School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Winter 2021

Final Project

Due Date April 29, 14:00 EST

This is an individual assessment. You need to solve these questions on your own. Instructors and TAs will only provide clarifications on the exam questions and it will be through Piazza. There will be no help provided to debug your project code. This is your final work, and it also requires you to demonstrate that you have learned the necessary debugging skills.

Obtaining a minimum of 50% in this project is mandatory to pass the course (in addition to obtaining an overall passing grade across all the course assessments). Students who do not achieve that threshold will receive an F grade. NO LATE submissions are accepted. Please account for Internet delays, outages, etc., and turn in your work in due time. These are not valid excuses for extensions.

This project description is for your personal use only. Sharing it with others, including websites, helping or seeking help from others for your project work will be deemed as copyright violation and plagiarism. Such actions will result in an automatic forfeit of your course grade to F in addition to any University disciplinary proceedings. You may leverage the course materials, solutions, etc., given to you, as well as parts of your own assignment solutions for your project implementation.

If you use concepts and snippets of code from other sources such as websites, they will have to be documented in your code as a proper comment. Lack of such citation will also be treated as plagiarism. In any case, utilizing such external resources must be limited to bare syntactical references (e.g - how to include a backslash - \ symbol in `sed` pattern) and should not result in you copying the major part of your program logic from such resources (e.g. - how do I traverse a linked list). **Both software and manual mechanisms will be employed for plagiarism detection.**

You MUST use `mimi.cs.mcgill.ca` to create the solution to this project. You must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can access `mimi.cs.mcgill.ca` from your personal computer using **ssh** or **putty** and also transfer files to your computer using **filezilla** or **scp** as seen in class and in Lab A and mini assignment 1.

Instructors/TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not compile/execute at all. (Commands that execute, but provide incorrect behavior/output will be given partial marks.) All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade.

Please read through the entire project description before you start working on it. You can loose several points for not following the instructions. Points can be deducted for violating the constraints set forth in the questions, including for the steps where an explicit deduction is not specified. It will also help you get the bigger picture of the project. There are also some helpful hints given at the end of this document that can be useful to you. Go through the checklist at the end to ensure that you have turned in all the required files. Files that are not turned-in will not be given a second chance.

The total points allocated to this project is 25.

Project Synopsis

For your final project, you will implement a software solution that consists of shell scripts and C program to compute the lab attendance associated with each student.

Exercise 1 - Version Control (2 Points)

Create a local git repository (private) in your home directory. You will be using this for the rest of your final project work to write any shell scripts, C programs, etc. **DO NOT use public repositories such as github**.

We expect to see at the least 4 commits, each of which are at the least 10 minutes (or more) apart. So commit your work frequently as you reach some logical milestone of your project work. **No points will be allocated for this exercise unless this requirement is completely met.** You must turn in your `git log` command output copied/redirected as `git.txt` file, once you have finished your complete project work.

Exercise 2 - A Shell Script to Pre-process the Information in the CSV Files. (6 Points)

At the end of a lab session, TAs will download a CSV file that contains the attendance information recorded by zoom into their respective folders. Let us start by taking a peak at the directory. The output below is truncated for brevity.

```
$ tree LabAttendance
LabAttendance
|-- lab1
|   |-- lab-A.csv
|   |-- lab-B.csv
........
|   |-- lab-I.csv
...
|-- lab3
|   |-- Lab-A.csv
...
|-- lab4
|   |-- LAB-A.csv
......
|   |-- LAB-I.csv
|
|-- lab5
......
|-- lab6
...
|-- lab7
    |-- lab-a.csv
    ...
    |-- lab-i.csv
```

As we can see, each of the seven lab groups (1 through 7) has an attendance file for each of the nine labs (A through I). While the CSV filename conventions are consistent, their case (uppercase/lowercase) can be in different combinations depending on how the TA named their files.

