Chapter 3: Streams

It's time to begin our serious foray into the magical world of C++ programming. In this first chapter, we'll explore C++'s *streams library*, a collection of functions that allow you to read and write formatted data from a variety of sources. The streams library allows your program to print text to the user and read back responses. It also lets you load persistent data from external files and to save custom information on-disk. As you continue your exploration of C++, you will use the contents of this chapter time and time again, whether for simple error-reporting or more complex data management.

Streams: An Overview

In the physical world, all interesting devices have some way of interacting with their environment. Take a common alcohol thermometer, for example. The thermometer has a liquid-filled bulb that is warmed up by the environment and a graduated meter which allows the user to read off the temperature near the bulb. Or consider a car, which has an accelerator, brake, gearbox, and steering wheel to control the direction and speed of the vehicle and a dashboard which reports the current state of the automobile. C++'s streams library is the primary means by which a C++ program can interact with its environment, namely the user and the file system.

The basic unit of communication between a program and its environment is a *stream*. A stream is a channel between a *source* and a *destination* which allows the source to push formatted data to the destination. The type of the source and the sink varies from stream to stream. In some streams the source is the program itself and the destination is a file on disk, and the stream can be used to write persistent data to the user's hard drive. In others, the source is the keyboard and the destination is the program, and the stream can be used to read user input from the physical world into the computer.

The use of the term "stream" in the context of the streams library is similar to the use of "stream" in the context of "streaming video." When a data provider (for example, YouTube) streams video over the Internet, the video is not sent all at once. Instead, the program receiving the video continuously queries the server for more and more information, and the video is sent in fixed-size chunks and reassembled by the video player. When using the streams library to read or write data, you do not need to read or write all of the data at once. It's perfectly legal (and quite common) to read the data one piece at a time. For example, if you want to read data from a file, instead of loading all of the file contents at once, you can read the file line-by-line, or character-by-character, or using some hybrid approach. This gives you great flexibility, since you can read different pieces of the file in different ways to get the data in a format appropriate to your application.

To give you a better sense of how streams work in practice, let's consider an actual stream, `cout`. `cout` (for **c**haracter **out**put) is a stream connected to the *console*, a text window that displays plain text data. Any information pushed across `cout` displays in the console, and so you can think of `cout` as a way of displaying data to the user. For example, here's a simple program which displays a message to the user and then quits:

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

int main() {
    cout << "I'm sorry Dave, I'm afraid I can't do that." << endl;
    return 0;
}
```

There's a lot of code here, so let's take a few minutes to dissect it. The first line of the program, `#include <iostream>`, instructs the C++ compiler to import the `cout` stream into the program. The line `using namespace std` is covered in the previous chapter and makes the `cout` stream available. Inside of `main`, we have the following line of code:

```
cout << "I'm sorry Dave, I'm afraid I can't do that." << endl;
```

The special `<<` operator is called the *stream insertion operator* and is a C++ operator that is used to push data into a stream object. Here, we push the text string `I'm sorry Dave, I'm afraid I can't do that` into the `cout` stream. This causes the this text to display on-screen. Afterwards, we push the special object `endl` into the stream. `endl` stands for "**end l**ine" and prints a newline character to the `cout` stream. This means that the next time we push text into `cout`, the text will display on the next line, rather than directly after the text string we just printed. We'll discuss `endl` in more detail later in this chapter.

Streams are very versatile and you can write data of multiple types to stream objects. In fact, you can push data of any primitive type into a stream. For example, here's a program showing off the sorts of data that can move across a stream:

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

int main() {
    cout << "Streams can take in text." << endl;
    cout << 137 << endl;     // Streams can take in integers.
    cout << 2.71828 << endl; // Streams can take in real numbers.
    cout << "Here is text followed by a number: " << 31415 << endl;
    return 0;
}
```

Running this program will produce the following output:

```
Streams can take in text.
137
2.71828
Here is text followed by a number: 31415
```

In the first line of this program, we sent a text string to the console. In the second and third, we sent an integer and a natural number, respectively. The last line is perhaps the most interesting. In it, we push both a string *and* an integer to the console by chaining together the stream insertion operator. The designers of the streams library were fairly clever, and so it's perfectly legal to chain together as many stream insertions as you'd like.

To give you a better feel for why each of the stream operations in the above program end by pushing `endl` into the stream, let's consider what would happen if this weren't the case. Here's a revised version of the above program will all instances of `endl` removed:

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

int main() {
    cout << "Streams can take in text.";
    cout << 137;
    cout << 2.71828;
    cout << "Here is text followed by a number: " << 31415;
    return 0;
}
```

This produces the following output:

```
Streams can take in text.1372.71828Here is text followed by a number: 31415
```

Notice that all of this text runs together. C++ will not "automatically" insert newlines into any text you write, and when outputting data to the console you will need to manually insert line breaks. As a general rule, most of the time that you use `cout` to push data to the console, you will need to append `endl` to ensure the output doesn't all run together.

All of the stream examples we have seen so far have revolved around `cout` and pushing data from the program to the console. To build a truly interactive program, however, we'll need to get input from the user. In CS106B/X, we provide the `simpio.h` header file, which exports the input functions `GetLine`, `GetInteger`, `GetReal`, and `GetLong`. Though useful, these functions are not part of the C++ standard library and will not be available outside of CS106B/X. Don't worry, though, because by the end of this chapter we'll see how to implement them using only standard C++.

The streams library exports another stream object called `cin` (**c**haracter **in**put) which lets you read values directly from the user. To read a value from `cin`, you use the *stream extraction operator* `>>`. Syntactically, the stream extraction operator mirrors the stream insertion operator. For example, here's a code snippet to prompt the user for an integer.

