AU 2023 CSE 2421 LAB 4

Assigned: Monday, September 25th
Early Due Date: Thursday, October 5th, by noon
Due: Friday, October 6th, by 11:30 p.m.

IMPORTANT: READ THESE INSTRUCTIONS AND FOLLOW THEM CAREFULLY. ALSO, REFER TO THE CLASS SLIDES ON STRUCTURES, LINKED LISTS, AND FUNCTION POINTERS. IF YOU DO, YOU CAN FINISH THIS LAB IN MUCH LESS TIME. IF YOU DO NOT, YOU WILL SPEND MUCH MORE TIME ON THE LAB AND YOU MAY STILL NOT FINISH ON TIME! YOU HAVE BEEN GIVEN OVER 10 DAYS TO COMPLETE THIS LAB; IT WILL LIKELY TAKE SOME OF YOU EVERY BIT OF THAT TIME TO COMPLETE. DON'T PROCRASTINATE!

The greater the difficulty, the more the glory in surmounting it. – Epicurus

Epicurus was an ancient Greek thinker who taught a popular philosophy based on simple pleasures. He believed our purpose on Earth was to fill our days with happiness, avoiding fear and pain. But even Epicurus knew that life wasn't without its difficulties. In this quote, he shares his belief that we can find joy in overcoming obstacles — and the more complex the task, the greater the satisfaction that awaits on the other side.

The Confucius quote from last lab still applies.

Objectives:

- Structures and Structure Pointers
- Strings and C library string functions
- Linked Lists
- Function Pointers

- Passing Command Line Arguments into main()
- Header files and Make files
- Reading/Writing from/to files

REMINDERS and GRADING CRITERIA:

- This is an individual lab.
- Not starting to work on this lab immediately would be an unbelievably bad idea.
- Make a point to check any Piazza post with "GUIDANCE:LAB 4" in the header. It will help you.
- **Effort has been made to ensure that this lab description is complete and consistent. That does not mean that you will not find errors or inconsistency. If you do, please ask for clarification.**
- **Every lab requires a Readme file** (for this lab, it must be called **lab4Readme – use this name.** This file must include the following:

  · Required Header:

  BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I STRICTLY ADHERED TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY.
  THIS IS THE README FILE FOR LAB 4.

· Your name
· Total amount of time (effort) it took for you to complete the lab
· Short description of any concerns, interesting problems or discoveries encountered, or comments in general about the contents of the lab
· Describe, with 4-5 sentences, how you used gdb to find a bug in your program while debugging it. Or, if you had no bugs in your program, how you used gdb to verify that your program was working correctly. Include how you set breakpoints, variables you printed out, what values they had, what you found that enabled you to fix the bug. . If you didn't have to use gdb to find any bugs, then use gdb to print out the value of each address you received from a malloc()/calloc() call, then use a breakpoint for the free() system call to verify that each addressed is passed to free at the end of the program. You may use ddd instead of gdb, if you wish.

- You should aim to always hand an assignment in on time or early. If you are late, you will receive 75% of your earned points for the designated grade as long as the assignment is submitted by 11:30 pm the following day, based on the due date given above. If you are more than 24 hours late, you will receive a zero for the assignment and your assignment will not be graded at all.

- Any lab submitted that does not compile – without errors or warnings - and run **WILL RECEIVE AN AUTOMATIC GRADE OF ZERO!** No exceptions will be made for this rule – to achieve even a single point on a lab, your code must minimally build (compile to an executable without errors or warnings) on stdlinux and execute on stdlinux without crashing, the gcc command must use the **–ansi** and **-pedantic** options.

Since a Makefile is required for this lab, you must create the appropriate compile statements to create a .o file for every .c file you create, which would then create a lab4 executable. Graders will be downloading your lab4.zip file from Carmen, unzipping it, and then executing **make** from a linux command line prompt. Your program must compile – without errors or warnings – via commands within the Makefile. The gcc options your program must be compiled with are: **-ansi -pedantic -Wimplicit-function-declaration -Wreturn-type -c**; you may also want to include the **-g** options so that debug information is generated. **Given valid input, your program must also run without having a seg fault, hanging or other abnormal termination.**

LAB DESCRIPTION

PART 1. CLASS GRADE COMPUTING SYSTEM:

Mandatory file names: 1) lab4main.c – will hold main() and **no other functions.**
2) lab4.h – will hold any local typedefs or struct definitions, function prototypes, etc. NO VARIABLES **that allocate space** or any executable code can be declared here.
3) you must have an additional file name for **every function that you write**. (e.g. insert_node.c, delete_node.c, option1.c, option2.c, etc.)
4) you must create (at a minimum) a function to do each of the following tasks:
insert a node in the linked list
delete a node from the linked list
malloc() a node and populate it with student data
free all memory associated with the linked list
a function for each option listed below
a function to write all data to disk

You will write a program to access, process and store data in order to calculate grades for a class. The program will execute in the following format:

lab4 filename1 filename2

where lab4 is the name of the executable
filename1 is the data file that contains the current class records to read in from disk
filename2 is a data file that your program will create with the updated class records

This means that you will have to read two parameters from the command line. The two parameters can be the same filename, so you will have to close filename1 before opening filename2 to insure that you do not end up with corrupted data.

