

I/O Streams as an Introduction to Objects and Classes

6

6.1 STREAMS AND BASIC FILE I/O 306

Why Use Files for I/O? 307

File I/O 308

Introduction to Classes and Objects 312

Programming Tip: Check Whether a File Was
Opened Successfully 314

Techniques for File I/O 316

Appending to a File (*Optional*) 320

File Names as Input (*Optional*) 321

6.2 TOOLS FOR STREAM I/O 323

Formatting Output with Stream Functions 323

Manipulators 329

Streams as Arguments to Functions 332

Programming Tip: Checking for the
End of a File 332

A Note on Namespaces 335

Programming Example: Cleaning Up a File
Format 336

6.3 CHARACTER I/O 338

The Member Functions **get** and **put** 338

The **putback** Member Function (*Optional*) 342

Programming Example: Checking Input 343

Pitfall: Unexpected '\n' in Input 345

Programming Example: Another **new_line**
Function 347


Default Arguments for Functions (*Optional*) 348

The **eof** Member Function 353

Programming Example: Editing a Text File 355

Predefined Character Functions 356

Pitfall: **toupper** and **tolower** Return Values 358



Fish say, they have their stream and pond; But is there anything beyond?

RUPERT BROOKE, "Heaven" (1913)

As a leaf is carried by a stream, whether the stream ends in a lake or in the sea, so too is the output of your program carried by a stream not knowing if the stream goes to the screen or to a file.

WASHROOM WALL OF A COMPUTER SCIENCE DEPARTMENT (1995)

INTRODUCTION

I/O refers to program input and output. Input can be taken from the keyboard or from a file. Similarly, output can be sent to the screen or to a file. This chapter explains how you can write your programs to take input from a file and send output to another file.

Input is delivered to your program via a C++ construct known as a *stream*, and output from your program is delivered to the output device via a stream. Streams are our first examples of *objects*. An object is a special kind of variable that has its own special-purpose functions that are, in a sense, attached to the variable. The ability to handle objects is one of the language features that sets C++ apart from earlier programming languages. In this chapter we tell you what streams are and explain how to use them for program I/O. In the process of explaining streams, we will introduce you to the basic ideas about what objects are and about how objects are used in a program.

PREREQUISITES

This chapter uses the material from Chapters 2 through 5.

6.1 STREAMS AND BASIC FILE I/O

Good Heavens! For more than forty years I have been speaking prose without knowing it.

MOLIÈRE, *Le Bourgeois Gentilhomme*

You are already using files to store your programs. You can also use files to store input for a program or to receive output from a program. The files used for program I/O are the same kind of files you use to store your programs. Streams, which we discuss next, allow you to write programs that handle file input and keyboard input in a unified way and that handle file output and screen output in a unified way.

A **stream** is a flow of characters (or other kind of data). If the flow is into your program, the stream is called an **input stream**. If the flow is out of your program, the stream is called an **output stream**. If the input

stream flows from the keyboard, then your program will take input from the keyboard. If the input stream flows from a file, then your program will take its input from that file. Similarly, an output stream can go to the screen or to a file.

Although you may not realize it, you have already been using streams in your programs. The `cin` that you have already used is an input stream connected to the keyboard, and `cout` is an output stream connected to the screen. These two streams are automatically available to your program, as long as it has an `include` directive that names the header file `iostream`. You can define other streams that come from or go to files; once you have defined them, you can use them in your program in the same way you use the streams `cin` and `cout`.

`cin` and `cout` are streams

For example, suppose your program defines a stream called `in_stream` that comes from some file. (We'll tell you how to define it shortly.) You can then fill an `int` variable named `the_number` with a number from this file by using the following in your program:

```
int the_number;
in_stream >> the_number;
```

Similarly, if your program defines an output stream named `out_stream` that goes to another file, then you can output the value of this variable to this other file. The following will output the string "`the_number is`" followed by the contents of the variable `the_number` to the output file that is connected to the stream `out_stream`:

```
out_stream << "the_number is" << the_number << endl;
```

Once the streams are connected to the desired files, your program can do file I/O the same way it does I/O using the keyboard and screen.

Why Use Files for I/O?

The keyboard input and screen output we have used so far deal with temporary data. When the program ends, the data typed in at the keyboard and the data left on the screen go away. Files provide you with a way to store data permanently. The contents of a file remain until a person or program changes the file. If your program sends its output to a file, the output file will remain after the program has finished running. An input file can be used over and over again by many programs without the need to type in the data separately for each program.

Permanent storage

The input and output files used by your program are the same kind of files that you read and write with an editor, such as the editor you use to write your programs. This means you can create an input file for your program or read an output file produced by your program whenever it's convenient for you, as opposed to having to do all your reading and writing while the program is running.

Files also provide you with a convenient way to deal with large quantities of data. When your program takes its input from a large input file, the program receives a lot of data without making the user do a lot of typing.

File I/O

When your program takes input from a file, it is said to be **reading** from the file; when your program sends output to a file, it is said to be **writing** to the file. There are other ways of reading input from a file, but the method we will use reads the file from the beginning to the end (or as far as the program gets before ending). Using this method, your program is not allowed to back up and read anything in the file a second time. This is exactly what happens when the program takes input from the keyboard, so this should not seem new or strange. (As we will see, the program can reread a file starting from the beginning of the file, but this is “starting over,” not “backing up.”) Similarly, for the method we present here, your program writes output into a file starting at the beginning of the file and proceeding forward. It is not allowed to back up and change any output that it has previously written to the file. This is exactly what happens when your program sends output to the screen. You can send more output to the screen, but you cannot back up and change the screen output. The way that you get input from a file into your program or send output from your program into a file is to connect the program to the file by means of a stream.

A stream is a variable

In C++, a stream is a special kind of variable known as an *object*. We will discuss objects in the next section, but we will first describe how your program can use stream objects to do simple file I/O. If you want to use a stream to get input from a file (or give output to a file), you must declare the stream and you must connect the stream to the file.

You can think of the file that a stream is connected to as the value of the stream. You can disconnect a stream from one file and connect it to another file, so you can change the value of these stream variables. However, you must use special functions that apply only to streams in order to perform these changes. You *cannot* use a stream variable in an assignment statement the way that you can use a variable of type *int* or *char*. Although streams are variables, they are unusual sorts of variables.

Declaring streams
ifstream and
ofstream

The streams `cin` and `cout` are already declared for you, but if you want a stream to connect to a file, you must declare it just as you would declare any other variable. The type for input-file stream variables is named `ifstream` (for “input-file stream”). The type for output-file stream variables is named `ofstream` (for “output-file stream”). Thus, you can declare `in_stream` to be an input stream for a file and `out_stream` to be an output stream for another file as follows:

```
ifstream in_stream;  
ofstream out_stream;
```

The types `ifstream` and `ofstream` are defined in the library with the header file `fstream`, and so any program that declares stream variables in this way must contain the following directive (normally near the beginning of the file):

```
#include <fstream>
```

When using the types `ifstream` and `ofstream`, your program must also contain the following, normally either at the start of the file or at the start of the function body that uses the types `ifstream` or `ofstream`:

```
using namespace std;
```

Stream variables, such as `in_stream` and `out_stream` declared earlier, must each be **connected** to a file. This is called **opening the file** and is done with a function named `open`. For example, suppose you want the input stream `in_stream` connected to the file named `infile.dat`. Your program must then contain the following before it reads any input from this file:

Connecting a
stream to a file
open

```
in_stream.open("infile.dat");
```

This may seem like rather strange syntax for a function call. We will have more to say about this peculiar syntax in the next section. For now, just notice a couple of details about how this call to `open` is written. First, the stream variable name and a dot (that is, a period) is placed before the function named `open`, and the file name is given as an argument to `open`. Also notice that the file name is given in quotes. The file name that is given as an argument is the same as the name you would use for the file if you wanted to write in it using the editor. If the input file is in the same directory as your program, you probably can simply give the name of the file in the manner just described. In some situations you might also need to specify the directory that contains the file. The details about specifying directories varies from one system to another. If you need to specify a directory, ask your instructor or some other local expert to explain the details.

Once you have declared an input stream variable and connected it to a file using the `open` function, your program can take input from the file using the extraction operator `>>`. For example, the following reads two input numbers from the file connected to `in_stream` and places them in the variables `one_number` and `another_number`:

```
int one_number, another_number;  
in_stream >> one_number >> another_number;
```

An output stream is opened (that is, connected to a file) in the same way as just described for input streams. For example, the following declares the output stream `out_stream` and connects it to the file named `outfile.dat`:

```
ofstream out_stream;  
out_stream.open("outfile.dat");
```

When used with a stream of type `ofstream`, the member function `open` will create the output file if it does not already exist. If the output file does already exist, the member function `open` will discard the contents of the file so that the output file is empty after the call to `open`.

After a file is connected to the stream `out_stream` with a call to `open`, the program can send output to that file using the insertion operator `<<`. For example, the following writes two strings and the contents of the variables `one_number` and `another_number` to the file that is connected to the stream `out_stream` (which in this example is the file named `outfile.dat`):

```
out_stream << "one_number = " << one_number
<< " another_number = " << another_number;
```

Notice that when your program is dealing with a file, it is as if the file had two names. One is the usual name for the file that is used by the operating system. This name is called the **external file name**. In our sample code the external file names were `infile.dat` and `outfile.dat`. The external file name is in some sense the “real name” for the file. It is the name used by the operating system. The conventions for spelling these external file names vary from one system to another; you will need to learn these conventions from your instructor or from some other local expert. The names `infile.dat` and `outfile.dat` that we used in our examples might or might not look like file names on your system. You should name your files following whatever conventions your system uses. Although the external file name is the real name for the file, it is typically used only once in a program. The external file name is given as an argument to the function `open`, but *after the file is opened, the file is always referred to by naming the stream that is connected to the file*. Thus, within your program, the stream name serves as a second name for the file.

The sample program in Display 6.1 reads three numbers from one file and writes their sum, as well as some text, to another file.

A File Has Two Names

Every input and every output file used by your program has two names. The **external file name** is the real name of the file, but it is used only in the call to the function `open`, which connects the file to a stream. After the call to `open`, you always use the stream name as the name of the file.

Every file should be closed when your program is finished getting input from the file or sending output to the file. Closing a file disconnects the stream from the file. A file is closed with a call to the function `close`. The following lines from the program in Display 6.1 illustrate how to use the function `close`:

```
in_stream.close( );
out_stream.close( );
```

Notice that the function `close` takes no arguments. If your program ends normally but without closing a file, the system will automatically close the file for you. However, it is good to get in the habit of closing files for at least two reasons. First, the system will only close files for you if your program ends in a normal fashion. If your program ends abnormally due to an error, the file will not be closed and may be left in a corrupted state. If your program closes files as soon as it is finished with them, file corruption is less likely. A second reason for closing a file is that you may want your program to send output to a file and later read that output back into the program. To do this, your program should close the file after it is finished writing to the file, and then your program should connect the file to an input stream using the function

DISPLAY 6.1 Simple File Input/Output

```

1  //Reads three numbers from the file infile.dat, sums the numbers,
2  //and writes the sum to the file outfile.dat.
3  //(A better version of this program will be given in Display 6.2.)
4  #include <fstream>
5  int main( )
6  {
7      using namespace std;
8      ifstream in_stream;
9      ofstream out_stream;
10
11     in_stream.open("infile.dat");
12     out_stream.open("outfile.dat");
13     int first, second, third;
14     in_stream >> first >> second >> third;
15     out_stream << "The sum of the first 3\n"
16                 << "numbers in infile.dat\n"
17                 << "is " << (first + second + third)
18                 << endl;
19     in_stream.close( );
20     out_stream.close( );
21     return 0;
22 }
```

infile.dat

(Not changed by program.)

```

1
2
3
4
```

outfile.dat

(After program is run.)

```

The sum of the first 3
numbers in infile.dat
is 6
```

There is no output to the screen and no input from the keyboard.

open. (It is possible to open a file for both input and output, but this is done in a slightly different way and we will not be discussing this alternative.)

Introduction to Classes and Objects

The streams `in_stream` and `out_stream` discussed in the last section and the predefined streams `cin` and `cout` are objects. An **object** is a variable that has functions as well as data associated with it. For example, the streams `in_stream` and `out_stream` both have a function named `open` associated with them. Two sample calls of these functions, along with the declarations of the objects `in_stream` and `out_stream`, are given below:

```
ifstream in_stream;
ofstream out_stream;
in_stream.open("infile.dat");
out_stream.open("outfile.dat");
```

There is a reason for this peculiar notation. The function named `open` that is associated with the object `in_stream` is a different function from the function named `open` that is associated with the object `out_stream`. One function opens a file for input, and the other opens a file for output. Of course, these two functions are similar. They both “open files.” When we give two functions the same name, it is because the two functions have some intuitive similarity. However, these two functions named `open` are different functions, even if they may be only slightly different. When the compiler sees a call to a function named `open`, it must decide which of these two functions named `open` you mean. The compiler determines this by looking at the name of the object that precedes the dot, in this case, either `in_stream` or `out_stream`. A function that is associated with an object is called a **member function**. So, for example, `open` is a member function of the object `in_stream`, and another function named `open` is a member of the object `out_stream`.

As we have just seen, different objects can have different member functions. These functions may have the same names, as was true of the functions named `open`, or they may have completely different names. The type of an object determines which member functions the object has. If two objects are of the same type, they may have different values, but they will have the same member functions. For example, suppose you declare the following stream objects:

```
ifstream in_stream, in_stream2;
ofstream out_stream, out_stream2;
```

The functions `in_stream.open` and `in_stream2.open` are the same function. Similarly, `out_stream.open` and `out_stream2.open` are the same function (but they are different from the functions `in_stream.open` and `in_stream2.open`).

A type whose variables are objects—such as `ifstream` and `ofstream`—is called a **class**. Since the member functions for an object are completely determined by its class (that is, by its type), these functions are called *member functions of the class* (as well as being called *members of the object*). For example, the class `ifstream` has a member function called `open`, and the class `ofstream`

has a different member function called `open`. The class `ofstream` also has a member function named `precision`, but the class `ifstream` has no member function named `precision`. You have already been using the member function `precision` with the stream `cout`, but we will discuss it in more detail later.

