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

COMP 2401 A/C – Assignment #2

Due: Thursday, October 19 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, to provide the end user with the ability to encrypt and decrypt messages, using a secret key. The algorithm that you implement will use a *chaining* approach. With chaining, each byte (i.e. each character) is encrypted in a slightly different way, because the encryption of each byte is dependent on the encryption result of the previous byte. This prevents the same character value from being encrypted identically every time. This approach is based on the Cipher block chaining (CBC) mode of the Advanced Encryption Standard (AES) encryption algorithm.

If the end user of your program chooses to encrypt a message (called the *plaintext*), your program will output the encrypted sequence (called the *ciphertext*) to the screen. If the user chooses to decrypt an encrypted sequence (ciphertext), your program will print out the decrypted message (plaintext).

2. Learning Outcomes

With this assignment, you will:

- write a program in C that is correctly designed into modular functions, and correctly documented
- organize a C program by reusing existing functions as much as possible
- use bit masks and bitwise operators to manipulate values at the bit level

3. Instructions

Your program will prompt the user to indicate whether they wish to encrypt a readable message (the plaintext), or decrypt an already encrypted sequence of numbers (the ciphertext) back to its readable form. You will implement the encryption algorithm described below, and you will deduce and implement the corresponding decryption algorithm.

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 encryption algorithm

3.1.1. Both the plaintext and the ciphertext must be stored as arrays of `unsigned char`s. We will manipulate the individual characters using bit masks, exactly as we saw in the the course material, section 2.1.2. Remember: plaintext characters are stored in memory using their numeric ASCII value.

3.1.2. The encryption algorithm processes each plaintext byte, one at a time, with a key that is initialized at the beginning of the program. The initial key value must be identical for both the encryption and the decryption processes, and it must be updated identically with every byte that's encrypted or decrypted. If we encrypt a message with a specific initial key, we must use the **same** key for decryption. If we use a different initial key for decryption, we will get garbage instead of plaintext when we decrypt the ciphertext.

3.1.3. Cipher-block chaining means that part of the encryption process for each plaintext byte requires the encrypted value of the *previous* plaintext byte. So our algorithm needs an initial value for encrypting the first plaintext byte, since there is no previous ciphertext byte to use. This *initial value* (IV) is set to `0b11001011`.

3.1.4. The encryption process has two separate parts:
- the initialization of the key, and
- the encryption of each plaintext byte into a ciphertext byte, using that key

3.1.5. **Initialization of the key:**

Definition #1: Two *mirror bit positions* inside a byte are two bit positions that are the same as each other, relative to each end of the byte. For example, bit positions 0 and 7 are mirror bit positions, because they are both in the first position at either end of a byte. Bits 1 and 6 are also mirror bit positions (both are in second position from either end of the byte), as are bits 2 and 5, and so on.

Definition #2: The *mirror bit value* of a bit in a specific position is the value (zero or one) of the bit at the mirror bit position. For example, the mirror bit value of the bit in position 2 is the value (zero or one) of the bit in position 5.

The initialization of the key takes place as follows:

(a) prompt the user to enter a partial key value; this must be a number between 1 and 15, inclusively, so that only the four least significant bits have values; this number is stored as an `unsigned char`

(b) left-shift the partial key by 4 bits, so that the four least significant bits are moved into the four most significant bit positions

(c) loop over the four most significant bits (from bit position 7 to 4, inclusively) of the partial key

(d) for each of the four most significant bits, assuming `i` as the current bit position:

  (i) compute a secondary bit position as `(i-1) mod 4`

  (ii) set the key bit at the secondary position to the *inverse* of the key bit at position `i`

3.1.6. **The encryption algorithm:**

Once the key has been entered and initialized as described above, the program must prompt the user to enter the plaintext. This is entered from the command line as a single string, and it is stored as a null-terminated array of `unsigned char`s. You have been provided with a function in the base code to assist you with this.

