COMP 2401 A/C – Assignment #5

Due: Thursday, December 7 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, that simulates the escape of two heroes, Timmy Tortoise and Prince Harold the Hare, through a small valley known as Dragon's Hollow, as they are pursued by an evil wizard's attack birds and flying monkeys.

Unbeknownst to our heroes, the escape is being broadcast by the evil wizard over a stream socket, where any spectator can connect. So all of Timmy and Harold's friends back in their home district are watching our heroes' progress on their screens in real-time.

Your program will simulate the escape by our heroes through Dragon's Hollow, as the wizard's flying monkeys and attack birds try to stop them. The program will be implemented as multi-process, with *two separate instances of the same program*. One process computes all participant moves and displays them on the screen in one terminal window, and the other process allows a spectator to view the identical escape output, at the same time, in a different window. Your code will use well-designed modular functions, as well as inter-process communications (IPC) over stream sockets, to implement this simulation.

2. Learning Outcomes

With this assignment, you will:

- write a multi-process program that uses stream socket IPC techniques
- use command line arguments to determine program behaviour
- optionally, practice writing code with threads and a mutex

3. Instructions

Your program will simulate Timmy and Harold's escape through Dragon's Hollow, as a variation of the client-server architecture. Each instance of the program will use command line arguments (covered in section 2.3 of the course material) to determine whether to behave as either a *server* process which simulates our heroes' escape and displays its output, or as a *client* process which connects to a running server in order to receive and display the identical escape output.

Because the movements of each participant (hero, bird, and flying monkey) are randomly generated, every execution of the simulation will have a different outcome. Your code must show the simulation as it progresses, including the changing positions of each participant, in both terminal windows. It must print out the outcome of the simulation at the end, specifically whether the heroes escaped the Hollow or died in the attempt.

Your program must follow the programming conventions that we saw in the lectures, including but not restricted to the correct separation of code into modular functions, basic error checking, and documenting each function. A more comprehensive list of constraints can be found in the *Constraints* section of this document.

3.1. Understand the provided base code

The base code provided in the `a5-posted.tar` file contains the constant definitions and structure data types that your program must use, *without modifications*, as well as the required function prototypes.

3.1.1. The `PositionType` stores the row and column for a participant's current location in the Hollow.

3.1.2. The `ParticipantType` contains the basic data for every participant (hero, bird, or monkey).

3.1.3. The `HeroType` stores the data for one hero (Timmy or Harold).

3.1.4. The `FlyerType` stores the data for one of the evil wizard's minions, either a bird or a flying monkey.

3.1.5. The `HeroArrayType` and `FlyerArrayType` types store collections of heroes or flyers, respectively. All hero and flyer instances *must be dynamically allocated*.

3.1.6. The `EscapeType` contains the necessary data for the simulation. This includes a collection of all heroes and a collection of all flyers, and the two TCP/IP sockets required for the server process to communicate with the client process.

3.1.7. The `connect.c` source file contains the functions necessary for the server and client to connect and communicate with each other. To understand these functions and how to use them, you must be familiar with the course material covered in section 5.3.

3.1.8. A `randomInt()` function has also been provided for you. You must use this function throughout the program to randomize participant behaviour. The pseudorandom number generator must be seeded **once** at the beginning of the program, using the statement: `srand( (unsigned)time( NULL ) );`

3.2. Understand the overall control flow

3.2.1. Each instance of the program must begin by determining whether it must behave as a server that computes the participant moves and displays them, or as a client that connects to a server and displays the server's escape output for its end users, who are the spectators.

3.2.2. If the program is launched *with no command line arguments*, then it becomes a server process and waits for a client connection request. Once a client has connected, the server run the escape simulation and displays the output. Every time the Hollow is printed out by the server, and when the simulation result is printed, that same information must be sent to the client process over a TCP/IP socket, so that the client can display the progress of the simulation in real-time to the spectators.

3.2.3. If the program is launched *with a command line argument*, then it becomes a client process. The command line argument must indicate the IP address where the server process is already executing. The client connects to that server, and every time the server sends data, the client prints it out. The client must be able to connect to a server at *any* IP address (use `127.0.0.1` for the same host).

3.3. Understand the overall rules of the escape

NOV 20 1:00
NOV 22 0:14

3.3.1. The escape simulates the behaviour of multiple participants: Timmy and Harold are trying to make it out of Dragon's Hollow, and the wizard's attack birds and flying monkeys are trying to stop them.

3.3.2. Dragon's Hollow is displayed as a 2D grid of rows and columns, with the origin point `(0,0)` located in the top-left corner. However, this grid is **not** stored in this program. Instead, it is reconstructed into a temporary variable, using participant positions, every time that the Hollow is printed.

