keep the old string we could have written a function to make a lowercase copy. Prefer **tolower()** to **toupper()** because that works better for text in some natural languages, such as German, where not every lowercase character has an uppercase equivalent.

11.7 Using nonstandard separators

This section provides a semi-realistic example of the use of **iostream**s to solve a real problem. When we read strings, words are by default separated by whitespace. Unfortunately, **istream** doesn't offer a facility for us to define what characters make up whitespace or in some other way directly change how **>>** reads a string. So, what do we do if we need another definition of whitespace? Consider the example from §4.6.3 where we read in “words” and compared them. Those words were whitespace-separated, so if we read

```
As planned, the guests arrived; then,
```

We would get the “words”

```
As
planned,
the
guests
arrived;
then,
```

This is not what we’d find in a dictionary: **planned,** and **arrived;** are not words. They are words plus distracting and irrelevant punctuation characters. For most purposes we must treat punctuation just like whitespace. How might we get rid of such punctuation? We could read characters, remove the punctuation characters – or turn them into whitespace – and then read the “cleaned-up” input again:

```
string line;
getline(cin,line);          // read into line
for (char& ch : line)       // replace each punctuation character by a space
        switch(ch) {
        case ';': case '.': case ',': case '?': case '!':
                ch = ' ';
        }

stringstream ss(line);              // make an istream ss reading from line
vector<string> vs;
for (string word; ss>>word; )   // read words without punctuation characters
        vs.push_back(word);
```

Using that to read the line, we get the desired

```
As
planned
the
guests
arrived
then
```

Unfortunately, the code above is messy and rather special-purpose. What would we do if we had another definition of punctuation? Let's provide a more general and useful way of removing unwanted characters from an input stream. What would that be? What would we like our user code to look like? How about

```
ps.whitespace(";:,.");  // treat semicolon, colon, comma, and dot as whitespace
for (string word; ps>>word; )
        vs.push_back(word);
```

How would we define a stream that would work like **ps**? The basic idea is to read words from an ordinary input stream and then treat the user-specified "whitespace" characters as whitespace; that is, we do not give "whitespace" characters to the user, we just use them to separate words. For example,

```
as.not
```

should be the two words

```
as
not
```

We can define a class to do that for us. It must get characters from an **istream** and have a **>>** operator that works just like **istream**'s except that we can tell it which characters it should consider to be whitespace. For simplicity, we will not provide a way of treating existing whitespace characters (space, newline, etc.) as non-whitespace; we'll just allow a user to specify additional "whitespace" characters. Nor will we provide a way to completely remove the designated characters from the stream; as before, we will just turn them into whitespace. Let's call that class **Punct_stream**:

```
class Punct_stream {       // like an istream, but the user can add to
                           // the set of whitespace characters
public:
    Punct_stream(istream& is)
        : source{is}, sensitive{true} { }

    void whitespace(const string& s)         // make s the whitespace set
        { white = s; }
    void add_white(char c) { white += c; }   // add to the whitespace set
    bool is_whitespace(char c);              // is c in the whitespace set?
```

```
        void case_sensitive(bool b) { sensitive = b; }
        bool is_case_sensitive() { return sensitive; }

        Punct_stream& operator>>(string& s);
        operator bool();
private:
        istream& source;          // character source
        istringstream buffer;     // we let buffer do our formatting
        string white;             // characters considered "whitespace"
        bool sensitive;           // is the stream case-sensitive?
};
```

The basic idea is – just as in the example above – to read a line at a time from the **istream**, convert "whitespace" characters into spaces, and then use the **istringstream** to do formatting. In addition to dealing with user-defined whitespace, we have given **Punct_stream** a related facility: if we ask it to, using **case_sensitive()**, it can convert case-sensitive input into non-case-sensitive input. For example, if we ask, we can get a **Punct_stream** to read

```
Man bites dog!
```

as

```
man
bites
dog
```

Punct_stream's constructor takes the **istream** to be used as a character source and gives it the local name **source**. The constructor also defaults the stream to the usual case-sensitive behavior. We can make a **Punct_stream** that reads from **cin** regarding semicolon, colon, and dot as whitespace, and that turns all characters into lower case:

```
Punct_stream ps {cin};          // ps reads from cin
ps.whitespace(";:.");           // semicolon, colon, and dot are also whitespace
ps.case_sensitive(false);       // not case-sensitive
```