The Encryption Algorithm:

For every byte of plaintext:

(1) process the key and update its value, as follows:

  (a) if the current key value is a multiple of 3, perform a circular left-shift of the key by 3 bits

  (b) otherwise, perform a circular left-shift of the key by 2 bits

(2) encrypt the plaintext byte using the key updated in step (1) above and the previous byte of ciphertext (or the IV if encrypting the first byte of plaintext), as follows:

  (a) initialize a temp byte to zero

  (b) loop over every bit position, starting at bit 0; for every bit position `i`:

    (i) if the key bit at position `i` is one, perform circular right-shift of the previous ciphertext byte by 1 bit, otherwise keep the current value

    (ii) set the temp byte bit at position `i` to the result of an xor operation between the plaintext bit at position `i` and the previous ciphertext bit at the mirror bit position

  (c) once the loop ends, return the temp byte value as the ciphertext byte

3.2. Implement the encryption functionality

3.2.1. You will begin with the base code in the `a2-posted.c` file that is posted in *Brightspace*. **Do not** make any changes to the provided code, including the function prototypes.

3.2.2. Implement the `getBit()`, `setBit()`, and `clearBit()` functions, exactly as we saw in class.
Your program **must** reuse these functions to access and make all changes to the bits inside each byte. Do not access or modify any bits without using these functions.

3.2.3. Implement a function that takes a byte and performs a circular left shift on it by one bit, and another function that does a circular right shift. As we discussed in class, in a *circular* shift, the bits shifted in are the same bits that were shifted out.

3.2.4. Implement the `unsigned char initKey()` function that prompts the user to enter a correct partial key, then initializes the full key as described in instruction 3.1.5. The function returns the initialized key as the return value.

3.2.5. Implement the `unsigned char processKey(unsigned char currKey)` function that updates the key from its current value in `currKey`, as described in step (1) of the encryption algorithm above. The function returns the updated key as the return value.

3.2.6. Implement the `unsigned char encryptByte(unsigned char pt, unsigned char key, unsigned char prev)` function that encrypts the given plaintext byte `pt`, using the given key and the previous byte of ciphertext `prev`, as described in step (2) of the encryption algorithm. The function returns the corresponding encrypted ciphertext byte as the return value.

3.2.7. Implement the `void encode(unsigned char *pt, unsigned char *ct, unsigned char key, int numBytes)` function that takes an array of plaintext characters stored in parameter `pt`, which contains `numBytes` bytes. The function encrypts each plaintext character in the array into its corresponding ciphertext byte using the given key, as described in the encryption algorithm in instruction 3.1.6. Each encrypted byte is stored into the ciphertext array `ct`.

3.3. **Implement the decryption functionality**

Write the functions required to implement the decryption algorithm. The decryption processes the key identically to the encryption, but the decryption of each byte using the updated key must perform the xor operation between different values than the encryption does.

NOTE: The xor operator ($\oplus$) has a very useful property. Assume that `a` and `b` are any numeric value, and that `a` $\oplus$ `b` results in `c`. Then it is also true that `b` $\oplus$ `c` results in `a`, and that `a` $\oplus$ `c` results in `b`. So your decryption algorithm does not need to "reverse" the xor operation that was performed during encryption. Instead, it must perform the xor between different values than during encryption.

3.3.1. Implement the `unsigned char decryptByte(unsigned char ct, unsigned char key, unsigned char prev)` function that decrypts the given ciphertext byte `ct`, using the given key and the previous byte of ciphertext `prev`. The function returns the corresponding decrypted plaintext byte as the return value.

3.3.2. Implement the `void decode(unsigned char *ct, unsigned char *pt, unsigned char key, int numBytes)` function that takes an array of ciphertext bytes stored in parameter `ct`, which contains `numBytes` bytes. The function decrypts each ciphertext byte in the array into its corresponding plaintext byte using the given key. Each decrypted byte is stored into the plaintext array `pt`.

3.4. **Implement the main control flow**

3.4.1. Implement the `main()` function to first prompt the user for whether they want to encrypt a string, or decrypt a sequence of numbers.

3.4.2. The user is then prompted for a partial key, and the full key is initialized as described in instruction 3.1.5. Your code must perform all error checking. Remember, a recoverable error like entering a value out of range is never a reason to terminate a program. Your code should keep prompting for a correct value until one is entered.

