Text File I/O

We can use essentially the same techniques we've been using to input from the keyboard and output to the screen and just apply them to files instead.

If you want to prepare input data ahead, you may store the data in a file and direct the program to read its input from a file. If you want to save output data in a file to use later, you may direct the program to write data to a file. To read and/or write to a text file, perform the following steps:

1. Request the preprocessor to include file **<fstream>** as well as file **<iostream>**. The former contains the declarations for defining input and output streams other than **cin** and **cout**.
2. Declare an input stream to be of type **ifstream** or an output stream to be of type **ofstream** (or both).
3. Prepare the streams for use by using the function named **open** provided in file **<fstream>**. The parameter for function **open** is the external name of the file. The external name is the name under which the file is stored on the disk.
4. Put the file name to the left of the insertion or extraction operator.
5. When finished, use the **close** function to close the file.

Here is an example program that reads four floating point data values from a file and writes them to another file in reverse order.

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

int main()
{
    float val1, val2, val3, val4;   // declares 4 variables
    ifstream inData;                // declares input stream
    ofstream outData;               // declares output stream

    inData.open("Data.In");
    // binds program variable inData to file "Data.In"
    outData.open("Data.Out");
    // binds program variable outData to file "Data.Out"

    inData  >> val1 >> val2 >> val3 >> val4;    // inputs 4 values
    outData  << val4  << endl;
    outData  << val3  << endl;
    outData  << val2  << endl;
    outData  << val1  << endl;                  // outputs 4 values
    inData.close();
    outData.close();
    return 0;
}
```

To reference a file in your program you need both an internal name and an external name. The internal name is what you call it in your program; the external name is the name the operating system knows it by. Somehow these two names must be associated with one

another. This association is called *binding* and is done in function **open**. Notice that **inData** and **outData** are identifiers declared in your program; "**Data.In**" and "**Data.Out**" are character strings. **Data.In** is the name that was used when the input data file was created; **Data.Out** is the name of the file where the answers are stored.

When the above program is run, if Data.In contains:

3.4
5.1
3.13145
55.23

then afterwards, a new file name Data.Out will be created that contains:

55.23
3.13145
5.1
3.4

Note that the filenames are case-sensitive on Unix systems. **If an entire path is not declared, it is expected that the file is in the current working directory. In Visual Studio, this is in the main project folder, then in the sub-folder with the project name.** If you specify the entire path to your file don't forget to use the double slashes to escape the file:

```
myFile.open("c:\\My Documents\\In.txt");
```

C++11 allows you to use a literal string to avoid escaping the backslash:

```
myFile.open(R"(c:\MyDocs\In.txt)");
```

It is generally desirable to avoid putting in the full pathname. Instead, a relative pathname using the default folder is preferred. This is because a full pathname is normally unique to just your machine, so relying on this pathname will usually not work on someone else's computer. However, if the default folder is used, this will generally operate on anyone's computer.

After opening the file, it's a very good idea to check to see if the open has failed. Upon failure the file variable will be equal to false. Failure could occur for many reasons (file not found, disk full, etc.) If successful, the file variable will be non-null:

```
myFile.open("somefile.txt");
if (myFile)
       cout << "Open succeeded");
else
       cout << "Open failed";
```

What if we didn't know in advance how many data items were in a file, and wanted to process it until we reached the end of the file? There is a function we can use to test for the end of the file, the **eof()** function. To test for the end of a file, use the variable name for the input stream, with the dotted notation: **ifvar.eof().** This returns true if the end of the file has been reached, and false otherwise. The following example reads the file named "test.txt" and outputs it to the screen:

```
int main()
{
      string s;
      ifstream inData;

      inData.open("test.txt");
      while (!inData.eof())           // While not the end of the file
      {
            getline(inData, s);       // Read a line of text
            cout << s << endl;        // Print it to the screen
      }
      inData.close();
      return 0;
}
```

As long as we haven't reached the end of the file, we'll continue reading a line of text in and print it out to the screen. For most programs, instead of printing the data to the screen, you would be doing some other form of processing.

This code can run differently on different operating systems!
If you run this in Visual Studio on Windows and also in Unix, you'll notice an extra blank line printed in Unix.

On a Unix system, the eof() call does not become true until we try to read **past** the end of the file. The program above actually reads past the end of the file and tries to print out the string s after we read the EOF marker. In this case we get a blank string when we try to use getline on the end of the file, so this program will always print a blank line at the end. If you don't want this behavior, the last line of the loop should be a getline.

Another way to check to see if we've reached the end of the file is to use the return value of getline(). This function will return NULL or false when it has reached the end of the file. This also works using cin >> var. This is generally the preferred method over using EOF. The result is we input from the file and check for the end of the file at the same time:

```
int main()
{
	string s;
	ifstream inData;

	inData.open("test.txt");
	// Get a line of text and check for EOF at the same time
	while (getline(inData, s))
	{
		cout << s << endl;		// Print it to the screen
	}
	inData.close();
	return 0;
}
```

```
int main()
{
	string s;
	ifstream inData;

	inData.open("test.txt");
	// Read a word at a time and output it until EOF
	while (inData >> s)
	{
		cout << s << endl;		// Print it to the screen
	}
	inData.close();
	return 0;
}
```

Passing File Variables to Functions

One final note about using file variables, if you wish to pass a file stream variable as a parameter in a function, the variable **must be passed by reference**.

We have only covered the basics on file manipulation here. If you would like to know more (e.g. random access into files, appending to files, peeking into files, putting data back into file streams) then please refer to the textbook.

Binary File I/O

To read and write files in binary we open them the same way, except use functions to read and write the file instead of the stream operators. There is a flag we append to the file to indicate we want to open it in binary.

Here is an example that writes a double and an array of integers to a binary file. We can use the write function to save to a binary file. This function requires the first parameter be a char pointer, so we have to typecast to a (char *) from whatever data we are writing. The second parameter is the number of bytes to write.

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

int main()
{
    double d = 3.14;
    int x[] = {0, 1, 2, 15, 14};
    ofstream outfile;
    outfile.open("binary.dat", std::ios::binary);

    outfile.write((char *) &d, sizeof(double));
    // put type into sizeof

    outfile.write(reinterpret_cast<char *>(x), 5 * sizeof(int));
    // How many bytes to write.  Reinterpret_cast is another way
    // to typecast at runtime to a char pointer.
    // We don't need &x since x is already a pointer.

    outfile.close();

    return 0;
}
```

This creates a binary file named **binary.dat**. If you try to open it with a text editor you will get garble. You could read it with a hex or binary file editor:

```
00000000   1F 85 EB 51  B8 1E 09 40  00 00 00 00  01 00 00 00  02 00 00 00  0F
00 00 00  ...Q...@.............
00000018   0E 00 00 00
```

To read from the file we use the read function in a manner similar to the write function:

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

int main()
{
    double d = 0;
    int x[5];
```

```
    ifstream infile;
    infile.open("binary.dat", std::ios::binary);

    infile.read((char *) &d, sizeof(double));
    infile.read(reinterpret_cast<char *>(x), 5 * sizeof(int));
    infile.close();

    cout << "we read in: " << d << endl;
    for (int i = 0; i < 5; i++)
        cout << i << " " << x[i] << endl;

    return 0;
}
```

This program outputs the data values stored from the first program.

Binary has the advantage over text that it will generally take up less storage and makes it easy to write out large structures or arrays. It is generally more efficient than text. Text has the advantage of being easier to edit in file format.