You will be supplied with a "test" class grade file called **class_records**. All of the data for the students in the class grade computing system will be in this file. Once the program has read in the student data, a user will be able to select options related to what to do with the data. You do not know the number of different students which will be in the file, so your code must be able to deal with an indeterminate number of different students (up to the limits of memory). If any grade slot does not have a current valid score, its value will be -1.

First, your program should open the file specified by the filename1 parameter and read in the initial class records. There will be options for adding or deleting students provided to the user later by the program after the initial data is read in and used to build the linked list. The data will be entered in the following format, with each item of data listed entered on a different line:

- A single line with 4 alphanumeric strings separated by spaces specifying the names of the 4 Grade Categories. These can be different in each input file your program uses.

Then for each student:

- ✓ Student Name (Assume the name may contain white spaces; check the initial class_records file on Piazza)

- ✓ Student ID number (1 - 20000)
- ✓ 3 Scores for Grade Category1 (3 floating pointvalues separated by spaces)
- ✓ 3 Scores for Grade Category2 (3 floating pointvalues separated by spaces)
- ✓ 3 Scores for Grade Category3 (3 floating pointvalues separated by spaces)
- ✓ 3 Scores for Grade Category4 (3 floating pointvalues separated by spaces)

Data about each student will not be separated from the data for the following student; that is, the first line of input for the next student will immediately follow the student which precedes it on the following line. The end of the input for the students will be indicated when an fscanf() function returns EOF. There will be data about at least one student in the file. The linked list that your program builds for the students should store them in order of increasing student ID number; the students will not necessarily appear in the input file in this order.

While reading the input data, and constructing the linked list, for each student your program should:

1) Calculate the cumulative score for each category (omitting any score listed as -1 from the

calculation). If all individual scores are -1, then mark the category cumulative score as -1.
Examples:

a) If the scores 95 88 89, then add the three scores and divide by 3.
b) If the scores are 95 88 -1, then add the first two scores and divide by 2.
c) If a category has the scores 95 -1 -1, then the cumulative score is 95.00.

2) Calculate the Current grade based on the following weighting system: Category 1 cumulative = 15% of the grade, Category 2 cumulative = 30% of the grade, Category 3 cumulative = 20% of the grade and Category 4 cumulative is 35% of the grade. If any category cumulative score is -1, then use a score of 100 in this calculation.
3) All Final Grades should be set to -1 when reading in data.

After reading the input data, and constructing the linked list, your program should:

1) Tell the user that the student data has been read in from the file and how many student records were read in.
2) Prompt the user to select from the following options for processing of the data. **(REQUIRED: You must use an Array of Function Pointers to call the User's Choice of Function for options 1 through 7. It is my expectation that options 1 through 7 will be able to return the same type and that the parameter(s) passed can be made to be identical.)**

OPTIONS:

1. Print a single student record with all stored information along with the calculated scores. The student record is requested by student ID number; if you see a score is listed as -1, print "n/a". Format it in a readable manner. All information for a single student should fit on a single line. You may want to print out a line with column headers on it prior to printing out the data. **There is a printHeader.c file uploaded on Piazza that prints a header line with appropriate spacing. You may use it if you wish.**

2. Print a single student record with all stored information along with the calculated scores. The student record is requested by student last name; (This function will likely require using the strstr() or a similar C Library function. If you see a score is listed as -1, print "n/a". Format it in a readable manner. All information for a single student should fit on a single line. You may want to print out a line with column headers on it prior to printing out the data.
3. Print all student records with all stored information along with the calculated scores sorted by student ID number; if you see a score is listed as -1, print "n/a". Format it in a readable manner. All information for a single student should fit on a single line. You may want to print out a line with column headers on it prior to printing out the data. Include with this data a summary line that lists the average score for each category across all students and an average
4. Recalculate all of a single student's grades (assume that new scores have been entered since the last calculation) and print out the student's name, the four current cumulatives by category and their current overall grade. The student will be selected by student ID number.
5. Recalculate all student's grades (assume that new scores have been entered since the last

calculation) and print out each student's name, the four current cumulatives by category and their current overall grade.