3.4.3. If the user chooses to encrypt a string, your program does the following:

(a) read a plaintext string from the command line, using a function provided in the base code; because this string will contain multiple space-delimited words, you cannot use the `scanf()` library function for this; the provided function uses `fgets()` instead

(b) encrypt the plaintext entered by the user, as described in the encryption algorithm

(c) print out to the screen the space-separated ciphertext bytes, as decimal numeric values

3.4.4. If the user chooses to decrypt an encrypted sequence, your program does the following:

(a) read a space-separated sequence of decimal numeric values from the command line, up to a sentinel value of `-1`; the `scanf()` library function is the best choice for this, because one call to `scanf()` reads a single value up to a delimiter, which can be either a space or a newline character

(b) decrypt the sequence entered by the user

(c) print out to the screen the resulting plaintext as a string of ASCII characters

NOTE: The user should be able to decrypt any ciphertext that was produced by the algorithm with the same key value, and it should produce the original string as plaintext.

3.5. **Test your program**

You must test your program thoroughly. If your algorithm is correctly implemented, you will be able to decrypt the following sequence to its correct plaintext, using partial key 9 (bragging rights go to the first student who emails me the correct decrypted message!):

```
086 144 168 059 158 250 223 183 142 109 019 161 114 208 203 118 144 132 112 220 157 231
026 172 077 038 035 045 126 131 102 024 180 150 168 059 158 250 141 039 069 103 005 033
114 145 237 031 185 150 253 223 135 106 029 177 193 115 159 230 062 163 068 -1
```

NOTE: You must try different values for the partial key. As long as you use the same value for both encryption and decryption, the algorithm should work.

3.6. **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.

3.7. **Package your files**

3.7.1. Your program must be contained within one (1) source file that is correctly named, for example `a2.c`, that contains your `main()` function at the top of the file, followed by the remaining functions.

3.7.2. 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

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

3.7.4. 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 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 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.

4.11. To earn full program execution marks, your code must compile and execute without warnings or errors, 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

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

COMP 2401 A/C – Assignment #3

Due: Thursday, November 9 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, that allows the end user to manage a book club, with books that are stored in a collection structure. The program will encapsulate collection structure manipulations within single-purpose modular functions.

2. Learning Outcomes

With this assignment, you will:

- write a program that uses structure data types to represent both a collection and its data
- practice with pointers and memory management techniques, including dynamically allocated memory
- write code that is separated into different header and source files

3. Instructions

The program will present the end user with a menu of options to manipulate a collection of books that have been rated (on a scale of 0 to 5) by members of a book club. The user will have the option to print all the books in the collection, or print the top rated books only, or add a new book to the collection.

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 data representation

3.1.1. The base code provided in the `a3-posted.tar` file, which is available in *Brightspace*, contains the skeleton code that your program must use. **Do not** make any changes to the provided code, including the function prototypes.

3.1.2. The `defs.h` header file contains the constants and data types that your program must use. Among those data types, it defines an enumerated data type that represents the order in which the books must be ordered in a collection (either by author, or by rating). You will note that enumerated data types are represented internally as numeric values, so it's fine to pass them by value as function parameters. However, your program must always use the *constant names* of the enumerated values as provided, and not their numeric equivalents.

3.1.3. The base code also contains the following structure data types that your program must use:

(a) the `BookClubType` contains the information for the book club, including its name and a collection of the books that the club members have read and rated

(b) the `BookType` contains the data for one book, including a unique id, the title, the author name formatted as "LastName, FirstName", the year of publication, and the rating (0 to 5)

(c) the `BookArrayType` represents a collection of books

3.1.4. The `BookArrayType` structure type contains the following fields:

(a) the *size* of the collection is the number of elements that it currently stores

(b) the *elements* of the collection is an array of `BookType` structure pointers; each element of the array (each book) will be *dynamically allocated* before being added to the array, similar to what we did in the coding examples of section 3.2, program #5

(c) the *order* in which the elements must be stored in the array; books in the array can be stored in order by alphabetical order of its author name, or by its rating

3.2. **Organize the code**

3.2.1. For this assignment, and all future assignments in this course, your functions will be separated across multiple *source files* (i.e. the `.c` files). This is a standard Unix convention for packaging software to minimize program building time and to facilitate multi-developer software development.