3.3.3. Timmy and Harold move exclusively along the ground of the Hollow (the bottom row). Each hero begins their escape on the left-hand side, at a randomly determined column between 0 and 4 inclusively, and the starting column for each hero must be different from each other. A hero successfully escapes the Hollow if they reach the right-most column without dying. Each hero begins the escape with a health indicator at 20 points, and they lose health points with every collision with a bird or flying monkey. If a hero's health indicator reaches zero, the hero dies.

3.3.4. The other rows represent the air above our heroes, where birds and monkeys are flying.

3.3.5. Every participant is represented by an avatar, which is defined as a single character. Timmy's avatar is `'T'`, and Harold's is `'H'`. Every bird is represented as `'v'` (lowercase `v`), and every monkey as a `'@'`. When a hero dies, their avatar is changed to a cross `'+'` that represents their grave.

3.3.6. Your program must declare a single instance of the escape, stored as an instance of `EscapeType`, in the `runEscape()` function. This structure is continuously updated with participant data, throughout the execution of the program. All helper functions must update that same instance of the escape.

3.3.7. The escape is implemented as a continuous game loop that executes until each hero has either escaped the Hollow by reaching its right-hand side, or died in the attempt.

3.3.8. The simulation begins with no birds and no monkeys in the Hollow. At every iteration of the game loop, there is a 80% probability that a new bird is spawned and joins the others in attempting to stop our heroes. A newly spawned bird is initially positioned in a randomly determined row between rows 0 and 4 inclusively, and a randomly generated valid column. Each bird is spawned with a randomized amount of strength, between 3 and 5 inclusively.

3.3.9. At every iteration of the game loop, there is a 40% probability that a new flying monkey is spawned. A newly spawned monkey is initially positioned in a randomly determined row between rows 0 and 14 inclusively, and a randomly generated valid column. Each monkey is spawned with a randomized amount of strength, between 8 and 11 inclusively.

3.3.10. At every iteration of the game loop, every participant (hero and spawned bird and monkey) is moved from their existing position to a new, randomly computed one, as described below. *All row and column positions must be valid within the Hollow*. If a new row or column is computed below zero, then it is repositioned at zero. If a new row or column is computed beyond the maximum, then it is reset to that maximum.

3.3.11. At every iteration of the game loop, for each of our two heroes:

(a) if the hero has already escaped or is dead, then nothing happens; an escaped hero disappears from the grid, and a dead one is replaced with a cross avatar in the location of their death

(b) if the hero is still alive and in the Hollow, then a new position is computed for that hero, based on a randomly selected move, as described in instruction 3.4; the hero's position is then updated to the newly computed one

3.3.12. At every iteration of the game loop, for every bird and flying monkey (let's give them the generic name *flyers*) that has been spawned:

(a) if the flyer has already reached the ground in a previous iteration, then it has disappeared from the simulation; it is no longer participating, and nothing happens

(b) if the flyer is still participating, then a new position is computed for that flyer, as described in instructions 3.4; the flyer's position is then updated to the newly computed one

(c) if the flyer's new position results in a *collision* with a live hero, because the flyer has moved into the exact same position, then that hero incurs damage, by having its health indicator decreased by the amount of strength of the flyer that collided with it; if a hero's health indicator reaches zero (or less), then the hero dies

(d) if a hero dies, their avatar is changed to a cross `'+'` that represents their grave

3.3.13. At the end of every iteration of the game loop, the Hollow and both heroes' health indicators must be printed to the screen and sent to the client process, as follows:

(a) a temporary grid is declared, as a 2D array of chars, and it is populated to represent Dragon's Hollow, with the avatars of all participants placed in their current positions, except flyers that have reached the ground

(b) the 2D grid and the heroes' health indicator information is serialized; for our purposes, this means that the information to be printed out is translated and formatted into a 1D array of chars that contains all the necessary borders, spacing, and newline characters, so that the output is contained in one very long string, which can be printed out with a single call to `printf()`

(c) the serialized Hollow is printed to the screen, then it is sent to the client process

3.3.14. Once the game loop has concluded, Dragon's Hollow must be printed to the screen one final time, and the outcome of the escape must be printed out, in both server and client terminal windows. Both processes must state either that both heroes have escaped, or that both heroes are dead, or they must indicate which hero escaped the Hollow and which one died. The server must also send a "quit" message to the client process, so that it can exit its loop and terminate.

3.4. **Understand the participant moves**

3.4.1. Each hero has a set of possible moves that represents how they progress in Dragon's Hollow. Each type of move is associated with the probability that the move is selected for that hero, and the number of columns (to the left or right) that the move represents. At every iteration of the game loop, a new move is randomly chosen for each hero, with a specific probability, as shown in Table 1.

