Project Preparation Activity 3 [30 marks]

Watch Briefing Video! **Not An Autograded Evaluation**

You must submit all the source codes to the Git Repository before the deadline of lab 3.

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

About PPA 3

- Use the following link to accept the PPA3 invitation: https://classroom.github.com/a/RZhN7P8Y
- PPA 3 code submission is due on **Friday, November 10, 2023, at 11:59 pm.**
- PPA 3 in-person demonstration is due in your designated lab day after the code due date.

About Project Preparation Activities in General

- The non-autograded programming activities that contribute to the cumulative knowledge and programming experience required to complete the COMPENG 2SH4 course project.

- Evaluation Format
    - More instructions will be provided in the lecture(s).
    - 5-minute feature-based demonstration in your designated lab session after code submission due date.
    - Knowledge-based evaluation will be carried out at the in-person cross examinations on the designated dates (keep an eye on announcements).

Introduction to Random Number Generation

In PPA3, we will create our very first playable game using what we have learned in 2SH4 up to this point. Nonetheless, a game won't be entertaining without the flavour of unpredictability. As a result, let's learn how to generate random integer numbers in C to spice up our program.

1. Pseudo-Random Integer Generation Function – `int rand(void)`

Source Library `stdlib.h`

Functionality A table of pre-programmed non-sequential integers is stored under `stdlib`, with values ranging from 0 to RAND_MAX (typically > 32767). Each invocation of `rand()` returns the next integer in the table. Since these integers are non-sequential, the return value of `rand()` will *appear to be random*.

Limitation `rand()` returns the same entry in the pseudo-random number table for every program launch, because the initial read-out position from the table is always 0th. In other words, `rand()` is still predictable.

Usage:

```
int myRandomNumber = rand();
// Obtain a pseudo-random number from the table
```

2. Seeding Random Number Generation – `void srand(unsigned int num)`

Source Library `stdlib.h`

Functionality Alter the initial read-out position of `rand()` with the seed number `num`, so to make the pseudo-random number generation less predictable. This function returns nothing.

Usage:

```
srand(100);     // Seed the random number generator

int myRandomNumber = rand();
// Obtain a *better* pseudo-random number from the table
```

3. Pseudo-Randomized Seeding

With srand(), we still need to overcome another limitation – predictable seeding, thus predictable generated numbers. To add another layer of unpredictability, let's seed with the **current time**, using `time()` function from `time.h`. time() returns the number of seconds since January 1, 1970.

Improved pseudo-random number generation

```
srand(time(NULL));    // seed with time()

int myRandomNumber = rand();
// Obtain a much more random pseudo-random integer
```

You may repeatedly seed with different time stamps in your program to obtain an even more randomized pseudo-random number generation.

Lastly, if we were to generate a random integer between the range **(0, INTMAX]**, we can call `rand()` this way:

```
int myRandomNumber = rand() % INTMAX;
```

Think carefully about how the modulus operator delivers our desired range-limiting functionality.

Introduction to Basic Collision Detection

In PPA3, we will need to determine whether the player controllable object overlaps an item on the game board, then determine the corresponding actions when such collisions take place. The good news is that we've already laid down a very effective framework in PPA2 to simplify the collision detection.

Recall the object position structure defined in PPA2.

```
struct objPos
{
    int x;            // x-coordinate of an object
    int y;            // y-coordinate of an object
    char symbol;      // The ASCII symbol of the object to be drawn on the screen
};
```

Assuming that we now have two instances of this struct: player and item. Then, determining whether they have collided is as simple as checking their x- and y-coordinates are identical in every program loop ***after*** the player movement is completed.

```
if(item.x == player.x && item.y == player.y)
{
    // Collision  Happened!  Take some actions accordingly…
}
```

PPA3 requires you to keep track of minimally 5 randomly-generated items on the game board. You will have to think about a clever way to store the 5 instances of the collectable item objects and check for their collisions with the player object in every program loop iteration.

Project Preparation Activity 3 – Create a Scavenger Hunt "Guess the Word" Game

The work done in PPA2 has laid the groundwork for PPA3. We can now add a few interesting features to the existing PPA2 work and make a simple scavenger hunt game – refer to the sample executable.

The basic game mechanism is simple:

- The game has a built-in Goal String containing 15-25 ASCII characters, hidden away from the player.
- Upon startup
  - 5 random characters are picked from the Goal String and randomly placed on the game board.
  - The Mystery String consisting of unrevealed characters '?' is displayed at the bottom of the game board, indicating the number of characters to be collected / revealed to match the Goal String.
  - The Move Count is displayed also at the bottom of the game board, indicating the distance (number of game board blocks) the player covered on the game board since the start of the game.
- The player must control the player object to collect the ASCII characters on the board using WASD control.
  - For every collected character
    - Its occurrences will be revealed in the Mystery String; the uncollected characters will remain '?'.
    - The remaining items on the board are removed, and another 5 randomly selected characters from the Goal String will be generated and randomly placed on the board.
- The goal of the game is to consume minimal move count to win the game by revealing all characters in the Mystery String to match the Goal String.

