Project Preparation Activity 1 [30 marks]

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

You must submit all the source codes to the Git Repository by the PPA1 deadline.

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

About PPA 1

- Use the following link to accept the PPA 1 invitation: https://classroom.github.com/a/ge6s4NTw
- PPA 1 code submission is due on **Sunday, September 29, 2024, by 11:59 pm** on Github.
- PPA 1 in-person interview and demonstration is due in the lab session immediately 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 lectures.
    - 5 to 10-minute feature-based demonstration and interview in your designated lab session.
    - Knowledge-based evaluation will be carried out at the in-person cross examinations on the designated dates.

Introduction to Program Loop Setup

All persisting programs would have minimally one master program loop performing the following minimal operations as illustrated in the flow chart. This persisting loop may be implemented by you explicitly (e.g., ARM embedded firmware in COMPENG 2DX3), or hidden away from you in the compilation background (e.g., Arduino C, Visual Studio application design template, Unity / Unreal game engines, etc.).

In the COMPENG 2SH4 project, you will learn to build a simple console screen game from scratch, thus you will receive the skeleton code from the GitHub repository implemented from the flow chart below. You will have to know how the skeleton code behaves to correctly implement all the project logics.

Initialize
The initialization routine run once before entering the program loop. It typically contains the parameter and memory initialization, and library initialization calls.

Loop with Exit Flag
Known as the program loop, it ensures the program repeatedly executes the intended logic without shutting down unless the exitFlag is set by the program.

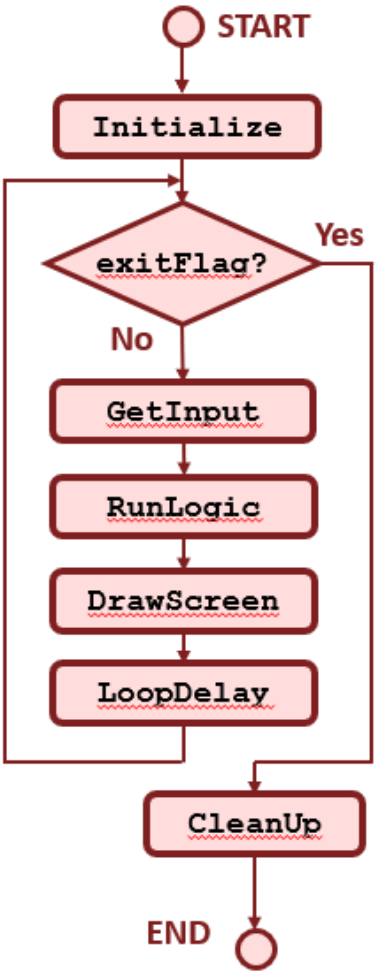
GetInput
This routine should contain all input collection logics. The remaining program logic will make its computational decision based on the presence of the input.

RunLogic
This routine contains all the computational logics. Typically, the logic uses both the current user input (or lack thereof) and the current program status to produce a state-specific output. This computational output is then drawn on the screen.

DrawScreen
This routine contains the screen drawing logics. It typically clears the screen, then uses the RunLogic output to draw a set of new contents (aka. A frame) on the screen.

LoopDelay
This routine typically contains a delay function to ensure the program loop repeats at a determined rate, which will roughly set the baseline program frame rate.

CleanUp
This routine is called only once after the program loop is broken, and right before the program shuts down. It typically contains memory deallocation logics, library de-initialization calls, and possibly an exit message.



Introduction to McMaster UI Library

This is the first custom user interface (UI) C library documentation you will need to get familiar with the COMPENG 2SH4 project. The McMaster UI Library (MacUILib) is a cross-platform API (application programming interface) for asynchronous console screen UI, allowing you to control the behaviour of your program like using your controller to control a video game on an old-school computer terminal screen. In the following project preparation activities, you will have to use the API functions in the MacUILib to deliver the required functionalities. The description of functions in this library follows.

```
void MacUILib_init(void);
```

Description

McMaster UI Library initialization sequence for **asynchronous input**. It must be called at the beginning of the program to make sure the UI library is set up for your program.

