Project One: Structs and File I/O in C

This project is a modification of exercises three and four from the end of Chapter 5 of McDowell.

Write a C program that uses structs, file input/output, and your own implementation of your favorite sorting algorithm. Your program will read in the contact information from a file, sort the contact information records, and then write the sorted information to another file. The two files are specified as command-line arguments to your main function. Your program must run correctly on agora. You program should be called from the command line as illustrated by the following example.

struct_sort contacts.txt sorted_contacts.txt

or

./struct_sort contacts.txt sorted_contacts.txt

Program Files

- You need a README file that includes your author names, the date of submission, descriptions of how to compile your program and how to run your program, a description of what the program is supposed to do from the user viewpoint, and identify any errors that still exist in your program.
- You need to write your own header file, struct_sort.h, that includes prototypes for all the functions in your program, your typedef declarations, and the declarations of your named constants using preprocessor directives. You should then include your header file in your C program. Recall that in C the declaration of a function has to occur before its use. With this header approach the order of your functions in your C program will not matter since the function prototypes are declarations.
- Besides the header file that you will write, you will have at least one C source code file, struct_sort.c. Your C source code (not including the header file) can all be in our file or split over more than one file if you wish.
- Note your executable file needs to be named **struct_sort**. In particular, do not name it **sort**. The reason is that on agora there is already an executable file named sort in the directory /usr/bin/sort. When you enter a command the agora shell (named bash) looks in directories in the order specified by the PATH environment variable. See the value of the PATH environment variable by the command

  printenv PATH

  Bash will only look in the current directory if an executable by the command name is not in any of the directories specified by the PATH environment variable. So the program /usr/bin/sort will be used instead of your own program. You can cause bash to ignore the PATH environment variable by calling the executable with a “./” in front as in “./sort”. However, just using the name struct_sort for your executable is simpler.
- When debugging your program by using printf() statements always end the string inside the printf with a newline character, '\n', as in “got to point A\n”. The newline character forces the string to be printed to the console instead of being held indefinitely in the output buffer of the input/output runtime.

Input File

- The format of the input file is the contact information for one person is on one line with a comma followed by optional whitespace (blank characters and tab characters) after each field except for the last field. The fields in order are first name, last name, street address, city, state, zip code, and phone number.
- An example input file, **example_contacts.txt**, is on the course website next to this handout. Your code must work with this input file. You can assume that the input file has no errors. For example, no blank lines, the correct number of fields, and fields always ending with a comma. However, a line in the input file can start with whitespace (blank characters or tabs) which is the case for the example input file. You can assume that a line is no longer than 100 characters.
- It is important that you actually download the example input file to your machine and then ftp it to agora. Do not copy and paste its content! Windows and Linux handle newlines differently, so doing copy and paste on a Windows machine will cause the invisible characters in the file to be changed. I will test your program with the original input file, not the Windows version.
- You can assume the input file will not have more than 100 lines (that is, 100 person records; see the definition of a person record below).

Data Structures

- Create a struct using **typedef** to represent the address of person with fields for street address, city, state, and zip code. Create another struct for the contact information of a person that has fields for first name, last name, address, and phone. The address field uses the address struct. This illustrates that fields in structs can be structs themselves. Thus, the definition of the outer struct will have the format of

  typedef struct {…} person_t;

- Then create an array of 100 person records (assuming you defined the named constant NUM_PEOPLE to have the value 100) with

  person_t data[NUM_PEOPLE];

- Common mistakes students make include
  - not allocating space for a char array. For example,

    ```
    char *first_name;
    ```

    does not allocate any space for the char array. It just creates a pointer.
  - Not including space for the null character in the char array. For example,

    ```
    #define STATE 2
    char state[STATE];
    ```

    will not work since you need an array of size three to include the null character at the end.

Input Processing Phase

- Your program should detect if the correct number of command-line arguments is not entered and output a useful error message. See Section 6.1 of McDowell.

