COMP 2401 B/D

Final Project

Threads, and Putting it All Together

Goals:

This project covers most of the major topics we've covered in the course, and largely leaves the design of the program up to you. You are allowed to reuse, and reference, old and similar code from previous assignments in the course. Many of the data structures and operations are repeated from what we've done in other assignments.

This is a big project, and you are expected to reuse and improve upon functionality from previous assignments to help you get through sections quicker. The goal is that you can take time before programming to plan your code and identify suitable areas for code reuse. Some areas of the design are specified to test specific knowledge. Areas unspecified are up to you.

Optional Pairs:

For this final project, you are allowed to work in pairs - following some specific rules and allowances.

- **Requirements and Restrictions:**
  - If you wish to work as a pair, you must self-enroll in a group on Brightspace **before November 23, 11:59PM**
    - At the same time as your partner, go to Tools -> Groups and join the same group.
  - If you encounter difficulties with your partner, it is your responsibility to reach out to the instructor via email with your team mate, note the issue and note that you are no longer pairs, and leave the group on Brightspace.
    - Any work done by the group up until the point of separation may still be used by either partner. Collaboration beyond this point is considered misconduct. No other action will be taken beyond this.
  - You are **restricted from using a "Divide and Conquer" technique**. This is to say, you **must** collaborate on the program at the same time wherever possible, working together on each piece. You are responsible for understanding the full code, not just parts that you work on.
    - A short video on pair programming will be released to help facilitate this, but you can research the technique on your own
    - For remote collaboration, consider installing Visual Studio Code on the virtual machine and utilizing the "Live Share" plugin. Alternatively, you can screenshare via Discord.
- **Allowances**
  - As an established pair, registered on Brightspace, you are allowed to review and discuss the code of each other's previous assignments at your own discretion to help you identify possible areas for code reuse.

Please be aware that the project is fully capable of being completed individually. Pair options *may* increase speed, but it is primarily meant as a way to open communication about your approach with another student and contrast design ideas.

Attempting to use a "divide and conquer" approach can lead to severe difficulties at the deadline with code not being put together correctly, and thus a pair-programming approach is required if you choose to work in pairs.

If you would like to be placed in a random pairing, please let me know. Positions are not guaranteed.

Weekly Check-Ins

This is a large and complex project, and you are expected to be working early and regularly on it. As part of this, you will be required to fill out a small weekly reflection each week to check-in on the state of your work.

The reflections will be a few short answer questions in a Brightspace quiz, location under the Project submodule.

- The first reflection will run from **Wednesday November 23rd, to Monday November 28th, at 11:59PM.**
- The second reflection will run from **Wednesday, November 30th, to Monday, December 5th, at 11:59PM.**

Each reflection is worth 5% of the final project grade, and will be evaluated SAT/UNSAT, if meaningful (even if brief) details are provided. You are required to make progress on the project before submitting each reflection. Reflections represent the state of the project up to the time you submit the reflection.

If you are in a pair, there will be additional questions in the reflection for you to answer relating to your team mate.

Assignment Context (Optional Read):

It's been a long time coming, but I think we're ready to start using the communications network you have been building for some real ghost hunts! Of course, we don't want anything to go wrong in the field, so we want you to create a simulator for us.

Listen, we know you have programming chops at this point. We aren't going to tell you how to work. I'm sure you can find some use from all of those old programs you've been writing; you have been writing them for reusability and readability, right? Well, either way, all that hard work should pay off and help move things forward here.

What we need you to build is a full communication simulation. A map of connected rooms. Four hunters, each with a single device to take readings of evidence from a room. A ghost, wandering around and leaving evidence. When a hunter is in a room with a ghost, their fear level goes from 0 to 100, and when they reach 100, they're *out of there*.

See, every ghost leaves behind three kinds of evidence. We get readings in every room of course; there's also sounds and temperatures to track, but a ghost leaves behind a special form of evidence. If we can find all three of those special kinds of evidence, we can identify the ghost and get rid of it!

For this simulation, you'll be generating data for evidence when we look for it. If there's a ghost, make sure to use its ghostly data.

Background Information:

This is a summative project, meant to evaluate many aspects of your course knowledge. Primarily, this project looks at:

- User Input
- Dynamically and Statically allocated memory
- Linked Lists
- Static Arrays of Pointers
- Multi-threaded programming
- Makefiles

Make sure to review these topics before beginning the project.