```
cout << "Please enter an integer: ";

int myInteger;
cin >> myInteger; // Value stored in myInteger
```

When the program encounters the highlighted line, it will pause and wait for the user to type in a number and hit enter. Provided that the user actually enters an integer, its value will be stored inside the `myInteger` variable. What happens if the user *doesn't* enter an integer is a bit more complicated, and we'll return to this later in the chapter.

You can also read multiple values from `cin` by chaining together the stream extraction operator in the same way that you can write multiple values to `cout` by chaining the stream insertion operator:

```
int myInteger;
string myString;
cin >> myInteger >> myString; // Read an integer and string from cin
```

This will pause until the user enters an integer, hits enter, then enters a string, then hits enter once more. These values will be stored in `myInteger` and `myString`, respectively.

Note that when using `cin`, you should not read into `endl` the way that you write `endl` when using `cout`. Hence the following code is illegal:

```
int myInteger;
cin >> myInteger >> endl; // Error: Cannot read into endl.
```

Intuitively, this makes sense because `endl` means "print a newline." Reading a value into `endl` is therefore a nonsensical operation.

In practice, it is not a good idea to read values directly from `cin`. Unlike `GetInteger` and the like, `cin` does not perform any safety checking of user input and if the user does not enter valid data, `cin` will begin behaving unusually. Later in this chapter, we will see how the `GetInteger` function is implemented and you will be able to use the function in your own programs. In the meantime, though, feel free to use `cin`, but make sure that you always type in input correctly!

Reading and Writing Files

So far, we have seen two examples of streams – `cout`, which sends data to the console, and `cin`, which reads data from the keyboard. In this next section we'll see two new kinds of streams – `ifstream`s and `ofstream`s – which can be used to read or write files on disk. This will allow your program to save data indefinitely, or to read in configuration data from an external source.

C++ provides a header file called `<fstream>` (**f**ile **stream**) that exports the `ifstream` and `ofstream` types, streams that perform file I/O. The naming convention is unfortunate – `ifstream` stands for **i**nput **f**ile **stream** (not "something that might be a stream") and `ofstream` for **o**utput **f**ile **stream**. There is also a generic `fstream` class which can do both input and output, but we will not cover it in this chapter. Unlike `cin` and `cout`, which are concrete stream objects, `ifstream` and `ofstream` are *types*. To read or write from a file, you will create an object of type `ifstream` or `ofstream`, much in the same way that you would create an object of type `string` to store text data or a variable of type `double` to hold a real number. Once you have created the file stream object, you can read or write to it using the stream insertion and extraction operators just as you would `cin` or `cout`.

To create an `ifstream` that reads from a file, you can use this syntax:

```
ifstream myStream("myFile.txt");
```

This creates a new stream object named `myStream` which reads from the file `myFile.txt`, provided of course that the file exists. We can then read data from `myStream` just as we would from `cin`, as shown here:

```
ifstream myStream("myFile.txt");
int myInteger;
myStream >> myInteger; // Read an integer from myFile.txt
```

Notice that we wrote `myStream >> myInteger` rather than `ifstream >> myInteger`. When reading data from a file stream, you must read from the stream variable rather than the `ifstream` type. If you read from `ifstream` instead of your stream variable, the program will not compile and will give you a fairly cryptic error message.

You can also open a file by using the `ifstream`'s `open` member function, as shown here:

```
ifstream myStream;            // Note: did not specify the file
myStream.open("myFile.txt"); // Now reading from myFile.txt
```

When opening a file using an `ifstream`, there is a chance that the specified file can't be opened. The filename might not specify an actual file, you might not have permission to read the file, or perhaps the file is

locked. If you try reading data from an `ifstream` that is not associated with an open file, the read will fail and you will not get back meaningful data. After trying to open a file, you should check if the stream is valid by using the `.is_open()` member function. For example, here's code to open a file and report an error to the user if a problem occurred:

```
ifstream input("myfile.txt");
if(!input.is_open())
    cerr << "Couldn't open the file myfile.txt" << endl;
```

Notice that we report the error to the `cerr` stream. `cerr`, like `cout`, is an output stream, but unlike `cout`, `cerr` is designed for error reporting and is sometimes handled differently by the operating system.

The output counterpart to `ifstream` is `ofstream`. As with `ifstream`, you specify which file to write to either by using the `.open()` member function or by specifying the file when you create the `ofstream`, as shown below:

```
ofstream myStream("myFile.txt"); // Write to myFile.txt
```

A word of warning: if you try writing to a nonexistent file with an `ofstream`, the `ofstream` will create the file for you. However, if you open a file that already exists, the `ofstream` will overwrite all of the contents of the file. Be careful not to write to important files without first backing them up!

The streams library is one of the older libraries in C++ and the `open` functions on the `ifstream` and `ofstream` classes predate the `string` type. If you have the name of a file stored in a C++ `string`, you will need to convert the `string` into a C-style string (covered in the second half of this book) before passing it as a parameter to `open`. This can be done using the `.c_str()` member function of the `string` class, as shown here:

