COMP 2401 B/D (Last Updated Oct. 13th)

Assignment #3: Finding Evidence

Structs and Dynamic Memory

Goals:

For this assignment, you will write a program in C, in the Ubuntu Linux environment of the course VM, that allows the end user to manage a database of categorized evidence.

Learning Outcomes:

If you are keeping up with the lectures, completing tutorials, and read the course notes as needed, then by the end of this assignment you will have demonstrated that you can:

- Write a program that uses structure data types to represent both a collection and the data
- Practice with dynamically allocated memory and using pointers to pass-by-reference
- Write code that is separated into different header and source files
- Practice with string manipulation and formatting

Assignment Context (Optional Read):

Great work on the communication line! We can now send and receive data between our field agents and the communication's van. I bet along the way, you also found some ways to make the encryption unique to us!

Now that our devices can communicate via the walkie-talkie, we want to streamline our process for recording evidence. You remember a few weeks back you wrote a program that let the user enter EMF values and store them in a big, sorted database? We want to extend that to handle multiple devices.

Out in the field, we use electromagnetic field (EMF) readers, thermometers, and microphones to detect extremely supernatural things like electricity, cold rooms, and sound levels.

In this week's work assignment, you will be implementing a database that lets us enter data while we're in the field, clean it up a little, and then let us see our organized evidence in a nice way.

PS: I read the decrypted transcript from last time. Don't let those bullies get you down. If you're having fun, it doesn't matter what they think! It's okay to enjoy yourself, even if other people don't understand why you like it, as long as nobody is getting hurt. Go out and ghost hunt, friend!

Instructions:

This assignment is a bit more structured than last time, in order to give you some framework for getting started with this new content. As usual, make sure to follow good coding practice, including but not limited to documenting functions, using good variable names, re-using functions where possible, avoiding global variables, and having reasonably robust error handling.

You must read and understand the code that has been provided to you. In this assignment, you will be compiling programs with multiple files. We have not yet talked about the "good" way to compile and link using Makefiles, so we will be using a somewhat "bad" way of compiling. For example, if your program contains three source files *file1.c, file2.c,* and *file3.c*, then you can use the following command to create an executable called *a3*: `gcc -o a3 file1.c file2.c file3.c`

(You *may* use Makefiles if you wrote each line of the Makefile yourself *and* you provide reproducible instructions for the TA in your README)

We are also using header files; these are just like regular c files, except they end in ".h". The purpose of these is to split our implementation away from our definitions - that is, we can forward declare our functions and define structures in the header, include that header (as seen in *main.c*), and have access to all of those definitions without making our implementation messy!

For this assignment, you will write the code to support both *Evidence* data and a *Collection of Evidence* - this evidence array is referred to as the "**Notebook**". This includes:

- Initialization functions
- An evidence deep copy function that creates a brand new Evidence Struct from another
- Growth/Add/Remove/Print/Cleanup functions for the Notebook (the Evidence Array)
- A main function that allows us to add/remove/print our evidence

Provided Code:

Code has been provided. Submit code that has the following code (bolded functions are provided):

- **Main.c**: main(), **printMenu()**
- **Init.c**: initNotebook(), initEvidence(), **loadEvidenceData()**
- **Defs.h:** Forward declarations, type definitions, error codes and other definitions
- Evidence.c: addEvidence(), delEvidence(), growNotebook(), copyEvidence(), printNotebook(), cleanupNotebook()

Note: You should only need to make changes to the main() function, to create the two functions in init.c, and to create the new file Evidence.c; you *may* add functions or definitions to defs.h **only** if they promote better code readability - otherwise **no other code should be modified**.

Note: All files should include the definitions file at the top, with the line **#include "defs.h"**

Approach for Completing:
This specification will cover each section more carefully, but here is the overall plan for approaching this assignment:

1. Read and understand the layout of the files and the data, and where the files need to go

2. Create your “Evidence.c” file, include the definitions, and add in all of the blank functions as placeholders; add your function comment documentation to give you an idea of the function’s requirements and purpose

3. Read the “init.c” file (and create the blank functions that are needed), to understand how the functions will be getting called. loadEvidenceData() has been provided to populate with some example data once you have initEvidence(), initNotebook, and addEvidence() written.

