Programming Notes 8: File IO

As usual, more examples of the notes are in the example program.

1. Files

In the past, you've been limited to input and output to and from the console window. No longer! File IO is simply how you can take input data from files, for example a text file, and output other data to files! It's not very complicated, so this should be a pretty short topic.

2. Output to files

File output is a bit simpler than file input—it's essentially the same as what you've been doing with "cout," except with files. First, if you are going to do any file input or output, you need to include the "fstream" header, in the same was as you include the "iostream" header. Then, if you want to output to a file, you declare a variable of type "ofstream." Example:

```
ofstream fout;      Declares a file stream called "fout" that will be used for output.
```

Then, to open a file, you can call the method "open" with the name of your file as the parameter. If the file you specify already exists in the same directory as your executable, it will overwrite the file, and if not, your program will create a new file with your file name. Additionally, while it can be useful to create ".txt" files, as windows will automatically recognize it as a text file, but you can use any filename and extension that you want, and it will all work just the same. Note that the file name must be a c-style string, meaning either a character array or a string literal. Example:

```
fout.open("outfile.txt");
```

Finally, you can use your "fout" variable in the exact same way you would use "cout," and your data will be written to the file as you would expect. Example:

```
fout << "Start of file:" << endl;
fout << data << " " << moreData << endl << endl;
```

That's pretty much all there is to basic file output—just remember it works in exactly the same way as outputting to the console.

3. Input from files and .good()

File input is a bit more complicated, as you have to deal with the fact that you may not get the data you expect from the file. However, like file output, it works in much the same way that your input had in the past. Again, to work with files, include the "fstream" header, and to declare a file input variable, declare a variable of type "ifstream." Example:

```
ifstream fin;          Declares a file stream called "fin" that will be used for input.
```

To open a file, it's the same as output—call "open" with a c-style string of your file name. However, if the file does not exist, your program will not create the file—in fact, it won't do anything at all. Well, not absolutely nothing. If your program can't find the specified file, your "fin" variable will be marked as invalid. To test if your file stream is still valid, you can call the method "good." Hence, you can use this to test if you received a legitimate file, or, as you will see in a second, to test if your inputs worked correctly. Example:

```
fin.open("infile.txt");

if(fin.good()) {
  //Do stuff
} else {
  cout << "File was invalid!" << endl;
}
```

Once you have opened your input file, you can proceed how you would expect. File input works in the exact same was as input from the console, except the user doesn't have to enter anything, as the data should already be described in the file. Remember that when you input a value using the extraction operator, where you are in the file will move forward until past the data. Additionally, when you input using the extraction operator, if your program sees whitespace (spaces, tabs, or newlines) in the file, it will automatically skip over them until it comes to a valid piece of data. Finally, when creating a data file, because of how the system works, be sure to add a blank line after the end of your data, or your program may see the last data input from the file as invalid. Example:

```
fin >> someData >> moreData;
```

If the file was simply "100 234," the first input to "someData" would move your program forward in the file, so it would see " 234" – note that there is still a space in front of the 234. Then, when you input to the variable "moreData," your proram will skip over any white space in the file (spaces, tabs, newlines), until it comes to the next piece of data, 234, and input it.

Finally, you may be wondering what happens when your file does not have the correct data, or the file ends before you're done inputting it. Well, this works in the same way as if you try to open a bad file—if your program fails at inputting the data, meaning either for example, you try to input a character into an integer, or the file doesn't have any more data to input, your file stream variable will be marked as invalid, and you can test if it is alright by using the "good" method. For example, after you input data, you usually want to test if the data was inputted correctly, as if it was not, you don't know if you have good data or not, so you should probably output an error and maybe try to input the data again. Example:

```
for(int i = 0; i < 10; i++) {
  fin >> array[i];                    Input 10 values from the file.
}

if(!fin.good())
  cout << "Data did not input correctly!" << endl;
```

5. Three methods of file input

Using file input in the exact same way as console input is useful for inputting single values and the like, for inputting more data (which you will normally be doing) there are tree general methods of doing so.