Data and and Behaviours:

This section describes the overall definitions and control flow at a high level. More detailed instructions and requirements will be provided below. Be aware that your goal is to have a functioning, multi-threaded simulation, which follows the rules laid out here, and which executes without crashing, without valgrind warnings, and without memory leaks. While some implementation details are specified and must be adhered to, a large part of this assignment is using the tools from class to design your implementation. While you may create new collections and *temporary* structures, data (i.e. evidence, ghosts, rooms, and hunters) should only exist once in memory.

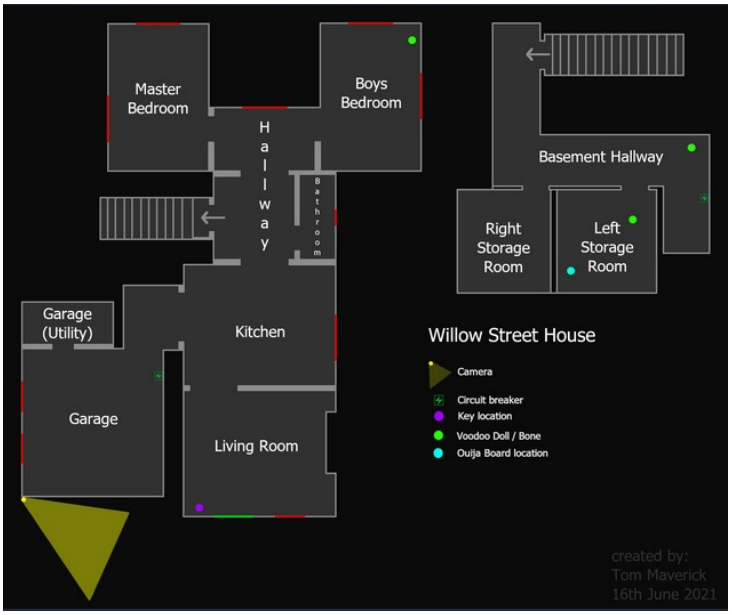
Please be aware: Some of the data types will be used somewhat differently than we have used in class, and you are allowed to modify structures where requirements have not been specified. You are allowed to add additional data to these, as long as it follows good design principles.

Data Definitions and Behaviours:

1. **Room:** A room represents a room in the building that the hunters are hunting in. A room has a name, a linked list of other rooms that it is connected to, a linked list of evidence that the ghost has left in the room, a collection of hunters currently in the room, and a pointer to the ghost. If the ghost is not in the room, the ghost pointer should be NULL.
   Note: You will be required to create a Linked List of RoomNodes, with RoomNodes containing Rooms.
   We must be able to append rooms to the end of the linked list, but we do not need to account for random insertion.
   You will likely want to write a separate, reusable function that takes in two Rooms and connects them together.

2. **Building**: A building is used to hold all of the information about the current hunt. It contains the ghost (either statically or dynamically allocated), a collection of all hunters, and a linked list of all rooms. The first room in every building is always the ghost hunter's vehicle, the Van. The building that we will use is based on the "**Willow Street House**" from the video game Phasmaphobia. A map is available via the Steam Community and below [link].

   Note: A sample populate function has been provided in the code building.c, moving from the van to the hallway. Your own design decisions may require modifying this code in order to work. It is provided for demonstration only. You may use your own building design, but it should have a similar complexity, i.e. at minimum 10 rooms, no single room is connected to all other rooms, and at least one room requires moving four times to reach. You must also include a visualization of the rooms, either as part of your program, as an easily understandable data file, or as a separate image included with the submission.



3. **Ghost:** A ghost is an entity that moves between rooms and leaves evidence behind. A ghost contains a GhostClass, which is an enumerated data type representing the type of ghost it is. It also contains a pointer to the room that it is in. It contains this pointer so that ghost related functions need only to know about the ghost to operate. It also contains a boredom timer - an integer initially set to BOREDOM_MAX. Each time the ghost is in a room with a hunter, it resets the counter to BOREDOM_MAX. Each time the ghost is in a room without a hunter, it decreases the timer variable by 1. If the boredom timer reaches <= 0, it is too bored, and ends its haunting.

   A ghost takes one of two actions: It has a random chance to generate evidence to add to the room, or it will move, or it will take no action. If a ghost is in a room with a hunter, it will not choose to move, but it may choose to take no action. These actions are selected at random.

   Note: There is only ever one ghost in a building. Each ghost leaves 3 different types of evidence at random from a total of 4 pieces of evidence, defined in the table below: Table 1. A ghost can be a POLTERGEIST, BANSHEE, BULLIES, or PHANTOM.