```
ifstream input(myString.c_str()); // Open the filename stored in myString
```

When a file stream object goes out of scope, C++ will automatically close the file for you so that other processes can read and write the file. If you want to close the file prematurely, you can use the `.close()` member function. After calling `close`, reading or writing to or from the file stream will fail.

As mentioned above in the section on `cin`, when reading from or writing to files you will need to do extensive error checking to ensure that the operations succeed. Again, we'll see how to do this later.

Stream Manipulators

Consider the following code that prints data to `cout`:

```
cout << "This is a string!" << endl;
```

What exactly is `endl`? It's an example of a *stream manipulator*, an object that can be inserted into a stream to change some sort of stream property. `endl` is one of the most common stream manipulators, though others exist as well. To motivate some of the more complex manipulators, let's suppose that we have a file called `table-data.txt` containing four lines of text, where each line consists of an integer value and a real number. For example:

File: `table-data.txt`

```
137     2.71828
42      3.14159
7897987 1.608
1337    .01101010001
```

We want to write a program which reads in this data and prints it out in a table, as shown here:

```
--------------------+----------------------+---------------------
                  1 |                  137 |              2.71828
                  2 |                   42 |              3.14159
                  3 |              7897987 |                1.608
                  4 |                 1337 |              0.01101
```

Here, the first column is the one-indexed line number, the second the integer values from the file, and the third the real-numbered values from the file.

Let's begin by defining a few constants to control what the output should look like. Since there are four lines in the file, we can write

```
const int NUM_LINES = 4;
```

And since there are three columns,

```
const int NUM_COLUMNS = 3;
```

Next, we'll pick an arbitrary width for each column. We'll choose twenty characters, though in principle we could pick any value as long as the data fit:

```
const int COLUMN_WIDTH = 20;
```

Now, we need to read in the table data and print out the formatted table. We'll decompose this problem into two smaller steps, resulting in the following source code:

```
#include <iostream>
#include <fstream>
using namespace std;

const int NUM_LINES = 4;
const int NUM_COLUMNS = 3;
const int COLUMN_WIDTH = 20;

int main() {
    PrintTableHeader();
    PrintTableBody();
    return 0;
}
```

`PrintTableHeader` is responsible for printing out the top part of the table (the row of dashes and pluses) and `PrintTableBody` will load the contents of the file and print them to the console.

Despite the fact that `PrintTableHeader` precedes `PrintTableBody` in this program, we'll begin by implementing `PrintTableBody` as it illustrates exactly how much firepower we can get from the stream manipulators. We know that we need to open the file `table-data.txt` and that we'll need to read four lines of data from it, so we can begin writing this function as follows:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        /* ... process data ... */
    }
}
```

You may have noticed that at the end of this `for` loop I've written `++k` instead of `k++`. There's a slight difference between the two syntaxes, but in this context they are interchangeable. When we talk about operator overloading in a later chapter we'll talk about why it's generally considered better practice to use the prefix increment operator instead of the postfix.

Now, we need to read data from the file and print it as a table. We can start by actually reading the values from the file, as shown here:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;
    }
}
```

Next, we need to print out the table row. This is where things get tricky. If you'll recall, the table is supposed to be printed as three columns, each a fixed width, that contain the relevant data. How can we ensure that when we print the values to `cout` that we put in the appropriate amount of whitespace? Manually writing space characters would be difficult, so instead we'll use a stream manipulator called `setw` (**set w**idth) to force `cout` to pad its output with the right number of spaces. `setw` is defined in the `<iomanip>` header file and can be used as follows:

```
cout << setw(10) << 137 << endl;
```

This tells `cout` that the next item it prints out should be padded with spaces so that it takes up at least ten characters. Similarly,

```
cout << setw(20) << "Hello there!" << endl;
```

Would print out `Hello there!` with sufficient leading whitespace.

By default `setw` pads the next operation with spaces on the left side. You can customize this behavior with the `left` and `right` stream manipulators, as shown here:

```
cout << '[' << left  << setw(10) << "Hello!" << ']' << endl;  // [     Hello!]
cout << '[' << right << setw(10) << "Hello!" << ']' << endl;  // [Hello!     ]
```

Back to our example. We want to ensure that every table column is exactly `COLUMN_WIDTH` spaces across. Using `setw`, this is relatively straightforward and can be done as follows:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}
```

This produces the following output when run on the input file described above:

```
                   1 |                 137 |             2.71828
                   2 |                  42 |             3.14159
                   3 |             7897987 |               1.608
                   4 |                1337 |             0.01101
```

The body of the table looks great, and now we just need to print the table header, which looks like this:

```
--------------------+----------------------+--------------------
```