3.2.2. Your submission must separate the program's functions into the following files:

- `main.c` : `main()`, `printMenu()`
- `load.c` : `loadBookData()`
- `club.c` : `initBookClub()`, `addBookToClub()`, `printBooks()`, `printTopRatedBooks()`
- `books.c` : `initBook()`, `initBookArray()`, `addBookToArray()`, `findInsPt()`, `printBook()`, `printBookArray()`, `cleanupBookArray()`

3.2.3. You must include the definitions file at the top of every source file, with the line: `#include "defs.h"`

3.2.4. Because we have not yet covered the details of compiling and linking, we have to use a somewhat "bad" way of compiling these multiple source files together. For example, if your program contains source files `file1.c`, `file2.c`, and `file3.c`, then you can use the following command to create the `a3` executable: `gcc -Wall -o a3 file1.c file2.c file3.c`

3.3. **Implement the initialization functions**

3.3.1. Implement the `void initBookArray(BookArrayType *arr, OrderType o)` function that initializes every field of the given `arr` parameter to default values. The number of books stored in a new array is always zero, and the ordering of books in the array must be initialized to the given parameter.

3.3.2. Implement the `void initBook(BookType *b, int id, char *t, char *a, int y, float r)` function that initializes every field of the given `BookType` structure, using the values found in the given parameters.

3.3.3. Implement the `void initBookClub(BookClubType *club, char *n)` function that initializes the given `BookClubType` structure, including the club name using the `n` parameter. The club's book array must be initialized by calling an existing function that you implemented in a previous step.

3.4. **Implement the book collection manipulation functions**

3.4.1. Implement the `int findInsPt(BookArrayType *arr, BookType *b, int *insPt)` function that searches an array to find the index where a new book should be inserted, as follows:

(a) loop over the elements of the given array `arr`, with the goal of finding the index where the given book `b` belongs, so that the elements of the array remain in the correct order

(b) if the array must be ordered by author, then find the array index where the new book belongs so that the array elements remain in *ascending* (increasing) alphabetical order by author; if the array contains multiple books by the same author, then these must be ordered by title

(c) if the array must be ordered by rating, then find the array index where the new book belongs so that the array elements remain in *descending* (decreasing) numerical order by rating

(d) when the insertion point is found, as an index in the given array where the new book belongs, it must be returned in the `insPt` parameter

(e) if any error occurs, the function returns an error flag, provided in the base code as a constant, or it returns a success flag otherwise

3.4.2. Implement the `int addBookToArray(BookArrayType *arr, BookType *b)` function that stores the given book directly into its *correct position* in the given book collection.

At all times, the book collection must be ordered as indicated by the `order` field: either in ascending alphabetical order by author name, then by title for multiple books by the same author, or in descending order by book rating. To add a book to the collection, first its correct insertion point must be found, then the other books must be *moved* within the array to make room for it.

The `addBookToArray()` function must be implemented as follows:

(a) check whether there is room in the array for the new book; if not, the book cannot be added, and an error flag must be returned

(b) find the insertion point for the new book by calling an existing function

(c) if the insertion point for the new book was found successfully, you must move each element one position towards the *back* (the end) of the array, from the last book down to the insertion point

NOTE: the insertion of the new book requires "moving" the other books 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

(d) store the new book into the array at the insertion point

(e) increase the size of the array

3.4.3. Implement the `void printBook(BookType *b)` function that prints all the data for one book. Every field of the book must be printed out and perfectly aligned with the other books (this will require you to research the options for the format string parameter of the `printf()` function). You must refer to the assignment introduction video for the expected output format.

3.4.4. Implement the `void printBookArray(BookArrayType *arr)` function that prints all the books in the given collection.

3.4.5. Implement the `void cleanupBookArray(BookArrayType *arr)` function that deallocates the dynamically allocated memory inside the given book collection.