4. **Hunter:** A hunter is an entity that moves between rooms, reads the room for evidence, and communicates evidence. They contain a pointer to the room they are currently in, an enumerated type representing the type of evidence their equipment can read (note: there is no equipment type, only the type of evidence they collect), a collection of evidence that they have personally collected, a name, and a fear integer which starts at 0. Hunters choose their action at random. If a hunter is in a room with a ghost when they make their decision, they gain 1 point of fear, specified by a definition in defs.h. If they reach 100 fear, they are removed from the simulation by closing their thread. If you wish, you may add logic to the hunter's actions beyond random selection. Document these behaviours in your README. If a hunter is in the same room as another hunter, they can choose to communicate. If they communicate, append **ghostly evidence data** to the other hunter's evidence collection. Ignore standard data.
   A hunter should also have a an integer timer variable. Each time they detect ghostly evidence, reset the timer to BORING_MAX. Each time they take an action other than detecting ghostly evidence, decrease the timer by 1. If the timer is <= 0, they are bored, and will leave the building.

5. **Evidence:** Evidence is a simple data type containing the Evidence Class (or Category) and a value for the data of that reading. Evidence can be one of the following categories: EMF, TEMPERATURE, FINGERPRINTS, SOUND. The value for each piece of evidence is selected from the below table, Table 1. A piece of evidence can have either a Ghost Reading or a Standard Reading; note, these readings are not reflected as fields. Instead, when we create evidence, we generate the value for the evidence randomly following the table.

   Note: If a ghost produces a value that is within the standard results, it can not be used as evidence for confirming that ghost type. It is a distraction. Evidence is stored by each hunter, and in each room. Multiple hunters may point to the same data, but once it is found, it is no longer in the room.

Evidence Category	Standard Results	Ghostly Results
EMF	0.00 - 4.90	4.70 - 5.00
TEMPERATURE	0.00 - 27.00	-10.00 - 1.00
FINGERPRINTS	0.00	1.00
SOUND	40.00 - 70.00	65.00 - 75.00

Table 1: Ranges of data to use when generating evidence values, either if the correct ghost is in the room, or if the ghost is not in the room.

Ghosts and their Evidence Types:

- **POLTERGEIST:** Leaves ghostly EMF, TEMPERATURE, and FINGERPRINTS
- **BANSHEE:** Leaves ghostly EMF, TEMPERATURE, and SOUND
- **BULLIES:** Leaves ghostly EMF, FINGERPRINTS, and SOUND
- **PHANTOM:** Leaves ghostly TEMPERATURE, FINGERPRINTS, and SOUND

Hunters obtain evidence by searching a room, or by communicating with other hunters in the same room.

Multi-Threading and Control Flow:

It is mandatory that each entity - the ghost, and each of the four hunters, runs their behaviour in a single thread each.

At every stage, display the current state of the simulation. You may use print statements to print the current action being taken, and the relevant data.

The main control flow should be loosely as follows:

1. **Initialize**
   - **1.1.** Ask the user to input 4 names for our hunters
   - **1.2.** Populate the building with rooms
   - **1.3.** Place the 4 hunters in the head of our room list, which should be the Van
   - **1.4.** Place the ghost in a random room, that is not the van. The ghost *may* move into the van later, but can not start there.
   - **1.5.** Initialize one thread for each hunter and one thread for the ghost.
2. **Ghost Thread**
   - **2.1.** If the Ghost is in the room with a hunter, reset the Ghost's boredom timer to BOREDOM_MAX, and it cannot move. Randomly choose to leave evidence or to do nothing.
   - **2.2.** Otherwise, if the Ghost is **not** in the room with a hunter, decrease the Ghost's boredom counter by 1. Randomly choose to move to an adjacent room, to leave evidence, or to do nothing.
   - **2.3.** If moving to an adjacent room, remember to update the room's Ghost pointer and to update the Ghost's Room pointer.
   - **2.4.** If leaving evidence, generate a new evidence structure from the ghost's list of three ghostly evidence types and add it to the room's evidence collection.
     - **2.4.1.** Randomly select one of the three evidence types
     - **2.4.2.** Randomly generate a value within the ghostly evidence range
     - **2.4.3.** Create a new Evidence structure with these two pieces of information, and add it to the room's evidence collection
   - **2.5.** **A room should contain a mutex semaphore to ensure only one thread is modifying it at a time.**
   - **2.6.** If the ghost's boredom counter has reached <= 0, exit the thread.
3. **Hunter Threads**
   - **3.1.** If the hunter is in a room with a ghost, increase the fear field of the hunter by 1 (which should be defined by a definition in defs.h so that it can be modified). Reset the hunter's boredom timer.
   - **3.2.** Randomly (or using your own, programmed logic) choose to either collect evidence, move, or communicate evidence if the hunter is in the same room as another hunter
     - **3.2.1.** If you the hunter is collecting evidence, look through the room's evidence collection. If there is an evidence that matches the type of evidence the hunter can detect, remove it from the evidence collection and add it to the hunter's evidence collection.
     - **3.2.2.** If the evidence is ghostly evidence, reset the hunter's boredom timer.
     - **3.2.3.** **Note: When working with the room's evidence collection, the room should be locked using its mutex semaphore**.
     - **3.2.4.** If there is **no** evidence, randomly generate a new piece of evidence using the Standard Evidence values of the type that the hunter can detect and add it to the hunter's evidence collection. This