The additional features added to your existing PPA2 work are:

- Deploy a constant Goal String (hidden),
- Deploy a Mystery String (shown) **instantiated on the heap.**
- Deploy a Move Count (OR, just the number of program loop iterations), instantiated anywhere.
- An array of 5 randomly generated collectable items, **instantiated on the heap.**
  - Use the pseudo-random integer generation method priorly discussed.
  - The generated items ***must NOT overlap*** the game board boundaries or the player object.
- Deploy the respective collection detection logic and character revelation logic.
  - When the player collides with an item on the board, reveal the corresponding ASCII character(s) in the Mystery String, if not revealed yet.
  - Then, remove the existing items on the board and generate another 5 items on the board.
- Deploy the game-end conditions.
  - When Mystery String is fully revealed and matches the Goal String, end the game, and show a winning message.
  - When an exit key is pressed, end the game without showing any winning message.
- Make sure to clean up everything on the heap before shutting down the program.
  - **Memory Leakage will be penalized.**

- **Advanced Features (Above and Beyond Activities)**
  - Make the Game More Challenging!
    - The generated ASCII character symbols are not limited to those in the Goal String, but to all the alphanumerical characters in the ASCII table.
    - Must guarantee that among 5 items generated, at least two of them are from the Goal String.

Use the sample executable **PPA3.exe** for deliverable references.

Required: Configure MacUILib Library and Makefile for your System

- Modify MacUILib.h as follows according to your OS:
  - **Windows**
    - Uncomment line 4 (`#define WINDOWS`)
    - Comment out line 5 (`#define POSIX`)
  - **Mac/Linux/Chrome**
    - Comment out line 4 (`#define WINDOWS`)
    - Uncomment line 5 (`#define POSIX`)
- Modify Makefile as follows according to your OS:
  - **Windows**
    - Comment out line 5 (`POSTLINKER=...`)
  - **Mac/Linux/Chrome**
    - Uncomment line 5 (`POSTLINKER=...`)

Marking Scheme

- Basic Features **[Total 25 marks]**
  - **[10 marks, breakdown provided]** Random non-repeating items generation algorithm implementation.
    - [6 marks] Non-repeating x-y coordinate generation within correct ranges.
    - [4 marks] Non-repeating character choices from Goal String.
  - **[2 marks]** Correct player-item collision detection
  - **[3 marks]** Correct Mystery String display, character revelation, and new item generation upon collision
  - **[2 marks]** Correct Move Count implementation and display
  - **[2 marks]** Correct End-Game implementation – Winning vs. Forced Exit.
  - **[6 marks]** No Memory Leakages. Item Bin and Mystery String created on Heap.

- Advanced Features **[Total 5 marks]**
  - **[1 mark]** Correctly choose only 2 characters from the Goal String
  - **[4 marks]** Correctly choosing 3 more characters from characters other than the Goal String.

Recommended Workflow

This is the last PPA before entering the 2SH4 project. While the number of lines of code to be added to complete PPA3 may not be more than 100 lines, it will take a series of critical thinking activities to complete the design. With the experience you've accumulated in the previous two PPAs, you should be able to complete most of the implementation without much coding guidance. The recommended workflow below will give you conceptual tips and pseudo-code algorithms to assist you in completing the implementation.

Remember, unless you are an experienced programmer, **DO NOT** attempt to develop this program in one shot. Always, **ALWAYS** make sure the current feature is working as intended before adding more features to prevent creating a debugging nightmare.

Get familiar with this workflow! Iterative design strategy is an absolute must as you progress on to higher level software courses. We will also incrementally reduce the number of hand-holding tips as we progress into later labs / project preparation activities under the assumption that you are more experienced with software development practices.

Pay Attention to Dynamic Memory Allocation! This is the first activity mandating you to place data on the heap for better memory management practices. You must remember to return every single piece of heap memory back to the computer when done with it!