When you call a member function in a program, you always specify an object, usually by writing the object name and a dot before the function name, as in the following example:

```
in_stream.open("infile.dat");
```

One reason for naming the object is that the function can have some effect on the object. In the preceding example, the call to the function `open` connects the file `infile.dat` to the stream `in_stream`, so it needs to know the name of this stream.

In a function call, such as

```
in_stream.open("infile.dat");
```

the dot is called the **dot operator** and the object named before the dot is referred to as the **calling object**. In some ways the calling object is like an additional argument to the function—the function can change the calling object as if it were an argument—but the calling object plays an even larger role in the function call. The calling object determines the meaning of the function name. The compiler uses the type of the calling object to determine the meaning of the function name. For example, in the earlier call to `open`, the type of the object `in_stream` determines the meaning of the function name `open`.

Calling a member function

Calling a Member Function

SYNTAX

```
Calling_Object.Member_Function_Name(Argument_List);
```

Dot Operator

EXAMPLES

```
in_stream.open("infile.dat");  
out_stream.open("outfile.dat");  
out_stream.precision(2);
```

The meaning of the *Member_Function_Name* is determined by the class of (that is, the type of) the *Calling_Object*.

Classes and Objects

An **object** is a variable that has functions associated with it. These functions are called **member functions**. A **class** is a type whose variables are objects. The object's class (that is, the type of the object) determines which member functions the object has.



VideoNote
Objects and File I/O
Streams

The function name `close` is analogous to `open`. The classes `ifstream` and `ofstream` each have a member function named `close`. They both “close files,” but they close them in different ways because the files were opened and were manipulated in different ways. We will be discussing more member functions for the classes `ifstream` and `ofstream` later in this chapter.

■ PROGRAMMING TIP Check Whether a File Was Opened Successfully

A call to `open` can be unsuccessful for a number of reasons. For example, if you open an input file and there is no file with the external name that you specify, then the call to `open` will fail. When this happens, you might not receive an error message and your program might simply proceed to do something unexpected. Thus, you should always follow a call to `open` with a test to see whether the call to `open` was successful and end the program (or take some other appropriate action) if the call to `open` was unsuccessful.

You can use the member function named `fail` to test whether a stream operation has failed. There is a `fail` member function for each of the classes `ifstream` and `ofstream`. The `fail` function takes no arguments and returns a `bool` value. A call to the function `fail` for a stream named `in_stream` would be as follows:

```
in_stream.fail( )
```

This is a Boolean expression that can be used to control a *while* loop or an *if-else* statement.

You should place a call to `fail` immediately after each call to `open`; if the call to `open` fails, the function `fail` will return *true* (that is, the Boolean expression will be satisfied). For example, if the following call to `open` fails, then the program will output an error message and end; if the call succeeds, the `fail` function returns *false*, so the program will continue.

```
in_stream.open("stuff.dat");
if (in_stream.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);  ← Ends the program
}
```

`fail` is a member function, so it is called using the stream name and a dot. Of course, the call to `in_stream.fail` refers only to a call to `open` of the form `in_stream.open`, and not to any call to the function `open` made with any other stream as the calling object.

The member
function `fail`

The `exit` statement shown earlier has nothing to do with classes and has nothing directly to do with streams, but it is often used in this context. The `exit` statement causes your program to end immediately. The `exit` function returns its argument to the operating system. To use the `exit` statement, your program must contain the following `include` directive:

```
#include <cstdlib>
```

When using `exit`, your program must also contain the following, normally either at the start of the file or at the start of the function body that uses `exit`:

```
using namespace std;
```

The function `exit` is a predefined function that takes a single integer argument. By convention, 1 is used as the argument if the call to `exit` was due to an error, and 0 is used otherwise.¹ For our purposes, it makes no difference what integer you use, but it pays to follow this convention since it is important in more advanced programming.

The `exit` Statement

The `exit` statement is written

```
exit(Integer_Value);
```

When the `exit` statement is executed, the program ends immediately. Any `Integer_Value` may be used, but by convention, 1 is used for a call to `exit` that is caused by an error, and 0 is used in other cases. The `exit` statement is a call to the function `exit`, which is in the library with header file named `cstdlib`. Therefore, any program that uses the `exit` statement must contain the following directives:

```
#include <cstdlib>
using namespace std;
```

(These directives need not be given one immediately after the other. They are placed in the same locations as similar directives we have seen.)

¹UNIX and Windows use 1 for error and 0 for success, but other operating systems may reverse this convention. You should ask your instructor what values to use.

Display 6.2 contains the program from Display 6.1 rewritten to include tests to see if the input and output files were opened successfully. It processes files in exactly the same way as the program in Display 6.1. In particular, assuming that the file `infile.dat` exists and has the contents shown in Display 6.1, the program in Display 6.2 will create the file `outfile.dat` that is shown in Display 6.1. However, if there were something wrong and one of the calls to `open` failed, then the program in Display 6.2 would end and send an appropriate error message to the screen. For example, if there were no file named `infile.dat`, then the call to `in_stream.open` would fail, the program would end, and an error message would be written to the screen.

Notice that we used `cout` to output the error message; this is because we want the error message to go to the screen, as opposed to going to a file. Since this program uses `cout` to output to the screen (as well as doing file I/O), we have added an `include` directive for the header file `iostream`. (Actually, your program does not need to have `#include <iostream>` when the program has `#include <fstream>`, but it causes no problems to include it, and it reminds you that the program is using screen output in addition to file I/O.)

Techniques for File I/O

As we already noted, the operators `>>` and `<<` work the same for streams connected to files as they do for `cin` and `cout`. However, the programming style for file I/O is different from that for I/O using the screen and keyboard. When reading input from the keyboard, you should prompt for input and echo the input, like this:

```
cout << "Enter the number: ";  
cin >> the_number;  
cout << "The number you entered is " << the_number;
```

When your program takes its input from a file, you should not include such prompt lines or echoing of input, because there is nobody there to read and respond to the prompt and echo. When reading input from a file, you must be certain the data in the file is exactly the kind of data the program expects. Your program then simply reads the input file assuming that the data it needs will be there when it is requested. If `in_file` is a stream variable that is connected to an input file and you wish to replace the previous keyboard/screen I/O shown with input from the file connected to `in_file`, then you would replace those three lines with the following line:

```
in_file >> the_number;
```

You may have any number of streams opened for input or for output. Thus, a single program can take input from the keyboard and also take input from one or more files. The same program could send output to the screen and

to one or more files. Alternatively, a program could take all of its input from the keyboard and send output to both the screen and a file. Any combination of input and output streams is allowed. Most of the examples in this book will use `cin` and `cout` to do I/O using the keyboard and screen, but it is easy to modify these programs so that the program takes its input from a file and/or sends its output to a file.

DISPLAY 6.2 File I/O with Checks on open

```

1  //Reads three numbers from the file infile.dat, sums the numbers,
2  //and writes the sum to the file outfile.dat.
3  #include <fstream>
4  #include <iostream>
5  #include <cstdlib>
6  int main( )
7  {
8      using namespace std;
9      ifstream in_stream;
10     ofstream out_stream;
11     in_stream.open("infile.dat");
12     if (in_stream.fail( ))
13     {
14         cout << "Input file opening failed.\n";
15         exit(1);
16     }
17     out_stream.open("outfile.dat");
18     if (out_stream.fail( ))
19     {
20         cout << "Output file opening failed.\n";
21         exit(1);
22     }
23     int first, second, third;
24     in_stream >> first >> second >> third;
25     out_stream << "The sum of the first 3\n"
26                << "numbers in infile.dat\n"
27                << "is " << (first + second + third)
28                << endl;
29     in_stream.close( );
30     out_stream.close( );
31     return 0;
32 }
```

Screen Output (If the file `infile.dat` does not exist)

```
Input file opening failed.
```

Summary of File I/O Statements

In this sample the input comes from a file with the directory name `infile.dat`, and the output goes to a file with the directory name `outfile.dat`.

- Place the following **include** directives in your program file:

```
#include <fstream>    ← For file I/O
#include <iostream>    ← For cout
#include <cstdlib>      ← For exit
```

- Choose a stream name for the input stream (for example, **in_stream**), and declare it to be a variable of type **ifstream**. Choose a stream name for the output file (for example, **out_stream**), and declare it to be of type **ofstream**:

```
using namespace std;
ifstream in_stream;
ofstream out_stream;
```

- Connect each stream to a file using the member function **open** with the external file name as an argument. Remember to use the member function **fail** to test that the call to **open** was successful:

```
in_stream.open("infile.dat");
if (in_stream.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}
out_stream.open("outfile.dat");
if (out_stream.fail( ))
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

- Use the stream **in_stream** to get input from the file **infile.dat** just like you use **cin** to get input from the keyboard. For example:

```
in_stream >> some_variable >> some_other_variable;
```

- Use the stream **out_stream** to send output to the file **outfile.dat** just like you use **cout** to send output to the screen. For example:

```
out_stream << "some_variable = "
            << some_variable << endl;
```

- Close the streams using the function **close**:

```
in_stream.close( );
out_stream.close( );
```

SELF-TEST EXERCISES

1. Suppose you are writing a program that uses a stream called `fin` that will be connected to an input file, and a stream called `fout` that will be connected to an output file. How do you declare `fin` and `fout`? What `include` directive, if any, do you need to place in your program file?
2. Suppose you are continuing to write the program discussed in the previous exercise and you want it to take its input from the file `stuff1.dat` and send its output to the file `stuff2.dat`. What statements do you need to place in your program in order to connect the stream `fin` to the file `stuff1.dat` and to connect the stream `fout` to the file `stuff2.dat`? Be sure to include checks to make sure that the openings were successful.
3. Suppose that you are still writing the same program that we discussed in the previous two exercises and you reach the point at which you no longer need to get input from the file `stuff1.dat` and no longer need to send output to the file `stuff2.dat`. How do you close these files?
4. Suppose you want to change the program in Display 6.1 so that it sends its output to the screen instead of the file `outfile.dat`. (The input should still come from the file `infile.dat`.) What changes do you need to make to the program?
5. What `include` directive do you need to place in your program file if your program uses the function `exit`?
6. Continuing Self-Test Exercise 5, what does `exit(1)` do with its argument?
7. Suppose `b1a` is an object, `dobedo` is a member function of the object `b1a`, and `dobedo` takes one argument of type `int`. How do you write a call to the member function `dobedo` of the object `b1a` using the argument 7?
8. What characteristics of files do ordinary program variables share? What characteristics of files are different from ordinary variables in a program?
9. Name at least three member functions associated with an `iostream` object, and give examples of usage of each.
10. A program has read half of the lines in a file. What must the program do to the file to enable reading the first line a second time?
11. In the text it says “a file has two names.” What are the two names? When is each name used?

Appending to a File (Optional)

When sending output to a file, your code must first use the member function `open` to open a file and connect it to a stream of type `ofstream`. The way we have done that thus far (with a single argument for the file name) always gives an empty file. If a file with the specified name already exists, its old contents are lost. There is an alternative way to open a file so that the output from your program will be appended to the file after any data already in the file.

To append your output to a file named `important.txt`, you would use a two-argument version of `open`, as illustrated by the following:

```
ofstream outStream;  
outStream.open("important.txt", ios::app);
```

If the file `important.txt` does not exist, this will create an empty file with that name to receive your program's output, but if the file already exists, then all the output from your program will be appended to the end of the file so that old data in the file is not lost. This is illustrated in Display 6.3.

DISPLAY 6.3 Appending to a File (Optional) (part 1 of 2)

```
1  //Appends data to the end of the file data.txt.  
2  #include <fstream>  
3  #include <iostream>  
4  
5  int main( )  
6  {  
7      using namespace std;  
8  
9      cout << "Opening data.txt for appending.\n";  
10     ofstream fout;  
11     fout.open("data.txt", ios::app);  
12     if (fout.fail( ))  
13     {  
14         cout << "Input file opening failed.\n";  
15         exit(1);  
16     }  
17  
18     fout << "5 6 pick up sticks.\n"  
19         << "7 8 ain't C++ great!\n";  
20  
21     fout.close( );  
22     cout << "End of appending to file.\n";  
23  
24     return 0;  
25 }
```

(continued)

DISPLAY 6.3 Appending to a File (Optional) *(part 2 of 2)*

*Sample Dialogue***data.txt**

(Before program is run.)

```
1 2 buckle my shoe.  
3 4 shut the door.
```

data.txt

(After program is run.)

```
1 2 buckle my shoe.  
3 4 shut the door.  
5 6 pick up sticks.  
7 8 ain't C++ great!
```

Screen Output

```
Opening data.txt for appending.  
End of appending to file.
```

The second argument `ios::app` is a special constant that is defined in `iostream` and so requires the following include directive:

```
#include <iostream>
```

Your program should also include the following, normally either at the start of the file or at the start of the function body that uses `ios::app`:

```
using namespace std;
```

File Names as Input (Optional)

Thus far, we have written the literal file names for our input and output files into the code of our programs. We did this by giving the file name as the argument to a call to the function `open`, as in the following example:

```
in_stream.open("infile.dat");
```

This can sometimes be inconvenient. For example, the program in Display 6.2 reads numbers from the file `infile.dat` and outputs their sum to the file `outfile.dat`. If you want to perform the same calculation on the numbers in another file named `infile2.dat` and write the sum of these numbers to another file named `outfile2.dat`, then you must change the file names in the two calls to the member function `open` and then recompile your program. A preferable alternative is to write your program so that it asks the user to type in the names of the input and output files. This way your program can use different files each time it is run.

Appending to a File

If you want to append data to a file so that it goes after any existing contents of the file, open the file as follows.

SYNTAX

```
Output_Stream.open(File_Name, ios::app);
```

EXAMPLE

```
ofstream outStream;  
outStream.open("important.txt", ios::app);
```

A file name is a *string* and we will not discuss string handling in detail until Chapter 8. However, it is easy to learn enough about strings so that you can write programs that accept a file name as input. A **string** is just a sequence of characters. We have already used string values in output statements such as the following:

```
cout << "This is a string.";
```

We have also used string values as arguments to the member function `open`. Whenever you write a literal string, as in the `cout` statement shown, you must place the string in double quotes.

In order to read a file name into your program, you need a variable that is capable of holding a string. We discuss the details of strings in Chapter 8, but for now we will cover just enough to store a file name. A variable to hold a string value is declared as in the following example:

```
char file_name[16];
```

This declaration is the same as if you had declared the variable to be of type *char*, except that the variable name is followed by an integer in square brackets that specifies the maximum number of characters you can have in a string stored in the variable. This number must be one greater than the maximum number of characters in the string value. So, in our example, the variable `file_name` can contain any string that contains 15 or fewer characters. The name `file_name` can be replaced by any other identifier (that is not a keyword), and the number 16 can be replaced by any other positive integer.

You can input a string value to a string variable the same way that you input values of other types. For example, consider the following piece of code:

```
cout << "Enter the file name (maximum of 15 characters):\n";  
cin >> file_name;  
cout << "OK, I will edit the file " << file_name << endl;
```

A possible dialogue for this code is

Enter the file name (maximum of 15 characters):
 myfile.dat
 OK, I will edit the file myfile.dat

Once your program has read the name of a file into a string variable, such as the variable `file_name`, it can use this string variable as the argument to the member function `open`. For example, the following will connect the input-file stream `in_stream` to the file whose name is stored in the variable `file_name` (and will use the member function `fail` to check whether the opening was successful):

String variables
as arguments to
`open`

```
ifstream in_stream;
in_stream.open(file_name);
if (in_stream.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

Note that when you use a string variable as an argument to the member function `open`, you do not use any quotes.

