MECH215
Lecture Notes 11

File Input and Output

The C++ library <fstream> contains definitions of three STREAM class types:

ifstream - this class is used to create streams which are used to input values from
ofstream - this class is used to create streams which are used for output
fstream - used to create streams capable of both input and output

We will study the ifstream and ofstream classes only.

Here is a simple program which creates an input file and an output file and simply reads integers from the input file and writes them to the output file:

```
// Author: Ted Obuchowicz and John Peach
// FileIO.cpp

#include <fstream> // fstream library defines stream classes ifstream
                   // ofstream and fstream

using namespace std;

int main() {

  ifstream infile("myinput.dat"); // associate the file myinput.dat with
                                  // infile - object of class ifstream

  ofstream outfile("myoutput.dat");  // associate the file myoutput.dat with
                                     // the outfile object of class ofstream

  int number;

  // a simple while loop which reads numbers from the infile
  // and writes them to the outfile object.  Loop will terminate
  // when the end of the infile object is reached

  while ( infile >> number ) {
    outfile << number << endl;
  }

  return 0;
}
```

When this program is executed, the input will be taken from a disk file called myinput.dat, the output will be written to a disk file called myoutput.dat.

For example, the contents of the disk file myinput.dat are:

```
1 3 5 7 9
2 4 6 8 10
0 1 0 1 0
```

After running the program, the file myoutput.dat will contain:

```
1
3
5
7
9
2
4
6
8
10
0
1
0
1
0
```

The following example shows a variation of the method used to associate a file name with an object of type class ifstream or ofstream:

```
// Author: Ted Obuchowicz and John Peach
// FileIOOpen.cpp

#include <fstream>
using namespace std;

int main() {

  ifstream infile;  // object called infile of type ifstream
  ofstream outfile; // object called ofstream of type ofstream

  infile.open("myinput.dat");  // associate the diskfile called
                               // myinput.dat with object infile

  outfile.open("myoutput.dat"); // associate the diskfile called
                                // myoutput.dat with object outfile

  int number;
```

```
  // a simple while loop which reads numbers from the infile
  // and writes them to the outfile object.  Loop will terminate
  // when the end of the infile object is reached

  while ( infile >> number ) {
    outfile << number << endl;
  }

  return 0;
}
```

The program does the exactly same thing as the FileOI.cpp program.

File Modes

The default mode for output to a stream object is to OVERWRITE the file every time the program is invoked. For example, with the previous two examples, if we run the program with different data in the "myinput.dat" file, then the contents in the output file will be overwritten with new output every time the program is executed. If we wish to APPEND to the end of the existing output file, then the file must be opened in a slightly different fashion with an extra parameter:

```
ofstream outfile("myoutput.dat", (ios_base::out | ios_base::app));
```

The second parameter specifies that the file "myoutput.dat" is to be opened for output and that any writing to the file is to be APPENEDED to the current end of the file.

We can also specify the mode as a parameter to the open() method if we use the second style of specifying a file name with a stream object:

```
ofstream outfile;
outfile.open("myoutput.dat", (ios_base::out | ios_base::app));
```

Some compilers do not support the ios_base::out | ios_base::app (for example the g++ compiler does not support it). Check your compiler documentation to see which modes are supported.

Opening and Closing Files

We have already seen examples of opening files. Let us see some examples of closing files. In addition to the member function open(), objects of type ifstream or ofstream also have the member function close() - this is typically used to break a current association of a physical file with a file stream object. The program can then make a new association between a new file and the given stream object.

The following program reads from two different files and write the output to to different files using only two file stream objects : infile and outfile:

```
// Author: Ted Obuchowicz and John Peach
// FileClose.cpp

#include <fstream>
using namespace std;

int main() {

  ifstream infile;
  ofstream outfile;

  infile.open("input1.dat") ;
  outfile.open("output1.dat");

  int number;

  // a simple while loop which reads numbers from the infile
  // and writes them to the outfile object.  Loop will terminate
  // when the end of the infile object is reached

  while ( infile >> number ) {
    outfile << number << endl;
  }

  // now we close the files and associate the file streams infile
  // and outfile with new disk files:

  infile.close();
  outfile.close();

  infile.open("input2.dat");
  outfile.open("output2.dat");

  // now simply loop reading numbers from the "input2.dat" file
  // and write them to the "output2.dat" file

  while ( infile >> number ) {
    outfile << number << endl;
  }

  // although not absolutely necessary, it is good
  // programming practice to close any open files
  // before the program terminates...

  infile.close();  // "input2.dat" will be closed
  outfile.close(); // "output2.dat" will be closed
```

```
  return 0;
}
```