- Section 6.1 of McDowell illustrates the first two steps of the input processing.
  - First, you need to use **fopen()** to get a file pointer for the input file.
  - Second, you need to read the input from the input line by line.
    - As illustrated in Section 6.1 you can use **fgets()** to move the input from the input file into a character buffer one line at a time.
    - Alternatively, you can use the **getline()** function to do the same thing. Here is a link showing how to use the **getline()** function: http://c-for-dummies.com/blog/?p=1112.
    - Which should you use? The getline() function was introduced into the C language in 2010 so it is quite new. I find getline() a bit less intuitive to use. The fgets() function works well and the example code using it in McDowell is easy to follow.
- Once you have a line from the input file in a char array buffer, you then need to process the characters in that buffer to put them each set of characters into a field of the record that is for this line of the input file.
  - One approach is to do this collecting of characters and assigning them to a field in the record manually by looking at each char separately.
  - A second approach is to use the string library functions discussed in Section 4.2 of McDowell.
  - A third approach is to use the **getdelim()** function which is in stdio.h.
  - A fourth approach is that you can use **sscanf()** which is like **scanf()** but takes as its first argument a char array that is read from. You would use the regular expression support in **sscanf()** to read the sequence of chars you want for a particular field. In particular, to read the char array to find all the characters that are to be a field in a person record, you can specify that you want to match any characters until a specified delimiter character (or characters) is reached.
- The regular expression support of **scanf()** and **sscanf()** is discussed in McDowell as the last entry in the table on page 15 (about the […] conversion character) and in the last two paragraphs on page 54. The […] conversion character is called a **scanset** in the C99 language standard.
- The brackets of the scanset mean that the regex matches one of the enclosed characters. For example, [ab] matches either the character 'a' or the character 'b'. If the first character is ^, then the ^ has a special meaning which is to invert the set of accepted characters. In other words [^ab] matches any character except for the character 'a' and the character 'b'. Thus, %[^ab] will keep reading characters until the character 'a' or the character 'b' is encountered. In fact, %s is just shorthand for %[^\t\n] which means match any sequence of characters until a tab character, a newline character, or a blank character is encountered.
- So one approach to the input file processing is to have a while loop that in the loop does a fgets() or getline() function call to read the next line of the input file and then
  - uses a sscanf() function call. The sscanf() function calls copies characters from the char array buffer into each of the fields of the person record until a character that does not match the scanset is reached. You need to think about what character you want sscanf() to stop the matching on for a field. Specify that character in your scanset.
  - Alternatively, use the string library functions in Section 4.2 of McDowell instead of the sscanf() function.
- If you read and process the characters in a line of the input file one-by-one (instead of using **fgets()/ getline()** and **sscanf()** or some other approach) there is a great deal of commonality in the reading in of each field of each person record. Instead of repeating the similar code over for each field, in this case you would need to have a **read_field** function that is called once for each field to read in that field.

Sorting the Person Records

- Sort on the person's last name. If two people have the same last name, then sort on their first names.

You need to write your own C code for the sorting algorithm. Using one of the standard $O(n^2)$ sorting algorithms is fine. You cannot use a sorting algorithm that is worse than $O(n^2)$!

- You sorting algorithm needs to be case-insensitive with respect to the names of the people. In other words, it does not matter if a last name is entered as ``Jones" or ``jones". To help with this the **ctype.h** header file can be included to give you access to the **tolower()** and **toupper()** functions to make a single character lowercase or uppercase, respectively. Thus, to make all the characters in a string the same case you need to loop through all of its characters and call **tolower()** or **toupper()** on each one.

Output File Processing

- The format of the output file is like that of the input file including the commas except that any whitespace before the first field on a line has been removed. After the first field there needs to be some whitespace in the output between each field, but the number of whitespace characters does not have to be exactly the same as in the input file. The output must maintain the case of all the characters in the input (though the person records were sorted in a manner that ignores case).

Course Coding Standards

Your solution must follow the course coding standards including

- no lines longer than 80 characters
- no magic numbers; use named constants
- javadoc-like comments at the top of each file (with @author and @version) and above each function (with all the normal javadoc tags)
- use // comments to document all the code in method bodies.
- Function bodies usually should be relatively short and represent a small set of actions that closely relate to each other. Use helper functions to maintain this property and to make the code more “self-documenting” (i.e. have your structure of function calls help explain the structure of your logic).

