COMP 2401 B/D (Updated: Nov. 7th)

Assignment #4: Tracking the Hunt

Collections and Pointers

Goals:

For this assignment, you will write a program in C, in the Ubuntu Linux environment of the course VM, that allows the user to manage their estimated likelihood of various “ghosts” and the rooms they might be in by storing them in different data structures.

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 manipulates collections using encapsulated functionality
- Implement different kinds of collections, including an array of pointers and a singly linked list with a tail
- Practice writing code that manages multiple pointers to the same data

Assignment Context (Optional Read):

Excellent stuff with that evidence database! Now that our team is able to track evidence in real time, they’re able to make some better guesses about the kinds of spooky ghosts (or neighbourhood bullies, I guess) that are wandering the buildings.

This week, you’ll be building another data manager that will help us track the estimated likelihood of which kinds of spirits (or otherwise) are in each room. You’ll need to track a list of rooms, and each room has a list of ghosts that we think might be in it. Each ghost has a type, and a likelihood of it being in that room based on the evidence we collected using your previous work. We’ll also give a unique ID to each one, to make sure our likelihoods aren’t getting all confused. Long term, we want to be able to track lots of different ghost types and information, so our Building data should track both a list of rooms *and* a list of all the suspected ghosts.

This program is just a proof-of-concept, though, so we won’t be checking against our full database of ghost types just yet. We’ll keep it simple for our junior hunters - we’ve expanded our club to include a once-weekly after school program for local children looking for some fun scares, but they have to be home before 8:00PM. One of them was inspired by your decryption algorithm and even started working with Arduino coding!

Oh, and we’ll be keeping an eye out for any rooms we think were set up by the local bullies. Real or not, I don’t think they get this is all for fun.

Overview:

For this assignment, you must make sure that you are following the good code practices that we've been following for the previous assignments. This assignment is quite similarly structured to the previous assignment, but you'll be working with data that's pointed to in multiple locations. Be extra careful to have no memory leaks, and to avoid accessing freed data.

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 *a4*: `gcc -o a4 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)

For this assignment, you will be writing the code to manage a Ghost database. There are Buildings, and each Building maintains a collection of all Ghosts in the building and all Rooms. Each Room maintains a collection of Ghosts in that room. Each Ghost tracks the Room it is in and uses an enumerated type to track what type of ghost it is, and the likelihood that the ghost is actually present (a floating point value between 0.0 and 100.0, but you do not need to validate this).

The implementation details are important to follow - while some of the approaches to data structures and operations may not be "optimal", they are here to provide you with practice with pointers and freeing memory in more complex scenarios.

Provided Code:
Your submission must separate the program's functions into the following source files, with structures, forward declarations, and constant definitions in the **defs.h** file:

- **main.c** : `main(), printMenu()`
- **building.c** : `initBuilding(), loadBuildingData(), cleanupBuilding()`
- **ghost.c** : `initGhostList(), initGhost(), addGhost(), addGhostByLikelihood(), printGhosts(), printByLikelihood(), printGhost(), cleanupGhostData(), cleanupGhostList()`
- **room.c** : `initRoomArray(), initRoom(), addRoom(), printRooms(), cleanupRoomArray()`

You will need to create function prototypes, forward declarations, and structs as defined in the instructions below.

Deductions Reminder: All of the standard penalties and requirements continue to apply. You may add additional helper functions if they help you to better organize and otherwise write cleaner code, but otherwise you should not modify the code or structure of code unless specified.

Print Formatting: There is no specified way of printing. The print formatting **may be assessed**, but it is up to you to make sure that it is printed in a nice and legible way. Group like data together, make sure to print all data of the structures, make sure that data is in a consistent "shape" (eg. fixed-size columns) and make sure that each element of a list is visibly separated. It does not need to be perfect, but it should be easy to read and understand at a glance.

Instructions:

Data Structures
While there is some inference required for the exact definitions of these data structures, make sure that all of the data structures follow the notes below. You must define the following data types:

1. **RoomArrayType:** A static array containing many rooms. The `RoomArrayType` contains an `elements` field, which must be declared as a statically allocated array of `RoomType` **pointers**. You will set the maximum capacity to an existing, predefined constant, and the structure must contain a `size` field to track the current number of rooms in the elements array.

2. **GhostListType:** A singly linked list meant to hold `NodeType` elements which point to `GhostTypes`. It must have both a head and a tail.

3. **NodeType:** The `NodeType` works with the `GhostListType` structure to implement a singly linked list of `GhostType` data.

4. **BuildingType:** This is a kind of "Overall Data" collection, which represents a building in our data, and holds all of the data at a higher level for us to reference. It contains two statically allocated fields: A linked list of ghosts, stored as a `GhostListType`, and a static array of rooms, stored as an instance of the `RoomArrayType`.

Generally, you can think of the structures as:
- A room contains many ghosts
- A ghost is in one room
- The building contains a list of every ghost and an array every room, like a big database
- We are only storing one copy of each room and ghost in memory, outside of temporary variables

1. **Implement the ghost management functions**

1.1. Implement the `void initGhostList(GhostListType *list)` function that initializes both fields of the given list parameter to default values. Since collections always begin as empty, the head and tail of the list should be set to NULL.

1.2. Implement the `void initGhost(int id, GhostEnumType gt, RoomType *r, float like, GhostType **ghost)` function that dynamically allocates memory for a GhostType structure, initializes its fields to the given parameters, and "returns" this new structure using the ghost parameter.

