CST8234 – C Programming F17 (Assign. 2)

Programming Exercise

In this assignment, we will demonstrate what we have learned with respect to

- File I/O
- Revisiting memory allocation
- Revisiting command line arguments

Statement of the problem

We'll be building a simulation of a student enrolment program, but working from data contained in files.

Your job is to read registration records (that contain a course code, a student ID and a timestamp) from multiple files, group them by course, and sort them by registration time, and then print them into a number of files (one file per course). Each file contains the registration records for a particular day.

Your program must support both reading in the input files in one execution, or over multiple executions (in append mode), and both approaches must generate the same results.

I will provide you with a number of test input files (some good, and some with errors) and a number of output files to which you can compare your results (generated using the good test input files). I.e., you should be able to run the program on the good test input files, and then do a command line "diff" to ensure you get **exactly** the same results as I provide with the sample output files.

It is important that your results obtained using the sample input files be **exactly** the same as the sample output files, because when I mark your assignment, I will use a different set of input files (that I won't share with you!) and do a "diff" to compare the results I get with your program with the results I get with my program.

Command Line Arguments

The basic usage of your program will be

```
assignment2 [--append] [--help] FILES
```

Alternatively, you may use POSIX arguments if you'd prefer (e.g., [-a] [-h]). But if you do, you must still support "--help" in addition to "-h".

Using "--help" (or "-h") will print out a suitable usage message, as per Lab 6.

Any error encountered while parsing the arguments will print out a helpful error message, on stderr, followed by the usage message.

If the "--append" flag is not specified, it defaults to false.

The one or more FILES that are specified on the command line will be opened, and processed. If a file cannot be read, a helpful error message will be printed to stderr, and all further processing will be skipped.

Arguments can occur in any order… e.g., this is a valid invocation of your program: "./assignment2.exe input-1.in --append input-2.in"

Input File Format

To test that your program works I have provided you with seven input files that you can use to test your results: reg-20171203.in, reg-20171204.in, reg-20171205.in, reg-20171206.in and reg-bad.in.

The first four files contain valid information, with each row of the file containing a registration record that is

- A course code
- A student ID
- A timestamp that the student registered for the course

There are also some files (reg-bad<n>.in) that contain malformatted rows that should cause an error message to be printed out to stderr, and all further input and output to be skipped.

Processing Input

You will read each row of each specified input files, and scan it (fscanf) into its constituent fields.

You'll use the course code to look up the appropriate course record (i.e., struct) from a global list of courses, and if a match cannot be found, you'll create a new course record and append it to the list of courses (using realloc).

You'll create a registration record (i.e., a struct) that contains the student ID and the timestamp, and append it (using realloc) to the course's list of registration records.

You cannot make any assumptions about how many input files there will be, or how many records will in a file, or how many courses will be referenced by those records. I.e., in the good sample input files, there are only two courses, and a handful of students. But I will test your results against more files containing more students and more courses!

You may assume that the courses codes all follow a format like "CST8234", e.g., 7 characters. You may also assume that student ID's are all like "myna1234", e.g., 8 characters. I.e., all the course codes and student ID's you will encounter will be able to fit into the space available within the structs listed in types.h. I.e., you don't need to do any validation to check that they are valid or too long.

You'll probably find the following fscanf formatting string very useful for parsing the input file records

```
"%[^,], %[^,], %ld%c"
```

The special format specifier of "%[^,]" means "look for characters that are NOT a ','".

The "%ld" says look for a 'long' integer (timestamps are too big to fit into a 'int').

And the '%c' is what we use to check for the error condition of extra characters after our numeric data (as per the "Input Validation" lecture).

So when combined, the format string above means "look for characters that are NOT a ',', followed by a comma, then look for characters that are NOT a ',' followed by a comma, followed by a long integer, followed by a character. If the result of an fscanf using this format string is 3 (i.e., all matched except the extra character at the end), then we know we got a valid record.

Output File Format

You will generate one output file per course, and name each one "<courseCode>.out". E.g., if the input files contain a reference to a course "CST8234" and "CST8299", then you'd generate output files named "CST8234.out" and "CST8299.out". If there is a problem creating a file, you'll print out a helpful error message to stderr.

The file format will consist of one line per registration record, which will look like

```
<timestamp>: <studentID>
```

The spacing and format must match the sample files exactly.

The rows must be printed out in **ascending** timestamp value. I.e., sort your registration records by timestamp before writing them out to the file(s).