Table 1: Hero moves

Participant	Type of move	Probability	What happens
Timmy	Fast walk	50%	move 2 columns right
	Slide	30%	move 1 column left
	Slow walk	20%	move 1 column right
Harold	Sleep	20%	no move
	Big hop	10%	move 5 columns right
	Big slide	10%	move 4 columns left
	Small hop	40%	move 3 columns right
	Small slide	20%	move 2 columns left

3.4.2. Each move by a bird is computed as follows:
   (a) the bird moves down from its current row in the Hollow by one row
   (b) the bird moves from its current column by a randomly determined one column to the left, or one column to the right, or stays in the same column

3.4.3. Each move by a flying monkey is computed as follows:
   (a) the monkey moves vertically from its current row in the Hollow by a randomly determined number of rows, between -3 and +3 inclusively; a negative amount means that the monkey is flying upwards from its current row, and a positive amount means that it's moving downwards
   (b) the monkey moves horizontally from its current column by a randomly determined number of columns, between 1 and 3 inclusively; however, the *direction* of the monkey's horizontal move (to the left, to the right, or the same column) depends on the location of the closest live, participating hero to the monkey's current position:
      (i) if the closest live hero is to the left of the monkey, then the monkey moves from its current column by a *negative* amount (towards the hero)
      (ii) if the closest live hero is to the right, the monkey moves by a *positive* amount
      (iii) if the hero is in the same column, the monkey stays in the same column

3.5. Implement the program behaviour

3.5.1. You must implement your program in accordance with all the escape rules described in instructions 3.2, 3.3, and 3.4. All participant data must be dynamically allocated.

3.5.2. The program design has been provided for you as a set of function prototypes in the `defs.h` header file. A short description of each function has also been provided.

3.5.3. The program output in both server and client terminal windows must match the output shown in the workshop video.

3.6. BONUS: Implement multi-threaded behaviour

For five (5) bonus marks, you may implement a separate *communications manager* thread that takes care of outputting the Hollow to the screen and to the client process. Instead of the main control flow outputting the Hollow at every iteration (instruction 3.3.13), the communications manager does this independently, at predefined time intervals (500,000 micro-seconds is a good amount). This thread contains a loop that alternates between sleeping and outputting the Hollow. Your code must use a mutex to lock the escape data when the communications manager is constructing the Hollow from current participant positions, and when the main control flow is updating these positions.

3.7. Document your program

You must document your program, as we covered in the course material, section 1.2.4. Every function except `main()` must have a block of comments before the function to indicate: (1) the function purpose, (2) the correct role for **each** of its parameters (covered in section 1.2.3 of the course material), and (3) the possible return value(s) if applicable. The program as a whole is documented in a `README` file. **Do not** use inline comments as a substitute, as they cannot correctly document a procedural design.

3.8. Package your files

3.8.1. Your program must be separated into a minimum of seven (7) source files, including `main.c` and `connect.c`. Each file (except `main.c` and the client code) should contain at least four (4) functions that are functionally related to each other.

3.8.2. You must provide a `Makefile` that separately compiles each source file into an object file, then links all the object files to produce the program executable. Your `Makefile` must also include the `clean` target that removes all the object files and the program executable.

3.8.3. You must provide a plain text `README` file that includes:
   (a) a preamble (program author, purpose, list of source, header, and data files, if applicable)
   (b) compilation and launching instructions, including any command line arguments

3.8.4. Use either the `tar` or the `zip` utility in Linux to package into one `.tar` or `.zip` file **all the files** required to build and run your program.

3.8.5. Do not include any additional files or directories in your submission.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. Your program must be correctly separated into modular, reusable functions, and it must adhere to the correct programming conventions for C programming in Linux.

4.2. The code must be written using the C11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.3. Your program must not use any library functions or techniques not covered in this course, unless otherwise permitted in the instructions.

4.4. Do not use any global variables.

4.5. If base code is provided, do not make any unauthorized changes to it.

4.6. Your program must reuse the functions that you implemented, as well as predefined constants, everywhere possible.

4.7. Your program must perform all basic error checking.

4.8. Do not use recursion where iteration is the better choice.

4.9. Compound data types must always be passed by reference, never by value.

4.10. Return values must be used only to indicate function status (success or failure). Except where otherwise permitted in the instructions, data must be returned using parameters, and never using the return value.

4.11. All dynamically allocated memory must be explicitly deallocated.

4.12. You may implement helper functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

4.13. All functions must be documented, as described in the course material, section 1.2.

4.14. To earn full program execution marks, your code must compile and execute without warnings or errors or memory leaks, and it must be implemented in accordance with all instructions.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Package your files** instruction.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

- 35 marks: code quality and design
- 35 marks: coding approach and implementation
- 25 marks: program execution
- 5 marks: program packaging
- *5 marks: bonus for implementing multi-threaded behaviour*