Iteration 0 – Port PPA2 into PPA3 Skeleton, and Lab 3 Q1 into myStringLib

- Read through the PPA3 skeleton code and copy everything applicable from your PPA2 implementation into the correct sections. The PPA3 skeleton code contains sufficient [COPY AND PASTE] instructions for you to follow.
- **STOP!** After copying is done, compile and make sure that you can run PPA3 with identical features from PPA2 without any syntactic or semantic bugs.
- **DO NOT PROCEED UNTIL TESTED**

- Then, examine myStringLib.h; it contains the prototypes of the four string functions you've deployed and tested in Lab 3 Q1. Copy the implementations of all four functions into myStringLib.c in PPA3. This is our own String Library to be used in PPA3.
    - **This is how a functional C library is typically developed – prototyped, unit tested, then modularized.**

Iteration 1 – Implement Game Feature Framework

- Declare a constant string "McMaster-ECE" in the global scope. This is our **Goal String**.
- Declare a character array pointer in global scope, then allocate sufficient space in the heap to hold identical number of characters as the Goal String. This is our **Mystery String**, and it MUST be allocated on the heap as a requirement.
    - After allocation, populate all the characters in the Mystery String with '?', except the last character as NULL.
    - Think about in which routine should the allocation call take place.
- Declare a struct objPos pointer in global scope, then allocate sufficient space in the heap to hold 5 struct objPos instances in an array. This is our **Item Bin**, and it MUST be allocated on the heap as a requirement.
    - Again, think about what to initialize for all the elements in the item bin, and in which routine should the allocation and initialization call be done.
    - Choose an alphanumerical character to initialize the items, and some predictable x-y coordinates for debug simplicity.
- Declare a moveCount integer in global scope to track the number of program loop iterations since the program start.
- **IMPORTANT!** After deploying these variables, go to the CleanUp() routine and make sure all heap-allocated variables are freed. You will receive a penalty with memory leak!

- Once done deploying these features, proceed to display the Mystery String, the Move Count, and the five items from the Item Bin on the game board.
    - Think about what to modify in the Main Logic and the Draw routines to achieve these features.

- **STOP!** DO NOT PROCEED until you can correctly draw these features on the display.
- **STOP!** DO NOT PROCEED until your memory profiler tells you the program has zero memory leak.

- **Recommended Validation Setup**
  - Goal String `"McMaster-ECE"`
  - Mystery String `"???????????"`
  - Move Count `0`, update every program loop iteration
  - Player `{10, 5, '@'}`
  - Item Bin `{1, 1, 'A'}, {2, 2, 'B'}, {3, 3, 'C'}, {4, 4, 'D'}, {5, 5, 'E'}`

  - Expected Display Contents:

```
####################
#A                   #
# B                  #
#  C                 #
#   D                #
#    E    @          #
#                    #
#                    #
#                    #
####################
Mystery String: ???????????
Move Count: 0
```

Iteration 2 – Random Item Position Generation

With all the static features deployed in the previous iteration, let's first deploy the random item position generation logic. In PPA3 skeleton code, the signature of the Random Item Generation function has already been prototyped as:

```
void GenerateItems(struct objPos list[], const int listSize, const struct objPos *playerObj \
                   const int xRange, const int yRange, const char* str)
```

where

- `list[]` The pointer to the Item Bin. All randomly generated items should be placed in here.
- `listSize` The size of the Item Bin. Default is 5 based on PPA3 specification.
- `playerObj` The pointer to the player struct so to make sure the randomly generated items do not fall on the player position.
- `xRange` The exclusive upper bound of the randomly generated integer x-coordinate
- `yRange` The exclusive upper bound of the randomly generated integer y-coordinate
- `str` The Goal String from which the 5 random characters should be selected for the 5 random items in the Item Bin.

This function should be called whenever we need to generate 5 new random items on the gameboard.

Based on the principle of incremental engineering, we will tackle this function implementation in two phases: a) Random non-repeating x-y coordinate generation within the game board range, and b) Random non-repeating character choices from the Goal String. Iteration 2 deals with phase a).

You are not restricted to the recommended algorithms to be introduced below. You can use any algorithms you have come up with as long as they meet the PPA3 software specifications.

STOP! Before engaging in implementation, change the initial coordinates of the items in the Item Bin to somewhere outside the game board. Think about why this is necessary.

- **General Algorithm for Non-Repeating x-y Coordinate Generation**
  - Generate a candidate integer x-coordinate value in the range of [0, xRange-1].
    (Think about whether this range is correct! If not, you are responsible to make it right)
  - Generate a candidate integer y-coordinate value in the range of [0, yRange-1]. Again, think about its correctness.
  - With the candidate x-y coordinate, check whether it matches 1) The player location, and 2) The location of any previously generated items in the Item Bin within this instance of the function call.
    - If yes, discard this candidate and proceed to generate another x-y candidate.
    - Otherwise, write this x-y coordinate to the target item in the Item Bin, and move on to the next item.
  - The algorithm stops when all items in the list[] have received their newly generated random x-y coordinates.

  - As a good programming practice, while developing this algorithm, print out the following items as the debugging message for tracking the implementation correctness. You are encouraged to use the debugger whenever required.
    - Player Location
    - Locations of All Items in the Item Bin
    - Newly Generated x-y Coordinate Candidate
    - Recall from Lecture – See It before Believing It!