Obviously, the most interesting operation is the input operator **>>**. It is also by far the most difficult to define. Our general strategy is to read a whole line from the **istream** into a string (called **line**). We then convert all of "our" whitespace characters to the space character (**' '**). That done, we put the line into the **istringstream** called **buffer**. Now we can use the usual whitespace-separating **>>** to read from **buffer**. The code looks a bit more complicated than this because we simply try reading from the **buffer** and try to fill it only when we find it empty:

```
Punct_stream& Punct_stream::operator>>(string& s)
{
        while (!(buffer>>s)) {                        // try to read from buffer
                if (buffer.bad() || !source.good()) return *this;
                buffer.clear();

                string line;
                getline(source,line);                 // get a line from source

                // do character replacement as needed:
                for (char& ch : line)
                        if (is_whitespace(ch))
                                ch = ' ';             // to space
                        else if (!sensitive)
                                ch = tolower(ch);     // to lower case

                buffer.str(line);                     // put string into stream
        }
        return *this;
}
```

Let's consider this bit by bit. Consider first the somewhat unusual

```
while (!(buffer>>s)) {
```

If there are characters in the **istringstream** called **buffer**, the read **buffer>>s** will work, and **s** will receive a "whitespace"-separated word; then there is nothing more to do. That will happen as long as there are characters in **buffer** for us to read. However, when **buffer>>s** fails – that is, if **!(buffer>>s)** – we must replenish **buffer** from **source**. Note that the **buffer>>s** read is in a loop; after we have tried to replenish **buffer**, we need to try another read, so we get

```
while (!(buffer>>s)) {                                // try to read from buffer
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        // replenish buffer
}
```

If **buffer** is **bad()** or the source has a problem, we give up; otherwise, we clear **buffer** and try again. We need to clear **buffer** because we get into that "replenish loop" only if a read failed, typically because we hit **eof()** for **buffer**; that is, there were no more characters in **buffer** for us to read. Dealing with stream state is always messy and it is often the source of subtle errors that require tedious debugging. Fortunately the rest of the replenish loop is pretty straightforward:

```
string line;
getline(source,line);            // get a line from source

// do character replacement as needed:
for (char& ch : line)
        if (is_whitespace(ch))
                ch = ' ';            // to space
        else if (!sensitive)
                ch = tolower(ch);    // to lower case

buffer.str(line);                // put string into stream
```

We read a line into **line**. Then we look at each character of that line to see if we need to change it. The **is_whitespace()** function is a member of **Punct_stream**, which we'll define later. The **tolower()** function is a standard library function doing the obvious, such as turning **A** into **a** (see §11.6).

Once we have a properly processed **line**, we need to get it into our **istringstream.** That's what **buffer.str(line)** does; it can be read as "Set the **istringstream buffer**'s **string** to **line**."

Note that we "forgot" to test the state of **source** after reading from it using **getline()**. We don't need to because we will eventually reach the **!source.good()** test at the top of the loop.

As ever, we return a reference to the stream itself, ***this**, as the result of **>>**; see §17.10.

Testing for whitespace is easy; we just compare a character to each character of the string that holds our whitespace set:

```
bool Punct_stream::is_whitespace(char c)
{
        for (char w : white)
                if (c==w) return true;
        return false;
}
```

Remember that we left the **istringstream** to deal with the usual whitespace characters (e.g., newline and space) in the usual way, so we don't need to do anything special about those.

This leaves one mysterious function:

```
Punct_stream::operator bool()
{
        return !(source.fail() || source.bad()) && source.good();
}
```

The conventional use of an **istream** is to test the result of **>>**. For example:

```
while (ps>>s) { /* . . . */ }
```

That means that we need a way of looking at the result of **ps>>s** as a Boolean value. The result of **ps>>s** is a **Punct_stream**, so we need a way of implicitly turning a **Punct_stream** into a **bool**. That's what **Punct_stream**'s **operator bool()** does. A member function called **operator bool()** defines a conversion to **bool**. In particular, it returns **true** if the operation on the **Punct_stream** succeeded.

Now we can write our program:

```
int main()
        // given text input, produce a sorted list of all words in that text
        // ignore punctuation and case differences
        // eliminate duplicates from the output
{
        Punct_stream ps {cin};
        ps.whitespace(";:,.?!()\"{}<>/&$@#%^*|~");    // note \" means " in string
        ps.case_sensitive(false);

        cout << "please enter words\n";
        vector<string> vs;
        for (string word; ps>>word; )
                vs.push_back(word);           // read words

        sort(vs.begin(),vs.end());            // sort in lexicographical order
        for (int i=0; i<vs.size(); ++i)       // write dictionary
                if (i==0 || vs[i]!=vs[i–1]) cout << vs[i] << '\n';
}
```

