Lab 6 – File I/O, Binary Edition

Overview

In this assignment, you are going to load a series of files containing data and then searching the loaded data for specific criteria (This sounds oddly familiar…). The files that you load will contain information about various hero characters, some of their attributes (i.e. strength, hit-points, etc.) as well as any items they may be carrying. The data that you load will also be in a **binary format**, which needs to be handled *differently* than **text-based files**.

Description

First things first, the files, there are 2 main files that you will be loading in this assignment:

- *superheroes.dat*
- *fantasyheroes.dat*

There is a sample file included with this lab document, which you can use to test your code. The file is called *SAMPLE_heroes.dat*.

The data you are loading is information about the heroes. In binary files, you have to know the format or pattern of the data in order to read it, as you can't open the file up in a text editor to see what's in it. (Well, you CAN, but… it won't be pretty.)

Hero Data

The output for a hero would look something like this:

```
Name: Gandalf
Str: 9
Int: 22
Agi: 16
HP: 65 / 73
Armor: 15%
Magic armor: 64%
Inventory:
Oak staff, 100, 2.17
Gray robes, 20, 2.44
Wizard hat, 15, 1.5
```

The hero has:

1) A name
2) *Three* attributes stored as *short* variables
3) Hitpoints / Max-hitpoints variables stored as *integers*
4) *Two* armor attributes stored as *floats* from 0-1 (but represented as a **percentage**)
5) An inventory containing a variable number of items, each of which contain a *name*, *integer* cost, and *float* for the weight. If a hero doesn't have any items, the file will still have to indicate a 0. Output-wise, you can just print out "**Inventory empty.**"

Reading binary data

Reading data in binary is all about copying *bytes* (1 byte := 2 nibbles := 8 bits) from a location in a file to a location in memory. When reading data you will **always** use the *read()* function, and when **writing** data you will always use the *write()* function. **For this assignment**, you will only need to *read()* data.

Strings are always an exceptional case. In the case of strings, you should read them in a 4 or 5 step process:

1. Read the length of the string from the file. Unless you are dealing with fixed-length strings (in which case you know the length of the string from somewhere else), it will be *there*, promise. (If someone didn't write this data out to a file, shame on them, they screwed up.)
2. Dynamically allocate an array equal to the size of the string, plus 1 for the null terminator. If the length already includes the null terminator, **do not** add one to the count here — you'd be accounting for it twice, which is bad.
3. Read the string into your newly created buffer.
4. (**OPTIONAL**) Store your dynamic char * in something like a std::string, which manages its own internal memory. Then you don't have to worry about it anymore.
5. Delete the dynamically allocated array. If you did step 4, this should be immediately after you store it in the std::string (so you don't forget to delete it later). If you are planning to use this variable later, be sure to delete it later on down the line.

Refer back to the Powerpoint slides about Binary File I/O for information on how to read and write binary files.

File format

The structure of the files is as follows:

4 bytes	A *count* of how many characters are in the file
"Count" number of characters follow the first 4 bytes. Each character is as follows:	
4 bytes	The *length* of the *name*, including the null terminator
"Length" bytes	The string data for the name, including the null terminator
2 bytes	The character's *strength*
2 bytes	The character's *intelligence*
2 bytes	The character's *agility*
4 bytes	The character's *current hitpoints*
4 bytes	The character's *maximum hitpoints*
4 bytes	A float representing the character's *physical armor* a percentage from 0-1.
4 bytes	A float representing the character's *magical armor* a percentage from 0-1.
4 bytes	A *count* representing the number of items in the character's inventory
"Count" number of Items follow the previous 4 bytes. Each Item is as follows:	
4 bytes	The length of the string representing the *name* of the item, including the null terminator
"Length" bytes	The string *data* for the name of the item, including the null terminator
4 bytes	The value of the item (in gold pieces, diamonds, gil, gems, dollars, whatever currency you want to imagine for this project)
4 bytes	A *floating-point* number for the weight of the item

Searches

After you've loaded the data, you will do a few searches on your heroes:

1. Print all the heroes
2. Print the hero with the most items
3. Who's the strongest?
4. Who has an intelligence greater than 18?
5. Who are the 2 clumsiest heroes (lowest and second-lowest agility)?
6. Which hero has the most valuable inventory?

Sample outputs

```
Clumsiest hero:
Name: Arthur
Str: 12
Int: 18
Agi: 11
HP: 50 / 70
Armor: 35%
Magic armor: 82%
Inventory:
Excalibur, 127, 0.9
Random Item #1, 102, 0.63
Random Item #2, 159, 0.46
Random Item #3, 341, 0.71
Random Item #4, 300, 0.81

Second clumsiest hero:
Name: Star Lord
Str: 14
Int: 15
Agi: 13
HP: 96 / 96
Armor: 78%
Magic armor: 24%
Inventory:
Awesome Mix Vol. 2, 227, 0.97
Random Item #1, 67, 0.71

Press any key to continue . . .
```

2 clumsiest heroes

```
3
Name: Bob
Str: 19
Int: 12
Agi: 16
HP: 44 / 53
Armor: 10%
Magic armor: 40%
Inventory:
   Inventory empty.

Press any key to continue . . .
```

Strongest hero

```
6
Name: Superman
Str: 14
Int: 12
Agi: 13
HP: 46 / 46
Armor: 48%
Magic armor: 90%
Inventory:
Red outer-underwear, 205, 0.91
Random Item #1, 396, 0.94
Random Item #2, 423, 0.08
Random Item #3, 471, 0.46
Random Item #4, 413, 0.03

Press any key to continue . . .
```

Hero with the most valuable stuff
(Superman the hoarder!)

Tips

1. Choices you make at the start of a program can have a big impact on how the rest of the program gets developed. Think about how you want to store the information retrieved from the file, and how you could easily pass that data to various functions you might write.
2. If you have a process for easily loading and accessing the data, the rest of the functionality should be a lot easier to write. Make sure the loading process is all taken care of before worrying about anything else.
3. The code to load 1 file containing 1 piece of data (no matter how complex that data is) should not be much different than loading 100 files, each containing 100 elements. Start by thinking about just 1 element from the file first. Do the values you read match the values in the file? What about 2 entries, does everything add up? Then 3, 4, etc…
4. When passing containers of data, make sure you pass them by REFERENCE, not by value. Creating copies of massive data sets is generally a bad, bad thing… unless you specifically need to duplicate the data. If not? Then pass by reference (in C++ that means either by pointer or the reference data type)
5. There may be a fair amount of repetition in a program like this. Think about where you can create helper functions to reduce the number of times you write the exact same (or just slightly different) code.
6. Boy these tips seem familiar. It's almost like there's a lot of similarity between this and things you've done before… use this to your advantage, now and in the future.
7. Try reading one element at a time. Read the first 4 bytes, try printing it out to the screen. Is the number something reasonable, or something that seems incorrect, like -20? If that works, move on to the next piece of data in the file.