If you'll notice, this is formed by printing twenty dashes, then the pattern `-+-`, another twenty dashes, the pattern `-+-`, and finally another twenty dashes. We could thus implement `PrintTableHeader` like this:

```
void PrintTableHeader() {
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column) {
        for(int k = 0; k < COLUMN_WIDTH; ++k)
            cout << '-';
        cout << "-+-";
    }

    /* Now print the ---...--- pattern for the last column. */
    for(int k = 0; k < COLUMN_WIDTH; ++k)
        cout << '-';

    /* Print a newline... there's nothing else on this line. */
    cout << endl;
}
```

As written there's nothing wrong with this code and the program will work just fine, but we can simplify the implementation by harnessing stream manipulators. Notice that at two points we need to print out `COLUMN_WIDTH` copies of the dash character. When printing out the table body, we were able to use the `setw` stream manipulator to print multiple copies of the space character; is there some way that we can use it here to print out multiple dashes? The answer is yes, thanks to `setfill`. The `setfill` manipulator

accepts a parameter indicating what character to use as a fill character for `setw`, then changes the stream such that all future calls to `setw` pad the stream with the specified character. For example:

```
cout << setfill('0') << setw(8) << 1000 << endl; // Prints 00001000
cout << setw(8) << 1000 << endl; // Prints 00001000 because of last setfill
```

Note that `setfill` does not replace all space characters with instances of some other character. It is only meaningful in conjunction with `setw`. For example:

```
cout << setfill('X') << "Some   Spaces" << endl; // Prints Some   Spaces
```

Using `setfill` and `setw`, we can print out `COLUMN_WIDTH` copies of the dash character as follows:

```
cout << setfill('-') << setw(COLUMN_WIDTH) << "" << setfill(' ');
```

This code is dense, so let's walk through it one step at a time. The first part, `setfill('-')`, tells `cout` to pad all output with dashes instead of spaces. Next, we use `setw` to tell `cout` that the next operation should take up at least `COLUMN_WIDTH` characters. The trick is the next step, printing the empty string. Since the empty string has length zero and the next operation will always print out at least `COLUMN_WIDTH` characters padded with dashes, this code prints out `COLUMN_WIDTH` dashes in a row. Finally, since `setfill` permanently sets the fill character, we use `setfill(' ')` to undo the changes we made to `cout`.

Using this code, we can rewrite `PrintTableHeader` as follows:

```
void PrintTableHeader() {
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column)
        cout << setfill('-') << setw(COLUMN_WIDTH) << "" << "-+-";

    /* Now print the ---...--- pattern for the last column and a newline. */
    cout << setw(COLUMN_WIDTH) << "" << setfill(' ') << endl;
}
```

Notice that we only call `setfill(' ')` once, at the end of this function, since there's no reason to clear it at each step. Also notice that we've reduced the length of this function dramatically by having the library take care of the heavy lifting for us. The code to print out a table header is now three lines long!

There are many stream manipulators available in C++. The following table lists some of the more commonly-used ones:

`boolalpha`	`cout << true << endl;` `// Output: 1`
	`cout << boolalpha << true << endl;` `// Output: true`
	
	Determines whether or not the stream should output boolean values as 1 and 0 or as "true" and "false." The opposite manipulator is `noboolalpha`, which reverses this behavior.

`setw(n)`	`cout << 10 << endl;` `// Output: 10`
	`cout << setw(5) << 10 << endl;` `// Output:  10`
	
	Sets the minimum width of the output for the next stream operation. If the data doesn't meet the minimum field requirement, it is padded with the default fill character until it is the proper size.

Common stream manipulators, contd.

`hex, dec, oct`	`cout << 10 << endl;  // Output: 10`
	`cout << dec << 10 << endl; // Output: 10`
	`cout << oct << 10 << endl; // Output: 12`
	`cout << hex << 10 << endl; // Output: a`
	`cin >> hex >> x;  // Reads a hexadecimal value.`
	
	Sets the radix on the stream to either octal (base 8), decimal (base 10), or hexadecimal (base 16). This can be used either to format output or change the base for input.

`ws`	`myStream >> ws >> value;` Skips any whitespace stored in the stream. By default the stream extraction operator skips over whitespace, but other functions like `getline` do not. `ws` can sometimes be useful in conjunction with these other functions.

When Streams Go Bad

Because stream operations often involve transforming data from one form into another, stream operations are not always guaranteed to succeed. For example, consider the following code snippet, which reads integer values from a file:

```
ifstream in("input.txt"); // Read from input.txt
for(int i = 0; i < NUM_INTS; ++i) {
    int value;
    in >> value;
    /* ... process value here ... */
}
```

If the file `input.txt` contains `NUM_INTS` consecutive integer values, then this code will work correctly. However, what happens if the file contains some other type of data, such as a string or a real number?

If you try to read stream data of one type into a variable of another type, rather than crashing the program or filling the variable with garbage data, the stream fails by entering an *error state* and the value of the variable will not change. Once the stream is in this error state, any subsequent read or write operations will automatically and silently fail, which can be a serious problem.

You can check if a stream is in an error state with the `.fail()` member function. Don't let the name mislead you – `fail` checks if a stream is in an error state, rather than putting the stream into that state. For example, here's code to read input from `cin` and check if an error occurred:

```
int myInteger;
cin >> myInteger;
if(cin.fail()) { /* ... error ... */ }
```

If a stream is in a fail state, you'll probably want to perform some special handling, possibly by reporting the error. Once you've fixed any problems, you need to tell the stream that everything is okay by using the `.clear()` member function to bring the stream out of its error state. Note that clear won't skip over the input that put the stream into an error state; you will need to extract this input manually.

