C++ IOStream Library

Introduction

The IOStream library is C++'s way of formatting input and output to a variety of sources, including the console, files, and string buffers. However, like most parts of the C++ standard library, the IOStream library has a large number of features and idiosyncrasies that can take some time to adjust to. These first two lectures serve as an introduction to many features of the streams library and include some very useful tips and tricks for practical programming.

`cout` and `cin`

As you will see in CS106B/X, C++ provides a stream object called `cout` (**c**haracter **out**put) which you can use to write formatted data to the console. For example, to print out a message to the user, you can write code that looks like this:

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

Here, the << operator is called the *stream insertion operator* and instructs C++ to push data into a stream.

To build a truly interactive program, however, we'll need to have some way to get input from the user. In CS106B/X, we provide you several functions that take care of user input in the `simpio.h` header, namely `GetLine`, `GetInteger`, `GetReal`, and `GetLong`. While these functions are useful, they're not part of the C++ standard library, and outside of CS106 will not be available. Don't worry, though, because by the end of this handout we'll see how to rewrite them using only standard C++.

Just as C++ provides `cout` for character output, the IOStream library exports another stream object called `cin` (**c**haracter **in**put) which lets you read values from the user. To use the `cin` stream to read a value, you use the *stream extraction operator* >> to read data from `cin` into the specified variable. 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
```

You can also read multiple values from `cin` by chaining together the stream extraction operator, much in the same way that you can write multiple values to `cout` by chaining the stream insertion operator. For example, the following code is perfectly legal:

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

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

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

In practice, it is not a particularly 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 malfunction. We will cover how to fix these problems in later sections.

Reading and Writing Files

One major advantage of the IOStream library is that the syntax for using different types of stream objects is uniform. That is, once you know how to read and write values to and from the console, you can use similar syntax to read and write files on disk.

C++ provides a header file called `<fstream>` (for **f**ile **stream**) which exports the `ifstream` and `ofstream` types. These objects represent streams that can read from and write to files on disk. As you might have guessed, `ifstream`s are used for input, while `ofstream`s are used for output. There is also a generic `fstream` class which can do both input and output, but we won't cover it in this class.

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 the final line looks almost identical to code that reads an integer from the console.

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

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

The output counterpart to `ifstream` is `ofstream`. As with `ifstream`, you specify which file to write to with an `ofstream` either by using the `open` member function or by specifying the file when you create the object, 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!

When Streams Go Bad

When sending output data to a stream, chances are the operation will succeed. However, when reading data from a stream it's possible to come across data that doesn't match the expected format. For example,

consider the following code snippet, which reads a certain number of integer values from of 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` does indeed contain `NUM_INTS` integer values in a row, then this code will work correctly. However, what happens if the file contains some other type of data, such as the string "This is not an integer"? In this case, you will run into trouble because your program attempted to read an integer value from the stream, but the stream instead contained string data.

If you try to read data from a stream that does not match the expected format, rather than crashing the program or filling the value 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 on it will automatically and silently fail, which can be a serious problem.

C++ defines three types of error states: end-of-file state, fail state, and bad state. In an end-of-file ("eof") state, the stream reached the end of its input source and was unable to continue reading. In this case, your program has no more data and should respond appropriately. The bad state is rare and encountered only during serious I/O errors from which recovery might not be possible (reading from a corrupted file, for example). In this case, the stream object is most likely unusable and there's not much you can do to fix the problem.

The fail state is the most common and most vexing of the above error states and results when a stream tries to read in data that doesn't match the expected formatting. For example, given this code fragment:

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

If the user enters something other than a number, for example, "Young," the stream has no idea what it's looking at. The line `cin >> age` is supposed to read in an integer, but the user provided it a string instead. Instead of filling age in with any value at all, the stream goes into a fail state and doesn't modify the contents of the integer. Worse, because the stream is in a fail state, any later use of `cin` to read data will automatically fail, meaning code you've written in entirely different functions might stop working properly. Hardly what you wanted!

You can check if a stream is in an error state by using the `fail` member function. Don't let the name mislead you – `fail` checks if a stream is in a fail state, rather than putting it 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. To do this, use the `clear()` member function to bring the stream out of its error state.

The following table summarizes the member functions you can use to check for error states:

Note: We have not covered the `const` keyword yet. For now, it's safe to ignore it.

`bool good() const`	`if(myStream.good()) { /* ... no problems ... */ }` Returns `true` if there are no errors associated with this stream, `false` otherwise. This function is *not* the opposite of `bad()`.
`bool bad() const`	`if(myStream.bad()) { /* ... serious I/O error ... */ }` Returns `true` if a serious I/O error occurred on the stream, `false` otherwise. If this function returns `true`, the stream may be entirely invalid. This function is *not* the opposite of `good()`.

`bool fail() const`
`bool operator !() const`
	`if(myStream.fail()) { /* ... operation failed ... */}`
	`if(!myStream) { /* ... I/O operation failed ... */ }`
	
	Returns `true` if a stream operation has failed that wasn't caused by reaching the end of the file, `false` otherwise. Unlike an error signaled by `bad()`, it's usually possible to recover from an error signaled by `fail()`.

`bool eof() const`	`if(myStream.eof()) { ... hit the end of the file ... }` Returns whether the stream has reached the end of the file.
`void clear()`	`myStream.clear();  // Stream is now valid.` Clears any associated error states on the stream. If the stream is in an error state that is recoverable, make sure to call `clear()` to remove the error state before proceeding.

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 go into a fail state. But what if the user provides *too much* information? For example, 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 a fail state. However, this isn't quite what happens. The first call, `cin >> age`, will fill the value of `age` in with `2`. The next call, `cin >> hourlyWage`, rather than prompting the user, will find the value `.71828` from the earlier input and fill in `hourlyWage` with that information. Essentially, the stream stored too much information (or, rather, you didn't take out enough). As if this wasn't bad enough, suppose you had 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();
}
```

If the password is "password," what happens if someone enters `password y` as the password? The first call, `cin >> password`, will only pull out `password` from the stream. Once you reach the second `cin` call, it automatically fills it in with the leftover `y`, and oh dear, there goes your hard drive! Clearly this is not what you intended.