Append Mode

If the "--append" (or "-a") flag was specified, then if the named output file already exists, you will append to it, otherwise you will create the file. If the append option is not specified, then the results of your execution will replace the contents.

More specifically, you can either run

```
./assignment2.exe reg-20171204.in reg-20171203.in reg-20171205.in reg-20171206.in
```

Or you can run it multiple times where the 2nd and subsequent executions use the "--append" flag

```
./assignment2.exe reg-20171203.in
./assignment2.exe --append reg-20171205.in reg-20171204.in
./assignment2.exe --append reg-20171207.in
```

And both must generate the **exact** same output files.

Note, if the program is to be run multiple times with the --append option, it is assumed that all subsequent executions use files from later dates. E.g., you'll process the 20171203 (Dec 3rd) file before trying to append the 20171204/20171205/20171206 (Dec 4th-6th) registrations, and you'll process the Dec 4th files before trying to process the 20171205/20171206 files, and the Dec 5th files before trying to append the 20171206 (Dec 6th) file.

However, in any single execution, if there are multiple input files, you may not assume that they have to be on the command line in any particular order (i.e., you need to read all of the specified files, before trying to sort the result and write/append them to the output files).

Other Constraints

I have provided you with a types.h, a globals.h and a globals.c. **You must use these files, and you may not modify them in any way.**

You may not use any global variables, other than the one I provided in globals.c.

You may not modify the typedefs for 'Course' and 'Registration'. This means that if you want to get the number of courses, or the number of registrations for a course, you'll have to calculate the length (hint, use a terminator record, and write a getNumCourses() function… just like strings are terminated with a NULL, and rely on a strlen() function).

Code Division

My solution had the following source files (in addition to the mandatory globals.c)

- main.c – initialize variables, call the command line parsing arguments, generate the output
- parse.c – parsing the arguments and processing the input
- course.c – all the functions associated with manipulating/printing/querying courses
- registration.c – all the functions associated with manipulating/printing registrations

You are not obligated to use the same files, but if you toss everything into a single source file, I will deduct marks for a lack of clear organization. Think about which functionality you'd add to which classes in a Java program… the same organization ought to apply here.

Requirements

1. Create a folder called algonquinUserID1_algonquinUserID2_A2 (e.g., "me0000123_you45678_A1") that represents the two members of your team. Do all of your work in this folder, and when complete, submit the zipped folder as per the "Lab Instructions" posted on Blackboard.
2. Write a program that will implement ALL of the requirements, explicit and implicit, listed in the "Statement of the problem" above.
3. You must distribute your functions in a meaningful manner across **multiple .**c files.
    a. The .c files should contain functions that represent sensible groupings of functionality
    b. You must define .h files as appropriate
4. Each function must have a header comments that explain what it does, and describe/explain its inputs (if any) and return value (if any)
5. You must use a makefile called "makefile"), with the **`CC_FLAGS`** set to "**`-g -ansi -pedantic -Wall -w`**"), and **`OUT_EXE`** set to "**`assignment2`**"

Marking

This assignment is out of 40

- 15 for coding correctness (i.e., correct results). This will be marked very harshly. You have been provided with sample output files, so you know **exactly** what output your program must produce.
- 10 for appropriate and efficient logic (i.e., no spaghetti code!)
- 5 for sensible distribution of well-contained functions between files
- 10 for clear comments and coding convention (not necessarily K&R, but it should be clean and consistent)

You can also lose marks for incorrect submission (e.g., including unnecessary files in the zipped folder), compiler warnings, typographic errors (including in comments).

You will lose marks for overly long functions. Each function should do a specific task. Other functions should call those specific tasks.

Submission

When you are done, ONE person on the team will submit your program to Blackboard, as per the instructions for each Lab.

You ***must*** include a makefile, as part of your submission! When grading your reports I will unpack your zip file and expect to find a **folder** that I can copy to my hard drive, I then expect to be able to 'cd' to that folder, and type '`make`' (no arguments!)… and if I don't end up with a 'assignment2.exe' to run, ***I will not grade your assignment***.

Take the time to read the submission instructions AGAIN, and make sure that you are following the submission guidelines.

Double check with your partner whether you are, in fact, submitting things correctly.

Assignments must be handed in before the due date, or they will receive zero. The only exceptions will be for true emergencies. I strongly recommend using a back solution (USB drive, drop box, git, etc.) to protect yourself against hard-disk failures and/or corrupted VM's, as no extensions will be given for such easily-preventable problems.