Next, let us take a peek into some of these CSV files. (These are synthetic data, not actual student names).

```
$ head -3 lab1/lab-A.csv
Name (Original Name),User Email,Total Duration (Minutes),Guest
Sharda Freedman,sharda.freedman@mail.mcgill.ca,64,No
Sterling Boone,sterling.boone@mail.mcgill.ca,64,No
```

```
$ head -3 lab5/Lab-A.csv
Name (Original Name),User Email,Join Time,Leave Time,Duration (Minutes),Guest
Mack Boyd,mack.boyd@mail.mcgill.ca,01/21/2021 08:58:30 PM,01/21/2021 10:17:06 PM,79,No
Chung Tibbs,chung.tibbs@mail.mcgill.ca,01/21/2021 08:58:31 PM,01/21/2021 10:03:25 PM,65,No
```

The CSV files have can have one of two possible formats (each file follows one or the other and not a mix). The second format contains the join time and leave time as extra attributes compared to the first format.

Your first task is to collect the information from all these CSV files and build a consolidated CSV file that contains the records in a common format.

1. Create a shell script `fixformat.sh` to do this task. You can use any shell/Unix commands already available in `mimi` to implement this script (other than sorting).

2. The shell script expects two arguments, the first being the directory under which it should recursively search for the CSV files that follows the naming convention mentioned above. You should find such CSV files no matter under which subdirectory they are stored in (i.e., subdirectory names and depths should not influence the outcome). In other words, even if we add a new lab group, etc., it should not impact your script. Your shell script's objective must be to look for the CSV files that follow the naming convention anywhere under that directory hierarchy. The second argument is the CSV file into which the script will store the re-formatted (and consolidated) output that it processed from all the input CSV files. Either of these arguments could be absolute or relative paths.

3. **(1 Point)** If the shell script is not invoked with sufficient arguments, display a usage message and terminate with code 1.

```
$ ./fixformat.sh
Usage fixformat.sh <dirname> <opfile>
$ echo $?
1
```

4. **(1 Point)** If the first argument is not a name of an existing directory, the script should throw an error message and terminate with code 1.

```
$ ./fixformat.sh /data/LabAttendance /data/labdata.csv
Error /data/LabAttendance is not a valid directory
```

5. Do not create an output file in the error/usage situations described above. **(-1 Points)**.

6. **(4 Points)** When invoked correctly, the script will find all the CSV files under the given directory hierarchy that follows the naming convention (mentioned previously), reformat the information and consolidate them into a single output CSV file, after which it will terminate with code 0.

```
$ ./fixformat.sh /data/LabAttendance /data/labdata.csv
$ echo $?
0
```

   A sample of the output is given below.

```
$ head -3 /data/labdata.csv
User Email,Name (Original Name),Lab,Total Duration (Minutes)
usha.rush@mail.mcgill.ca,Usha Rush,D,64
bessie.thompson@mail.mcgill.ca,Bessie Thompson,D,58
```

   There is a header at the top of the file, followed by the actual information records. The shell script has done the following transformations.

   (a) For each record, it included only the email, name and duration information from the original CSV files.

   (b) To each record, it added a new column, `Lab`, which it derived from the name of each CSV file and added to the records that it read from that CSV file (the values of this column represents the labs A through I). The values for this column should be uppercase alphabets irrespective of the case of the CSV filenames.

   The output produced by your shell script should follow the same exact formatting (order of columns, etc.). There is no particular order in which the actual attendance information records are to be stored in the CSV file. **Deliberately sorting attendance records when producing the output is not allowed and can result in a 0 for this exercise**. That will be the task of the C program in the next exercise.

7. Existing output files must be overwritten by your script (not appended).

8. You need not handle any other error/failure situations other than those mentioned above.

9. You do not have to handle conditions where a directory may not have any relevant CSV files or the CSV files not having headers/records (i.e, any file that matches the naming convention will be valid CSV files).

Exercise 3 - A C Program to Compute the Attendance. (10 Points)

Now that we have done some preliminary cleaning of the attendance data, it is time to compute the attendance associated with each student. At a high level, first we need to figure out for each student how much time (duration) they attended a lab. **Keep in mind that even for a given lab, (say lab D) a student may have multiple records**. This is because the student might have got disconnected and had to connect back (which results in multiple zoom records for the same meeting). Another reason could be that the student could only attend part of the lab for their group (say lab 2) and had to catchup on another TA's lab (say lab 5). In all of these cases, our objective is to find the net amount of time (duration) that the student spent on a given lab (say lab D).