does not modify the room's evidence in any way.

3.2.5. If the hunter is moving, move to a random, connected room. Make sure to update the room pointer in the hunter and the hunter collection in the room. Decrease the boredom counter by 1.

3.2.6. If the hunter is communicating, pick a hunter at random from the room and append any ghostly evidence readings to their evidence collection; do not append standard evidence readings.

3.3. Check if the hunter's evidence collection contains three different pieces of ghostly evidence. If it does, exit the thread

3.4. Check if the fear level of the hunter is greater than or equal to 100. If it is, exit the thread.

3.5. Check if the hunter's boredom timer is <= 0. If it is, exit the thread.

4. Finalize Results

4.1. When all threads have completed, print the results to the screen.

4.2. List all hunters that have fear >= 100.

4.3. If all hunters have fear >= 100, print the ghost type, and that the ghost has won.

4.4. If at least one hunter has fear < 100, then they should have collected enough ghostly data to confirm what type of ghost it is. Print the ghost type, print the speculated ghost type, and if it is correct (which it should be every time), print that the hunters have won.

Design & Submission Requirements:

While there is flexibility in the design, there are some requirements that must be met.

1. Anticipate separating your program into 15-35 modular, and reusable functions, that could be used to extend the program functionality even further.
2. Your functions should be single-purpose and reusable, and they must have a clear interface, with parameters that are defined as either input, output, or input-output parameters.
3. Your functions must include, at minimum:
    a. Individual functions which initialize each structure, as needed
    b. Functions for cleaning up dynamically allocated data once it is no longer needed
    c. Functions for appending and removing from collections
    d. One function which executes the ghost thread, and one function that executes the hunter thread
        i. They should take in a void pointer to a ghost and hunter, respectively
    e. One function that computes the ghost that was identified based on one hunter's evidence collection
4. Every function must be documented, along with each of its parameters and the return value
5. You must use a `Makefile` that separates compiling and linking of your code with a `clean` target to remove object files and executables.
6. If you use a map other than the one provided, include an image representation of the map to show connectivity
7. Submit all of your files as a `.zip` or `.tar` file with everything needed to run your program
8. There must be no memory leaks, no warnings, and no valgrind warnings.

You may use static or dynamic memory allocated where it makes sense, and where it is required for the program.
You must **not** duplicate ghosts, rooms, evidence, or hunters.
You must **not** use global variables - functions must communicate via parameters, but global constant definitions (eg. `#define FEAR_RATE 1`) are acceptable and encouraged.

The program **must** be multi-threaded, and **not** multi-process. Not using multi-threading techniques will lead to severe penalties.

Grading:

A regular, approximate grading scheme will be provided soon, in an updated specification. For this project, some bonus marks are available for creative input. Note that your mark cannot exceed 100%.

Bonus Marks:
If you choose to do any of the following, you will receive some bonus marks, up to a maximum of 100%. If you have a bonus in mind that is not covered by these, let me know and I will confirm if it will receive bonus marks or not. You can receive a maximum of 5% through bonus marks. Each bonus is equally weighted at 1% each, regardless of time or effort required to implement.

If you implement any bonuses, list them in the README file.

- Display the map using an ASCII representation during the run of the program
- Read the map data from a file
- Write a log of all of the events to a file
- Use Git and either GitLab or GitHub (using private repositories only) to version control your code
    - Include a screenshot or log of your commit history to show that these tools were used
- Use print formatting to make the simulation data clear and easy to read
- Allow two hunters to hunt, and include an action to change which evidence type they are collecting. Two hunters cannot collect the same evidence type at the same time.
- Program smarter behaviour than purely random movement and action selection

Grading:

Additional clarifications may be posted over time. Make sure to keep up to date with announcements, just in case any clarifications are needed.