This will produce a properly sorted list of words from input. The test

```
if (i==0 || vs[i]!=vs[i–1])
```

will suppress duplicates. Feed this program the input

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
```

and it will output

```
about
and
are
complain
don't
kind
languages
of
only
people
that
there
two
use
```

Why did we get **don't** and not **dont**? We left the single quote out of the **whitespace()** call.

Caution: **Punct_stream** behaves like an **istream** in many important and useful ways, but it isn't really an **istream**. For example, we can't ask for its state using **rdstate()**, **eof()** isn't defined, and we didn't bother providing a **>>** that reads integers. Importantly, we cannot pass a **Punct_stream** to a function expecting an **istream**. Could we define a **Punct_istream** that really is an **istream**? We could, but we don't yet have the programming experience, the design concepts, and the language facilities required to pull off that stunt (if you – much later – want to return to this problem, you have to look up stream buffers in an expert-level guide or manual).

Did you find **Punct_stream** easy to read? Did you find the explanations easy to follow? Do you think you could have written it yourself? If you were a genuine novice a few days ago, the honest answer is likely to be "No, no, no!" or even "NO, no! Nooo!! – Are you crazy?" We understand – and the answer to the last question/outburst is "No, at least we think not." The purpose of the example is

- To show a somewhat realistic problem and solution
- To show what can be achieved with relatively modest means
- To provide an easy-to-use solution to an apparently easy problem
- To illustrate the distinction between the interface and the implementation

To become a programmer, you need to read code, and not just carefully polished solutions to educational problems. This is an example. In another few days or weeks, this will become easy for you to read, and you will be looking at ways to improve the solution.

One way to think of this example is as equivalent to a teacher having dropped some genuine English slang into an English-for-beginners course to give a bit of color and enliven the proceedings.

11.8 And there is so much more

The details of I/O seem infinite. They probably are, since they are limited only by human inventiveness and capriciousness. For example, we have not considered the complexity implied by natural languages. What is written as **12.35** in English will be conventionally represented as **12,35** in most other European languages. Naturally, the C++ standard library provides facilities for dealing with that and many other natural-language-specific aspects of I/O. How do you write Chinese characters? How do you compare strings written using Malayalam characters? There are answers, but they are far beyond the scope of this book. If you need to know, look in more specialized or advanced books (such as Langer, *Standard C++ IOStreams and Locales*, and Stroustrup, *The C++ Programming Language*) and in library and system documentation. Look for "locale"; that's the term usually applied to facilities for dealing with natural language differences.

Another source of complexity is buffering: the standard library **iostream**s rely on a concept called **streambuf**. For advanced work – whether for performance or functionality – with **iostream**s these **streambuf**s are unavoidable. If you feel the need to define your own **iostream**s or to tune **iostream**s to new data sources/sinks, see Chapter 38 of *The C++ Programming Language* by Stroustrup or your system documentation.

When using C++, you may also encounter the C standard **printf()**/**scanf()** family of I/O functions. If you do, look them up in §27.6, §B.10.2, or in the excellent C textbook by Kernighan and Ritchie (*The C Programming Language*) or one of the innumerable sources on the web. Each language has its own I/O facilities; they all vary, most are quirky, but most reflect (in various odd ways) the same fundamental concepts that we have presented in Chapters 10 and 11.

The standard library I/O facilities are summarized in Appendix B.

The related topic of graphical user interfaces (GUIs) is described in Chapters 12–16.

Drill

1. Start a program called **Test_output.cpp**. Declare an integer **birth_year** and assign it the year you were born.
2. Output your **birth_year** in decimal, hexadecimal, and octal form.
3. Label each value with the name of the base used.
4. Did you line up your output in columns using the tab character? If not, do it.
5. Now output your age.
6. Was there a problem? What happened? Fix your output to decimal.
7. Go back to 2 and cause your output to show the base for each output.
8. Try reading as octal, hexadecimal, etc.:

```
cin >> a >>oct >> b >> hex >> c >> d;
cout << a << '\t'<< b << '\t'<< c << '\t'<< d << '\n' ;
```

   Run this code with the input