Streams can also go into error states if a read operation fails because no data is available. This occurs most commonly when reading data from a file. Let's return to the table-printing example. In the `PrintTableData` function, we hardcoded the assumption that the file contains exactly four lines of data. But

what if we want to print out tables of arbitrary length? In that case, we'd need to continuously read through the file extracting and printing numbers until we exhaust its contents. We can tell when we've run out of data by checking the `.fail()` member function after performing a read. If `.fail()` returns true, something prevented us from extracting data (either because the file was malformed or because there was no more data) and we can stop looping.

Recall that the original code for reading data looks like this:

```
void PrintTableBody() {
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}
```

The updated version of this code, which reads all of the contents of the file, is shown here:

```
void PrintTableBody() {
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    int rowNumber = 0;
    while(true) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        if(input.fail()) break;

        cout << setw(COLUMN_WIDTH) << (rowNumber + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;

        rowNumber++;
    }
}
```

Notice that we put the main logic into a `while(true)` loop that `break`s when `input.fail()` returns `true` instead of a `while(!input.fail())` loop. These two structures may at first appear similar, but are quite different from one another. In a `while(!input.fail())` loop, we only check to see if the stream encountered an error after reading and processing the data in the body of the loop. This means that the loop will execute once more than it should, because we don't notice that the stream malfunctioned until the top of the loop. On the other hand, in the above loop structure (`while(true)` plus `break`), we stop looping as soon as the stream realizes that something has gone awry. Confusing these two loop structures is a common error, so be sure that you understand why to use the "loop-and-a-half" idiom rather than a simple `while` loop.

A Useful Shorthand

In the above code, we used the loop-and-a-half idiom to determine whether we should continue reading and printing data out of the file or whether we should stop looping. The general pattern for this idiom is as follows:

```
while(true) {
    int intValue;
    double doubleValue;
    input >> intValue >> doubleValue;

    if(input.fail()) break;

    /* ... process values here ... */
}
```

This code is perfectly valid, but it's a bit clunky. The outermost loop is a `while(true)` loop, which means "loop forever," but in reality the idea we want to represent is "loop until there is no more available data." The designers of the streams library anticipated this use case and provided a remarkably simple shorthand to alleviate this complexity. The above code is entirely equivalent to

```
int intValue;
double doubleValue;

while(input >> intValue >> doubleValue) {
    /* ... process values here ... */
}
```

Notice that the condition of the `while` loop is now `input >> intValue >> doubleValue`. Recall that in C++, any nonzero value is interpreted as "`true`" and any zero value is interpreted as "`false`." The streams library is configured so that most stream operations, including stream insertion and extraction, yield a nonzero value if the operation succeeds and zero otherwise. This means that code such as the above, which uses the read operation as the looping condition, is perfectly valid. One particular advantage of this approach is that while the syntax is considerably more dense, the code is more intuitive. You can read this while loop as "while I can successfully read data into `intValue` and `doubleValue`, continue executing the loop." Compared to our original implementation, this is much cleaner.

This syntax shorthand is actually a special case of a more general technique. In any circumstance where a boolean value is expected, it is legal to place a stream object or a stream read/write operation. We will see this later in this chapter when we explore the getline function.

When Streams Do Too Much

Consider the following code snippet, which prompts a user for an age and hourly salary:

```
int age;
double hourlyWage;

cout << "Please enter your age: ";
cin >> age;

cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
```

As mentioned above, if the user enters a string or otherwise non-integer value when prompted for their age, the stream will enter an error state. There is another edge case to consider. Suppose the input is `2.71828`. You would expect that, since this isn't an integer (it's a real number), the stream would go into an error state. However, this isn't what happens. The first call, `cin >> age`, will set `age` to `2`. The next call, `cin >> hourlyWage`, rather than prompting the user for a value, will find the `.71828` from the earlier input and fill in `hourlyWage` with that information. Despite the fact that the input was malformed for the first prompt, the stream was able to partially interpret it and no error was signaled.

As if this wasn't bad enough, suppose we have this program instead, which prompts a user for an administrator password and then asks whether the user wants to format her hard drive:

```
string password;
cout << "Enter administrator password: ";

cin >> password;
if(password == "password") { // Use a better password, by the way!
    cout << "Do you want to erase your hard drive (Y or N)? ";

    char yesOrNo;
    cin >> yesOrNo;

    if(yesOrNo == 'y')
        EraseHardDrive();
}
```

What happens if someone enters `password y`? The first call, `cin >> password`, will read only `password`. Once we reach the second `cin` read, it automatically fills in `yesOrNo` with the leftover `y`, and there goes our hard drive! Clearly this is not what we intended.

As you can see, reading directly from `cin` is unsafe and poses more problems than it solves. In CS106B/X we provide you with the `simpio.h` library primarily so you don't have to deal with these sorts of errors. In the next section, we'll explore an entirely different way of reading input that avoids the above problems.