3.5. Implement the book club manipulation functions

3.5.1. Implement the `int addBookToClub(BookClubType *club, int id, char *t, char *af, char *al, int y, float r)` function that takes as parameters a new book id, title, author's first name, author's last name, year of publication, and rating. The function then creates a new book and adds it to the given book club's collection. The function must do the following:

(a) validate the id and the year as positive numbers

(b) validate the rating to be in the range between 0 and 5, inclusively

(c) if any parameter is invalid, then the function returns an error flag

(d) create a single string for the author's name, using `sprintf()`, so that the author's name is stored in the format "LastName, FirstName"

(e) dynamically allocate the memory for a new book, and initialize it by calling an existing function with the correct information

(f) add the new book to the book club's collection, using an existing function

(g) return an error flag if the book could not be added to the collection, or success otherwise

3.5.2. Implement the `void printBooks(BookClubType *club)` function that prints out the book club's name, and all the books in the club's book collection. You must reuse an existing function for this.

3.5.3. Implement the `void printTopRatedBooks(BookClubType *club)` function that prints out the top 20% best-rated books in the given book club's collection. The function **must** do the following:

(a) declare two temporary `BookArrayType` structures: one for all books ordered by rating, and one for the top-rated books only

(b) initialize both temporary book collections, using an existing function, so that both store the books in order by rating

(c) loop over the given club's book collection and add each of its books to the temporary all books collection

(d) compute the number that represents 20% of the total number of books in the club

(e) loop over the temporary all books collection to add only that number of books to the temporary top-rated collection

(f) print out the club's name and the top-rated books collection

3.6. Implement the main control flow

Implement the `main()` function as follows:

3.6.1. Declare a local `BookClubType` variable to store all the book club data in the program.

3.6.2. Initialize the book club structure by calling a function previously implemented.

3.6.3. Load the book data into the book club by calling a function provided in the base code.

3.6.4. 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.

3.6.5. If any error occurs during the execution of a selected feature, a detailed error message must be printed out, and the main menu shown again for a new selection. The program should not terminate unless the user chooses to exit.

3.6.6. The "print all books" functionality must be implemented by calling an existing function.

3.6.7. The "print top rated books" functionality must be implemented by calling an existing function.

3.6.8. The "add book" functionality must be implemented as follows:

(a) prompt the user to enter the new book's data, including id, title, author first and last names, publication year, and rating

(i) the statement `scanf("%[^\n]", title)` will allow the user to enter a space separated sequence that's stored in `title` as a single string

(ii) it may be necessary to read a single character from the command line using `scanf()` just *before* the above statement

(iii) the author's first and last names must be entered as two separate strings

(b) use an existing function to create and add a new book with the user-provided information to the book club

(c) if any error occurs, the `main()` function must notify the user

(d) if the book is added successfully, the user must be notified as well

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

If any errors are encountered in the above, a detailed error message must be printed out to the end user. Existing functions must be reused everywhere possible.

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 four (4) source files, as indicated in instruction 3.2.2.

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 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 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. 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

- 30 marks: code quality and design
- 35 marks: coding approach and implementation
- 30 marks: program execution
- 5 marks: program packaging

COMP 2401 A/C – Assignment #4

Due: Thursday, November 23 at 11:59 pm

1. Goal

For this assignment, you will write a program in C, in the Ubuntu Linux environment, that allows the end user to manage a restaurant, with patrons and reservations that are stored in different types of collection structures.

2. Learning Outcomes

With this assignment, you will:

- write a program that manipulates collections using encapsulated functionality
- implement different kinds of collections, including a dynamically allocated array and a doubly linked list
- practice writing code that dynamically allocates its data and manages multiple pointers to the same data

3. Instructions

The program will present the end user with a menu of options to view restaurant data, including reservations and patrons. The user will have the option to print all the patron (customer) data, or print all the reservations at the restaurant, or print out the reservations for one patron only.

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 data representation

You will begin by understanding the base code provided in the `a4-posted.tar` file, which is available in *Brightspace*.

The base code contains the following structure data types that your program must use:

3.1.1. the `PatronType` contains the information for one patron (customer) of the restaurant

3.1.2. the `ResvTimeType` contains the day and time information for a reservation at the restaurant

3.1.3. the `ResvType` represents one reservation at the restaurant, including the date and time for that reservation, and a pointer to the patron that made the reservation

NOTE: It is mandatory that only **one instance** of each patron and each reservation exist in the program. Your program must not create any copies of this data. Instead, it must store multiple pointers to the same instance of data, where required.