```
1234 1234 1234 1234
```

   Explain the results.
9. Write some code to print the number **1234567.89** three times, first using **defaultfloat**, then **fixed**, then **scientific** forms. Which output form presents the user with the most accurate representation? Explain why.
10. Make a simple table including last name, first name, telephone number, and email address for yourself and at least five of your friends. Experiment with different field widths until you are satisfied that the table is well presented.

Review

1. Why is I/O tricky for a programmer?
2. What does the notation **<< hex** do?
3. What are hexadecimal numbers used for in computer science? Why?
4. Name some of the options you may want to implement for formatting integer output.
5. What is a manipulator?
6. What is the prefix for decimal? For octal? For hexadecimal?
7. What is the default output format for floating-point values?
8. What is a field?
9. Explain what **setprecision()** and **setw()** do.
10. What is the purpose of file open modes?
11. Which of the following manipulators does not “stick”: **hex**, **scientific**, **setprecision()**, **showbase**, **setw**?

12. What is the difference between character I/O and binary I/O?
13. Give an example of when it would probably be beneficial to use a binary file instead of a text file.
14. Give two examples where a **stringstream** can be useful.
15. What is a file position?
16. What happens if you position a file position beyond the end of file?
17. When would you prefer line-oriented input to type-specific input?
18. What does **isalnum(c)** do?

Terms

binary	hexadecimal	octal
character classification	irregularity	output formatting
decimal	line-oriented input	regularity
defaultfloat	manipulator	**scientific**
file positioning	nonstandard separator	**setprecision()**
fixed	**noshowbase**	**showbase**

Exercises

1. Write a program that reads a text file and converts its input to all lower case, producing a new file.
2. Write a program that given a file name and a word outputs each line that contains that word together with the line number. Hint: **getline()**.
3. Write a program that removes all vowels from a file ("disemvowels"). For example, **Once upon a time!** becomes **nc pn tm!**. Surprisingly often, the result is still readable; try it on your friends.
4. Write a program called **multi_input.cpp** that prompts the user to enter several integers in any combination of octal, decimal, or hexadecimal, using the **0** and **0x** base suffixes; interprets the numbers correctly; and converts them to decimal form. Then your program should output the values in properly spaced columns like this:

```
0x43  hexadecimal  converts to  67  decimal
0123  octal        converts to  83  decimal
  65  decimal      converts to  65  decimal
```

5. Write a program that reads strings and for each string outputs the character classification of each character, as defined by the character classification functions presented in §11.6. Note that a character can have several classifications (e.g., **x** is both a letter and an alphanumeric).
6. Write a program that replaces punctuation with whitespace. Consider **.** (dot), **;** (semicolon), **,** (comma), **?** (question mark), **-** (dash), **'** (single

quote) punctuation characters. Don't modify characters within a pair of double quotes (**"**). For example, " **- don't use the as-if rule.**" becomes " **don t use the as if rule** ".

7. Modify the program from the previous exercise so that it replaces **don't** with **do not**, **can't** with **cannot**, etc.; leaves hyphens within words intact (so that we get " **do not use the as-if rule** "); and converts all characters to lower case.
8. Use the program from the previous exercise to make a dictionary (as an alternative to the approach in §11.7). Run the result on a multi-page text file, look at the result, and see if you can improve the program to make a better dictionary.
9. Split the binary I/O program from §11.3.2 into two: one program that converts an ordinary text file into binary and one program that reads binary and converts it to text. Test these programs by comparing a text file with what you get by converting it to binary and back.
10. Write a function **vector<string> split(const string& s)** that returns a **vector** of whitespace-separated substrings from the argument **s**.
11. Write a function **vector<string> split(const string& s, const string& w)** that returns a **vector** of whitespace-separated substrings from the argument **s**, where whitespace is defined as "ordinary whitespace" plus the characters in **w**.
12. Reverse the order of characters in a text file. For example, **asdfghjkl** becomes **lkjhgfdsa**. Warning: There is no really good, portable, and efficient way of reading a file backward.
13. Reverse the order of words (defined as whitespace-separated strings) in a file. For example, **Norwegian Blue parrot** becomes **parrot Blue Norwegian**. You are allowed to assume that all the strings from the file will fit into memory at once.
14. Write a program that reads a text file and writes out how many characters of each character classification (§11.6) are in the file.
15. Write a program that reads a file of whitespace-separated numbers and outputs a file of numbers using scientific format and precision 8 in four fields of 20 characters per line.
16. Write a program to read a file of whitespace-separated numbers and output them in order (lowest value first), one value per line. Write a value only once, and if it occurs more than once write the count of its occurrences on its line. For example, **7 5 5 7 3 117 5** should give

```
3
5	3
7	2
117
```

Postscript

Input and output are messy because our human tastes and conventions have not followed simple-to-state rules and straightforward mathematical laws. As programmers, we are rarely in a position to dictate that our users depart from their preferences, and when we are, we should typically be less arrogant than to think that we can provide a simple alternative to conventions built up over time. Consequently, we must expect, accept, and adapt to a certain messiness of input and output while still trying to keep our programs as simple as possible – but no simpler.