In Display 6.4 we have rewritten the program in Display 6.2 so that it takes its input from and sends its output to whatever files the user specifies. The input and output file names are read into the string variables `in_file_name` and `out_file_name` and then these variables are used as the arguments in calls to the member function `open`. Notice the declaration of the string variables. You must include a number in square brackets after each string variable name, as we did in Display 6.4.

String variables are not ordinary variables and cannot be used in all the ways you can use ordinary variables. In particular, you cannot use an assignment statement to change the value of a string variable.

Warning!

6.2 TOOLS FOR STREAM I/O

You shall see them on a beautiful quarto page, where a neat rivulet of text shall meander through a meadow of margin.

RICHARD BRINSLEY SHERIDAN, *The School for Scandal*

Formatting Output with Stream Functions

The layout of a program's output is called the **format** of the output. In C++ you can control the format with commands that determine such details as the number of spaces between items and the number of digits after the decimal point. You already used three output formatting instructions when you learned the formula for outputting dollar amounts of money in the usual way (not in

DISPLAY 6.4 Inputting a File Name (Optional) *(part 1 of 2)*

```

1  //Reads three numbers from the file specified by the user, sums the numbers,
2  //and writes the sum to another file specified by the user.
3  #include <fstream>
4  #include <iostream>
5  #include <cstdlib>
6
7  int main( )
8  {
9      using namespace std;
10     char in_file_name[16], out_file_name[16];
11     ifstream in_stream;
12     ofstream out_stream;
13
14     cout << "I will sum three numbers taken from an input\n"
15           << "file and write the sum to an output file.\n";
16     cout << "Enter the input file name (maximum of 15 characters):\n";
17     cin >> in_file_name;
18     cout << "Enter the output file name (maximum of 15 characters):\n";
19     cin >> out_file_name;
20     cout << "I will read numbers from the file "
21           << in_file_name << " and\n"
22           << "place the sum in the file "
23           << out_file_name << endl;
24
25     in_stream.open(in_file_name);
26     if (in_stream.fail( ))
27     {
28         cout << "Input file opening failed.\n";
29         exit(1);
30     }
31
32     out_stream.open(out_file_name);
33     if (out_stream.fail( ))
34     {
35         cout << "Output file opening failed.\n";
36         exit(1);
37     }
38     int first, second, third;
39     in_stream >> first >> second >> third;
40     out_stream << "The sum of the first 3\n"
41               << "numbers in " << in_file_name << endl
42               << "is " << (first + second + third)
43               << endl;
44
45     in_stream.close( );

```

(continued)

DISPLAY 6.4 Inputting a File Name (Optional) *(part 2 of 2)*

```

46     out_stream.close( );
47
48     cout << "End of Program.\n";
49     return 0;
50 }

```

numbers.dat
(Not changed by program.)

```

1
2
3
4

```

sum.dat
(After program is run.)

```

The sum of the first 3
numbers in numbers.dat
is 6

```

Sample Dialogue

```

I will sum three numbers taken from an input
file and write the sum to an output file.
Enter the input file name (maximum of 15 characters):
numbers.dat
Enter the output file name (maximum of 15 characters):
sum.dat
I will read numbers from the file numbers.dat and
place the sum in the file sum.dat
End of Program.

```

e-notation) with two digits after the decimal point. Before outputting amounts of money, you inserted the following “magic formula” into your program:

```

cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);

```

Now that you’ve learned about object notation for streams, we can explain this magic formula and a few other formatting commands.

The first thing to note is that you can use these formatting commands on any output stream. If your program is sending output to a file that is connected to an output stream called `out_stream`, you can use these same commands to ensure that numbers with a decimal point will be written in the way we normally write amounts of money. Just insert the following in your program:

```

out_stream.setf(ios::fixed);

```

```
out_stream.setf(ios::showpoint);  
out_stream.precision(2);
```

To explain this magic formula, we will consider the instructions in reverse order.

Every output stream has a member function named `precision`. When your program executes a call to `precision` such as the previous one for the stream `out_stream`, then from that point on in your program, any number with a decimal point that is output to that stream will be written with a total of two significant figures, or with two digits after the decimal point, depending on when your compiler was written. The following is some possible output from a compiler that sets two significant digits:

```
23.      2.2e7      2.2      6.9e-1      0.00069
```

The following is some possible output from a compiler that sets two digits after the decimal point:

```
23.56      2.26e7      2.21      0.69      0.69e-4
```

In this book, we assume the compiler sets two digits after the decimal point.

A call to `precision` applies only to the stream named in the call. If your program has another output stream named `out_stream_two`, then the call to `out_stream.precision` affects the output to the stream `out_stream` but has no effect on the stream `out_stream_two`. Of course, you can also call `precision` with the stream `out_stream_two`; you can even specify a different number of digits for the numbers output to the stream `out_stream_two`, as in the following:

```
out_stream_two.precision(3);
```

The other formatting instructions in our magic formula are a bit more complicated than the member function `precision`. We now discuss these other instructions. The following are two calls to the member function `setf` with the stream `out_stream` as the calling object:

```
out_stream.setf(ios::fixed);  
out_stream.setf(ios::showpoint);
```

`setf` is an abbreviation for *set flags*. A **flag** is an instruction to do something in one of two possible ways. If a flag is given as an argument to `setf`, then the flag tells the computer to write output to that stream in some specific way. What it causes the stream to do depends on the flag.

In the previous example, there are two calls to the function `setf`, and these two calls set the two flags `ios::fixed` and `ios::showpoint`. The flag `ios::fixed` causes the stream to output numbers of type *double* in what is called **fixed-point notation**, which is a fancy phrase for the way we normally write numbers. If the flag `ios::fixed` is set (by a call to `setf`), then all floating-point numbers (such as numbers of type *double*) that are output to that stream will be written in ordinary everyday notation, rather than e-notation.

DISPLAY 6.5 Formatting Flags for `setf`

Flag	Meaning	Default
<code>ios::fixed</code>	If this flag is set, floating-point numbers are not written in e-notation. (Setting this flag automatically unsets the flag <code>ios::scientific</code> .)	Not set
<code>ios::scientific</code>	If this flag is set, floating-point numbers are written in e-notation. (Setting this flag automatically unsets the flag <code>ios::fixed</code> .) If neither <code>ios::fixed</code> nor <code>ios::scientific</code> is set, then the system decides how to output each number.	Not set
<code>ios::showpoint</code>	If this flag is set, a decimal point and trailing zeros are always shown for floating-point numbers. If it is not set, a number with all zeros after the decimal point might be output without the decimal point and following zeros.	Not set
<code>ios::showpos</code>	If this flag is set, a plus sign is output before positive integer values.	Not set
<code>ios::right</code>	If this flag is set and some field-width value is given with a call to the member function <code>width</code> , then the next item output will be at the right end of the space specified by <code>width</code> . In other words, any extra blanks are placed <i>before</i> the item output. (Setting this flag automatically unsets the flag <code>ios::left</code> .)	Set
<code>ios::left</code>	If this flag is set and some field-width value is given with a call to the member function <code>width</code> , then the next item output will be at the left end of the space specified by <code>width</code> . In other words, any extra blanks are placed <i>after</i> the item output. (Setting this flag automatically unsets the flag <code>ios::right</code> .)	Not set

The flag `ios::showpoint` tells the stream to always include a decimal point in floating-point numbers, such as numbers of type *double*. So if the number to be output has a value of 2.0, then it will be output as 2.0 and not simply as 2; that is, the output will include the decimal point even if all the digits after the decimal point are 0. Some common flags and the actions they cause are described in Display 6.5.

Another useful flag is `ios::showpos`. If this flag is set for a stream, then positive numbers output to that stream will be written with the plus sign in front of them. If you want a plus sign to appear before positive numbers, insert the following:

```
cout.setf(ios::showpos);
```

Minus signs appear before negative numbers without setting any flags.

One very commonly used formatting function is width. For example, consider the following call to width made by the stream cout:

```
cout << "Start Now";  
cout.width(4);  
cout << 7 << endl;
```

This code causes the following line to appear on the screen:

```
Start Now   7
```

This output has exactly three spaces between the letter 'w' and the number 7. The width function tells the stream how many spaces to use when giving an item as output. In this case the item (namely, the number 7) occupies only one space, and width said to use four spaces, so three of the spaces are blank. If the output requires more space than you specified in the argument to width, then as much additional space as is needed will be used. The entire item is always output, no matter what argument you give to width.

A call to width applies only to the next item that is output. If you want to output 12 numbers, using four spaces to output each number, then you must call width 12 times. If this becomes a nuisance, you may prefer to use the manipulator setw that is described in the next subsection.

Any flag that is set may be unset. To unset a flag, you use the function unsetf. For example, the following will cause your program to stop including plus signs on positive integers that are output to the stream cout:

```
cout.unsetf(ios::showpos);
```

Flag Terminology

Why are the arguments to setf, such as ios::showpoint, called *flags*? And what is meant by the strange notation ios::?

The word **flag** is used for something that can be turned on or off. The origin of the term apparently comes from some phrase similar to “when the flag is up, do it.” Or perhaps the term was “when the flag is down, do it.” Moreover, apparently nobody can recall what the exact originating phrase was because programmers now say “when the flag is set” and that does not conjure up any picture. In any event, when the flag ios::showpoint is set (that is, when it is an argument to setf), the stream that called the setf function will behave as described in Display 6.5; when any other flag is set (that is, is given as an argument to setf), that signals the stream to behave as Display 6.5 specifies for that flag.

The explanation for the notation ios:: is rather mundane for such exotic notation. The ios indicates that the meaning of terms such as fixed or showpoint is the meaning that they have when used with an input or output stream. The notation :: means “use the meaning of what follows the :: in the context of what comes before the ::.” We will say more about this :: notation later in this book.

Manipulators

A **manipulator** is a function that is called in a nontraditional way. In turn, the manipulator function calls a member function. Manipulators are placed after the insertion operator `<<`, just as if the manipulator function call were an item to be output. Like traditional functions, manipulators may or may not have arguments. We have already seen one manipulator, `endl`. In this subsection we will discuss two manipulators called `setw` and `setprecision`.

The manipulator `setw` and the member function `width` (which you have already seen) do exactly the same thing. You call the `setw` manipulator by writing it after the insertion operator `<<`, as if it were to be sent to the output stream, and this in turn calls the member function `width`. For example, the following outputs the numbers 10, 20, and 30, using the field widths specified:

```
cout << "Start" << setw(4) << 10
    << setw(4) << 20 << setw(6) << 30;
```

The preceding statement will produce the following output:

```
Start  10  20   30
```

(There are two spaces before the 10, two spaces before the 20, and four spaces before the 30.)

The manipulator `setprecision` does exactly the same thing as the member function `precision` (which you have already seen). However, a call to `setprecision` is written after the insertion operator `<<`, in a manner similar to how you call the `setw` manipulator. For example, the following outputs the numbers listed using the number of digits after the decimal point that are indicated by the call to `setprecision`:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
    << "$" << 20.5 << endl;
```

The statement above will produce the following output:

```
$10.30
$20.50
```

When you set the number of digits after the decimal point using the manipulator `setprecision`, then just as was the case with the member function `precision`, the setting stays in effect until you reset it to some other number by another call to either `setprecision` or `precision`.

To use either of the manipulators `setw` or `setprecision`, you must include the following directive in your program:

```
#include <iomanip>
```

Your program should also include the following:

```
using namespace std;
```

SELF-TEST EXERCISES

12. What output will be produced when the following lines are executed (assuming the lines are embedded in a complete and correct program with the proper include directives)?

```
cout << "*";  
cout.width(5);  
cout << 123  
    << "*" << 123 << "*" << endl;  
cout << "*" << setw(5) << 123  
    << "*" << 123 << "*" << endl;
```

13. What output will be produced when the following lines are executed (assuming the lines are embedded in a complete and correct program with the proper include directives)?

```
cout << "*" << setw(5) << 123;  
cout.setf(ios::left);  
cout << "*" << setw(5) << 123;  
cout.setf(ios::right);  
cout << "*" << setw(5) << 123 << "*" << endl;
```

14. What output will be produced when the following lines are executed (assuming the lines are embedded in a complete and correct program with the proper include directives)?

```
cout << "*" << setw(5) << 123 << "*" << endl;  
    << 123 << "*" << endl;  
cout.setf(ios::showpos);  
cout << "*" << setw(5) << 123 << "*" << endl;  
    << 123 << "*" << endl;  
cout.unsetf(ios::showpos);  
cout.setf(ios::left);  
cout << "*" << setw(5) << 123 << "*" << endl;  
    << setw(5) << 123 << "*" << endl;
```

15. What output will be sent to the file `stuff.dat` when the following lines are executed (assuming the lines are embedded in a complete and correct program with the proper include directives)?