An Alternative: `getline`

Up to this point, we have been reading data using the stream extraction operator, which, as you've seen, can be dangerous. However, there are other functions that read data from a stream. One of these functions is *`getline`*, which reads characters from a stream until a newline character is encountered, then stores the read characters (minus the newline) in a `string`. `getline` accepts two parameters, a stream to read from and a `string` to write to. For example, to read a line of text from the console, you could use this code:

```
string myStr;
getline(cin, myStr);
```

No matter how many words or tokens the user types on this line, because `getline` reads until it encounters a newline, all of the data will be absorbed and stored in `myStr`. Moreover, because any data the user types in can be expressed as a string, unless your input stream encounters a read error, `getline` will not put the stream into a fail state. No longer do you need to worry about strange I/O edge cases!

You may have noticed that the `getline` function acts similarly to the CS106B/X `GetLine` function. This is no coincidence, and in fact the `GetLine` function from `simpio.h` is implemented as follows:*

```
string GetLine() {
    string result;
    getline(cin, result);
    return result;
}
```

At this point, `getline` may seem like a silver-bullet solution to our input problems. However, `getline` has a small problem when mixed with the stream extraction operator. When the user presses return after entering text in response to a `cin` prompt, the newline character is stored in the `cin` internal buffer. Normally, whenever you try to extract data from a stream using the `>>` operator, the stream skips over newline and whitespace characters before reading meaningful data. This means that if you write code like this:

```
int first, second;
cin >> first;
cin >> second;
```

The newline stored in `cin` after the user enters a value for `first` is eaten by `cin` before `second` is read. However, if we replace the second call to `cin` with a call to `getline`, as shown here:

```
int dummyInt;
string dummyString;
cin >> dummyInt;
getline(cin, dummyString);
```

`getline` will return an empty string. Why? Unlike the stream extraction operator, `getline` does *not* skip over the whitespace still remaining in the `cin` stream. Consequently, as soon as `getline` is called, it will find the newline remaining from the previous `cin` statement, assume the user has pressed return, and return the empty string.

To fix this problem, your best option is to replace all normal stream extraction operations with calls to library functions like `GetInteger` and `GetLine` that accomplish the same thing. Fortunately, with the information in the next section, you'll be able to write `GetInteger` and almost any `Get____` function you'd ever need to use. When we cover templates and operator overloading in later chapters, you'll see how to build a generic read function that can parse any sort of data from the user.

Reading Files with `getline`

Our treatment of `getline` so far has only considered using `getline` to read data from `cin`, but `getline` is in fact much more general and can be used to read data from any stream object, including file streams. To give a better feel for how the `getline` function works in practice, let's go over a quick example of how to use `getline` to read data from files. In this example, we'll write a program that takes in a data file containing some useful information and display it in a nice, pretty format. In particular, we'll write a program that reads a data file called `world-capitals.txt` containing a list of all the world's countries and their capitals, then displays them to the user. We will assume that the `world-capitals.txt` file is formatted as follows:

---

* Technically, the implementation of `GetLine` from `simpio.h` is slightly different, as it checks to make sure that `cin` is not in an error state before reading.

File: `world-capitals.txt`

```
Abu Dhabi
United Arab Emirates
Abuja
Nigeria
Accra
Ghana
Addis Ababa
Ethiopia
...
```

In this file, every pair of lines represents a capital city and the country of which it is the capital. For example, the first two lines indicate that Abu Dhabi is the capital of the United Arab Emirates, the second two that Abuja is the capital of Nigeria, etc. Our goal is to write a program that prints this data in the following format:

```
Abu Dhabi is the capital of United Arab Emirates
Abuja is the capital of Nigeria
Accra is the capital of Ghana
...
```

How can we go about writing a program like this? Well, we can start by opening the file and printing an error if we can't find it:

```
int main() {
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    /* ... */
}
```

Now, we need to process pairs of lines in the file. Using the concepts from this chapter, we have two general lines of attack to consider. First, we could use the stream extraction operator `>>` to read the data from the file. Second, we could use the `getline` function to read lines of text from the file. In this particular circumstance, it is not a particularly good idea to use the stream extraction operator. Remember that the extraction operator reads data from files one token at a time, rather than one line at a time. Not all world capitals are a single token long (for example, Abu Dhabi or Addis Ababa) nor are all countries one token long (for example, United Arab Emirates). If we were to try to read the file data using the stream extraction operator, we would have no way of knowing when we had read in the complete name of a capital city or country, and it would be all but impossible to print the data out in a meaningful format. However, `getline` does not have this problem, since `getline` blindly reads lines of text and has no notion of whitespace-delineated tokens. Thus for this particular program, we'll use the `getline` function to read file data.

As with most file reading operations, we will need to keep looping until we've exhausted all of the data in the file. This can usually be done with the loop-and-a-half idiom. In our case, one possible version of the code is as follows:

```
int main() {
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    while (true) {
        string capital, country;
        getline(capitals, capital);
        getline(capitals, country);

        if (capitals.fail()) break;

        cout << capital << " is the capital of " << country << endl;
    }
}
```

The above code creates two strings, `capital` and `country`, and populates them with data from the file. It then checks whether the read succeeded, and, if so, prints out the formatted data string.