6. Insert a new score for a single student ID #. You will need to ask which category it's for and whether it should be stored in the first, second or third score position.
7. Calculate Final Grades. Calculate the final grade and store it in the appropriate spot for each student. The final grade should be calculated based on the following weighting system: Category 1 cumulative = 15% of the grade, Category 2 cumulative = 30% of the grade, Category 3 cumulative = 20% of the grade and Category 4 cumulative is 35% of the grade. If any category cumulative score is -1, then use a score of 0 in this calculation. Then call option 5 from here to print out all student information. NOTE that this calculation is different than the calculation for Current Cumulative Grade.
8. Add a new student to the class. You will have to prompt for each data item.
9. Delete a student (prompt the user to enter a student ID number to delete): if the student is not found in the linked list, print a message indicating an error, or if the list is empty, print an error indicatingthat);
10. Exit the program. (This option would write all current records in the linked list out to disk using filename2 from the command line and would also free all dynamically allocated memory.) **The information that you save in filename2 should be able to be used as input to this program the next time it is executed.**

The user will enter a choice of one of these ten options by entering 1 - 10 immediately followed by enter (new line). You can assume that all user input will be correct, except that the user may inadvertently attempt to delete a student which has not been added to the class. You should write a separate function to do the necessary computations and print output for each of these options. Some of these functions may call some of the other functions. The goal is to make main() as small and succinct as possible, while creating functions that make it as easy as possible to monitor, modify and execute the code. You should also write a separate function to read the data for a single student from stdin into a node after allocating storage for the node.

Be sure, for each of the functions above, which is required to print output, to print a label which describes the output, along with the appropriate output on the same line.

GUIDANCE

- All function prototypes should be included in your lab4.h file.
- Declare the structs shown below in your lab4.h file:

```
struct Cat_Grade{
        float score1;
        float score2;
        float score3;
        float Cumulative;
};
struct Data {
        char student_name[40];
        int student_ID;
        struct Cat_Grade Cat1;
        struct Cat_Grade Cat2;
        struct Cat_Grade Cat3;
```

```
        struct Cat_Grade Cat4;
        float Current_Grade;
        float Final_Grade;
};

typedef struct Node {
        struct Data Student;
        struct Node *next;
} Node;
```

- You should write and debug the following functions first:

  - ✓ main() (adding items as needed)
  - ✓ create your Makefile (adding items as needed)
  - ✓ a function to read in the student data from disk which may call other functions
  - ✓ a function to print the data in nodes in the list (i.e. student list)
  - ✓ a function insert(), to add a node to the list; find and carefully follow the algorithm for inserting a Node in the classslides or from the text.
  - ✓ a function delete(), to remove a node from the list

- Do yourself a HUGE favor: DO NOT WRITE THE CODE FOR OTHER FUNCTIONS UNTIL THE FUNCTIONS ABOVE ARE WORKING!!!

- Until you are ready to write the code for the other functions, you can write "stub" functions with empty parameters and empty blocks for the other functions in the program; if these functions are called in the program, they will do nothing, so this will create no problems for testing and debugging. For example, a stub function for change_grade() might look like this:


```
void change_grade(){
        return;
}
```


You can change the return value and the number and type of parameters passed when you have a better idea of what it will require later in your development cycle.

CONSTRAINTS

- All source code files (.c files and .h files) submitted to Carmen as a part of this program **must** include the following at the top of each file:

```
/* BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE
** STRICTLY ADHERED TO THE TENURES OF THE
** OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY.
   */
```

- Your Makefile submitted to Carmen as a part of this program **must** include the following at the top:

```
# BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE
# STRICTLY ADHERED TO THE TENURES OF THE
# OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY.
```

- You **must** comment your code

- You may **NOT** use any "global" or file scope variables.
- The student data must be stored in a singly-linked list, where each node in the list contains the data for one student. **The head of the list cannot be a NODE; it must be a list head (e.g.NODE *). Points will be deducted if this is not the case.**
- **You are not permitted to declare any variables at file scope**; you should, however, declare the Node data type used to store the data for the linked list at file scope (but no variables can be declared as part of this declaration). See the description of the declaration of the Node type above.
- All floating point data will be entered with up to two digits of precision and should also be output with two digits of precision.
- You must allocate the storage for each Node structure dynamically using malloc/calloc.
- If a student record is deleted, you should call free() to free the storage for the node.
- You must write the student data to disk (filename2, note format for this output file above) and then free the space for the individual nodes that still contain data before the program terminates when the user selects option 10.
- See the text file posted on Piazza with sample input.
- **You must create a Makefile that creates your executable lab4 and creates your .zip file to submit to Carmen. Insure that the Makefile contains all appropriate dependencies.**

LAB SUBMISSION

Always be sure your linux prompt reflects the correct directory or folder where all of your files to be submitted reside. If you are not in the directory with your files, the following will not work correctly.

You must submit all your lab assignments electronically to Carmen in .zip file format.

IMPORTANT: Prior to uploading the file to Carmen, I suggest creating a separate directory and putting a copy of your lab4.zip file there. Run unzip and then run **make**. This should verify for you that you have included all needed files in your lab4.zip that are required to create your lab4 executable. Many times, I've seen students inadvertently miss a file or two in the .zip file, then, while grading, we can't create a program to test and the student gets a 0 on the lab. This lab takes too much work to have such a simple error cause such an outcome. Please do something to check that all files are included. An alternative is to modify the **verify** shell file that I created for you for lab 1.