```
ofstream fout;  
fout.open("stuff.dat");  
fout << "*" << setw(5) << 123 << "*" << endl;  
    << 123 << "*" << endl;  
fout.setf(ios::showpos);  
fout << "*" << setw(5) << 123 << "*" << endl;  
    << 123 << "*" << endl;
```

```
fout.unsetf(ios::showpos);
fout.setf(ios::left);
fout << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

16. What output will be produced when the following line is executed (assuming the line is embedded in a complete and correct program with the proper include directives)?

```
cout << "*" << setw(3) << 12345 << "*" << endl;
```

17. In formatting output, the following flag constants are used with the stream member function `setf`. What effect does each have?

- a. `ios::fixed`
- b. `ios::scientific`
- c. `ios::showpoint`
- d. `ios::showpos`
- e. `ios::right`
- f. `ios::left`

18. Here is a code segment that reads input from `infile.dat` and sends output to `outfile.dat`. What changes are necessary to make the output go to the screen? (The input is still to come from `infile.dat`.)

```
//Problem for Self Test. Copies three int numbers
//between files.
#include <fstream>
int main( )
{
    using namespace std;

    ifstream instream;
    ofstream ostream;

    instream.open("infile.dat");
    ostream.open("outfile.dat");
    int first, second, third;
    instream >> first >> second >> third;
    ostream << "The sum of the first 3" << endl
        << "numbers in infile.dat is " << endl
        << (first + second + third) << endl;
    instream.close( );
    ostream.close( );
    return 0;
}
```

Stream
parameters
must be call-by-
reference

Streams as Arguments to Functions

A stream can be an argument to a function. The only restriction is that the function formal parameter must be call-by-reference. A stream parameter cannot be a call-by-value parameter. For example, the function `make_neat` in Display 6.6 has two stream parameters: one is of type `ifstream` and is for a stream connected to an input file; another is of type `ofstream` and is for a stream connected to an output file. We will discuss the other features of the program in Display 6.6 in the next two subsections.

■ PROGRAMMING TIP Checking for the End of a File

That's all there is, there isn't any more.

ETHEL BARRYMORE (1879–1959)

When you write a program that takes its input from a file, you will often want the program to read all the data in the file. For example, if the file contains numbers, you might want your program to calculate the average of all the numbers in the file. Since you might run the program with different data files at different times, the program cannot assume it knows how many numbers are in the file. You would like to write your program so that it keeps reading numbers from the file until there are no more numbers left to be read. If `in_stream` is a stream connected to the input file, then the algorithm for computing this average can be stated as follows:

```
double next, sum = 0;
int count = 0;
while (There are still numbers to be read)
{
    in_stream >> next;
    sum = sum + next;
    count++;
}
```

The average is `sum / count`.

This algorithm is already almost all C++ code, but we still must express the following test in C++:

(There are still numbers to be read)

Even though it may not look correct at first, one way to express the aforementioned test is the following:

```
(in_stream >> next)
```

The previous algorithm can thus be rewritten as the following C++ code (plus one last line in pseudocode that is not the issue here):

DISPLAY 6.6 Formatting Output (part 1 of 2)

```

1  //Illustrates output formatting instructions.
2  //Reads all the numbers in the file rawdata.dat and writes the numbers
3  //to the screen and to the file neat.dat in a neatly formatted way.
4  #include <iostream>
5  #include <fstream>
6  #include <cstdlib>
7  #include <iomanip>
8  using namespace std;
9  void make_neat(ifstream& messy_file, ofstream& neat_file,
10               int number_after_decimalpoint, int field_width);
11 //Precondition: The streams messy_file and neat_file have been connected
12 //to files using the function open.
13 //Postcondition: The numbers in the file connected to messy_file have been
14 //written to the screen and to the file connected to the stream neat_file.
15 //The numbers are written one per line, in fixed-point notation (that is, not in
16 //e-notation), with number_after_decimalpoint digits after the decimal point;
17 //each number is preceded by a plus or minus sign and each number is in a field
18 //of width field_width. (This function does not close the file.)
19 int main( )
20 {
21     ifstream fin;
22     ofstream fout;
23
24     fin.open("rawdata.dat");
25     if (fin.fail( ))
26     {
27         cout << "Input file opening failed.\n";
28         exit(1);
29     }
30     fout.open("neat.dat");
31     if (fout.fail( ))
32     {
33         cout << "Output file opening failed.\n";
34         exit(1);
35     }
36
37     make_neat(fin, fout, 5, 12);
38
39     fin.close( );
40     fout.close( );
41
42     cout << "End of program.\n";
43     return 0;
44 }
45

```

Needed for setw
Stream parameters must be call-by-reference.

(continued)

DISPLAY 6.6 **Formatting Output** *(part 2 of 2)*

```
46 //Uses iostream, fstream, and iomanip:
47 void make_neat(ifstream& messy_file, ofstream& neat_file,
48               int number_after_decimalpoint, int field_width)
49 {
50     neat_file.setf(ios::fixed);
51     neat_file.setf(ios::showpoint);
52     neat_file.setf(ios::showpos);
53     neat_file.precision(number_after_decimalpoint);
54     cout.setf(ios::fixed);
55     cout.setf(ios::showpoint);
56     cout.setf(ios::showpos);
57     cout.precision(number_after_decimalpoint);
58
59     double next;
60     while (messy_file >> next)
61     {
62         cout << setw(field_width) << next << endl;
63         neat_file << setw(field_width) << next << endl;
64     }
65 }
```

rawdata.dat
(Not changed by program.)

10.37	-9.89897
2.313	-8.950 15.0
7.33333	92.8765
-1.237568432e2	

neat.dat
(After program is run.)

+10.37000
-9.89897
+2.31300
-8.95000
+15.00000
+7.33333
+92.87650
-123.75684

Screen Output

+10.37000
-9.89897
+2.31300
-8.95000
+15.00000
+7.33333
+92.87650
-123.75684
End of program.

```
double next, sum = 0;
int count = 0;
while (in_stream >> next)
{
    sum = sum + next;
    count++;
}
The average is sum / count.
```

Notice that the loop body is not identical to what it was in our pseudocode. Since `in_stream >> next` is now in the Boolean expression, it is no longer in the loop body.

This loop may look a bit peculiar, because `in_stream >> next` is both the way you input a number from the stream `in_stream` and the controlling Boolean expression for the `while` loop. An expression involving the extraction operator `>>` is simultaneously both an action and a Boolean condition.² It is an instruction to take one input number from the input stream, and it is also a Boolean expression that is either satisfied or not. If there is another number to be input, then the number is read and the Boolean expression is satisfied, so the body of the loop is executed one more time. If there are no more numbers to be read in, then nothing is input and the Boolean expression is not satisfied, so the loop ends. In this example the type of the input variable `next` was `double`, but this method of checking for the end of the file works the same way for other data types, such as `int` and `char`. ■

A Note on Namespaces

We have tried to keep our `using` directives local to a function definition. This is an admirable goal, but now we have a problem—functions whose parameter type is in a namespace. In our immediate examples we need the stream type names that are in the namespace `std`. Thus, we need a `using` directive (or something) outside of the function definition body so that C++ will understand the parameter type names, such as `ifstream`. The easiest fix is to simply place one `using` directive at the start of the file (after the `include` directives). We have done this in Display 6.6.

Placing a single `using` directive at the start of a file is the easiest solution to our problem, but many experts would not consider it the best solution, since it would not allow the use of two namespaces that have names in common, and that is the whole purpose of namespaces. At this point we are

² Technically, the Boolean condition works this way: The overloading of operator `>>` for the input stream classes is done with functions associated with the stream. This function is named `operator >>`. The return value of this operator function is an input stream reference (`istream&` or `ifstream&`). A function is provided that automatically converts the stream reference to a `bool` value. The resulting value is `true` if the stream is able to extract data, and `false` otherwise.

only using the namespace `std`,³ so there is no problem. In Chapter 12, we will teach you another way around this problem with parameters and namespaces. This other approach will allow you to use any kinds of multiple namespaces.

Many programmers prefer to place *using* directives at the start of the program file. For example, consider the following *using* directive:

```
using namespace std;
```

Many of the programs in this book do not place this *using* directive at the start of the program file. Instead, this *using* directive is placed at the start of each function definition that needs the namespace `std` (immediately after the opening brace). An example of this is shown in Display 6.3. An even better example is shown in Display 5.11. All of the programs that have appeared so far in this book, and almost all programs that follow, would behave exactly the same if there were just one *using* directive for the namespace `std` and that one *using* directive were placed immediately after the *include* directives, as in Display 6.6. For the namespace `std`, the *using* directive can safely be placed at the start of the file (in almost all cases). For some other namespaces, a single *using* directive will not always suffice, but you will not see any of these cases for some time.

We advocate placing the *using* directives inside function definitions (or inside some other small units of code) so that it does not interfere with any other possible *using* directives. This trains you to use namespaces correctly in preparation for when you write more complicated code later in your programming career. In the meantime, we sometimes violate this rule ourselves when following the rule becomes too burdensome to the other issues we are discussing. If you are taking a course, do whatever your instructor requires. Otherwise, you have some latitude in where you place your *using* directives.

PROGRAMMING EXAMPLE

Cleaning Up a File Format

The program in Display 6.6 takes its input from the file `rawdata.dat` and writes its output, in a neat format, both to the screen and to the file `neat.dat`. The program copies numbers from the file `rawdata.dat` to the file `neat.dat`, but it uses formatting instructions to write them in a neat way. The numbers are written one per line in a field of width 12, which means that each number is preceded by enough blanks so that the blanks plus the number occupy 12 spaces. The numbers are written in ordinary notation; that is, they are not written in e-notation. Each number is written with five digits after the decimal

³ We are actually using two namespaces: the namespace `std` and a namespace called the global namespace, which is a namespace that consists of all names that are not in some other namespace. But this technical detail is not a big issue to us now.

point and with a plus or minus sign. The output to the screen is the same as the output to the file `neat.dat`, except that the screen output has one extra line that announces that the program is ending. The program uses a function, named `make_neat`, that has formal parameters for the input-file stream and the output-file stream.

SELF-TEST EXERCISES

19. What output will be produced when the following lines are executed, assuming the file `list.dat` contains the data shown (and assuming the lines are embedded in a complete and correct program with the proper include directives)?

```
ifstream ins;
ins.open("list.dat");
int count = 0, next;
while (ins >> next)
{
    count++;
    cout << next << endl;
}
ins.close( );
cout << count;
```

The file `list.dat` contains the following three numbers (and nothing more)

1	2
3	

20. Write the definition for a *void* function called `to_screen`. The function `to_screen` has one formal parameter called `file_stream`, which is of type `ifstream`. The precondition and postcondition for the function are as follows:

```
//Precondition: The stream file_stream has been connected
//to a file with a call to the member function open. The
//file contains a list of integers (and nothing else).
//Postcondition: The numbers in the file connected to
//file_stream have been written to the screen one per line.
//(This function does not close the file.)
```

21. (This exercise is for those who have studied the optional section entitled "File Names as Input.") Suppose you are given the following string variable declaration and input statement:

```
#include <iostream>
using namespace std;
// ...
char name[21];
cout >> name;
```

Suppose this code segment is embedded in a correct program. What is the longest name that can be entered into the string variable `name`?

6.3 CHARACTER I/O

Polonius: What do you read, my lord?

Hamlet: Words, words, words.

WILLIAM SHAKESPEARE, *Hamlet*

All data is input and output as character data. When your program outputs the number 10, it is really the two characters '1' and '0' that are output. Similarly, when the user wants to type in the number 10, he or she types in the character '1' followed by the character '0'. Whether the computer interprets this 10 as two characters or as the number 10 depends on how your program is written. But, however your program is written, the computer hardware is always reading the characters '1' and '0', not the number 10. This conversion between characters and numbers is usually done automatically so that you need not think about such detail. Sometimes, however, all this automatic help gets in the way. Therefore, C++ provides some low-level facilities for input and output of character data. These low-level facilities include no automatic conversions. This allows you to bypass the automatic facilities and do input/output in absolutely any way you want. You could even write input and output functions that read and write numbers in Roman numeral notation, if you wanted to be so perverse.

The Member Functions `get` and `put`

The function `get` allows your program to read in one character of input and store it in a variable of type `char`. Every input stream, whether it is an input-file stream or the stream `cin`, has `get` as a member function. We will describe `get` as a member function of the stream `cin`, but it behaves in exactly the same way for input-file streams as it does for `cin`, so you can apply all that we say about `get` to input-file streams as well as to the stream `cin`.

Before now, we have used `cin` with the extraction operator `>>` in order to read a character of input (or any other input, for that matter). When you use the extraction operator `>>`, as we have been doing, some things are done for you automatically, such as skipping blanks. With the member function `get`, nothing is done automatically. If you want, for example, to skip over blanks using `cin.get`, you must write code to read and discard the blanks.

The member function `get` takes one argument, which should be a variable of type `char`. That argument receives the input character that is read from the input stream. For example, the following reads in the next input character from the keyboard and stores it in the variable `next_symbol`:

```
char next_symbol;  
cin.get(next_symbol);
```

It is important to note that your program can read any character in this way. If the next input character is a blank, this code will not skip over the blank, but will read the blank and set the value of `next_symbol` equal to the blank character. If the next character is the new-line character `'\n'`, that is, if the program has just reached the end of an input line, then the call to `cin.get` shown earlier sets the value of `next_symbol` equal to `'\n'`.

Reading blanks
and `'\n'`

Although we write it as two symbols, `'\n'` is just a single character in C++. With the member function `get`, the character `'\n'` can be input and output just like any other character. For example, suppose your program contains the following code:

```
char c1, c2, c3;  
cin.get(c1);  
cin.get(c2);  
cin.get(c3);
```

and suppose you type in the following two lines of input to be read by this code:

```
AB  
CD
```

That is, suppose you type `AB` followed by Return and then `CD` followed by Return. As you would expect, the value of `c1` is set to `'A'` and the value of `c2` is set to `'B'`. That's nothing new. But when this code fills the variable `c3`, things are different from what they would be if you had used the extraction operator `>>` instead of the member function `get`. When this code is executed on the input we showed, the value of `c3` is set to `'\n'`; that is, the value of `c3` is set equal to the new-line character. The variable `c3` is not set equal to `'C'`.