This code is perfectly correct, but it's clunky. The loop-and-a-half idiom is never pretty, and there has to be a better way to structure this code. Fortunately, there is a wonderful shorthand we can use to condense this code. Recall that when using the stream extraction operator `>>`, we could write code to the following effect to read data from a file and continue looping while the read operation succeeds:

```
while (myStream >> myValue) {
    /* ... process myValue here ... */
}
```

We can use a similar trick with `getline`. In particular, the `getline` function returns a nonzero value if data can be read from a file and a zero value otherwise. Consequently, we can rewrite the above code as follows:

```
int main()
{
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    string capital, country;
    while (getline(capitals, capital) && getline(capitals, country))
        cout << capital << " is the capital of " << country << endl;
}
```

This code is considerably more concise than our original version and arguably easier to read. The condition of the `while` loop now reads "while we can read a line from the file into `capital` and a line from the file into `country`, keep executing the loop." If you ever find yourself reading a file line-by-line, feel free to adapt this trick into your own code – you'll save yourself a great deal of typing if you do.

A String Buffer: `stringstream`

Before we discuss writing `GetInteger`, we'll need to take a diversion to another type of C++ stream.

Often you will need to construct a string composed both of plain text and numeric or other data. For example, suppose you wanted to call this hypothetical function:

```
void MessageBoxAlert(string message);
```

and have it display a message box to the user informing her that the level number she wanted to warp to is out of bounds. At first thought, you might try something like

```
int levelNum = /* ... */;
MessageBoxAlert("Level " + levelNum + " is out of bounds."); // Error
```

For those of you with Java experience this might seem natural, but in C++ this isn't legal because you can't add numbers to strings (and when you can, it's almost certainly won't do what you expected; see the chapter on C strings).

One solution to this problem is to use another kind of stream object known as a *`stringstream`*, exported by the `<sstream>` header. Like console streams and file streams, `stringstream`s are stream objects and consequently all of the stream operations we've covered above work on `stringstream`s. However, instead of reading or writing data to an external source, `stringstream`s store data in temporary string buffers. In other words, you can view a `stringstream` as a way to create and read string data using stream operations.

For example, here is a code snippet to create a `stringstream` and put text data into it:

```
stringstream myStream;
myStream << "Hello!" << 137;
```

Once you've put data into a `stringstream`, you can retrieve the string you've created using the `.str()` member function. Continuing the above example, we can print out an error message as follows:

```
int levelNum = /* ... */;
stringstream messageText;

messageText << "Level " << levelNum << " is out of bounds.";
MessageBoxAlert(messageText.str());
```

`stringstream`s are an example of an `iostream`, a stream that can perform both input and output. You can both insert data into a `stringstream` to convert the data to a string and extract data from a `stringstream` to convert string data into a different format. For example:

```
stringstream myConverter;
int myInt;
string myString;
double myDouble;

myConverter << "137 Hello 2.71828";              // Insert string data
myConverter >> myInt >> myString >> myDouble;    // Extract mixed data
```

The standard rules governing stream extraction operators still apply to `stringstream`s, so if you try to read data from a `stringstream` in one format that doesn't match the character data, the stream will fail. We'll exploit this functionality in the next section.

Putting it all together: Writing `GetInteger`

Using the techniques we covered in the previous sections, we can implement a set of robust user input functions along the lines of those provided by `simpio.h`. In this section we'll explore how to write `GetInteger`, which prompts the user to enter an integer and returns only after the user enters valid input.

Recall from the above sections that reading an integer from `cin` can result in two types of problems. First, the user could enter something that is not an integer, causing `cin` to fail. Second, the user could enter too much input, such as `137 246` or `Hello 37`, in which case the operation succeeds but leaves extra data in `cin` that can garble future reads. We can immediately eliminate these sorts of problems by using the `getline` function to read input, since `getline` cannot put `cin` into a fail state and grabs all of the user's data, rather than just the first token.

The main problem with `getline` is that the input is returned as a `string`, rather than as formatted data. Fortunately, using a `stringstream`, we can convert this text data into another format of our choice. This suggests an implementation of `GetInteger`. We read data from the console using `getline` and funnel it into a `stringstream`. We then use standard stream manipulations to extract the integer from the `stringstream`, reporting an error and reprompting if unable to do so. We can start writing `GetInteger` as follows:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Process data here.  On error: */
        cout << "Retry: "
    }
}
```

At this point, we've read in all of the data we need, and simply need to check that the data is in the proper format. As mentioned above, there are two sorts of problems we might run into – either the data isn't an integer, or the data contains leftover information that isn't part of the integer. We need to check for both cases. Checking for the first turns out to be pretty simple – because `stringstream`s are stream objects, we can see if the data isn't an integer by extracting an integer from our `stringstream` and checking if this puts the stream into a fail state. If so, we know the data is invalid and can alert the user to this effect.

The updated code for `GetInteger` is as follows:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        if(converter >> result) {
            /* ... check that there isn't any leftover data ... */
        } else
            cout << "Please enter an integer." << endl;

        cout << "Retry: "
    }
}
```

