COMP 2401 A/C – Assignment #1

Due: Thursday, October 5 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Linux environment of the course virtual machine (VM), that simulates the escape of two heroes, Timmy Tortoise and Prince Harold the Hare, through a small valley known as Dragon's Hollow.

Timmy and Harold were attempting to rescue a baby Dragon from the clutches of the evil wizard who kidnapped it. Unfortunately for our heroes, they were caught before they could complete their mission. Now, Timmy and Harold are fleeing to safety through Dragon's Hollow, but the wizard has dispatched his minions after our heroes. Because the wizard's flying monkeys are currently on vacation, the only minions available for this simulation are the attack birds.

Your program will simulate the attempted escape by our heroes through Dragon's Hollow, as the wizard's attack birds try to stop them. Your code will use well-designed modular functions and array techniques to implement this simulation, using console-based output.

2. Learning Outcomes

With this assignment, you will:

- get familiar with the Linux programming environment and the course VM
- write a small program in C that is correctly designed into modular functions, and correctly documented
- practice with basic user output and the manipulation of primitive arrays in C
- write functions that communicate with each other by passing parameters and using return values

3. Instructions

Your program will simulate Timmy and Harold's escape through Dragon's Hollow. Because the movements of each hero and bird is 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 hero and bird. 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 course lectures, including but not restricted to the correct separation of code into modular functions, basic error checking, and the documentation of each function. A more comprehensive list of constraints can be found in the *Constraints* section of this document.

3.1. Understand the overall rules of the escape

3.1.1. The escape simulates the behaviour of multiple participants: Timmy and Harold, who are trying to make it out of Dragon's Hollow, and the wizard's attack birds who are trying to stop our heroes.

3.1.2. Dragon's Hollow is organized as a two-dimensional (2D) grid of rows and columns, with the origin point `(0,0)` located in the top-left corner.

3.1.3. Timmy and Harold move exclusively along the ground of the Hollow (the bottom row). Because of this, a hero's *position* is the same as the *column* where they are located. Since a hero's row is always fixed, the terms *position* and *column* are used interchangeably for hero participants.

3.1.4. Each hero begins their escape at the left-most column on the ground, and each one successfully escapes the Hollow if they reach the right-most column without dying.

3.1.5. The other rows represent the air above our heroes, where birds are flying.

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

3.1.7. Your `main()` function must declare a single instance of Dragon's Hollow as a 2D array of characters, declared in local scope. This array is continuously updated with participants' avatars and their changing positions in the Hollow, throughout the execution of the program. All helper functions must update that same instance of the Hollow.

3.1.8. The chase 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.

3.1.9. The simulation begins with no birds in the Hollow. At every iteration of the game loop, there is a 90% probability that a new bird is spawned and joins the others in attempting to stop our heroes. A newly spawned bird is initially positioned at row zero and a randomly generated valid column in the grid.

3.1.10. At every iteration of the game loop, every participant (hero and spawned bird) 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.

3.1.11. 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 3.2

(c) the hero is then moved in the Hollow from their old position to the newly computed one

3.1.12. At every iteration of the game loop, for every bird that has been spawned:

(a) if the bird 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 bird is still participating, then a new position is computed for that bird, as described in instruction 3.3, and the bird is moved to its newly computed position in the Hollow

(c) if the bird's new position results in a *collision* with a live hero, because the bird has moved into the exact same position, then that hero dies instantly

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

3.1.13. At the end of every iteration of the game loop, Dragon's Hollow and the avatars that it contains is printed to the screen.

3.1.14. 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. The program must state either that both heroes have escaped, or that both heroes are dead, or it must indicate which hero escaped the Hollow and which one died.

3.2. **Understand the hero moves**

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 3 columns right
	Slide	20%	move 2 columns left
	Slow walk	30%	move 1 column right
Harold	Sleep	20%	no move
	Big hop	10%	move 6 columns right
	Big slide	10%	move 4 columns left
	Small hop	40%	move 4 columns right
	Small slide	20%	move 2 columns left

3.3. **Understand the bird moves**

At every iteration of the game loop, a position is randomly computed for each spawned bird, as follows:

3.3.1. the bird moves in a downward direction from its current row in the Hollow by a randomly determined one or two rows

3.3.2. 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. **Declare the required data**

3.4.1. The base code provided in the `a1-posted.c` file, which is available in *Brightspace*, contains the constant definitions and function prototypes that are recommended for your program to use. It also provides the pseudo-random number generator (PRNG) function that your code must use to randomize hero and bird movements. The provided constants include the following:

(a) the `MAX_ROW` and `MAX_COL` preprocessor constants indicate the number of row and columns, respectively, that represent the dimensions of Dragon's Hollow

(b) the `MAX_BIRDS` constant represents the maximum number of birds that can be spawned by the program at runtime

(c) the `BIRD_FREQ` constant specifies the odds of a new bird being spawned during each iteration of the main loop

3.4.2. The `main()` function must declare the following data as *local variables* and initialize them:

(a) a 2D array of characters that represents Dragon's Hollow; this array must be initialized at the beginning of the program, and updated with each participant avatar at their current position throughout the program execution; remember, variables in the C programming language always contain garbage before they are initialized by the programmer