1. Use `vim` to create C program files `labapp.c`, `zoomrecs.c`, `zoomrecs.h` and a `makefile`. **You are not allowed to create any other C or H files for this exercise**.

2. You can use any functions and features available in the C libraries `sdtio.h`, `string.h` and `stdlib.h`, but your implementation should be solely on C language (i.e., you should not, for example, use the `system` function call to execute part of your work using another Unix command.)

3. The executable of the program should be named `labapp`. It will receive two arguments, an input CSV file and an output filename for the program to write its output to. Either of these arguments can be an absolute path or relative path. Below is an example of its execution syntax

```
$ ./labapp /data/labdata.csv /data/attendance.csv
```

4. `zoomrec.h` must contain the following C struct definition. **No changes are allowed to this structure**.

```
struct ZoomRecord
{
  char email[60]; // email of the student
  char name[60]; // name of the student
  int durations[9]; // duration for each lab.
  struct ZoomRecord *next;
};
```

   This will serve as the "Node" of your linked list. As can be seen, the structure is used to record a student's email(**unique to a student**), name and the durations associated with each of the nine labs.

5. `zoomrecs.c` will contain at the least two functions (you can add more as needed for your logic), `addZoomRecord` and `generateAttendance`. These two functions are intended to be called from the code that is in `labapp.c`.

6. `addZoomRecord` is called once for EACH time a line (record) of lab attendance is read from the input file. The function will search for the student's information in a linked list (using `email` as the search attribute). If found, it will update/increment the duration associated with that lab for that student. If the student is not found, it will create a new `ZoomRecord` for the student, and add it to the linked list such that the list is maintained in an order of email ids (and update the relevant lab's duration).

7. `generateAttendance` is called ONCE, after all of the input information has been read into the linked list. It will then read through the linked list (which is now in the order of student email ids) and write to the output file the detailed attendance information associated with each student (thus the output is now sorted in the order of email ids, alphabetically and has one record per student).

   Below is an example format of the output (only parts of it are shown for brevity).

```
$ cat /data/attendance.csv
User Email,Name (Original Name),A,B,C,D,E,F,G,H,I,Attendance (Percentage)
adeline.larsen@mail.mcgill.ca,Adeline Larsen,62,57,0,0,45,51,58,60,60,77.78
...
melody.cohen@mail.mcgill.ca,Melody Cohen,65,58,57,64,43,50,53,60,59,88.89
...
```

The output contains a header followed by actual attendance information (ordered by the student email ids). Those records contain the email and name of a student, followed by the duration (nine columns) associated with each of the labs A through I. **If a student does not have a zoom record for a particular lab (in the above example, Adeline did not attend labs C and D), then it should be recorded as 0 duration**. The last column is the calculated attendance. For this purpose, you **count the number of labs where the student spent 45 minutes or more in the specific lab (in total) and calculate the percentage**. In the above example, Melody spent only 43 minutes in lab E, and therefore, her attendance would be 8 out of 9, calculated as 88.89 percentage.

Your program should follow the above output formatting style and keep decimal points of attendance percentage to two places (i.e. its values would be from 0.00 through 100.00). Existing output files must be overwritten.

8. `labapp.c` will contain the `main` of your program. It will call `addZoomRecord` and `generateAttendance` as needed (either directly or through other functions in `labapp.c`). Keep in mind that this may require you to put some additional information into `zoomrecs.h` to make it work. You have to figure that out based on modular programming concepts discussed in class.

9. The more finer details of rest of the program logic is up to you - including the arguments to be passed to these functions, adding other (helper) functions if required, etc. But the high-level design described above should be maintained and the attendance information must be maintained and processed using the linked list. Duplicating the attendance data from linked list to a new array, etc., is not acceptable. You have to demonstrate your ability to work with linked lists. Not following this would result in **(-3 points deduction)**.

10. You are not required to perform any explicit error checks in the C program or implement other capabilities not explicitly stated. However, it is highly recommended to implement some of the commonly used checks to help reduce your development and debugging effort.

11. Ensure that your program frees up any dynamic memory allocated to it once it is done with its use and before terminating the program. **(-2 points)**

12. **(7 Points)** For producing the correct output.