Reading and Writing Different Data Types From and to Text Files

The following program illustrates how different data types may be read from and written to ASCII text files. It makes use of the <string> library as well as the <fstream> library.

```
// Author: Ted Obuchowicz and John Peach
// ReadDifferentDataTypes.cpp

#include <string>
#include <fstream>
using namespace std;

int main() {

  ifstream infile("inputDiffData.dat");
  ofstream outfile("outputDiffData.dat");

  int number;
  string aString;
  float aFloat;

  while ( infile >> number >> aString >> aFloat ) {
    outfile << number << aString << aFloat << endl;
  }

  return 0;
}
```

```
The contents of the inputDiffData.dat file are:

123 Hello 3.4
2 goodbye 6.456
-12 different  999.9
99 a 2.3
```

When we run the program, the ASCII text file "outputDiffData.dat" will be created and have as contents:

```
123Hello3.4
2goodbye6.456
-12different999.9
99a2.3
```

Note how that during the input operation (reading in from the text file and storing the value read into the variables number, a_string, a_float, white space characters are skipped over. During the output of the variable, white spaces are not inserted into the associated output file stream. If they are desired, one would have to explicitly state so as in:

```
outfile << number << " " << aString << " " << aFloat << endl;
```

If we run the program with this change, the output now has (as expected) a blank character between each data item:

```
123 Hello 3.4
2 goodbye 6.456
-12 different 999.9
99 a 2.3
```

Binary Files

Consider the following program:

```
// Author: Ted Obuchowicz and John Peach
// FileIOFloat.cpp

#include <fstream>
using namespace std;

int main() {

  ifstream infile("myinputFloat.dat");
  ofstream outfile("myoutputFloat.dat");
  float  number;

  while ( infile >> number ) {
    outfile << number << endl;
  }

  return 0;
}
```

If the contents of the input file "myinputFloat.dat" are:

```
1.234567899999999
2.222222229999999
3.333333333333333
4.444444443333333
5.555555553333333
```

```
6.666666663333333
7.777777773333333
8.888888883333333
9.999999999999999
```

What do you think the output file will contain? The answer may surprise you:

```
1.23457
2.22222
3.33333
4.44444
5.55556
6.66667
7.77778
8.88889
10
```

We can note that only 5 digits of precision are used when writing to the output. If we want more digits after the decimal place, we have to explicitly tell the << operator by making use of MANIPULATORS contained in the <iomanip> header file:

```
// Author: Ted Obuchowicz and John Peach
// FileIOPrecision.cpp

#include <fstream>
#include <iomanip>  // needed for the setprecision manipulator
using namespace std;

int main() {

  ifstream infile("myinputFloat.dat");
  ofstream outfile("myoutputFloat.dat");
  float  number;

  // print floats with 16 decimal place precision
  outfile << setprecision(16);

  while ( infile >> number ) {
    outfile << number << endl;
  }

  return 0;
}
```

Now, the contents of the two files nearly identical:

```
1.234567880630493
2.222222328186035
3.333333253860474
4.44444465637207
5.55555534362793
6.666666507720947
7.777777671813965
8.888889312744141
10
```

Note, that both of these files are ASCII text files. When the input is read in, it is converted to a float but floats have a very limited level of precision. When the data is output, there is an error that is introduced and the resulting output is not the same. If the data type is changed from a float to a double, we still have this problem but the error is much smaller. Most engineering calculations are solved in an iterative manner and this small error gets magnified. Thus, it is generally preferable to use a double then a float.

```
// Author: Ted Obuchowicz and John Peach
// FileIODouble.cpp

#include <fstream>
#include <iomanip>  // needed for the setprecision manipulator
using namespace std;

int main() {

  ifstream infile("myinputFloat.dat");
  ofstream outfile("myoutputDouble.dat");
  double  number; // this has been changed from float to double

  // print floats with 16 decimal place precision
  outfile << setprecision(16);

  while ( infile >> number ) {
    outfile << number << endl;
  }

  return 0;
}
```

The contents of myoutputDouble.dat is now:

```
1.234567899999999
2.222222229999999
3.333333333333333
4.444444443333333
```

```
5.555555553333333
6.666666663333333
7.777777773333333
8.888888883333333
9.999999999999998
```

And this is identical to the myinputFloat.dat file.