Doxygen, www.doxygen.org, is a program that can be used to turn javadoc-like comment in a C program into a html documentation page. You do not need to use it, but it is available if you wish to use it.

Unit Testing in C (optional)

Unit testing frameworks are a helpful way to minimizing the number of bugs in your program. I have put on the course website the code for a very simple one in a file named c_unit_testing.c. The file has the url of the web page describing it.

Using GDB (optional)

Debuggers are very useful tools. Integrated Development Environments such as Eclipse have built-in graphical debuggers. The debugger on agora, gdb, is command-line but still quite useful and easy to use. Google to learn more about here are a few key points.

- Compile your program with the -g flag as in (the -Wall is not needed by gdb but displays all compiler warnings)

  gcc -g -Wall struct_sort.c -o struct_sort

- Then start the debugger with your executable as a command-line argument as in

  gdb struct_sort

- Then set some breakpoints which can be specified by the name of a function or by a line number as in

  break main

  to set a breakpoint at the start of your main function and

  break 54

  to set a breakpoint at line 54 in your source code

- The start your program running within gdb by

  run contacts.txt sorted_contacts_txt

- It will stop at your first breakpoint. Then use

  continue --- to run to the next breakpoint or the program termination if no more breakpoints

  n --- to execute one statement stepping over function calls

  s --- to execute one statement but stepping into function calls

  backtrace --- show a stack trace (the sequence of frames on the stack at this point in the execution)

  list --- to show a section of the source code

  print variable_name --- to print the value of the variable named variable_name

Valgrind, valgrind.org, is a tool that is useful for, among other things, detecting memory leaks due to space that is dynamically allocated, but not deleted. Recall that in C programs space is dynamically allocated using the **malloc()** and **calloc()** library calls and the space is allocated on the heap.

Style and Documentation

Your program must follow good programming style as reflected in the programs in the McDowell textbook. The programming style should also be consistent with the Sun's Java Coding Standard (see http://www.oracle.com/technetwork/java/codeconv-138413.html) to the extent possible given that your program is in C instead of Java.

Your program must be fully commented which means

- On agora create a project1 directory that has in it a README text file and the files for your project source code. The README file lists the author names, date, that this is Project 1, explains how to run your program (including showing the exact command to be entered), a description of the purpose of the program, and any aspects of the program that do not work. You will lose fewer points for parts that do not work that you tell me about.
- every file must have a start-of-file comment at the top with your name, date, and the name of the project and a brief description of what the code in that file does
- every function must have a comment above the header which is like a javadoc method header comment (including describing every parameter and return value).
- Comment other lines of code when you think it will help the readability of your program

Administrative

1. The project is due at 5 pm on Monday, the 24th of February.
2. The score is 0 if the program does not compile.
3. You need to compile with the command line flag -Wall as in

   gcc -Wall struct_sort.c -o struct_sort

   This flag shows all warnings. There should not be any warnings.
4. The project can be turned in late but with late points. Every twenty-four hour period late results in a deduction of three points not including weekends and holidays. Up to a maximum of thirty late points with a final deadline of 5 pm on the Friday of the second to last week of classes. So a project submitted at 5:01 pm on the 24th of February will have three points deducted.
5. You may work in teams of one or two students from the class. You may talk with other students in the class about the concepts involved in doing the project, but anything involving actual code needs to just involve your team. In other words, you can not show your team's code to students outside of your team and you can not look at the code of another team.
6. You need to turn in your solution as a gzipped tar file that includes your README file, your struct_sort.c file, and your struc_sort.h file.
7. Submit your gzipped tar file via handin on agora. If your tar file is called **project1.tar.gz**, then the command will be:

   handin.352.1 1 project1.tar.gz

   If you are in the directory containing the project1 directory as a subdirectory, then the command

   tar -cvzf project1.tar.gz project1

   will create the project1.tar.gz file in the current directory.