13. **(-3 Points)** If the program crashes at any point (only valid inputs will be used for testing).

14. **(1.5 Points)** Make sure that your source code is well commented and follows the basic modular programming principles covered in class.

15. **(1.5 Points)** Write a proper `makefile` for your C program. Make sure that it compiles any parts as needed and only when it is actually impacted in the event of source code changes, following the principles discussed in class. TAs will compile your program by executing the command `make` in the directory that has your source code and `makefile`. **If it does not build your executable program, no points will be given for this exercise**. Points are also deducted for any uncessary compilation steps (defeats the purpose of a `makefile`).

Exercise 4 - GDB. (4 Points)

This is a theoretical exercise. The response to this exercise must be turned in as `gdb.txt` as a combination of textual explanation (of what you are doing and why) and `gdb` commands. Your responses must be based on your C program.

1. Now imagine that your C program crashed (abort) while executing. You suspect that it is crashing inside the function `generateAttendance`. Describe what `gdb` commands (from compiling to executing the program) will you use, so that `gdb` will automatically stop execution at the beginning of the said function (using the most minimal steps to get to there).
2. At this point, since you are not sure which part of the code is causing the issue, describe the sequence of commands that you will be executing. Continue the above investigation till you reach the part of your code where you are accessing the first node of the linked list (as passed or accessed by the `generateAttendance` or its helper functions) for the first time. Print the address pointed by that particular variable.

You may end the `gdb` session at this time. (We will imagine the issue was that somehow that pointer was NULL which your code was not handling correctly. But you do not have to modify your working code to make it crash.)

Exercise 5 - A Shell Script to Convert CSV to HTML (3 Points)

Now that we have the attendance information in the form of a CSV file, we can generate other interesting visual formats of this information. In this exercise, you will convert the CSV into an HTML format using Unix commands.

1. Write a shell script `csv2html.sh` that accepts two filenames as arguments.

   ```
   $ csv2html.sh /data/attendance.csv /data/attendance.html
   ```

   The first argument is a CSV file in the format produced at the C program's output that contains attendance information. The second argument is the filename for the output html version. The script should work fine with both absolute and relative paths.

2. **You MUST use only `sed` (additionally may use `echo`, if you need to) command to generate the output html contents**. Violation of this will result in 0 points for this exercise.

3. The conversion can be easily done, once you understand the simple structure of HTML. Below is an example format (truncated for brevity).

   ```
   $ cat attendance.html
   <TABLE>
   <TR><TD>User Email</TD>...<TD>A</TD>...<TD>I</TD> <TD>Attendance (Percentage)</TD></TR>
   ...
   <TR><TD>adeline.larsen@mail.mcgill.ca</TD>...<TD>62</TD>...<TD>77.78</TD></TR>
   ...
   <TR><TD>melody.cohen@mail.mcgill.ca</TD>...<TD>65</TD>...<TD>88.89</TD></TR>
   ...
   </TABLE>
   ```

   Basically the entire contents is enclosed between the HTML tags `<TABLE>` and `</TABLE>`. Further, each row (line/record from CSV) is enclosed in `<TR>` and `</TR>` tags. Finally, each value themselves is enclosed between `<TD>` and `</TD>` tags.

   If you copy the `attendance.html` file to your computer and open it in your computer's browser, it will look something like this (truncated).