First, there is the simplest, but it requires that you know how much data you want to input before you actually run your program. The method is simply to loop the number of times you want to input, and input a value each time. Example:

```
for(int i = 0; i < 10; i++) {
  fin >> array[i];
}
```

This inputs 10 values from the fin file stream to an array.

Second, you can specify the amount of data you want to input within the file itself. Basically, you put the number of values before the actual values, and loop that many times to input the actual data. Example:

File: 5 9 34 23 1 8

```
fin >> numValues;
for(int i = 0; i < numValues; i++) {
  fin >> array[i];
}
```

This will first input that there are 5 values in the file, then loop 5 times to input the values.

Finally, the third type of input is looping input until you reach the end of the file. This is arguably the easiest way to input data, but it can only be used if you want to input all the data in the file until the end. To implement this type of input, you want to keep inputting values until the end of the file, so you can do something like this:

```
do {
  fin >> dummyData;
while(fin.good());
```

This code will input a value from the file, and if the file is still valid, it will input another value, and so on.

However, you have to be careful when using this type of file input. For example, you probably noticed that the code uses a do/while loop rather than simply a while loop. This is because you need to input the value before it checks if the file is still good. Why does the program need to do that? Well, if you test the validity of the file before you input the value, when you get to the last legitimate value, you will input another value before the program realized you've reached the end of the file. This is another one of those pesky off-by-one errors, except this time it will give you an extra, garbage value.

6. Useful IO functions

These are a few functions that can be called from your file variables, or, actually, from cin. These will be very useful in formatting and checking your input.

a. getline()

This first function is not actually called from "cin" or a file stream variable, but simply called on its own. As the name suggests, it is used to input an entire line of a file, rather than just one word. For example, if you entered the text "first last," and used the extraction operator to input a string, you'd only get the word "first." However, if you use the getline function, you can input the entire line "first last" as one string. getline takes three parameters: the input stream you want to take from (i.e. "cin" or "fin"), the string variable you want to input into, and the delimiting character. By default, the delimiting character is the newline character, which means the function will input text until the end of the line. However, if you want, you can specify a different character to input until. Example:

getline(cin,stringVar); Inputs a line from the console.

getline(fin,stringVar,':'); Inputs text from a file until it gets to a colon.

b. .getline()

This is almost exactly the same as getline(), except it is used with c-style strings, and it is called from cin or a file variable. It also takes slightly different parameters: the character array, or c-string, the maximum characters to input if the delimiting character is not reached, and again, optionally a delimiting character. Example:

cin.getline(charArray,1000); Inputs text from the console until it gets to the end of the line or inputs 1000 characters.

fin.getline(charArray,1000,':'); Inputs text from a file until it gets to a colon or inputs 1000 characters.

c. .ignore()

Ignore is a very simple but very useful function: it is used to simply skip data from a file if it is irrelevant. For example, if you had a file containing "Data: 1 2 3 4," you can't just start inputting the values, as the word "Data" is in the way. With ignore, you can skip the label and get straight to the data. Ignore is called from your file variable or "cin," and it takes two parameters: the maximum number of characters to ignore, and the delimiting character. Note that ignore will throw away the data up to and INCLUDING the delimiting character, so you would want to ignore until a colon in the above example. Example:

cin.ignore(1000,'\n'); Ignores the console input until the end of the line, or after 1000 characters.

fin.ignore(1000,':'); Ignores the data in the file until it reaches a colon, or has ignored 1000 characters.

d. .get()
e. .peek()

Get and peek are very simple functions: they return the next character in the input stream, but do not ignore white space like the extraction operator. This means that if the program is at " data" in the file, get() will give you a space, whereas the extraction operator would ignore the space and give you "data." The difference between get() and peek() is very simple—get() will remove the character from the input stream when it gives it to you, whereas peek() will return the character without removing it from the input stream. Both are called from "cin" or your file variable, and do not take any parameters. Example

charTest = cin.get(); Gives you the next character from the console input.

peekChar = fin.peek(); Gives you the next character in the file without removing it from the stream.