One thing you can do with the member function `get` is to have your program detect the end of a line. The following loop will read a line of input and stop after passing the new-line character `'\n'`. Then, any subsequent input will be read from the beginning of the next line. For this first example, we have simply echoed the input, but the same technique would allow you to do whatever you want with the input:

Detecting the end
of an input line

```
cout << "Enter a line of input and I will echo it:\n";  
char symbol;  
do  
{  
    cin.get(symbol);
```

```

        cout << symbol;
    } while (symbol != '\n');
    cout << "That's all for this demonstration.";

```

This loop will read any line of input and echo it exactly, including blanks. The following is a sample dialogue produced by this code:

```

Enter a line of input and I will echo it:
Do Be Do 1 2    34
Do Be Do 1 2    34
That's all for this demonstration.

```

Notice that the new-line character '\n' is both read and output. Since '\n' is output, the string that begins with the word "That's" is on a new line.

The Member Function `get`

Every input stream has a member function named `get` that can be used to read one character of input. Unlike the extraction operator `>>`, `get` reads the next input character, no matter what that character is. In particular, `get` reads a blank or the new-line character '\n' if either of these is the next input character. The function `get` takes one argument, which should be a variable of type `char`. When `get` is called, the next input character is read and the argument variable (called *Char_Variable* below) has its value set equal to this input character.

SYNTAX

```
Input_Stream.get(Char_Variable);
```

EXAMPLE

```

char next_symbol;
cin.get(next_symbol);

```

If you wish to use `get` to read from a file, you use an input-file stream in place of the stream `cin`. For example, if `in_stream` is an input stream for a file, then the following reads one character from the input file and places the character in the `char` variable `next_symbol`:

```
in_stream.get(next_symbol);
```

Before you can use `get` with an input-file stream such as `in_stream`, your program must first connect the stream to the input file with a call to `open`.

(continued)

'\n' and "\n"

'\n' and "\n" sometimes seem like the same thing. In a `cout` statement, they produce the same effect, but they cannot be used interchangeably in all situations. '\n' is a value of type *char* and can be stored in a variable of type *char*. On the other hand, "\n" is a string that happens to be made up of exactly one character. Thus, "\n" is not of type *char* and cannot be stored in a variable of type *char*.

The member function `put` is analogous to the member function `get` except that it is used for output rather than input. `put` allows your program to output one character. The member function `put` takes one argument, which should be an expression of type *char*, such as a constant or a variable of type *char*. The value of the argument is output to the stream when the function is called. For example, the following outputs the letter 'a' to the screen:

```
cout.put('a');
```

The function `cout.put` does not allow you to do anything you could not do by using the methods we discussed previously, but we include it for completeness.

If your program uses `cin.get` or `cout.put`, then just as with other uses of `cin` and `cout`, your program should include the following directive:

```
#include <iostream>
```

Similarly, if your program uses `get` for an input-file stream or `put` for an output-file stream, then just as with any other file I/O, your program should contain the following directive:

```
#include <fstream>
```

The Member Function `put`

Every output stream has a member function named `put`, which takes one argument which should be an expression of type *char*. When the member function `put` is called, the value of its argument (called *Char_Expression* below) is output to the output stream.

SYNTAX

```
Output_Stream.put(Char_Expression);
```

(continued)

EXAMPLES

```
cout.put(next_symbol);  
cout.put('a');
```

If you wish to use `put` to output to a file, you use an output-file stream in place of the stream `cout`. For example, if `out_stream` is an output stream for a file, then the following will output the character 'Z' to the file connected to `out_stream`:

```
out_stream.put('Z');
```

Before you can use `put` with an output-file stream, such as `out_stream`, your program must first connect the stream to the output file with a call to the member function `open`.

When using either of these `include` directives, your program must also include the following:

```
using namespace std;
```

The `putback` Member Function (*Optional*)

Sometimes your program needs to know the next character in the input stream. However, after reading the next character, it might turn out that you do not want to process that character and so you would like to simply put it back in the input stream. For example, if you want your program to read up to *but not including* the first blank it encounters in an input stream, then your program must read that first blank in order to know when to stop reading—but then that blank is no longer in the stream. Some other part of your program might need to read and process this blank. There are a number of ways to deal with this sort of situation, but the easiest is to use the member function `putback`. The function `putback` is a member of every input stream. It takes one argument of type `char` and it places the value of that argument back in the input stream. The argument can be any expression that evaluates to a value of type `char`.

For example, the following code will read characters from the file connected to the input stream `fin` and write them to the file connected to the output stream `fout`. The code reads characters up to, but not including, the first blank it encounters.

```
fin.get(next);  
while (next != ' ' )  
{  
    fout.put(next);  
    fin.get(next);  
}  
fin.putback(next);
```

Notice that after this code is executed, the blank that was read is still in the input stream `fin`, because the code puts it back after reading it.

Notice that `putback` places a character in an *input* stream, while `put` places a character in an *output* stream. The character that is put back into the input stream with the member function `putback` need not be the last character read; it can be any character you wish. If you put back a character other than the last character read, the text in the input file will not be changed by `putback`, although your program will behave as if the text in the input file had been changed.

PROGRAMMING EXAMPLE

Checking Input

If a user enters incorrect input, the entire run of the program can become worthless. To ensure that your program is not hampered by incorrect input, you should use input functions that allow the user to reenter input until the input is correct. The function `get_int` in Display 6.7 asks the user whether the input is correct and asks for a new value if the user says the input is incorrect. The program in Display 6.7 is just a driver program to test the function `get_int`, but the function, or one very similar to it, can be used in just about any kind of program that takes its input from the keyboard.

Notice the call to the function `new_line()`. The function `new_line` reads all the characters on the remainder of the current line but does nothing with them. This amounts to discarding the remainder of the line. Thus, if the user types in No, then the program reads the first letter, which is N, and then calls the function `new_line`, which discards the rest of the input line. This means that if the user types 75 on the next input line, as shown in the sample dialogue, the program will read the number 75 and will not attempt to read the letter o in the word No. If the program did not include a call to the function `new_line`, then the next item read would be the o in the line containing No instead of the number 75 on the following line.

DISPLAY 6.7 Checking Input (part 1 of 2)

```
1  //Program to demonstrate the functions new_line and get_input.
2  #include <iostream>
3  using namespace std;
4
5  void new_line( );
6  //Discards all the input remaining on the current input line.
7  //Also discards the '\n' at the end of the line.
8  //This version works only for input from the keyboard.
9
10 void get_int(int& number);
11 //Postcondition: The variable number has been
```

(continued)

DISPLAY 6.7 Checking Input *(part 2 of 2)*

```

12  //given a value that the user approves of.
13
14
15  int main( )
16  {
17      int n;
18
19      get_int(n);
20      cout << "Final value read in = " << n << endl
21          << "End of demonstration.\n";
22      return 0;
23  }
24
25
26  //Uses iostream:
27  void new_line( )
28  {
29      char symbol;
30      do
31      {
32          cin.get(symbol);
33      } while (symbol != '\n');
34  }
35  //Uses iostream:
36  void get_int(int& number)
37  {
38      char ans;
39      do
40      {
41          cout << "Enter input number: ";
42          cin >> number;
43          cout << "You entered " << number
44              << ". Is that correct? (yes/no): ";
45          cin >> ans;
46          new_line( );
47      } while ((ans != 'Y') && (ans != 'y'));
48  }

```

Sample Dialogue

```

Enter input number: 57
You entered 57. Is that correct? (yes/no): No
Enter input number: 75
You entered 75. Is that correct? (yes/no): yes
Final value read in = 75
End of demonstration.

```

Notice the Boolean expression that ends the *do-while* loop in the function `get_int`. If the input is not correct, the user is supposed to type No (or some variant such as no), which will cause one more iteration of the loop. However, rather than checking to see if the user types a word that starts with 'N', the *do-while* loop checks to see if the first letter of the user's response is *not* equal to 'Y' (and not equal to the lowercase version of 'Y'). As long as the user makes no mistakes and responds with some form of Yes or No, but never with anything else, then checking for No or checking for not being Yes are the same thing. However, since the user might respond in some other way, checking for not being Yes is safer. To see why this is safer, suppose the user makes a mistake in entering the input number. The computer echoes the number and asks if it is correct. The user should type in No, but suppose the user makes a mistake and types in Bo, which is not unlikely since 'B' is right next to 'N' on the keyboard. Since 'B' is not equal to 'Y', the body of the *do-while* loop will be executed, and the user will be given a chance to reenter the input.

When in doubt,
enter the input
again

But, what happens if the correct response is Yes and the user mistakenly enters something that begins with a letter other than 'Y' or 'y'? In that case, the loop should not iterate, but it does iterate one extra time. This is a mistake, but not nearly as bad a mistake as the one discussed in the last paragraph. It means the user must type in the input number one extra time, but it does not waste the entire run of the program. When checking input, it is better to risk an extra loop iteration than to risk proceeding with incorrect input.

PITFALL Unexpected '\n' in Input

When using the member function `get`, you must account for every character of input, even the characters you do not think of as being symbols, such as blanks and the new-line character '\n'. A common problem when using `get` is forgetting to dispose of the '\n' that ends every input line. If there is a new-line character in the input stream that is not read (and usually discarded), then when your program next expects to read a "real" symbol using the member function `get`, it will instead read the character '\n'. To clear the input stream of any leftover '\n' characters, you can use the function `new_line`, which we defined in Display 6.7. Let's look at a concrete example.

It is legal to mix the different forms of `cin`. For example, the following is legal:

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

However, this mixing can produce problems, as illustrated by the following dialogue:

```
Enter a number:  
21  
Now enter a letter:  
A
```

With this dialogue, the value of `number` will be 21, as you expect. However, if you expect the value of the variable `symbol` to be 'A', you will be disappointed. The value given to `symbol` is '\n'. After reading the number 21, the next character in the input stream is the new-line character, '\n', and so that is read next. Remember, `get` does not skip over line breaks and spaces. (In fact, depending on what is in the rest of the program, you may not even get a chance to type in the A. Once the variable `symbol` is filled with the character '\n', the program proceeds to whatever statement is next in the program. If the next statement sends output to the screen, the screen will be filled with output before you get a chance to type in the A.)

Either of the following rewritings of the previous code will cause the previous dialogue to fill the variable `number` with 21 and fill the variable `symbol` with 'A':

```
cout << "Enter a number:\n";  
int number;  
cin >> number;  
cout << "Now enter a letter:\n";  
char symbol;  
cin >> symbol;
```

Alternatively, you can use the function `new_line`, defined in Display 6.7, as follows:

```
cout << "Enter a number:\n";  
int number;  
cin >> number;  
new_line( );  
cout << "Now enter a letter:\n";  
char symbol;  
cin.get(symbol);
```

As this second rewrite indicates, you can mix the two forms of `cin` and have your program work correctly, but it does require some extra care. ■

Making Stream Parameters Versatile

If you want to define a function that takes an input stream as an argument and you want that argument to be `cin` in some cases and an input-file stream in other cases, then use a formal parameter of type `istream` (without an `f`). However, an input-file stream, even if used as an argument of type `istream`, must still be declared to be of type `ifstream` (with an `f`).

(continued)

Similarly, if you want to define a function that takes an output stream as an argument and you want that argument to be `cout` in some cases and an output-file stream in other cases, then use a formal parameter of type `ostream`. However, an output-file stream, even if used as an argument of type `ostream`, must still be declared to be of type `ofstream`. You cannot open or close a stream parameter of type `istream` or `ostream`. Open these objects before passing them to your function and close them after the call.

PROGRAMMING EXAMPLE

Another *new_line* Function

As another example of how you can make a stream function more versatile, consider the function `new_line` in Display 6.7. That function works only for input from the keyboard, which is input from the predefined stream `cin`. The function `new_line` in Display 6.7 has no arguments. Below we have rewritten the function `new_line` so that it has a formal parameter of type `istream` for the input stream:

```
//Uses istream:
void new_line(istream& in_stream)
{
    char symbol;
    do
    {
        in_stream.get(symbol);
    } while (symbol != '\n');
}
```

Now, suppose your program contains this new version of the function `new_line`. If your program is taking input from an input stream called `fin` (which is connected to an input file), the following will discard all the input left on the line currently being read from the input file:

```
new_line(fin);
```

On the other hand, if your program is also reading some input from the keyboard, the following will discard the remainder of the input line that was typed in at the keyboard:

```
new_line(cin);
```

If your program has only the rewritten version of `new_line` above, which takes a stream argument such as `fin` or `cin`, you must always give the stream name, even if the stream name is `cin`. But thanks to overloading, you can have both versions of the function `new_line` in the same program: the version with

Using both
versions of
`new_line`

no arguments that is given in Display 6.7 and the version with one argument of type `istream` that we just defined. In a program with both definitions of `new_line`, the following two calls are equivalent:

```
new_line(cin);
```

and

```
new_line( );
```

You do not really need two versions of the function `new_line`. The version with one argument of type `istream` can serve all your needs. However, many programmers find it convenient to have a version with no arguments for keyboard input, since keyboard input is used so frequently.



VideoNote
Default Arguments

Default Arguments for Functions (*Optional*)

An alternative to having two versions of the `new_line` function is to use **default arguments**. In the following code, we have rewritten the `new_line` function a third time:

```
//Uses istream:
void new_line(istream& in_stream = cin)
{
    char symbol;
    do
    {
        in_stream.get(symbol);
    } while (symbol != '\n');
}
```

If we call this function as

```
new_line( );
```

the formal parameter takes the default argument `cin`. If we call this as

```
new_line(fin);
```

the formal parameter takes the argument provided in the call to `fin`. This facility is available to us with any argument type and any number of arguments.

If some parameters are provided default arguments and some are not, the formal parameters with default arguments must all be together at the end of the argument list. If you provide several defaults and several nondefault arguments, the call may provide either as few arguments as there are nondefault arguments or more arguments, up to the number of parameters. The arguments will be applied to the parameters without default arguments in order, and then will be applied to the parameters with default arguments up to the number of parameters.