(b) two parallel arrays that work together to store the position of each bird that has been spawned

(i) one array represents the rows where the birds are located (`rows`), and the other array represents the columns (`cols`)

(ii) the location of each bird can be found at the same index in both arrays

(iii) for example, the location of the third bird is stored at row `rows[2]` and column `cols[2]`

3.4.3. To ensure that every execution of the program produces a different outcome, the PRNG must generate a *unique* sequence of pseudo-random numbers every time the program runs. For this to happen, you must initialize the PRNG by seeding it with the current time. This is done by executing the following statement **once**, at the beginning of your program: `srand( (unsigned)time( NULL ) );`

3.5. **Design the functions**

A valid design has been provided for you below, as a set of recommended functions that you can implement for the required program functionality. You **must** ensure that your program is correctly designed, either by implementing these functions as provided, or by coming up with your own design, subject to the *Constraints* listed in section 4 of this document, and in instruction 3.5.2 below.

3.5.1. The recommended functions include the following:

(a) a `void initHollow(char hollow[MAX_ROW][MAX_COL])` function that initializes the `hollow` parameter to all spaces

(b) a `void printHollow(char hollow[MAX_ROW][MAX_COL])` function that prints to the screen the current content of the grid contained in the `hollow` parameter; the layout and formatting of the output must be identical to what is shown in the assignment workshop video

(c) a `void moveInHollow(char avatar, int oldRow, int oldCol, int newRow, int newCol, char hollow[MAX_ROW][MAX_COL])` function that moves a participant's given `avatar` value, inside the given `hollow` parameter, from the participant's old position at `(oldRow,oldCol)` to its new position at `(newRow,newCol)`

(d) a `char isDone(char avatar, char col)` function that returns the provided, predefined constant equivalent to true if the hero with the given avatar and column position is dead or has escaped the hollow

(e) a `char escapeOver(char tAvatar, char hAvatar, int tCol, int hCol)` function that returns the provided constant equivalent to true if the escape has concluded, or the constant that represents false otherwise; the escape is over if each of the two heroes is either dead or has escaped

(f) a `void printResult(char tAvatar, char hAvatar, int tCol, int hCol)` function that prints to the screen the outcome of the escape, as one of the three possibilities: both heroes are dead, or both heroes have escaped, or one hero is dead and the other escaped; if the last case has occurred, the function must indicate by name which hero died and which one escaped

(g) a `int moveHero(char avatar, int oldPos, char hollow[MAX_ROW][MAX_COL])` function that computes a new position (a new column) for a hero, in accordance with the probabilistic hero moves listed in Table 1, then moves the hero's given avatar from their old position in the given Hollow to the new one, and returns that newly computed position as the return value

(h) a `int moveBird(int index, int rows[MAX_BIRDS], int cols[MAX_BIRDS], char hollow[MAX_ROW][MAX_COL])` function that computes a new position for the bird found at the given index in the given rows and columns arrays, in accordance with the probabilistic bird moves described in instruction 3.3, then moves the bird's avatar from its old position in the given Hollow to the new one, and updates the rows and columns arrays with the bird's new position

3.5.2. **Design alternative:** You may choose to come up with your own design. However, your design must be equal or superior to the provided design, in accordance with every principle of correct software engineering that we uphold in this course, including encapsulation, abstraction, modifiability, extensibility, and code readability. If you are unable to defend your design choices to the course instructor, based on documented principles found in textbooks or peer-reviewed research, or if your design choices undermine the learning outcomes of the assignment, then your design may result in a grade of zero. If you choose this option, it is *strongly recommended* that you discuss your intended design with the course instructor at office hours, before you begin coding.

3.6. Implement the program behaviour

3.6.1. You must implement your program in accordance with all the escape rules described in instructions 3.1. 3.2, and 3.3.

3.6.2. Your code must be separated into a minimum of eight (8) separate non-trivial modular functions, excluding the `main()` function, that communicate with each other exclusively through parameters, and it must reuse all the functions that you implemented.

3.6.3. Your simulation must be randomized, the Dragon's Hollow layout must be printed out after each execution of the game loop, and the printed layout must match the output shown in the workshop video. You may simulate the movement effect from the workshop video with the `system("clear")` statement before the Hollow is printed to the screen each time.

3.6.4. As well, to slow down the execution so that the end user can better see the progress of the participants, you can add a short sleep cycle to every execution of the game loop. For example, the statement `usleep(300000)` will pause the execution for 300,000 micro-seconds.

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 contained within one (1) source file that is correctly named, for example `a1.c`, that contains your `main()` function at the top of the file, followed by the remaining functions.

3.8.2. 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

3.8.3. Use either the `tar` or the `zip` utility in Linux to package the files above into one `.tar` or `.zip` file.

3.8.4. 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 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 predefined constants and functions that you implemented, where 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. 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.

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

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

- 20 marks: code quality and design
- 50 marks: coding approach and implementation
- 25 marks: program execution
- 5 marks: program packaging