User Email	Name (Original Name)	A	B	C	D	E	F	G	H	I	Attendance (Percentage)
aaron.ross@mail.mcgill.ca	Aaron Ross	0	55	51	46	48	53	62	53	56	88.89
adelaida.fogle@mail.mcgill.ca	Adelaida Fogle	61	64	62	64	61	60	60	56	58	100.00
adeline.larsen@mail.mcgill.ca	Adeline Larsen	62	57	0	0	45	51	58	60	60	77.78
adolfo.demarco@mail.mcgill.ca	Adolfo Demarco	46	54	59	58	56	55	48	37	0	77.78
agnes.becker@mail.mcgill.ca	Agnes Becker	48	51	58	0	40	56	0	45	0	55.56
agripina.delong@mail.mcgill.ca	Agripina Delong	64	60	60	61	61	60	60	55	59	100.00
ahmad.burgess@mail.mcgill.ca	Ahmad Burgess	60	65	0	0	61	48	66	0	0	55.56
akilah.muller@mail.mcgill.ca	Akilah Muller	60	64	64	66	62	63	63	63	62	100.00
albert.weaver@mail.mcgill.ca	Albert Weaver	70	65	0	72	44	0	74	67	63	66.67
aleen.ladd@mail.mcgill.ca	Aleen Ladd	64	63	65	0	24	33	65	67	64	66.67
alena.chase@mail.mcgill.ca	Alena Chase	65	5	57	3	44	49	48	57	0	55.56
alesia.link@mail.mcgill.ca	Alesia Link	60	55	57	58	58	54	59	53	58	100.00
alexandra.goodwin@mail.mcgill.ca	Alexandra Goodwin	61	56	58	58	58	54	54	52	0	88.89

There might be some minor visual differences based on your browser types and versions, but that is ok.

4. If you want to experiment with HTML table tags, here is a resouce that can be helpful.
5. There is no requirement for the shell script to perform any error checks, but it is once again recommended to include some to help you during development and debugging.

CHECKLIST: WHAT TO HAND IN

- `git.txt`
- `fixformat.sh`
- `labapp.c zoomrecs.h zoomrecs.c makefile`
- `gdb.txt`
- `csv2html.sh`

You may create an archive of the file to upload, if needed. If all of your code is in the directory `FinalProj`, you can archive them in the following manner.

```
$ tar -zcvf FinalProj.tar.gz FinalProj
```

Please download your submitted files to double check if they are good. Submissions that are corrupted cannot be unfortunately graded. Forgetting to submit required files will not get another chance.

You do not have to turn in the executables and object files associated with your C program. Any such files will be ignored and TAs will compile your C program on their own as indicated in the problem descriptions above.

ASSUMPTIONS

- For the C program, you can assume that each line in the CSV can be easily read into a char array of size 200.
- You can assume that the given C struct is sufficent to store any values that you may see in the C program's input.
- You can assume that the names of the fields in the headers and their order remains the same and that no empty files will be used. All data will be ASCII (what we have seen so far in class) no Unicode, etc.
- Alphabets in email ids follow lowercase letters and emails can additionally contain only period(`.`) and at sign (`@`).
- You do not have to look for "bad data" in the attendance records.
- You need not perform any explict error checks for the exercises other than those explicitly stated. However, your program must not crash/fail/error for valid inputs.

HINTS

You will find that the shell scripting techniques learned for `mini3` and the C programming logic that you implemented for `mini5` can be used to make your project development effort easier.

RESTRICTIONS

1. None of your programs, scripts should not create any other intermediate (temporary use) files. They should only create the final output file. Violating this can result in additional deductions for respective exercises.
2. **All of your scripts, programs should execute under a minute (in-total) for an estimate of 100 zoom records**. (For some perspective, a naive implementation will execute under a couple of seconds).
3. If your program crashes, goes into infinite loop, etc., TAs may not execute remaining test cases.
4. Other individual restrictions are stated with each exercise. There is no further constraints.

TESTING

- We recommend you start testing by creating only a couple of directories (say lab1, lab2) and a couple of files (say Lab-A, lab-C) and with just 1-2 students to begin with. Keep in mind that your objective should be about variety of data in testing and not necessarily its volume. For example if you test with 100 records that are already in sorted order, you may not realize that your C program's sort logic is not working. If you plan smart, you can test effectively with just a few records that you create.
- Once you get a hang of it, you can try out more number of students. Remember! debugging your logic for corectness using a large amount of data will be very difficult.
- TAs will test using their own scripts, but it will follow the assumptions that have been setforth in the project description.

QUESTIONS?

Please use piazza. However, before posting your question, use the search functionality to check if it has been already discussed. You should also look under "Final Project general clarifications" pinned post to check if a popular question has been already included there. They will not get individual responses again.

This is your final project and you are expected to be able to debug your program's logical errors and other issues using the tools and techniques taught in class. TAs will not help you do your final project.

If your program crashes, use `gdb` to debug your program.

Responses are limited to clarifications on the project descriptions and expectations.