Here is an example:

```
//To test default argument behavior
//Uses iostream
void default_args(int arg1, int arg2, int arg3 = -3,
                  int arg4 = -4)
{
    cout << arg1 << ' ' << arg2 << ' ' << arg3 << ' ' << arg4
        << endl;
}
```

Calls to this may be made with two, three, or four arguments. For example, the call

```
default_args(5, 6);
```

supplies the nondefault arguments and uses the two default arguments. The output is

```
5 6 -3 -4
```

Next, consider

```
default_args(6, 7, 8);
```

This call supplies the nondefault arguments and the first default argument, and the last argument uses the default. This call gives the following output:

```
6 7 8 -4
```

The call

```
default_args(5, 6, 7, 8);
```

assigns all the arguments from the argument list and gives the following output:

```
5 6 7 8
```

SELF-TEST EXERCISES

22. Suppose `c` is a variable of type `char`. What is the difference between the following two statements?

```
cin >> c;
```

and

```
cin.get(c);
```

23. Suppose `c` is a variable of type `char`. What is the difference between the following two statements?

```
cout << c;
```

and

```
cout.put(c);
```

24. (This question is for those who have read the optional section “The putback Member Function.”) The putback member function “puts back” a symbol into an input stream. Does the symbol that is put back have to be the last symbol input from the stream? For example, if your program reads an 'a' from the input stream, can it use the putback function to put back a 'b', or can it only put back an 'a'?

25. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
char c1, c2, c3, c4;
cout << "Enter a line of input:\n";
cin.get(c1);
cin.get(c2);
cin.get(c3);
cin.get(c4);
cout << c1 << c2 << c3 << c4 << "END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:
a b c d e f g

26. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin.get(next);
while (next != '\n')
{
    if ((count % 2) == 0)
        cout << next;
    count++;
    cin.get(next);
}
```

True if count is even →

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:
abcdef gh

27. Suppose that the program described in Self-Test Exercise 26 is run and the dialogue begins as follows (instead of beginning as shown in Self-Test Exercise 26). What will be the next line of output?

Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11

28. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin >> next;
while (next != '\n')
{
    if ((count % 2) == 0)
        cout << next;
    count++;
    cin >> next;
}
```

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11

29. Define a function called `copy_char` that takes one argument that is an input stream. When called, `copy_char` will read one character of input from the input stream given as its argument and will write that character to the screen. You should be able to call your function using either `cin` or an input-file stream as the argument to your function `copy_char`. (If the argument is an input-file stream, then the stream is connected to a file before the function is called, so `copy_char` will not open or close any files.) For example, the first of the following two calls to `copy_char` will copy a character from the file `stuff.dat` to the screen, and the second will copy a character from the keyboard to the screen:

```
ifstream fin;
fin.open("stuff.dat");
copy_char(fin);
copy_char(cin);
```

30. Define a function called `copy_line` that takes one argument that is an input stream. When called, `copy_line` reads one line of input from the input stream given as its argument and writes that line to the screen. You should be able to call your function using either `cin` or an input-file stream as the argument to your function `copy_line`. (If the argument

is an input-file stream, then the stream is connected to a file before the function is called, so `copy_line` will not open or close any files.) For example, the first of the following two calls to `copy_line` will copy a line from the file `stuff.dat` to the screen, and the second will copy a line from the keyboard to the screen:

```
ifstream fin;
fin.open("stuff.dat");
copy_line(fin);
copy_line(cin);
```

31. Define a function called `send_line` that takes one argument that is an output stream. When called, `send_line` reads one line of input from the keyboard and outputs the line to the output stream given as its argument. You should be able to call your function using either `cout` or an output-file stream as the argument to your function `send_line`. (If the argument is an output-file stream, then the stream is connected to a file before the function is called, so `send_line` will not open or close any files.) For example, the first of the following calls to `send_line` copies a line from the keyboard to the file `morestuff.dat`, and the second copies a line from the keyboard to the screen:

```
ofstream fout;
fout.open("morestuff.dat");
cout << "Enter 2 lines of input:\n";
send_line(fout);
send_line(cout);
```

32. (This exercise is for those who have studied the optional section on default arguments.) What output does the following function provide in response to the following calls?

```
void func(double x, double y = 1.1, double z = 2.3)
{
    cout << x << " " << y << " " << z << endl;
}
```

Calls:

- a. `func(2.0);`
- b. `func(2.0, 3.0);`
- c. `func(2.0, 3.0, 4.0);`

33. (This exercise is for those who have studied the optional section on default arguments.) Write several functions that overload the function name to get the same effect as all the calls in the default function arguments in the previous Self-Test Exercise.

The eof Member Function

Every input-file stream has a member function called `eof` that can be used to determine when all of the file has been read and there is no more input left for the program. This is the second technique we have presented for determining when a program has read everything in a file.

The letters `eof` stand for *end of file*, and `eof` is normally pronounced by saying the three letters e-o-f. The function `eof` takes no arguments, so if the input stream is called `fin`, then a call to the function `eof` is written

```
fin.eof( )
```

This is a Boolean expression that can be used to control a *while* loop, a *do-while* loop, or an *if-else* statement. This expression is satisfied (that is, is *true*) if the program has read past the end of the input file; otherwise, the expression above is not satisfied (that is, is *false*).

Since we usually want to test that we are *not* at the end of a file, a call to the member function `eof` is typically used with a *not* in front of it. Recall that in C++ the symbol `!` is used to express *not*. For example, consider the following statement:

eof is usually used with "not"

```
if (! fin.eof( ))
    cout << "Not done yet.";
else
    cout << "End of the file.";
```

The Boolean expression after the *if* means "not at the end of the file connected to `fin`." Thus, the *if-else* statement above will output the following to the screen:

```
Not done yet.
```

provided the program has not yet read past the end of the file that is connected to the stream `fin`. The *if-else* statement will output the following, if the program has read beyond the end of the file:

```
End of the file.
```

As another example of using the `eof` member function, suppose that the input stream `in_stream` has been connected to an input file with a call to `open`. Then the entire contents of the file can be written to the screen with the following *while* loop:

Ending an input loop with the eof function

```
in_stream.get(next);
while (! in_stream.eof( ))
{
    cout << next;
    in_stream.get(next);
}
```

If you prefer, you can use `cout.put(next)` here.

This *while* loop reads each character from the input file into the *char* variable *next* using the member function *get*, and then writes the character to the screen. After the program has passed the end of the file, the value of *in_stream.eof()* changes from *false* to *true*. So,

```
(! in_stream.eof( ))
```

changes from *true* to *false* and the loop ends.

Notice that *in_stream.eof()* does not become *true* until the program attempts to read one character beyond the end of the file. For example, suppose the file contains the following (without any new-line after the *c*):

```
ab
c
```

This is actually the following list of four characters:

```
ab<the new-line character '\n'>c
```

This loop reads an 'a' and writes it to the screen, then reads a 'b' and writes it to the screen, then reads the new-line character '\n' and writes it to the screen, and then reads a 'c' and writes it to the screen. At that point the loop will have read all the characters in the file. However, *in_stream.eof()* will still be *false*. The value of *in_stream.eof()* will not change from *false* to *true* until the program tries to read one more character. That is why the *while* loop ends with *in_stream.get(next)*. The loop needs to read one extra character in order to end the loop.

There is a special end-of-file marker at the end of a file. The member function *eof* does not change from *false* to *true* until this end-of-file marker is read. That's why the example *while* loop could read one character beyond what you think of as the last character in the file. However, this end-of-file marker is not an ordinary character and should not be manipulated like an ordinary character. You can read this end-of-file marker but you should not write it out again. If you write out the end-of-file marker, the result is unpredictable. The system automatically places this end-of-file marker at the end of each file for you.

The next Programming Example uses the *eof* member function to determine when the program has read the entire input file.

You now have two methods for detecting the end of a file. You can use the *eof* member function or you can use the method we described in the Programming Tip entitled "Checking for the End of a File." In most situations you can use either method, but many programmers use the two different methods in different situations. If you do not have any other reason to prefer one of these two methods, then use the following general rule: Use the *eof* member function when you are treating the input as text and reading the input with the *get* member function; use the other method when you are processing numeric data.

Deciding how to
test for the end
of an input file

SELF-TEST EXERCISES

34. Suppose `ins` is a file input stream that has been connected to a file with the member function `open`. Suppose your program has just read the last character in the file. At this point, would `ins.eof()` evaluate to *true* or *false*?
35. Write the definition for a *void* function called `text_to_screen` that has one formal parameter called `file_stream` that is of type `ifstream`. The precondition and postcondition for the function are as follows:

```
//Precondition: The stream file_stream has been connected  
//to a file with a call to the member function open.  
//Postcondition: The contents of the file connected to  
//file_stream have been copied to the screen character by  
//character, so that the screen output is the same as the  
//contents of the text in the file.  
//(This function does not close the file.)
```

PROGRAMMING EXAMPLE

Editing a Text File

The program discussed here is a very simple example of text editing applied to files. It might be used by a software firm to update its advertising literature. The firm has been marketing compilers for the C programming language and has recently introduced a line of C++ compilers. This program can be used to automatically generate C++ advertising material from the existing C advertising material. The program takes its input from a file that contains advertising copy that says good things about C and writes similar advertising copy about C++ in another file. The file that contains the C advertising copy is called `cad.dat`, and the new file that receives the C++ advertising copy is called `cp1usad.dat`. The program is shown in Display 6.8.

The program simply reads every character in the file `cad.dat` and copies the characters to the file `cp1usad.dat`. Every character is copied unchanged, except that when the uppercase letter 'C' is read from the input file, the program writes the string "C++" to the output file. This program assumes that whenever the letter 'C' occurs in the input file, it names the C programming language; thus, this change is exactly what is needed to produce the updated advertising copy.

Notice that the line breaks are preserved when the program reads characters from the input file and writes the characters to the output file. The new-line character '\n' is treated just like any other character. It is read from the input file with the member function `get`, and it is written to the output file using the insertion operator `<<`. We must use the member function `get` to

read the input. If we instead use the extraction operator `>>` to read the input, the program would skip over all the whitespace, which means that none of the blanks and none of the new-line characters `'\n'` would be read from the input file, so they would not be copied to the output file.

Also notice that the member function `eof` is used to detect the end of the input file and end the *while* loop.

Predefined Character Functions

In text processing, you often want to convert lowercase letters to uppercase or vice versa. The predefined function `toupper` can be used to convert a lowercase letter to an uppercase letter. For example, `toupper('a')` returns `'A'`. If the argument to the function `toupper` is anything other than a lowercase letter, then `toupper` simply returns the argument unchanged. So `toupper('A')` also returns `'A'`. The function `tolower` is similar except that it converts an uppercase letter to its lowercase version.

The functions `toupper` and `tolower` are in the library with the header file `cctype`, so any program that uses these functions, or any other functions in this library, must contain the following `include` directive:

```
#include <cctype>
```

DISPLAY 6.8 Editing a File of Text (part 1 of 2)

```
1  //Program to create a file called cplusplus.dat that is identical to the file
2  //cad.dat, except that all occurrences of 'C' are replaced by "C++".
3  //Assumes that the uppercase letter 'C' does not occur in cad.dat except
4  //as the name of the C programming language.
5  #include <fstream>
6  #include <iostream>
7  #include <cstdlib>
8  using namespace std;
9  void add_plus_plus(istream& in_stream, ostream& out_stream);
10 //Precondition: in_stream has been connected to an input file with open.
11 //out_stream has been connected to an output file with open.
12 //Postcondition: The contents of the file connected to in_stream have been
13 //copied into the file connected to out_stream, but with each 'C' replaced
14 //by "C++". (The files are not closed by this function.)
15 int main( )
16 {
17     ifstream fin;
18     ofstream fout;
19     cout << "Begin editing files.\n";
20     fin.open("cad.dat");
21     if (fin.fail( ))
22     {
```

(continued)

DISPLAY 6.8 Editing a File of Text (*part 2 of 2*)

```

23         cout << "Input file opening failed.\n";
24         exit(1);
25     }
26     fout.open("cplusad.dat");
27     if (fout.fail( ))
28     {
29         cout << "Output file opening failed.\n";
30         exit(1);
31     }
32     add_plus_plus(fin, fout);
33     fin.close( );
34     fout.close( );
35     cout << "End of editing files.\n";
36     return 0;
37 }
38
39 void add_plus_plus(istream& in_stream, ostream& out_stream)
40 {
41     char next;
42     in_stream.get(next);
43     while (! in_stream.eof( ))
44     {
45         if (next == 'C')
46             out_stream << "C++";
47         else
48             out_stream << next;
49         in_stream.get(next);
50     }
51 }

```

cad.dat

(Not changed by program.)

C is one of the world's most modern programming languages.
There is no language as versatile as C, and C is fun to use.

cplusad.dat

(After program is run.)

C++ is one of the world's most modern programming languages.
There is no language as versatile as C++, and C++ is fun to use.

Screen Output

```

Begin editing files.
End of editing files.

```

Display 6.9 contains descriptions of some of the most commonly used functions in the library `cctype`.

The function `isspace` returns *true* if its argument is a *whitespace* character. If the argument to `isspace` is not a whitespace character, then `isspace` returns *false*. Thus, `isspace('')` returns *true* and `isspace('a')` returns *false*.

For example, the following code reads a sentence terminated with a period and echoes the string with all whitespace characters replaced with the symbol '-':

```
char next;
do
{
    cin.get(next);
    if (isspace(next)) ← True if the character in
                        next is whitespace
        cout << '-';
    else
        cout << next;
} while (next != '.');
```

For example, if the code above is given the following input:

Ahhdo be do.

then it will produce the following output:

Ahhdo--be--do.

■ PITFALL toupper and tolower Return Values

In many ways, C++ considers characters to be whole numbers, similar to the numbers of type `int`. Each character is assigned a number, and when the character is stored in a variable of type `char`, it is this number that is placed in the computer's memory. In C++ you can use a value of type `char` as a number—for example, by placing it in a variable of type `int`. You can also store a number of type `int` in a variable of type `char` (provided the number is not too large). Thus, the type `char` can be used as the type for characters or as a type for small whole numbers.