As you can see, reading directly from `cin` can be a real hassle and almost poses more problems than it solves. In CS106B/X we provide you the simpio.h library primarily so you don't have to deal with these sorts of errors in your programs. In the next section, we'll explore an entirely different way of reading input from the console that bypasses most of the above problems.

getline

Up to this point, we have been reading data from streams using the stream extraction operator, which, as you've seen, has several problematic idiosyncrasies. However, there are several other ways to extract data from a stream. One of these functions, called `getline`, reads characters from a stream until a newline character is encountered, then stores the read characters into a string of your choice. The newline character is not stored.

`getline` is a free function which 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 the string. Moreover, because any text data can be expressed as a string, unless your input stream encounters a read error `getline` will not put a 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 CS106 `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;
}
```

* Technically, the implementation of `GetLine` from `simpio.h` is slightly different, as it checks to make sure that `cin` is not in a fail state before reading. These details are irrelevant, however.

At this point, `getline` may seem like a silver-bullet solution to our input problems. However, `getline` has a small problem when mixed with regular `cin` operations 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 operator >>, the stream passes over all 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 in. 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);
```

The `getline` function will return an empty string. Why? Because unlike a regular `cin` statement, `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 left over 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 using `cin` 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.

`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 them that the level number they wanted to warp to is out of bounds. At first thought, you might try something like

```
int levelNum;  // Initialized, out of bounds.
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).

One solution to this problem is to use another kind of stream object known as a `stringstream`. 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 a temporary string buffer. 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 get the string you've created out of the string using the `str` member function. For example, continuing the above example, we can print out an error message to the user using code like this:

```
int levelNum;     // Initialized, out of bounds.
stringstream messageText;
messageText << "Level " << levelNum << " is out of bounds.";
MessageBoxAlert(messageText.str());
```

Just as you can use `stringstream`s to build strings from data, you can also use them to extract formatted data from string input. For example:

```
stringstream myConverter;
int myInt;
string myString;
double myDouble;
myConverter << "137 Hello 2.71828";       // Fill in string data
myConverter >> myInt >> myString >> myDouble;  // Extract mixed data
```

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 same lines as those provided by `simpio.h`. In this section we'll explore how to write a function called `GetInteger` which prompts the user for an integer, reprompting as necessary.

Recall from the above sections that when reading an integer from `cin`, we encountered two problems. First, the user could enter something that is not an integer, causing `cin` to enter a fail state. Second, the user could enter too much input, such as `137 246` or `Hello 37`, in which case the operation might succeed but would leave extra data. 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 using `getline` for input is that that 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. Thus, using the `GetLine` function from above, 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 ... */
        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 of these 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 trying to extract an integer from our `stringstream` and then 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;
        converter >> result;
        if(!converter.fail())
        {
            /* ... check that there isn't any leftover data ... */
        }
        else cout << "Please enter an integer." << endl;

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

Finally, we need to check to see 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 otherwise can return the value we read. While there are several ways to check if there is any extra input lying around, one simple method is to try 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.

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;
        converter >> result;
        if(!converter.fail())
        {
            char remaining;
            converter >> remaining; // Check for stray input
            if(converter.fail()) // Couldn't read any more, so input is valid
                return result;
            else cout << "Unexpected character: " << remaining << endl;
        }
        else cout << "Please enter an integer." << endl;

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

Stream Manipulators

One of the more powerful features of the C++ IOStream library is its assortment of stream manipulators. Stream manipulators, some of which require the `<iomanip>` header, are objects that modify stream properties without necessarily performing any I/O. For example, you're probably familiar with the `endl` stream manipulator, which appends a newline to a stream and flushes it to its destination:

```
cout << "This is some text" << endl;
```

When you write `cout << endl`, syntactically, it seems like you're pushing `endl` into `cout` just as you would any other piece of data. Indeed, this is one of the main ideas underlying stream manipulators – that they should seamlessly blend into normal stream operations.

I've reprinted some of the more useful stream manipulators below:

`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 [three spaces, then 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 from the console.`
	
	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.

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++ IOStreams 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++ IOStreams let you directly write data from memory onto disk without any formatting. All `ostream`s support the `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 CS106 `Lexicon` class uses `read` to quickly load its data file into memory.

Practice Problems

Don't worry, I'm not going to collect and grade these questions. However, I highly encourage you to try them out to practice the material.

1. Write a function `ExtractFirstToken` that accepts a string and returns the first token from that string. For example, if you passed in the string "Eleanor Roosevelt," the function should return "Eleanor." For our purposes, define a token as a single continuous block of characters with no intervening whitespace. While it's possible to write this using the C library function `isspace` and a `for` loop, there's a much shorter solution leveraging off of a `stringstream`.
2. 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 for loops.)*
3. Using the code for `GetInteger` we covered in class and the `boolalpha` stream manipulator, write a function `GetBoolean` that waits for the user to enter "true" or "false" and returns the value.

Quick Reference: Simplified IOStream Hierarchy