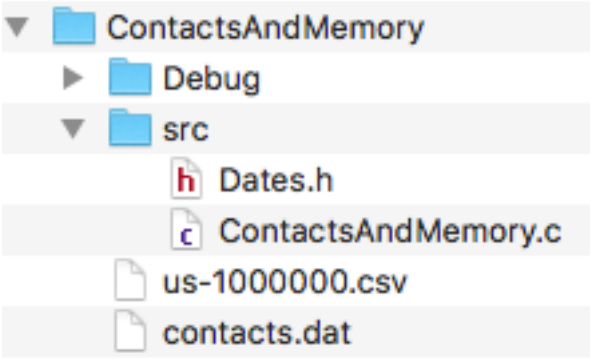
C Binary Files, Malloc, and Structs: Putting It All Together

We now have a good understanding of how C allocates memory, both on the stack and on the heap. We also know how to perform basic file I/O. In addition, we know how to work with basic structures, pointers, file pointers, and memory pointers (used with `malloc`). This assignment is going to help us to crystalize this basic knowledge in our minds; we will learn some practical programming skills around this knowledge that works well. Here, we will be performing memory management, binary file I/O, and storage of a large number of structures (both in memory and in binary files). You may refer to the last two lectures and the related chapters in the book to serve as a refresher for this project.

For this project you will find 3 files in the assignment zip file: Dates.h, Contacts.c, and us-1000000.csv. The .h and .c file will reside in the same directory together, and will be compiled together before you run your program. us-1000000.csv is a text file containing 1 million contact records – 1 record per row. A comma separates each field in the file. This type of file is a typical *comma-delimited file*, also known as file with ***comma-separated values*** (csv). The location of this file can be similar to your last project, or to simplify, you can place it in the working directory/working path of your program. For example, in Eclipse this will be in the root directory of your project (see image). If you have the csv file in the working path, you only need to refer to the file's name in your code, and it will find it in the local path. In the code I give to you, it assumes this. If your path is different, you may have to modify that part of the code to point to the proper path.



Oftentimes csv files hold temporary information that was *exported* from a system like a database or a spreadsheet. This information can be sent to another system where it is *imported* into a native (usually a binary format) data form that is more efficient for storing large amounts of information or for performing complex analysis.

The Assignment

The code included with this assignment has the functionality for reading data from us-1000000.csv and loading all of the contacts contained therein into memory. This is the first step in the import process. The next steps needed to complete the import process is your responsibility: writing the contact data into a binary format. You

will then test that binary file by reading all of the contacts from that file. So, once your program can perform the tasks below, your assignment will be considered a success:

(1) read all data from the csv file
(2) write all that data into a new binary file
(3) read it all back into memory from that new binary file

You are to complete two functions: `loadFromBinary()` and `writeToBinary()`. The prototypes are already done for you, and the code already calls those functions from `main()`. Note that you will need to manage memory for holding the contacts while you read or write the data. There is not much coding you actually need to do. In fact, I calculated that you only need to supply about 15% of the code in Contacts.c – I have already supplied the rest of the code. You probably need to write up to 65 lines of code in total – including comments. You can get away with less than that. Your `writeToBinaryCode()` only needs to be 5 lines of pure code inside of the function block (not counting comments). `loadFromBinary()` can be just 20 lines of code inside of the block (not counting comments). Your code (obviously) may vary, depending on how you decide to go about it. But if you listened in lecture, and took notes, you will know how to "chunk" your data into and out of a binary file. Do not make this harder than it needs to be. But **pay attention to details**… It can be easy to mess things up by not allocating memory properly, by not handling your files properly, or by not relinquishing resources properly.

*Run the program **as-is** before starting to code.* You will notice that **`loadContactCSV()`** has quite a bit of work to do in order to import the cvs data. Fortunately for you, I did not require you to write that code. But you should look it over carefully so that you understand what it does; this includes the subroutines it uses. You will also notice when you run your program that it takes considerable time to read from this csv file. That's because reading and parsing up the text for a million contacts is not very efficient. However, once you get your binary read/write functions working, loading contacts is done in a much simpler and efficient way. That's because we are allocating **chunks of storage** to hold structs that are already lined up accordingly. So, reading (and writing) the binary data should be much simpler and very fast.

Finally…

You will see code towards the end of main() that prints the first 10 contacts in the list after reading from the binary file. Change that code to **print every-other contact starting at 2500** to 2600. Print 2500, but not 2501, print 2502, but not 2503…

Lastly, I want you to **provide the code in `printContactRecord()`.** The output should look likt his for any contact struct passed into the function:

```
*** Contact Information ***
First Name: James
Last Name: Butt
Company: Benton| John B Jr
Address: 6649 N Blue Gum St
City: New Orleans
State: LA
Zip: 70116
Email: jbutt@gmail.com
Web: http://www.bentonjohnbjr.com
Phone: 504-621-8927
[LF HERE to provide separation for other records printed]
```

I want to see good code comments in this assignment. Zip your code up (using ZIP format only). Only the source files should be in your zip file; no paths, no data files (that means do not include the csv file or your new binary file