We can use a different representation to store the numbers in the file. Instead of using 1 ASCII character (which equals 1 byte of storage) to represent every digit of the number, the number can be stored in its binary floating point representation (using the IEEE 754 floating point format which uses 4 bytes for a float and 8 bytes for a double) to represent each number. To specify that we want to use BINARY representation instead of ASCII text, we must open the file and specify ios::binary and we must also use an explicit function called write() to perform the output instead of the outfile << number as in the earlier example.

```
// Author: Ted Obuchowicz and John Peach
// FileIOFloatBinary.cpp

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

int main() {

  ifstream infile("myinputFloat.dat");
  ofstream outfile("myoutputFloatBinary.dat", ios::binary | ios::out);
  double  number;

  outfile << setprecision(16);

  while ( infile >> number ) {
    outfile.write((char *) &number, sizeof(double));
  }
  return 0;
}
```

Let us examine the details of the:

```
outfile.write((char *) &number, sizeof(double))
```

We are invoking a method called 'write' belonging to the object outfile which is of type ofstream. This method 2 arguments:

1. A pointer to a character (actually a pointer to a character constant),
2. An integer specifying the number of bytes to write.

We can make use of the sizeof(double) operator to specify the second argument.

When we run this program, a binary file with filename "myoutputFloatBinary.dat" will be created and it will contain the 9 numbers stored in binary format (each number is presented in 8 bytes of binary information):

```
#> ls -l myoutputFloatBinary.dat
-rw-rw-r-- 1 jpeach jpeach 72 Nov  2 20:25 myoutputFloatBinary.dat
```

Binary files have the advantage of being small in size, so that large amounts of information may be stored in files which do not occupy much room on your hard drive.

The next example, opens the binary file created earlier for input and simply reads in every number and displays it on the screen. It makes use of the read() method.

```
// Author: Ted Obuchowicz and John Peach
// FileIOFloatBinaryRead.cpp

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main() {

  ifstream infile("myoutputFloatBinary.dat", ios::binary | ios::in);
  double  number;

  cout  << setprecision(16);

  while ( !infile.eof() ) { // check for the end of file of infile
    // read a number from the binary input file and save it in the
    // variable called number
    infile.read((char *) &number, sizeof(double));
    cout << number << endl;   // print out the number to the monitor
  }

  return 0;
}
```

The read method expects two arguments :

1. A pointer to a char and
2. An integer specifying the number of bytes to read.

When we run this program, the following output is produced:

```
1.234567899999999
2.222222229999999
3.333333333333333
4.444444443333333
5.555555553333333
6.666666663333333
7.777777773333333
8.888888883333333
9.999999999999998
9.999999999999998
```

IMPORTANT: If a program is reading or writing binary files, it is very important that the file be created on the same type of computer (operating system) as the program which reads the file was compiled . Errors will occur otherwise because different types of hardware store information in different formats.

To illustrate the above, let us see what happens if we we create the binary text file on a Sunblade 100 workstation and then try to read this file by compiling and running the program on a Intel Pentium running the Linux operating system:

I login to the ENCS Linux server called 'tux' and recompile my source code to produce a Linux/Intel executable file:

```
#>g++ -o FileIOFloatBinary FileIOFloatBinary.cpp
```

This program will now attempt to open the file myoutputFloatBinary.dat, which is a binary data file WHICH WAS CREATED using a different hardware/operating system (i.e a Sun workstation running Solaris 9):

```
#> FileIOFloatBinary

-1.70429597257665e+246
4.490634596939233e+300
-3.720662080976632e-103
-3.424421674952914e-29
9.812365294988344e-161
-2.811348471783243e-292
-3.970889723894514e+188
-9.26149346340087e-124
1.168913735987871e+250
1.168913735987871e+250
```

The program misinterprets the binary information stored in the file.