Usually you need not be concerned with this detail and can simply think of values of type `char` as being characters and not worry about their use as numbers. However, when using the functions in `cctype`, this detail can be important. The functions `toupper` and `tolower` actually return values of type `int` rather than values of type `char`; that is, they return the number corresponding to the character we think of them as returning, rather than the character itself. Thus, the following will not output the letter 'A', but will instead output the number that is assigned to 'A':

```
cout << toupper('a');
```

DISPLAY 6.9 Some Predefined Character Functions in `cctype`

Function	Description	Example
<code>toupper(Char_Exp)</code>	Returns the uppercase version of <i>Char_Exp</i> .	<pre>char c = toupper('a'); cout << c; Outputs: A</pre>
<code>tolower(Char_Exp)</code>	Returns the lowercase version of <i>Char_Exp</i> .	<pre>char c = tolower('A'); cout << c; Outputs: a</pre>
<code>isupper(Char_Exp)</code>	Returns <i>true</i> provided <i>Char_Exp</i> is an uppercase letter; otherwise, returns <i>false</i> .	<pre>if (isupper(c)) cout << c << << " isuppercase."; else cout << c << " is not uppercase.";</pre>
<code>islower(Char_Exp)</code>	Returns <i>true</i> provided <i>Char_Exp</i> is a lowercase letter; otherwise, returns <i>false</i> .	<pre>char c = 'a'; if (islower(c)) cout << c <<<< " islowercase."; Outputs: a is lowercase.</pre>
<code>isalpha(Char_Exp)</code>	Returns <i>true</i> provided <i>Char_Exp</i> is a letter of the alphabet; otherwise, returns <i>false</i> .	<pre>char c = '\$'; if (isalpha(c)) cout << c << " is a letter."; else cout << c << " is not a letter."; Outputs: \$ is not a letter.</pre>
<code>isdigit(Char_Exp)</code>	Returns <i>true</i> provided <i>Char_Exp</i> is one of the digits '0' through '9'; otherwise, returns <i>false</i> .	<pre>if (isdigit('3')) cout << "It's a digit."; else cout << "It's not a digit."; Outputs: It's a digit.</pre>
<code>isspace(Char_Exp)</code>	Returns <i>true</i> provided <i>Char_Exp</i> is a whitespace character, such as the blank or new-line symbol; otherwise, returns <i>false</i> .	<pre>//Skips over one "word" and //sets c equal to the first //whitespace character after //the "word": do { cin.get(c); } while (! isspace(c));</pre>

In order to get the computer to treat the value returned by `toupper` or `tolower` as a value of type `char` (as opposed to a value of type `int`), you need to indicate that you want a value of type `char`. One way to do this is to place the value returned in a variable of type `char`. The following will output the character 'A', which is usually what we want:

```
char c = toupper('a');  ← Places 'A' in the
                        variable c
cout << c;
```

Another way to get the computer to treat the value returned by `toupper` or `tolower` as a value of type `char` is to use a type cast as follows:

```
cout << static_cast<char>(toupper('a'));
```

(Type casts were discussed in Chapter 4 in the section “Type Casting.”) ■

SELF-TEST EXERCISES

36. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    cout << next;
} while ( (! isdigit(next)) && (next != '\n') );
cout << "<END OF OUTPUT";
```

If the dialogue begins as follows, what will be the next line of output?

Enter a line of input:
I'll see you at 10:30 AM.

37. Write some C++ code that will read a line of text and echo the line with all uppercase letters deleted.

CHAPTER SUMMARY

- A stream of type `ifstream` can be connected to a file with a call to the member function `open`. Your program can then take input from that file.
- A stream of type `ofstream` can be connected to a file with a call to the member function `open`. Your program can then send output to that file.
- You should use the member function `fail` to check whether a call to `open` was successful.

- An **object** is a variable that has functions associated with it. These functions are called **member functions**. A **class** is a type whose variables are objects. A stream is an example of an object. The types `ifstream` and `ofstream` are examples of classes.
- The following is the syntax you use when you write a call to a member function of an object:

Calling_Object.Member_Function_Name(Argument_List);

An example with the stream `cout` as the calling object and `precision` as the member function is the following:

```
cout.precision(2);
```

- Stream member functions, such as `width`, `setf`, and `precision`, can be used to format output. These output functions work the same for the stream `cout`, which is connected to the screen, and for output streams connected to files.
- Every input stream has a member function named `get` that can be used to read one character of input. The member function `get` does not skip over whitespace. Every output stream also has a member function named `put` that can be used to write one character to the output stream.
- The member function `eof` can be used to test for when a program has reached the end of an input file. The member function `eof` works well for text processing. However, when processing numeric data, you might prefer to test for the end of a file by using the other method we discussed in this chapter.
- A function may have formal parameters of a stream type, but they must be call-by-reference parameters; they cannot be call-by-value parameters. The type `ifstream` can be used for an input-file stream, and the type `ofstream` can be used for an output-file stream. (See the next summary point for other type possibilities.)
- If you use `istream` (spelled without the `f`) as the type for an input-stream parameter, then the argument corresponding to that formal parameter can be either the stream `cin` or an input-file stream of type `ifstream` (spelled with the `f`). If you use `ostream` (spelled without the `f`) as the type for an output stream parameter, then the argument corresponding to that formal parameter can be either the stream `cout` or an output-file stream of type `ofstream` (spelled with the `f`).

Answers to Self-Test Exercises

1. The streams `fin` and `fout` are declared as follows:

```
ifstream fin;  
ofstream fout;
```

The `include` directive that goes at the top of your file is

```
#include <fstream>
```

Your code also needs the following:

```
using namespace std;
```

2. `fin.open("stuff1.dat");`

```
if (fin.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

`fout.open("stuff2.dat");`

```
if (fout.fail( ))
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

3. `fin.close();`

`fout.close();`

4. You need to replace the stream `out_stream` with the stream `cout`. Note that you do not need to declare `cout`, you do not need to call `open` with `cout`, and you do not need to close `cout`.

5. `#include <cstdlib>`

Your code also needs the following:

```
using namespace std;
```

6. The `exit(1)` function returns the argument to the operating system. By convention, the operating system uses a 1 as an indication of error status and 0 as an indication of success. What is actually done is system-dependent.

7. `bla.dobedo(7);`

8. Both files and program variables store values and can have values retrieved from them. Program variables exist only while the program runs, whereas files may exist before a program is run and may continue to exist after a program stops. In short, files may be permanent; variables are not. Files provide the ability to store large quantities of data, whereas program variables do not provide quite so large a store.

9. We have seen the `open`, `close`, and `fail` member functions at this point. The following illustrate their use.

```

int c;
ifstream in;
ofstream out;
in.open("in.dat");
if (in.fail( ))
{
    cout << "Input file opening failed.\n";
    exit(1);
}
in >> c;

out.open("out.dat");
if (out.fail( ))
{
    cout << "Output file opening failed.\n";
    exit(1);
}
out << c;

out.close( );
in.close( );

```

10. This is the “starting over” the text describes at the beginning of this chapter. The file must be closed and opened again. This action puts the read position at the start of the file, ready to be read again.
11. The two names are the *external file name* and the *stream name*. The external file name is the one used by the operating system. It is the real name of the file, but it is used only in the call to the function `open`, which connects the file to a stream. The stream name is a stream variable (typically of type `ifstream` or `ofstream`). After the call to `open`, your program always uses the stream name as the name of the file.

12.

* 123*123*
 * 123*123*

Each of the spaces contains exactly two blank characters. Notice that a call to `width` or call to `setw` only lasts for one output item.

13.

* 123*123 * 123*

Each of the spaces consists of exactly two blank characters.

14.

* 123*123*
 * +123*+123*
 *123 *123 *

There is just one space between the `*` and the `+` on the second line. Each of the other spaces contains exactly two blank characters.

15. The output to the file `stuff.dat` will be exactly the same as the output given in the answer to Exercise 14.

16. `*12345*`

Notice that the entire integer is output even though this requires more space than was specified by `setw`.

17. a. `ios::fixed`. Setting this flag causes floating-point numbers not to be displayed in e-notation, that is, not in scientific notation. Setting this flag unsets `ios::scientific`.

b. `ios::scientific`. Setting this flag causes floating-point numbers to be displayed in e-notation, that is, in scientific notation. Setting this flag unsets `ios::fixed`.

c. `ios::showpoint`. Setting this flag causes the decimal point and trailing zeros to be always displayed.

d. `ios::showpos`. Setting this flag causes a plus sign to be output before positive integer values.

e. `ios::right`. Setting this flag causes subsequent output to be placed at the right end of any field that is set with the `width` member function. That is, any extra blanks are put before the output. Setting this flag unsets `ios::left`.

f. `ios::left`. Setting this flag causes subsequent output to be placed at the left end of any field that is set with the `width` member function. That is, any extra blanks are put after the output. Setting this flag unsets `ios::right`.

18. You need to replace `ostream` with `cout` and delete the `open` and `close` calls for `ostream`. You do not need to declare `cout`, `open cout`, or `close cout`. The `#include <fstream>` directive has all the `iostream` members you need for screen I/O, though it does no harm, and may make the program clearer, to `#include <iostream>`.

19.

```
1
2
3
3
```

20.

```
void to_screen(ifstream& file_stream)
{
    int next;
    while (file_stream >> next)
        cout << next << endl;
}
```


21. The maximum number of characters that can be typed in for a string variable is one less than the declared size. Here the value is 20.
22. The statement
- ```
cin >> c;
```
- reads the next *nonwhite* character, whereas
- ```
cin.get(c);
```
- reads the next character whether the character is nonwhite or not.
23. The two statements are equivalent. Both of the statements output the value of the variable *c*.
24. The character that is “put back” into the input stream with the member function *putback* need not be the last character read. If your program reads an 'a' from the input stream, it can use the *putback* function to put back a 'b'. (The text in the input file will not be changed by *putback*, although your program will behave as if the text in the input file had been changed.)
25. The complete dialogue is

```
Enter a line of input:
a b c d e f g
a b END OF OUTPUT
```

26. The complete dialogue is

```
Enter a line of input:
abcdef gh
ace h
```

Note that the output is simply every other character of the input, and note that the blank is treated just like any other character.

27. The complete dialogue is

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
01234567891 1
```

Be sure to note that only the '1' in the input string 10 is output. This is because *cin.get* is reading characters, not numbers, and so it reads the input 10 as the two characters, '1' and '0'. Since this code is written to echo only every other character, the '0' is not output. Since the '0' is not output, the next character, which is a blank, is output, and so there is one blank in the output. Similarly, only one of the two '1' characters in 11 is output. If this is unclear, write the input on a sheet of paper and use a small square for the blank character. Then, cross out every other character; the output shown above is what is left.

28. This code contains an infinite loop and will continue as long as the user continues to give it input. The Boolean expression (`next != '\n'`) is always *true* because `next` is filled via the statement

```
cin >> next;
```

and this statement always skips the new-line character `'\n'` (as well as any blanks). The code will run and if the user gives no additional input, the dialogue will be as follows:

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
0246811
```

Notice that the code in Self-Test Exercise 27 used `cin.get`, so it reads every character, *whether the character is a blank or not*, and then it outputs every other character. So the code in Self-Test Exercise 27 outputs every other character even if the character is a blank. On the other hand, the code in this Self-Test Exercise uses `cin` and `>>`, so it *skips over all blanks* and considers only nonblank characters (which in this case are the digits '0' through '9'). Thus, this code outputs every other *nonblank* character. The two '1' characters in the output are the first character in the input 10 and the first character in the input 11.

29. `void copy_char(istream& source_file)`

```
{
    char next;
    source_file.get(next);
    cout << next;
}
```

30. `void copy_line(istream& source_file)`

```
{
    char next;
    do
    {
        source_file.get(next);
        cout << next;
    } while (next != '\n');
}
```

31. `void send_line(ostream& target_stream)`

```
{
    char next;
    do
    {
        cin.get(next);
        target_stream << next;
    } while (next != '\n');
}
```

32. a. 2.0 1.1 2.3
 b. 2.0 3.0 2.3
 c. 2.0 3.0 4.0

33. One set of functions follows:

```
void func(double x)
{
    double y = 1.1;
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y)
{
    double z = 2.3;
    cout << x << " " << y << " " << z << endl;
}
void func(double x, double y, double z)
{
    cout << x << " " << y << " " << z << endl;
}
```

34. It would evaluate to *false*. Your program must attempt to read one more character (beyond the last character) before it changes to *true*.

35.

```
void text_to_screen(istream& file_stream)
{
    char next;
    file_stream.get(next);
    while (! file_stream.eof( ))
    {
        cout << next;
        file_stream.get(next);
    }
}
```

If you prefer, you can use `cout.put(next);` instead of `cout << next;`.

36. The complete dialogue is as follows:

```
Enter a line of input:
I'll see you at 10:30 AM.
I'll see you at 1 <END OF OUTPUT
```

37.

