File Input and Output

In addition to the input and output streams we are familiar with (cin and cout) we can also define streams that read and write information from an external file.

This is particularly useful when dealing with large amounts of data that you (or a user) don't want to take the time to input by hand.

For starters we will need to declare a new class to deal with opening, closing, input, and output of files.

```
#include <fstream>
```

Input

For starters lets look at reading data from a file.

To do this we first need to be able to read from a file.

To work with files we basically declare an instance of the object by using the I/O class as the type and the file name as a parameter.

For input the specific class is `ifstream`

So, opening a file and declaring an instance of it would look like:

```
ifstream inf("Sample.dat");
```

`Inf` becomes the object that we can then use like `cin`.

Notice we used a literal to pass the file's name. We could also do the same by passing it a string that came from the user. If we want to ask them what file to open.

From there we may want to make sure that the file opened successfully. If the file doesn't exist or the name is wrong there is no sense in continuing.

```
// If we couldn't open the output file stream for reading
if (!inf)
{
     // Print an error and exit
     cout << "Uh oh, Sample.dat could not be opened for
     reading!" << endl;

return 0;
}
```

This will treat the success of inf being declared as a boolen and will print this message if "not inf".

I also used a return 0 to kill the function I am in. If that function is not the main it may take more thought on how to end the program.

Then we will use the object like cin to run a loop that will pull and print all the information in the file.

```
// While there's still stuff left to read
while (inf)
{
     // read stuff from the file into a string and print it
     string strInput;
     inf >> strInput;
     cout << strInput << endl;
}
```

By using inf in the while loop the program will continue to read from the file until it doesn't find anything more in the file.

Speaking of the data lets look at the contents of Sample.dat:

```
     This is line 1
     This is line 2
```

The program should read this information from the file and print it to the screen.

However, it's not going to print it in quite the way we think.

Running it as is will produce the output:

```
This
is
line
1
This
is
line
2
```

Remember when we first learned to use the `cin` operator with strings we had issues when there was a space in the stream.

When there is going to be a space in the data file (which causes the stream to break) we should consider using the `getline` command.

```
// read stuff from the file into a string and print it
string strInput;
getline(inf,strInput);
cout << strInput << endl;
```

Running the program with this code will print to the screen exactly as it appears in the data file.

Output

Outputting to a file is very similar to the setup we used for inputting from a file. (It can be more complex as needed, more on that later.)

We need to once again declare our stream and the name of the file we wish to output to:

```
ofstream outf("Sample.dat");
```

This time we use "outf" in place of our "cout" when writing to the file.

Otherwise we can output to the file in a similar fashion as inputting by using commands such as:

```
outf << "This is line 1" << endl;
```

Notice that I can use conventions such as "endl" to format my output.

In fact, I can use such things as the iomanip commands to format my output when appropriate.

If you want to write to the file using a set number of decimal places use commands such as:

```
outf.setf(ios::fixed);
outf << setprecision(4);
```

An example of an outputting program similar to our infile example would be:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
// ofstream is used for writing files
// We'll make a file called Sample.dat

ofstream outf("Sample.dat");

// If we couldn't open the output file stream for writing

if (!outf)
{          // Print an error and exit
       cout << "Uh oh, Sample.dat could not be opened for
     writing!" << endl;
return 0;
}

// We'll write two lines into this file

outf << "This is line 1" << endl;
outf << "This is line 2" << endl;

return 0;

// When outf goes out of scope, the ofstream
// destructor will close the file
}
```

Output-File Modes

What happens if we try to write to a file that already exists?

If we run the outfile.cpp program more than once it will simply just overwrite the current file.

What if, instead, we wanted to append some more data to the end of the file?

It turns out that the file stream constructors take an optional second parameter that allows you to specify information about how the file should be opened.

This parameter is called mode, and the valid flags that it accepts live in the Ios class.

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks the end of the file before writing
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist
trunc	Erases the file if it already exists

To be able to add lines to the Sample.dat file we wrote in the outfile.cpp program consider outfile2.cpp where the file is opened in append mode.

```
ofstream outf("Sample.dat", ios::app);
```

This opens the same file, but opens it to write at the end instead of just erasing it all.

We will then add two lines to the bottom of the file:

```
outf << "This is line 3" << endl;
outf << "This is line 4" << endl;
```

Overall, file input and output works much the same as cin and cout if consideration is given to the format of the file given and the way we want to output the file (overwrite versus append).

Input Formatting

Consider an input file where the information is separated by spaces.

We saw in previous examples where the inputs were entire lines such as:

```
This is line 1
```

We had to make use of the getline command to get the entire line and not have the input stream break on the spaces.

We can easily manage this until the input we want is on the same line, but items separated by spaces need to be grouped.

Consider the example of names:

```
James T. Kirk  144
Leonard McCoy  223
```

If we pull separate inputs for first and last name McCoy will work fine but Kirk will have issues because of the middle initial.

If we use a getline command we will get everything, including the number in the same string.

Consider adding a '#' as a delimitating character after each name.

```
James T. Kirk# 144
Leonard McCoy# 223
```

We can then have getline read until it finds the delimitating character bypassing it as a third input:

```
getline(inf, name, '#');
```

The third character will tell getline where the input should break.