4. Write your init functions for the notebook and evidence. Try printing some of the evidence attributes to see if it’s working.

5. There are three functions that get used together, and you may wish to break them down into smaller, testable problems: copyEvidence, growNotebook, and addEvidence.

6. Finish writing the notebook functions, delEvidence and printNotebook

7. Write the cleanupNotebook() function, and use valgrind to make sure there are no memory leaks! Memory leaks can lead to deductions.

8. Clean your main() function and implement the main control flow for the program.

9. Make sure your README and other packaging is set up. Test, submit, download, test.

A note on deductions: There are some places that can lead to severe deductions, similar to previous assignments, that are worth highlighting for your approach. We **must** see functions running in order to fairly grade them. This means that code that requires a TA to fix before it will compile will have big deductions. If the function cannot be seen running, it will only receive part marks for how correct it *appears* to be.

If you write the code, verify that it compiles before (and after) submitting, and verify that each function that you want full grades for is executing at some point. If a function is working, but you did not have time to implement the main control flow, make it run and make it visible to get part marks (and lose marks for the main control flow).

Not compiling/executing may lead to larger and larger deductions.

Instructions:

This is a suggested order of consideration for each requirement.

1. **Initialize the collection and evidence data**
   1.1. Implement the `void initNotebook(NotebookType *arr, int cap)` function that initializes every field of the given `arr` parameter to default values. The initial array capacity is set to the given parameter `cap`, the quantity of evidence currently stored in the new array is always zero, and the elements array must be dynamically allocated to hold the number of elements indicated by the capacity.
   1.2. Implement the `void initEvidence(int id, char* room, char *device, float value, int timestamp, EvidenceType *ev)` function that initializes every field of the given `EvidenceType` structure, using the given parameters.
2. **Implement the collection manipulation functions**
   2.1. Implement the `void copyEvidence(EvidenceType* newEv, EvidenceType* oldEv)` function that performs a ***deep copy*** of the evidence found in the `oldEv` parameter into the evidence structure in the `newEv` parameter.

   Note: A deep copy duplicates every part of the field separately, so that all the data contained in the source structure and its contained structures is duplicated into the destination structure. A *shallow copy* has some, or all, of the fields *shared* between the source and the destination structures, using multiple pointers to the same data!

   Note: You must reuse a function that you've previously implemented for this to promote good quality code reuse.

   2.2. Implement the `void growNotebook(NotebookType* arr)` function that doubles the capacity of the evidence collection, using only the `malloc()` and/or `calloc()` library functions.

   Note: DO NOT use `realloc()` or any other memory copying library functions, as these undermine the problem solving goals of the assignment.

   `growNotebook()` **must** be implemented as follows:

      2.2.1. Declare a new `NotebookType` variable to temporarily store a new collection; for example, a new variable called `newNotebook`
      2.2.2. Initialize the `newNotebook` collection with an existing function, using double the capacity of the given collection in parameter `arr`

2.2.3. Loop over the given collection in `arr`, and make a deep copy of every piece of evidence from the old elements array in arr into the new elements array in `newNotebook`

Note: It is essential that you **DO NOT** use the assignment operator (=) to "copy" any structures or arrays from one place in memory to another; this operator usually only performs a *shallow copy* when used with compound data types. You **must** use the `copyEvidence()` function that we implemented in a previous step everywhere that evidence structures are copied.

2.2.4. Deallocate `arr`'s old elements array

2.2.5. Use the assignment operator to set arr's elements pointer to `newNotebook`'s elements array, which points to a new, increased array in the heap

2.2.6. Double the capacity of the given collection in parameter `arr`

2.3. Implement the `void addEvidence(NotebookType* arr, EvidenceType* ev)` function that makes a copy of the given evidence directly into its *correct* position in the given Notebook collection. This must be a deep copy of the evidence structure, and all copy operations must reuse a function that you implemented in a previous step.

