C++ IOStream Library

Introduction

The IOStream library is C++'s way of formatting input and output to a variety of sources, such as the console, files, or 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.

Stream Manipulators

One of the more powerful features of the C++ IOStream library is its assortment of stream manipulators. Stream manipulators, declared in the header file `<iomanip>`, are objects that modify stream properties while maintaining readability. 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.

`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. Useful with `getline`, as shown later.

When Streams Go Bad

When sending output data to a stream, chances are the operation will succeed. However, when reading data from an external (or even internal) source it's possible to come across data that doesn't match the expected format. For example, suppose you open a text file for reading expected to find several integer values, but instead come across a string. In that case, if your code looks something like:

```
ifstream in("input.txt");
int value;
for(int i = 0; i < NUM_LINES; i++)
{
      in >> value;
      // Process 'value' here
}
```

You will run into trouble because the stream won't know what to do when it encounters invalid data. Rather than crashing the program or filling the integer with garbage data, instead the stream totally fails and doesn't change the value of the variable `value`. Once the stream is in this "error state," any subsequent reading or writing operations on it will automatically fail, which can become a big 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!

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 reset a stream's error flags. Now the stream is back to normal and I/O operations on it won't automatically fail.

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()) { ... I/O 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

Suppose you have the following code fragment:

```
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 an automatic 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:

```
string password;
cout << "Enter administrator password: ";
cin >> password;
if(password == "Bubba")
{
      cout << "Do you want to erase your hard drive (Y or N)? ";
      char yesOrNo;
      cin >> yesOrNo;
      if(yesOrNo == 'y')
            EraseHardDrive();
}
```

If the password is "Bubba," what happens if someone enters `Bubba y` as the password? The first call, `cin >> password`, will only pull out `Bubba` 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 want to do.

As you can see, reading directly from `cin` can be a real hassle and almost seems to pose more problems than it solves. This is the reason that in CS106X we provide you the simpio.h library – 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`

The `getline` function is truly wonderful. It lets you read in an entire line of text as a single string. When used for file input, it provides an easy way to get all the data on a line without advance knowledge of the formatting. Used on the `cin` stream, it guarantees a way to read input without the possibility that the user left extra characters in `cin` that might get read later.

For example, to read a multiword string 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. No longer do you need to worry about strange I/O edge cases!

However, `getline` does have 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 dummyInt;
cin >> dummyInt;
cin >> dummyInt;
```

The newline stored in `cin` after the user enters a value for `dummyInt` is eaten by `cin` before the second value of `dummyInt` 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 always 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.

There are two main ways to fix this. First, you can use the `ws` (**w**hite**s**pace) stream manipulator to remove all the whitespace from `cin`, as shown here:

```
int dummyInt;
string dummyString;
cin >> dummyInt >> ws;  // Now works the way you'd expect it to, mostly...
getline(cin, dummyString);
```

While this works well, it is not necessarily the optimal solution because the user might have filled `cin` with additional garbage data in the `cin` call that will fill `dummyString` with information before the user has a chance to respond. The better option, and the one the CS106 libraries use, 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 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). To fix this, you can use a `stringstream`, a stream object that stores its data in a growing string buffer. You could, for example, do this:

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

Except for the last line, most of what is going on here should be self-explanatory. You're creating an object of type `stringstream` and filling it in with formatted data. In the last line, you call the `str()` member function to get the string the `stringstream` object has built for you.

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!
```

As with all streams, you can use stream manipulators on `stringstream`s to format the source data, and if you try to extract data of the wrong type from a `stringstream` the stream will go into a fail state.

`stringstream`s are especially useful in conjunction with `getline`. To read in an object of any type, you can call `getline` to read a line of text from the console, guaranteeing that nothing is left over in `cin`, then pass the string into a `stringstream` to format and extract the data. If the data isn't well-formatted, when you try to extract formatted data the stream will go into failure mode and you can report an error. Similarly, if after you read data from the `stringstream` the stream has any remaining contents (one way to check this is to see if you can read a `char` out of the `stringstream` successfully), you can report to the user that they've entered too much. In fact, this is exactly how the CS106 library function `GetInteger` works!

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 recommending exploring some of these topics:

- **Random Access:** Many data files store multiple pieces of information in contiguous blocks. 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. 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 RAM. To get around this, C++ IOStreams let you directly write data from RAM 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 Questions

Don't worry, I'm not going to collect and grade these questions. However, I highly encourage you to try them out. It's one thing to see the IOStream library worked out in front of you, but it's a whole other thing to try to make it work for you!

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