EE 1305. Instructor Ian Scott-Fleming

Lab 08: Binary input/output

For Fall 2015: you MUST do part 1, and turn in your program binary_io.cpp and explanation text file . You can either do part 2 yourself, or study and then compile and run my solution to part 2. If you do not write your own solution for part 2, compile my solution to part 2, run it, and time it, and put the timing info in your explanation file. This is so you have a good feel of how much faster binary I/O is than text-based I/O.

Part 1. Binary File Input and Output

Reading and Writing Binary data:

Background:

The textbook doesn't cover reading and writing binary data, but we covered it in class. If you had trouble following my lectures on it, here are a set of slides by Dr. Fernando Paniagua, on reading and writing binary data. [1] My slides on the subject are on Blackboard on the Course Materials page.

As a reminder:

1. You can read and write data in binary format by opening the file in binary mode and using the ifstream's read(...) and ofstream's write(...) functions. This is easy to do for data in char arrays, and only slightly more complicated for other data types such as int, float and double.

2. To open a file in binary format, you simply provide a second parameter along with the filename: ios::binary

   ```
   ifstream ifs("junk.dat", ios::binary);
   ```

3. read(...) and write(...) both take 2 parameters. The first is the address of a char array in memory where the data should be written from (or read into); the second is the number of bytes to write (or read). Recall that the name of an array is its address in C++.

   ```
   char buf[10];
   ifs.read(buf, 10);
   ```

   For data types other than char, we have to reinterpret_cast the array to make it look like a char array. To do this (assuming an array "int my_ints[10]"),

   ```
   int my_ints[10] = {1,2,3,4,5,6,7,8,9,10};
   ofs.write(reinterpret_cast<char *>(my_ints), 10*sizeof(int));
   ```

`reinterpret_cast<char *>` tells the compiler to treat the address my_ints as if it were the address of a char array. This is needed because read & write both take char *'s as their first arguments, so we have to change the address my my_ints from an int * to a char *.

[1] Note: Dr. Paniagua uses the same filestream (`fstream`) for both reading and writing, by using the flag `ios::read` or `ios::write`. This is the same as opening 2 different filestreams, one an **o**`fstream` (for writing) and the other an **i**`fstream` (for reading), as we've done in this class.

`10*sizeof(int)` is the number of bytes to write. In this case, it's ten times the size of an int (usually 4 bytes).

For those wishing to know a little more about binary file input & output, here's a nice tutorial, which covers the basics (and a little more than I covered in class):
http://courses.cs.vt.edu/cs2604/fall01/binio.html

Part 1.1: Study, understand, compile and run the sample program:

This exercise should help you get clear in your own mind how numbers are stored inside a computer, and what the relationship between the bit patterns for chars, short-int's and int's is, as well as demonstrates how to read and write binary data for all the standard data types.

Here is a sample program and data file (sample_binary.dat). It reads the data file 5 different times: first into a char array, then into a short int array, then into int, float and double arrays. It outputs the values in the arrays (in hex and in integer for the integer data types).

Study the program to understand what it does, then compile and run it, and examine the output. Take some time to make sure the output makes sense to you. Note that the bytes appear reversed (from the hex values for chars) when output in hexadecimal as short ints and ints. This is because Intel processors are "little-endian" – they store the least-significant byte first in memory. However, since we humans expect the most-significant digits first, when a little-endian value is output in hex, c++ outputs the most-significant byte first to keep us happy.

Verify that the integer values output for the first 2 bytes of the char output, 0x74 and 0x68 are correct. You can do that by googling up the ascii table, and finding the hexadecimal and integer values of the characters. Then calculate the integer value of the first short int value, sibuf[0], which is hexadecimal 0x6874, and convince yourself that the integer value of 26740 is correct. If you're not sure how to do that: try 256 * 'h' plus 't' (256*104 + 116) and see if it does add up to the same value as the first short int, sibuf[0].

Then, explain in your own words in a text file:

1. How to calculate the value of a short int and of an int by looking at the values in hexadecimal. Give an example for a short int and an int (suggestion: work out the values for sibuf[1] and ibuf[1]).
2. How the same bytes can represent such different values when read into the different data types
3. Explain in your own words why the hexadecimal values for the chars, short ints and ints aren't in the same order. If you're not clear, google "little-endian" and do a little reading.
4. Take a stab at explaining to yourself why the float and double values can be so wildly varying. (Hint, some of the bits must represent the exponent.) You don't need to write your explanation down; but if you're curious, google IEEE 754. (Wikipedia's page is rather involved, but there's a nice (vaguely) simple example at http://class.ece.iastate.edu/arun/Cpre305/ieee754/ie4.html . (A little googling should also turn up an IEEE 754 converter, which might be helpful to make sense of the values the sample program puts out.)

Part 1.2: Write a program which does the following:

1. Declares two integer arrays, **a** & **b,** both eleven elements long, containing the numbers:
   a. array a: 543516756 1667855729 1919033451 544110447 544763750 1886221674
      1986994291 1948283493 1814062440 544832097 778530660
   b. array b: 0 0 0 0 0 0 0 0 0 0 0
2. Opens a file called "binary.dat" and writes the **array a** to the file in binary format
3. closes the file (be sure to do this, or the rest of the program won't work right!)
4. Opens the file "binary.dat" for reading, and read the data into the **array b**
5. outputs both **arrays a** & **b** to the console so you can verify that you read the data back again properly.

Look at the contents of the file **binary.dat** with a text editor and compare it to the contents of sample_binary.dat. Does it make sense? (To open it with a plain-text editor, such as Notepad app on PC or textedit on Mac, you can copy it to "binary.txt", or select "all files" in the file-open dialog box.) As a text file, it should contain "The quick brown fox...", and if you're not sure how that happened, go back to the sample program above.

Here is the program from class, which may help if you get stuck.

***Name your program* binary_io.cpp**
***Name your explanation file* binary_input_output.txt**

Part 2. Reading and writing binary data is much faster than reading and writing text data.

Here is a program which does the following:
a. creates a char array 64 MB long (like 8-bit pixels, for example).
b. initializes it to random values between -128 and 127
c. writes the values to a data file as (text) integer values (like we have done with our PGM images)

Run this program and time how long it takes to run. (If it takes less than a few seconds, double or quadruple the size of the array.)

Then, write functions to write out and read back in the same data in binary format. Run your program and time how long it takes to run. Calculate approximately how much faster it is to read and write binary, and what the difference in size is between the ascii file and the binary file. Add a few lines to your *binary_input_output.txt* file giving the comparisons.

Here's my solution, if you get stuck.

Name your output binary_timing.cpp

Deliverables:

Final program (main and functions) for parts 1a and 1b, 2a and 2b Turn in:

1. `binary_io.cpp`
2. `binary_timing.cpp`
3. `binary_input_output.txt`

Please do NOT turn in the ascii or binary files of the 64 MB of data!