  - **Gotcha** – if C compiler is complaining about not being able to use variables (xRange, yRange) as array dimensions to create arrays, you can use the global preprocessor constants rather than the local input parameters. Not the most correct method but serves the purpose well.
  - **STOP!** Make sure your random coordinate generation is working well before moving on.

- Once the algorithm is correctly deployed, replace the manual initialization for the Item Bin with this function, so that the program can place the initial set of items on randomly generated coordinates upon program startup.

- For example, once the algorithm is developed correctly and invoked upon startup to initialize the items in the Item Bin, the initial output on the screen might look like the sample below, and the five items will appear in different non-repeating locations every time you launch the program.

```
####################
#                  #
#        A         #
#                  #
#         E        #
#         @    D   #
#              C   #
#                  #
#   B              #
####################
Mystery String: ???????????
Move Count: 0
```

- **STOP!** You should make sure the random non-repeating coordinate generation is correctly implemented before moving on to the next iteration.
  - We recommend that you keep the debugging message related to random item generation until the end of the next iteration.

Iteration 3 – Random Character Choices from Goal String

Next, we need to add the character choice algorithm into the GenerateItems function. This algorithm may be done independently from the x-y coordinate generation (think about why).

- **General Algorithm for Non-Repeating Character Choice from Reference String (Goal String)**
    - Generate a random integer index in an appropriate range such that all possible indices have a valid corresponding character in the Goal String ***excluding*** the NULL termination character. You may need to use my_strlen() from your own string library for this purpose.
    - With the newly generated index, check whether the ASCII character at this index in the Goal String matches the character of any previously generated item within this instance of the function call.
        - If yes, discard this candidate and proceed to generate another index.
        - Otherwise, write this ASCII character to the symbol member of the current item, and move on to the next item in the bin.
    - The algorithm stops when all items in the list[] have received their newly generated random choices of non-repeating characters from the Goal String.
    - Again, you should print out all the relevant debugging messages to ensure your algorithm implementation works as expected.

    - **STOP!** Make sure your random character choice implementation is working well before moving on.

- Once the algorithm is correctly deployed, the function should generate five items at random locations on the game board with randomly chosen characters from the Goal String.

- Sample Output that will vary upon every program startup. Notice all five characters are from the sample Goal String "McMaster-ECE".

```
####################
#                  #
#       M          #
#                  #
#        -         #
#        @    a    #
#             c    #
#                  #
#   E              #
####################
Mystery String: ???????????
Move Count: 0
```

- **STOP!** You should make sure the random non-repeating item generation is correctly implemented before moving on to the next iteration.
    - Once done, you may remove the debugging messages related to random item generation.

Iteration 4 – Item Collection, Character Revelation, Move Count Update, and Game-End Condition

This iteration is not difficult but consists of many small features to implement. Tips are provided below. Once completed, you will have a working PPA3 scavenger hunt game that can be refined in additional iterations, should you choose to perfect the game experience or shoot for above-and-beyond features. The recommended workflow will only support you up to this point.

- **Item Collection Implementation**
    - Read the Collision Detection section.
    - When player object collides with any one of the items in the Item Bin
        - Check whether the symbol character under the item has been collected. If not, reveal all instances of this character in the correct indices in the Mystery String. Otherwise, do not do anything.
        - Regardless of whether characters were revealed in Mystery String, always regenerate all 5 items in the Item Bin at the end of a detected collision.

- **Character Revelation**
  - To reveal the corresponding characters in the Mystery String at the correct locations, iterate through the Goal String and find all the indices of their appearance.

- **Move Count Update**
  - Really just counting the number of iterations the program loop has run.
  - You may want to stop the count when the player FSM is in STOP state. Think about why.

- **Game-End Condition**
  - At the end of every detected collision, check whether the contents of the Mystery String are identical to that in the Goal String using your own my_strcmp() function.
  - If yes, end the game with a Winning Message.
  - You may want to set up something to differentiate a winning end-game from the command-forced end-game.

- **DO NOT FORGET!!** Incremental engineering, see it before believing it, and use all the debugging techniques to ensure your program development is under control.

- **DO NOT FORGET!!** Check for memory leakages at the end of each development iteration.

Additional Iterations – Advanced Features (Above and Beyond Activities)

- **Only conceptual hints will be provided for above-and-beyond activities.**
  You need to apply the knowledge and skills from the previous design iterations to come up with additional features.

- **Above-and-Beyond Feature – Generating Randomly Selected Characters from not just the Goal String**
  - First, choose two characters from the Goal String using the non-repeating choosing algorithm in Iteration 4.
  - Then, choose the remaining three characters from all possible alphanumerical and punctuation characters and using again the non-repeating choosing algorithm. A possible ASCII character range might be (33, 126).
    - To prevent confusing playing experience, you should avoid using the player object symbol, boundary symbol, and space character.
  - If interested, you may even look into the bit vector method for non-repeating element generation. We won't cover this until possibly in 2SI3.