1.3. Implement the `void addGhost(GhostListType *list, GhostType *ghost)` function that adds the `ghost` *directly at the back* (i.e. the end) of the given list. The linked list must be implemented as we saw in class, specifically with no dummy nodes.

1.4. Implement the `void addGhostByLikelihood(GhostListType *list, GhostType *ghost)` function that adds the `ghost` to the given list, *directly in its correct place*, in descending order by likelihood. You must implement this by finding the correct insertion point in the list, as we saw in class, and inserting the new ghost. Do not add to the end of the list and then sort. Do not use any sorting function or algorithm.

1.5. Implement the `void cleanupGhostData(GhostListType *list)` function that traverses the given list and deallocates *its data only*.

1.6. Implement the `void cleanupGhostList(GhostListType *list)` function that traverses the given list and deallocates *its nodes only*.

1.7. Implement the `void printGhost(GhostType *ghost)` function that prints to the screen every field of the given ghost. The ghost type must be printed as a descriptive string (e.g. "Poltergeist", "Wraith", "Phantom", "Bullies", "Other", or "Unknown"). The name of the ghost's room must be printed out as well, or the string "Unknown" if the ghost has no room specified.

1.8. Implement the `void printGhosts(GhostListType *list, int ends)` function that prints every ghost in the given list to the screen, using an existing function. If the `ends` parameter is set to `C_TRUE`, then the two ghosts at the head and tail of the linked list must be printed out a second time, after all the ghosts have been output. If the `ends` parameter is set to `C_FALSE`, then the head and tail are not printed a second time at the end.

1.9. Implement the `void printByLikelihood(GhostListType *origList, int ends)` function that prints every ghost in the given list to the screen, in decreasing order by likelihood. The function **must** be implemented as follows:

(a) declare a temporary list as a local variable, and initialize it as empty
(b) traverse the given list `origList`, and add each of its data elements to the temporary list, in descending order by ghost `likelihood`; **do not** make copies of the data! The temporary list is a separate list with its own unique nodes, but with pointers to the same data as the original list
(c) print out the ghosts in the temporary list, using the given `ends` parameter as described in 1.8
(d) clean up the memory for the temporary list, *without deallocating the data* because the same data is still used in the original list

NOTE: You must reuse existing functions everywhere possible.

2. **Implement the room management functions**

2.1. Implement the `void initRoomArray(RoomArrayType *arr)` function that initializes the fields of the given `RoomArrayType` structure that require initialization. In this case, the size of the collection begins with zero elements.

2.2. Implement the `void initRoom(int id, char *name, RoomType **room)` function that does the following:

(a) dynamically allocate memory for a `RoomType` structure
(b) initialize the new structure's fields to the given parameters
(c) dynamically allocate a new `GhostListType` structure to store the new room's ghost list
(d) initialize the new ghost linked list as empty, by reusing an existing function
(e) "return" the new room structure using the `room` parameter

2.3. Implement the `void addRoom(RoomArrayType *arr, RoomType *r)` function that adds the room `r` to the back of the array structure in parameter `arr`.

2.4. Implement the `void cleanupRoomArray(RoomArrayType *arr)` function that deallocates the dynamically allocated memory in the given room collection. This includes each room, and each room's ghosts list and its nodes.

2.5. Implement the `void printRooms(RoomArrayType *arr)` function that traverses the given room array and prints the fields of each room structure to the screen. Every room's ghost list must also be printed out, using an existing function, but without printing out the head and tail. There is no specified way of printing this, but the rooms must be clearly separated from each other, and the ghost list must look clearly associated with the room.

3. **Implement the building management functions**

3.1. Implement the `void initBuilding(BuildingType *b)` function that initializes the two fields of the given building structure. This initialization must be performed by calling existing functions.

3.2. Implement the `void cleanupBuilding(BuildingType *b)` function that cleans up all the dynamically allo-cated memory in the given building `b`. This includes the room data, and both the nodes and data of the master list of ghosts.

4. **Implement the main control flow**

Implement the `main()` function as follows:

4.1. Declare a local `BuildingType` variable to store the building data in the program, including the rooms and the master list of all ghosts.

4.2. Initialize the local `BuildingType` variable by calling an existing function.

4.3. Load the building data into the local building structure, by calling a function provided in the base code.

4.4. 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.

4.5. The “print rooms”, “print ghosts”, and “print ghosts by likelihood” functionality must each be implemented by calling an existing function. The “print ghosts” feature prints out all the ghosts in the building’s master list in ascending order by ghost ID (you **may** assume that they are entered into the list in ascending order). The “print ghosts by influence” feature prints out all the ghosts in the building’s master list in descending order by ghost likelihood.

4.6. At the end of the program, the building data must be cleaned up.

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:

- 7 marks: correct design and implementation of main control flow
- 6 marks: correct design and implementation of ghost initialization functions
- 18 marks: correct design and implementation of add ghost functions
- 20 marks: correct design and implementation of print ghosts functions
- 6 marks: correct design and implementation of cleanup ghosts functions
- 20 marks: correct design and implementation of room management functions
- 5 marks: correct design and implementation of building management functions
- 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:

- 2022-11-04: Significant additions defining the data types needed to be implemented in the instructions.
- 2022-11-07: Correction: The RoomTypeArray should hold an array of **RoomType** pointers, not **GhostType** pointers.