3.2. Define your structure data types

You must update the `defs.h` header file with the new data types that you create, and you must add all function prototypes as forward references to this header file as well. **Do not** make any unauthorized changes to the provided code, including the function prototypes.

You will define the following new data types:

3.2.1. The `PatronArrayType` structure type must contain the following:

(a) an elements field, which must be declared as a dynamically allocated array of `PatronType` *structures* (not pointers), as we saw in the in-class coding example of section 3.3, program #1

(b) a size field that tracks the current number of patrons in the elements array

(c) a `nextId` integer field that stores the unique identifier that will be assigned to the next patron added to the array

3.2.2. The `ResvListType` structure type defines a doubly linked list, with only a head (and no tail). The list structure must also have a `nextId` integer field that stores the unique identifier that will be assigned to the next reservation added to the list.

3.2.3. The `NodeType` structure type works with the `ResvListType` structure to implement a *doubly* linked list of `ResvType` data (as a pointer, as we saw in the in-class coding examples).

3.2.4. The `RestaurantType` structure type represents a restaurant in our program. It contains three fields: the restaurant's name as a string, a linked list of all the reservations in the restaurant, stored as a `ResvListType` structure, and an array of all the restaurant patrons, as an instance of the `PatronArrayType` type. All three fields must be allocated statically.

3.3. **Organize the code**

3.3.1. Your submission must separate the program's functions into the following source files:

- `main.c` : `main()`, `printMenu()`
- `load.c` : `loadResData()`, `loadPatronData()`
- `restaurant.c` : `initRestaurant()`, `createResv()`, `printResByPatron()`, `cleanupRestaurant()`, `validateResvTime()`
- `resv.c` : `initResvList()`, `initResv()`, `initResvTime()`, `addResv()`, `lessThan()`, `printReservations()`, `printReservation()`, `cleanupResvList()`
- `patrons.c` : `initPatronArray()`, `addPatron()`, `findPatron()`, `printPatrons()`, `cleanupPatronArray()`

3.3.2. Each source file must `#include` the `defs.h` file, in order to have access to its contents.

3.3.3. Because we have not yet covered the details of compiling and linking, we have to use a somewhat "bad" way of compiling these multiple source files together. For example, if your program contains source files `file1.c`, `file2.c`, and `file3.c`, then you can use the following command to create the `a4` executable: `gcc -o a4 file1.c file2.c file3.c`

3.4. **Implement the patron management functions**

3.4.1. Implement the `void initPatronArray(PatronArrayType *arr)` function that initializes every field of the given patron collection to default values. The elements array must be dynamically allocated in this function, using a capacity provided as a constant in the base code; the number of patrons stored in a new array is always zero; and the starting point for unique patron identifiers can be found in a provided constant.

3.4.2. Implement the `void addPatron(PatronArrayType *arr, char *n)` function that adds a new patron to the back of the given patron collection. Because it consists of an array of structures (and not pointers), adding to the array means initializing the fields of the next element after the last initialized one. The new patron's name is set to the given `n` parameter, and its id is initialized from the next available id in the collection, so that unique ids are generated in a sequential manner.

3.4.3. Implement the `int findPatron(PatronArrayType *arr, int id, PatronType **p)` function that searches through the given patron collection to find the patron with the given id, and "returns" that patron using the `p` parameter. The function returns an error flag if the patron is not found, or a success flag otherwise.

3.4.4. Implement the `void printPatrons(PatronArrayType *arr)` function that prints out to the screen all the details of every patron in the given patron collection.

3.4.5. Implement the `void cleanupPatronArray(PatronArrayType *arr)` function that deallocates all the required dynamically allocated memory for the given patron collection.

3.5. **Implement the reservation management functions**

3.5.1. Implement the `void initResvList(ResvListType *list)` function that initializes both fields of the given `list` parameter to default values. Collections always start out empty, and the starting point for unique reservation identifiers can be found in a provided constant.

3.5.2. Implement the `void initResvTime(ResvTimeType **rt, int yr, int mth, int day, int hr, int min)` function that dynamically allocates memory for a `ResvTimeType` structure, initializes its fields to the given parameters, and "returns" this new structure using the `rt` parameter.