The function should be implemented as follows:

2.3.1. Check whether there is room in the array for new evidence and, if necessary, grow the array using a function that you implemented in a previous step

2.3.2. Find the insertion point of the new evidence in the array, so that the following order is maintained at all times: First, evidence is stored in *descending* (decreasing) order by room name, then by timestamp in *ascending* (increasing) order; consider the cases where a room is (currently) unique and where it is not.

2.3.3. Once you have found the insertion point for the new evidence, you must copy each element one position towards the back of the array, starting at the last evidence down to the insertion point.

Note: The insertion of new evidence requires "moving" the other evidence (by copying them) to make room for the new one. Do not add to the end of the array and sort, do not use any sort function or sorting algorithm anywhere in this program.

2.3.4. Copy the new evidence into the array at the insertion point

2.3.5. Increase the size of the array

2.4. Implement the `int delEvidence(NotebookType* arr, int id)` function that removes from the given collection the evidence with the given id. All copy operations

must reuse a function that you implemented in a previous step. This function must be implemented as follows:

2.4.1. Find the evidence with the given `id` in the given notebook

2.4.2. If the evidence is not found, return an error flag using a predefined constant; if the evidence is found, continue with the following steps

2.4.3. Copy each element of the evidence collection one position towards the front of the array, starting at the next element after the found one up to the last piece of evidence

Note: The removal of evidence requires "moving" the other evidence (by copying them) to close the gap left by the removed one. **Do not** leave empty positions in the evidence array. All evidence elements must be directly contiguous with each other.

2.4.4. Decrease the size of the array and return a success flag

2.5. Implement the `void printNotebook(NotebookType* arr)` function that prints all of the evidence in the given collection. This function must meet the following criteria:

2.5.1. Every field of every evidence must be printed out

2.5.2. Every printed field must be aligned with the other evidence

2.5.3. Any EMF device data should have at most 1 number after the decimal
An EMF reading above 4.0 should have "(HIGH)" written beside it

2.5.4. Any THERMAL device data should have at most 2 numbers after the decimal
A THERMAL reading below 0.0 should have "(COLD)" written beside it

2.5.5. Any SOUND device data should have at most 1 number after the decimal
A SOUND reading below 35.0 should have "(WHISPER)" written beside it
A SOUND reading above 70.0 should have "(SCREAM)" written beside it

2.5.6. Time is in "seconds since starting the hunt". It should be printed as hours, minutes, and seconds, separated by a colon, with exactly 2 digits each.

Eg. 8082 seconds would print as "02:14:42"
Note: Time / 3600 gives us the number of hours. (Time % 3600) / 60 gives us minutes, Time % 60 gives us seconds.

2.6. Implement the `void cleanupNotebook(NotebookType* arr)` function that deallocates the dynamically allocated memory in the given collection.

3. **Implement the main control flow**

3.1. Declare a local `NotebookType` variable to store all the evidence in the program.

3.2. Initialize the Notebook collection structure by calling a function previously implemented, using an initial capacity of 2, as found in the predefined constant provided. Do not use a

larger initial capacity, as this will prevent the grow function from being called, resulting in deduction for program execution.

3.3. Load the evidence data into memory by calling a function that was provided to you in the base code. **This should be loaded in your final submission to pre-allocate data.**

3.4. Repeatedly print out the main menu by calling the provided `printMenu()` function, and process each user selection as described below, until the user chooses to exit. Verify that the user enters a valid menu option; if they enter an invalid option, they must be prompted for a new selection.

3.5. The “add evidence” functionality must be implemented as follows:

3.5.1. Prompt the user to enter the evidence data, including id, room name, evidence type, and timestamp

3.5.1.a. The room name might have spaces in it; you can use `fgets()` and remember that you might need to clear the text buffer as shown in class and in the course notes. Also note: `fgets()` *contains the newline* character that is typed; you can remove this newline by manually setting the null terminator.