```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    if (!isupper(next))
        cout << next;
} while (next != '\n');
```

Note that you should use `!isupper(next)` and not use `islower(next)`. This is because `islower(next)` is *false* if `next` contains a character that is not a letter (such as the blank or comma symbol).

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a program that will search a file of numbers of type `int` and write the largest and the smallest numbers to the screen. The file contains nothing but numbers of type `int` separated by blanks or line breaks. If this is being done as a class assignment, obtain the file name from your instructor.
2. Write a program that takes its input from a file of numbers of type `double` and outputs the average of the numbers in the file to the screen. The file contains nothing but numbers of type `double` separated by blanks and/or line breaks. If this is being done as a class assignment, obtain the file name from your instructor.
3. a. Compute the median of a data file. The median is the number that has the same number of data elements greater than the number as there are less than the number. For purposes of this problem, you are to assume that the data is *sorted* (that is, is in increasing order). The median is the middle element of the file if there are an odd number of elements, or the average of the two middle elements if the file has an even number of elements. You will need to open the file, count the elements, close the file and calculate the location of the middle of the file, open the file again (recall the “start over” discussion in this chapter), count up to the file entries you need, and calculate the middle.

If your instructor has assigned this problem, ask for a data file to test your program with. Otherwise, construct several files on your own, including one with an even number of data points, increasing, and one with an odd number, also increasing.

- b. For a sorted file, a quartile is one of three numbers: The first has one-fourth the data values less than or equal to it, one-fourth the data values between the first and second numbers, one-fourth the data points between the second and the third, and one-fourth above the third quartile. Find the three quartiles for the data file you used for part (a).

(*Hint:* You should recognize that having done part (a) you have one-third of your job done—you have the second quartile already. You also should recognize that you have done almost all the work toward finding the other two quartiles as well.)

4. Write a program that takes its input from a file of numbers of type *double*. The program outputs to the screen the average and standard deviation of the numbers in the file. The file contains nothing but numbers of type *double* separated by blanks and/or line breaks. The standard deviation of a list of numbers n_1, n_2, n_3 , and so forth is defined as the square root of the average of the following numbers:

$$(n_1 - a)^2, (n_2 - a)^2, (n_3 - a)^2, \text{ and so forth}$$

The number a is the average of the numbers n_1, n_2, n_3 , and so forth. If this is being done as a class assignment, obtain the file name from your instructor.

(Hint: Write your program so that it first reads the entire file and computes the average of all the numbers, and then closes the file, then reopens the file and computes the standard deviation.)

5. Write a program that gives and takes advice on program writing. The program starts by writing a piece of advice to the screen and asking the user to type in a different piece of advice. The program then ends. The next person to run the program receives the advice given by the person who last ran the program. The advice is kept in a file, and the contents of the file change after each run of the program. You can use your editor to enter the initial piece of advice in the file so that the first person who runs the program receives some advice. Allow the user to type in advice of any length so that it can be any number of lines long. The user is told to end his or her advice by pressing the Return key two times. Your program can then test to see that it has reached the end of the input by checking to see when it reads two consecutive occurrences of the character '\n'.
6. Write a program that reads text from one file and writes an edited version of the same text to another file. The edited version is identical to the unedited version except that every string of two or more consecutive blanks is replaced by a single blank. Thus, the text is edited to remove any extra blank characters. Your program should define a function that is called with the input- and output-file streams as arguments. If this is being done as a class assignment, obtain the file names from your instructor.
7. Write a program that merges the numbers in two files and writes all the numbers into a third file. Your program takes input from two different files and writes its output to a third file. Each input file contains a list of numbers of type *int* in sorted order from the smallest to the largest. After the program is run, the output file will contain all the numbers in the two input files in one longer list in sorted order from smallest to largest. Your program should define a function that is called with the two input-file streams and the output-file stream as three arguments. If this is being done as a class assignment, obtain the file names from your instructor.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a program to generate personalized junk mail. The program takes input both from an input file and from the keyboard. The input file contains the text of a letter, except that the name of the recipient is indicated by the three characters `#N#`. The program asks the user for a name and then writes the letter to a second file but with the three letters `#N#` replaced by the name. The three-letter string `#N#` will occur exactly once in the letter.

(Hint: Have your program read from the input file until it encounters the three characters `#N#`, and have it copy what it reads to the output file as it goes. When it encounters the three letters `#N#`, it then sends output to the screen asking for the name from the keyboard. You should be able to figure out the rest of the details. Your program should define a function that is called with the input- and output-file streams as arguments. If this is being done as a class assignment, obtain the file names from your instructor.)

Harder version (using material in the optional section “File Names as Input”): Allow the string `#N#` to occur any number of times in the file. In this case, the name is stored in two string variables. For this version, assume that there is a first name and last name but no middle names or initials.

2. Write a program to compute numeric grades for a course. The course records are in a file that will serve as the input file. The input file is in exactly the following format: Each line contains a student’s last name, then one space, then the student’s first name, then one space, then ten quiz scores all on one line. The quiz scores are whole numbers and are separated by one space. Your program will take its input from this file and send its output to a second file. The data in the output file will be the same as the data in the input file except that there will be one additional number (of type *double*) at the end of each line. This number will be the average of the student’s ten quiz scores. If this is being done as a class assignment, obtain the file names from your instructor. Use at least one function that has file streams as all or some of its arguments.
3. Enhance the program you wrote for Programming Project 2 in all of the following ways.
 - a. The list of quiz scores on each line will contain ten or fewer quiz scores. (If there are fewer than ten quiz scores, that means the student missed one or more quizzes.) The average score is still the sum of the quiz scores divided by 10. This amounts to giving the student a 0 for any missed quiz.

- b. The output file will contain a line (or lines) at the beginning of the file explaining the output. Use formatting instructions to make the layout neat and easy to read.
- c. After placing the desired output in an output file, your program will close all files and then copy the contents of the “output” file to the “input” file so that the net effect is to change the contents of the input file.

Use at least two functions that have file streams as all or some of their arguments. If this is being done as a class assignment, obtain the file names from your instructor.

4. Write a program that will compute the average word length (average number of characters per word) for a file that contains some text. A word is defined to be any string of symbols that is preceded and followed by one of the following at each end: a blank, a comma, a period, the beginning of a line, or the end of a line. Your program should define a function that is called with the input-file stream as an argument. This function should also work with the stream `cin` as the input stream, although the function will not be called with `cin` as an argument in this program. If this is being done as a class assignment, obtain the file names from your instructor.
5. Write a program that will correct a C++ program that has errors in which operator, `<<` or `>>`, it uses with `cin` and `cout`. The program replaces each (incorrect) occurrence of

```
cin <<
```

with the corrected version

```
cin >>
```

and each (incorrect) occurrence of

```
cout >>
```

with the corrected version

```
cout <<
```

For an easier version, assume that there is always exactly one blank space between any occurrence of `cin` and a following `<<`, and similarly assume that there is always exactly one blank space between each occurrence of `cout` and a following `>>`.

For a harder version, allow for the possibility that there may be any number of blanks, even zero blanks, between `cin` and `<<` and between `cout` and `>>`. In this harder case, the replacement corrected version has only one blank between the `cin` or `cout` and the following operator. The program to be corrected is in one file and the corrected version is output

to a second file. Your program should define a function that is called with the input- and output-file streams as arguments.

If this is being done as a class assignment, obtain the file names from your instructor and ask your instructor whether you should do the easier version or the harder version.

(*Hint:* Even if you are doing the harder version, you will probably find it easier and quicker to first do the easier version and then modify your program so that it performs the harder task.)

6. Write a program that allows the user to type in any one-line question and then answers that question. The program will not really pay any attention to the question, but will simply read the question line and discard all that it reads. It always gives one of the following answers:

I'm not sure, but I think you will find the answer in Chapter #N.
That's a good question.

If I were you, I would not worry about such things.

That question has puzzled philosophers for centuries.

I don't know. I'm just a machine.

Think about it and the answer will come to you.

I used to know the answer to that question, but I've forgotten it.

The answer can be found in a secret place in the woods.

These answers are stored in a file (one answer per line), and your program simply reads the next answer from the file and writes it out as the answer to the question. After your program has read the entire file, it simply closes the file, reopens the file, and starts down the list of answers again.

Whenever your program outputs the first answer, it should replace the two symbols #N with a number between 1 and 18 (including the possibility of 1 and 18). In order to choose a number between 1 and 18, your program should initialize a variable to 18 and decrease the variable's value by 1 each time it outputs a number so that the chapter numbers count backward from 18 to 1. When the variable reaches the value 0, your program should change its value back to 18. Give the number 17 the name `NUMBER_OF_CHAPTERS` with a global named constant declaration using the `const` modifier.

(*Hint:* Use the function `new_line` defined in this chapter.)

7. This project is the same as Programming Project 6, except that in this project your program will use a more sophisticated method for choosing the answer to a question. When your program reads a question, it counts the number of characters in the question and stores the number in a variable named `count`. It then responds with answer number `count % ANSWERS`. The first answer in the file is answer number 0, the next is answer number 1, then 2, and so forth. `ANSWERS` is defined in a constant declaration, as shown next, so that it is equal to the number of answers in the answer file:


```
const int ANSWERS = 8;
```

This way you can change the answer file so that it contains more or fewer answers and you need change only the constant declaration to make your program work correctly for a different number of possible answers. Assume that the answer listed first in the file will always be the following, even if the answer file is changed:

I'm not sure, but I think you will find the answer in Chapter #N.

When replacing the two characters #N with a number, use the number $(\text{count} \% \text{NUMBER_OF_CHAPTERS} + 1)$, where count is the variable discussed above, and NUMBER_OF_CHAPTERS is a global named constant defined to be equal to the number of chapters in this book.

8. This program numbers the lines found in a text file. Write a program that reads text from a file and outputs each line to the screen and to another file preceded by a line number. Print the line number at the start of the line and right-adjusted in a field of three spaces. Follow the line number with a colon, then one space, then the text of the line. You should get a character at a time and write code to ignore leading blanks on each line. You may assume that the lines are short enough to fit within a line on the screen. Otherwise, allow default printer or screen output behavior if the line is too long (that is, wrap or truncate).

A somewhat harder version determines the number of spaces needed in the field for the line numbers by counting lines before processing the lines of the file. This version of the program should insert a new line after the last complete word that will fit within a 72-character line.

9. Write a program that computes all of the following statistics for a file and outputs the statistics to both the screen and to another file: the total number of occurrences of characters in the file, the total number of non-whitespace characters in the file, and the total number of occurrences of letters in the file.
10. The text file `babynames2012.txt`, which is included in the source code for this book and is available online from the book's Web site, contains a list of the 1000 most popular boy and girl names in the United States for the year 2012 as compiled by the Social Security Administration.

This is a space-delimited file of 1000 entries in which the rank is listed first, followed by the corresponding boy name and girl name. The most popular names are listed first and the least popular names are listed last. For example, the file begins with

```
1 Jacob Sophia  
2 Mason Emma  
3 Ethan Isabella
```

This indicates that Jacob is the most popular boy name and Sophia is the most popular girl name. Mason is the second most popular boy name and Emma is the second most popular girl name.

Write a program that allows the user to input a name. The program should then read from the file and search for a matching name among the girls and boys. If a match is found, it should output the rank of the name. The program should also indicate if there is no match.

For example, if the user enters the name "Justice," then the program should output:

```
Justice is ranked 519 in popularity among boys.  
Justice is ranked 518 in popularity among girls.
```

If the user enters the name "Walter," then the program should output:

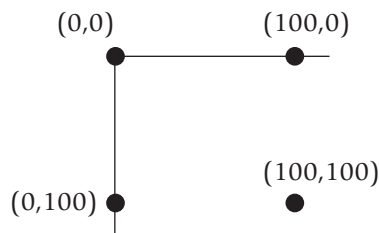
```
Walter is ranked 376 in popularity among boys.  
Walter is not ranked among the top 1000 girl names.
```



VideoNote
Solution to Programming
Project 6.11

11. To complete this problem you must have a computer that is capable of viewing Scalable Vector Graphics (SVG) files. Your Web browser may already be able to view these files. To test to see if your browser can display SVG files, type in the `rectline.svg` file below and see if you can open it in your Web browser. If your Web browser cannot view the file, then you can search on the Web and download a free SVG viewer.

The graphics screen to draw an image uses a coordinate system in which (0, 0) is located in the upper-left corner. The x coordinate increases to the right, and the y coordinate increases to the bottom. Consequently, coordinate (100, 0) would be located 100 pixels directly toward the right from the upper-left corner, and coordinate (0, 100) would be located 100 pixels directly toward the bottom from the upper-left corner. This is illustrated in the figure below.



The SVG format defines a graphics image using XML. The specification for the image is stored in a text file and can be displayed by an SVG viewer. Here is a sample SVG file that draws two rectangles and a line. To view it, save it to a text file with the ".svg" extension, such as `rectline.svg`, and open it with your SVG viewer.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="500" height="500"
xmlns="http://www.w3.org/2000/svg">

<rect x="20" y="20" width="50" height="250"
style="fill:blue;"/>
<rect x="75" y="100" width="150" height="50"
style="fill:rgb(0,255,0);"/>
<line x1="0" y1="0" x2="300" y2="300"
style="stroke:purple;stroke-width:2"/>

</svg>
```

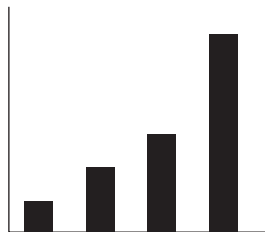
For purposes of this problem, you can ignore the first five lines and the last line and consider them “boilerplate” that must be inserted to properly create the image.

The lines that begins with `<rect x="20"...` draw a blue rectangle whose upper-left corner is at coordinate (20, 20) and whose width is 50 pixels and height is 250 pixels.

The lines that begin with `<rect x="75"...` draw a green rectangle (RGB color value of 0,255,0 is all green) whose upper-left corner is at coordinate (75, 100) and whose width is 150 pixels and height is 50 pixels.

Finally, the `<line>` tag draws a purple line from (0, 0) to (300, 300) with a width of 2.

Based on this example, write a program that inputs four nonnegative integer values and creates the SVG file that displays a simple bar chart that depicts the integer values. Your program should scale the values so they are always drawn with a maximum height of 400 pixels. For example, if your input values to graph were 20, 40, 60, and 120, you might generate a SVG file that would display as follows:



12. Refer to Programming Project 11 for information about the SVG format. Shown below is another example that illustrates how to draw circles, ellipses, and multiple lines:

```

<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="500" height="500"
xmlns="http://www.w3.org/2000/svg">

<circle cx="100" cy="50" r="30"
stroke="green" stroke-width="3" fill="gold"/>

<ellipse cx="100" cy="200" rx="50" ry="100"
style="fill:purple;stroke:black;stroke-width:2"/>

<polyline points="10,10 40,40 20,100 120,140"
style="fill-opacity:0;stroke:red;stroke-width:2"/>

</svg>

```

The `<circle>` tag draws a circle centered at (100, 50) with radius 30 and pen width of 3. It is filled in with gold and has a border in green.

The `<ellipse>` tag draws an ellipse centered at (100, 200) with x radius of 30 and y radius of 100. It is filled using purple with a black border.

The `<polyline>` tag draws a red line from (10, 10) to (40, 40) to (20, 100) to (120, 140). The fill-opacity is set to 0, making the fill of the polygon transparent.

Based on these examples and those presented in Project 18, write a program that creates an SVG image that draws a picture of your professor. It can be somewhat abstract and simple. If you wish to draw a fancier image, you can research the SVG picture format; there are additional tags that can draw using filters, gradients, and polygons.

13. Write a program that prompts the user to input the name of a text file and then outputs the number of words in the file. You can consider a “word” to be any text that is surrounded by whitespace (for example, a space, carriage return, newline) or borders the beginning or end of the file.
14. The following is an old word puzzle: “Name a common word, besides tremendous, stupendous and horrendous, that ends in dous.” If you think about this for a while, it will probably come to you. However, we can also solve this puzzle by reading a text file of English words and outputting the word if it contains “dous” at the end. The text file “words.txt” contains 87,314 English words, including the word that completes the puzzle. This file is available online with the source code for the book. Write a program that reads each word from the text file and outputs only those containing “dous” at the end to solve the puzzle.