Finally, we need to check if there's any extra data left over. If so, we need to report to the user that something is wrong with the input, and can otherwise return the value we read. While there are several ways to check this, one simple method is to read in a single `char` from the `stringstream`. If it is possible to do so, then we know that there must have been something in the input stream that wasn't picked up when we extracted an integer and consequently that the input is bad. Otherwise, the stream must be out of data and will enter a fail state, signaling that the user's input was valid. The final code for `GetInteger`, which uses this trick, is shown here:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        if(converter >> result) {
            char remaining;
            if(converter >> remaining) // Something's left, input is invalid
                cout << "Unexpected character: " << remaining << endl;
            else
                return result;
        } else
            cout << "Please enter an integer." << endl;

        cout << "Retry: "
    }
}
```

More To Explore

C++ streams are extremely powerful and encompass a huge amount of functionality. While there are many more facets to explore, I highly recommend exploring some of these topics:

- **Random Access**: Most of the time, when performing I/O, you will access the data sequentially; that is, you will read in one piece of data, then the next, etc. However, in some cases you might know in advance that you want to look up only a certain piece of data in a file without considering all of the data before it. For example, a ZIP archive containing a directory structure most likely stores each compressed file at a different offset from the start of the file. Thus, if you wanted to write a program capable of extracting a single file from the archive, you'd almost certainly need the ability to jump to arbitrary locations in a file. C++ streams support this functionality with the `seekg`, `tellg`, `seekp`, and `tellp` functions (the first two for `istream`s, the latter for `ostream`s). Random access lets you quickly jump to single records in large data blocks and can be useful in data file design.

- **`read` and `write`**: When you write numeric data to a stream, you're actually converting them into sequences of characters that represent those numbers. For example, when you print out the four-byte value `78979871`, you're using eight bytes to represent the data on screen or in a file – one for each character. These extra bytes can quickly add up, and it's actually possible to have on-disk representations of data that are more than twice as large as the data stored in memory. To get around this, C++ streams let you directly write data from memory onto disk without any formatting. All `ostream`s support a `write` function that writes unformatted data to a stream, and `istream`s support `read` to read unformatted data from a stream into memory. When used well, these functions can cut file loading times and reduce disk space usage. For example, The CS106B/X `Lexicon` class uses `read` to quickly load its data file into memory.

Practice Problems

Here are some questions to help you play around with the material from this chapter. Try some of these out; you'll be a better coder for the effort.

1. How do you write data to a file in C++?

2. What does the `setw` manipulator do? What does the `setfill` manipulator do? How do you use them?

3. What is stream failure? How do you check for it?

4. What is a `stringstream`?

5. Using a `stringstream`, write a function that converts an `int` into a `string`.

6. Modify the code for `GetInteger` to create a function `GetReal` that reads a real number from the user. How much did you need to modify to make this code work?

7. Using the code for `GetInteger` and the `boolalpha` stream manipulator, write a function `GetBoolean` that waits for the user to enter "true" or "false" and returns the corresponding boolean value.

8. In common usage, numbers are written in *decimal* or *base 10*. This means that a string of digits is interpreted as a sum of multiples of powers of ten. For example, the number 137 is 1·100 + 3·10 + 7·1, which is the same as $1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$. However, it is possible to write numbers in other bases as well. For example, *octal*, or base 8, encodes numbers as sums of multiples of powers of eight. For example, 137 in octal would be $1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$ in decimal.* Similarly, *binary*, or base 2, uses powers of two.

   When working in a particular base, we only use digits from 0 up to that base. Thus in base 10 we use the digits zero through nine, while in base five the only digits would be 0, 1, 2, 3, and 4. This means that 57 is not a valid base-five number and 93 is not a valid octal number. When working in bases numbered higher than ten, it is customary to use letters from the beginning of the alphabet as digits. For example, in *hexadecimal*, or base 16, one counts 0, 1, 2, ..., 9, A, B, C, D, E, F, 10. This means that 3D45E is a valid hexadecimal number, as is DEADBEEF or DEFACED.

   Write a function `HasHexLetters` that accepts an `int` and returns whether or not that integer's hexadecimal representation contains letters. *(Hint: you'll need to use the `hex` and `dec` stream manipulators in conjunction with a `stringstream`. Try to solve this problem without brute-forcing it: leverage off the streams library instead of using loops.)* ♦

9. Although the console does not naturally lend itself to graphics programming, it is possible to draw rudimentary approximations of polygons by printing out multiple copies of a character at the proper location. For example, we can draw a triangle by drawing a single character on one line, then three on the next, five on the line after that, etc. For example:

```
    #
   ###
  #####
 #######
#########
```

   Using the `setw` and `setfill` stream manipulators, write a function `DrawTriangle` that takes in an `int` corresponding to the height of the triangle and a `char` representing a character to print, then draws a triangle of the specified height using that character. The triangle should be aligned so that the bottom row starts at the beginning of its line.

10. Write a function `OpenFile` that accepts as input an `ifstream` by reference and prompts the user for the name of a file. If the file can be found, `OpenFile` should return with the `ifstream` opened to read that file. Otherwise, `OpenFile` should print an error message and reprompt the user. *(Hint: If you try to open a nonexistent file with an `ifstream`, the stream goes into a fail state and you will need to use `.clear()` to restore it before trying again)*.

* Why do programmers always confuse Halloween and Christmas? Because 31 Oct = 25 Dec. ☺