3.5.1.b. The evidence type must be entered as a numeric value: 1 represents EMF, 2 represents THERMAL, 3 represents SOUND. Your program must convert this numeric value to the corresponding evidence type string; Make sure that the entered type is valid before allowing them to proceed

3.5.1.c. The timestamp is entered as the number of hours, minutes, and seconds - separated by spaces, on a single line, and must be converted into a single seconds integer

3.5.2. Initialize a new, *statically allocated* evidence structure, using the data entered by the user

3.5.3. Add this new evidence to the evidence collection (the notebook), in its correct position, using an existing function

3.6. The “delete evidence” functionality must be implemented as follows:

3.6.1. Prompt the user for the ID of the evidence to be removed

3.6.2. Remove the evidence from the collection

3.6.3. If the removal fails, print out a detailed error message to the user

3.7. The ”print evidence” functionality must be implemented by calling an existing function

3.8. At the end of the program, the evidence collection must be cleaned up by calling an existing function

For any errors that occur, a detailed error message must be printed to the user. Your program should avoid terminating for invalid input where possible unless otherwise specified, by re-prompting the user for valid data. Existing functions must be reused everywhere possible.

Requirements:

Make sure that you follow these requirements for the whole program to get full marks.

1. Document your program as demonstrated in class. A formatting guide *may* be posted to Brightspace to assist with this, and you will be expected to follow this if it is. Minimally but not all encompassing, your documentation should include:
    a. A description of the purpose and process of the function
    b. The role for each parameter (in, out, in/out)
    c. Possible return values
2. Include a README file; a plaintext file named README or README.txt or README.md which contains the following information:
    a. A preamble: The author, student ID, description of the program, and a list of files
    b. Instructions for compilation and launching, including command line arguments
3. Submit all files in a single .tar or .zip file
4. Ensure the code compiles and executes on the course VM following the directions that you provide in the README
5. Functions will only receive full marks if they are implemented, called in the program, and not commented out (this will be assumed to be rough, discarded work)
6. All data should be printed to the screen as required to demonstrate functioning code
7. The code should follow good coding practices:
    a. Functions are modular, reusable, documented, and reused where possible
    b. There are no global variables
    c. Constants (defines) are used where applicable
    d. Return values indicate error state, unless otherwise specified
    e. Helper functions are used when it's consistent with the program design
8. Only use techniques and tools used during the course (tutorials, notes, lectures)
9. **There are no memory leaks on exit** or memory errors during execution
10. **Compound data types must always be passed by reference**
11. **A reminder: Functions that are not run, or do not compile, will receive severe deductions.**

Grading:

While specifics of the grading scheme may change, the following is an approximation of the weights of each section:

- 20 marks: correct design and implementation of main functionality
- 9 marks: correct design and implementation of initialization functions
- 20 marks: correct design and implementation of add evidence function
- 15 marks: correct design and implementation of add evidence support functions
- 10 marks: correct design and implementation of delete evidence function
- 5 marks: correct design and implementation of print notebook function
- 3 marks: correct design and implementation of clean up
- 4 marks: correct packaging
- 4 marks: correct documentation
- 10 marks: correct program execution
    - **Note**: Additional deductions may be included here when functions are not executed, and reduced grades will be given for any function that can not be verified during execution due to the code not being run, causing an error, or otherwise not compiling

Policy Reminders:

Please recall that all course materials are copyright and you may not share or distribute the materials to anyone outside of the class for any purpose.

Collaboration of any kind outside of general material discussion is strictly prohibited.

Late submissions will not be accepted, unless an announcement is made which overrides this policy.

Submit early, submit often. Technical issues at the deadline do not warrant an extension. You are expected to submit your work periodically to ensure at least part marks for work done prior to the deadlines.

For support or clarifications, attend TA office hours or post to the course Brightspace discussion forums.

*You have **one week** after you receive your grades back to contact the TA that graded your assignment to dispute your grades, after which it will not be considered. Disputes are only valid if the TA clearly graded incorrectly; disagreeing with the grading scheme is not a valid dispute.*

Changelog:

- October 13th, 2022:
    - **Fix:** Specification 1.2 showed an incorrect type for value in the initEvidence function. Value should be passed as a *float*, not as an int. The provided function prototype in the provided code correctly has this type and does not need to be updated.