3.5.3. Implement the `void initResv(ResvType **r, PatronType *p, ResvTimeType *rt)` function that dynamically allocates memory for a `ResvType` structure, initializes its patron and reservation time fields to the given parameters, and "returns" this new structure using the `r` parameter. The reservation id must be set to a temporary value of zero until the new reservation is added to a reservation list in another function.

3.5.4. Implement the `int lessThan(ResvTimeType *r1, ResvTimeType *r2)` function that compares two reservation times, and returns the constant associated with true if the first reservation begins earlier in date and time than the second one, and the constant for false otherwise.

3.5.5. Implement the `void addResv(ResvListType *list, ResvType *r)` function that adds the reservation `r` *in its correct position* in the given `list`, so that the list remains ordered in ascending (increasing) order by reservation time. The linked list must be implemented *in accordance with the in-class coding examples*, with no dummy nodes.

This function must ensure that a unique identifier is generated automatically, in a sequentially increasing manner, and assigned to the new reservation.

3.5.6. Implement the `void printReservation(ResvType *r)` function that prints out all the details of the given reservation. The output must match the workshop video, and every field must be aligned with the other reservations. The output for the month, day, hours, and minutes fields must be padded with zeros, as shown in the video.

3.5.7. Implement the `void printReservations(ResvListType *list)` function that prints out the details of every reservation in the given list. Because the list is doubly linked, it must be printed out **twice**: once in the forward direction, using the `next` links, and again in the backward direction, using the `prev` links. This step is mandatory to demonstrate that the links are correctly initialized when reservations are added to the list.

3.5.8. Implement the `void cleanupResvList(ResvListType *list)` function that deallocates all the required dynamically allocated memory for the given list.

3.6. **Implement the restaurant management functions**

3.6.1. Implement the `void initRestaurant(RestaurantType *r, char *n)` function that initializes the three fields of the given restaurant structure. The name must initialized from the given parameter, and the initialization of the other fields must be performed by calling existing functions.

3.6.2. Implement the `int validateResvTime(int yr, int mth, int day, int hr, int min)` function that checks that the parameters represent a valid date and time. The year must be the current year or a future one, and the other parameters must be within valid ranges. The function returns an error flag if one or more of the fields is invalid, or a success flag otherwise.

3.6.3. Implement the `void createResv(RestaurantType *r, int pId, int yr, int mth, int day, int hr, int min)` function that creates a new reservation and adds it to the given restaurant, as follows:

(a) validate the date and time parameters

(b) find the patron with the given id in the restaurant

(c) if an error is encountered in either step above, a detailed error message must be printed out, and the reservation cannot be created

(d) create and initialize a new reservation time, using the given parameters

(e) create and initialize a new reservation, using the new reservation time and the found patron

(f) add the new reservation to the restaurant

3.6.4. Implement the `void printResByPatron(RestaurantType *r, int id)` function that prints out the restaurant name and the details of every reservation made by the patron with the given id.

3.6.5. Implement the `void cleanupRestaurant(RestaurantType *r)` function that cleans up all the dynamically allocated memory for the given restaurant `r`. This includes the patron collection, and the reservation list.

3.7. **Implement the main control flow**

Implement the `main()` function as follows:

3.7.1. Declare a local `RestaurantType` structure variable to store all the restaurant data in the program. You may allocate the restaurant either statically or dynamically.

3.7.2. Initialize the local `RestaurantType` structure by calling an existing function.

3.7.3. Load the provided data into the local restaurant structure, by calling two functions given in the base code.

3.7.4. 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.

3.7.5. The "print patrons" and "print reservations" functionality must both print out the restaurant name before calling an existing function.

3.7.6. The "print reservations by patron" functionality must prompt the user to enter a patron id before calling an existing function.

3.7.7. At the end of the program, the restaurant data must be cleaned up.

If any errors are encountered in the above, a detailed error message must be printed out to the end user. Existing functions must be reused everywhere possible.

3.8. **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.

3.9. **Package your files**

3.9.1. Your program must be separated into five (5) source files, as indicated in instruction 3.3.1.

3.9.2. 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

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

3.9.4. 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 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

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**

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*