Input Parameter: None
Output: None

```
void MacUILib_init_sync(void);
```

Description

The **synchronous input** version of McMaster UI Library initialization sequence. Call this instead of the asynchronous version in debugging stage, so you can examine the program behaviour in a controlled manner.

** IMPORTANT ** Synchronous Input mode works slightly differently on Windows system vs Mac/Linux.

Windows – Use scanf() for input collection. You need to press ENTER after every key input.
Mac/Linux – Use MacUILib_getChar() for input collection. You do not need to press ENTER after every key input.

Input Parameter: None
Output: None

```
void MacUILib_Delay(int usec);
```

Description

When invoking this function, it will arbitrarily delay your program for the number of microseconds as specified in the input parameter `usec`. Use this to slow down your program logic whenever required.

Input Parameter:

`usec` (integer type)	Specifies the number of microseconds to delay the program execution. The maximum amount of delay you can achieve is 999,999 usec on most systems.

Output: None

```
int MacUILib_hasChar(void);
```

Description

Asynchronous check of whether a keyboard press was detected. It will return 1 / non-zero value (true) if a keypress was detected, and 0 otherwise. Use this to determine whether your program needs to process an input from user.

Input Parameter: None
Output: Integer type, non-zero indicating a keypress was detected, 0 otherwise.

```
char MacUILib_getChar(void);
```

Description

Call this function to obtain the most recently pressed keyboard key. You should always call MacUILib_hasChar() to check whether a keypress is detected before calling this function to process the target keypress.

Input Parameter: None
Output: Character type, the ASCII character corresponding to the most recently detected keypress.

```
void MacUILib_clearScreen(void);
```

Description

Call this function to clear the console screen in preparation for redrawing contents on the screen.

Input Parameter: None
Output: None

```
void MacUILib_uninit(void);
```

Description

Must call this function right before exiting the program to properly shut down the McMaster UI Library.
It will print the message “Press Any Key to Shut Down”, and will exit the program after pressing any key on the keyboard.

Input Parameter: None
Output: None

```
int MacUILib_printf(const char* str, ...);
```

Description

The OS-independent version of printf(). Use it for all the screen drawing operations.
*** Its behaviour is identical to printf() from <stdio.h> ****

Input Parameter:

`const char* str`	The string with conversion specifiers.
…	The parameters to be filled into the conversion specifiers in the string.

Output:

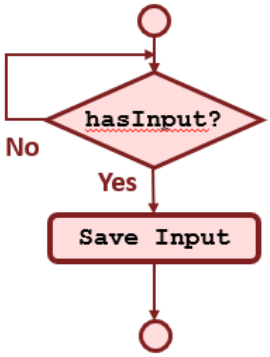
The total number of characters printed on the terminal screen.

Project Preparation Activity 1 – Console Screen Animation and Asynchronous Input

All the programming activities we've done up to this point are static – the program waits indefinitely for an input command before taking an action. This behaviour does not make an interesting and interactive program at all. The nature of synchronous console input makes your program one-movement-per-command. As a result, any games built on top of this synchronous blocking UI will only be as good as a text-based RPG.

Definition: Synchronous Input

- The program halts at the function call for collecting input.
  (e.g. `scanf()`, `getchar()`, etc)
  The program will wait for input indefinitely, and only progress onward after receiving an input.
- Also known as ***blocking*** input mode because synchronous input mode BLOCKS program progression.
- This mode is required when the subsequent program execution behaviour depends on the user input.



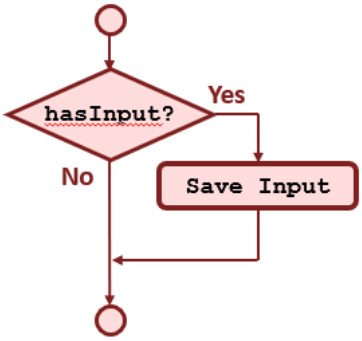
Real-Life Example

A vending machine won't know what to vend until you press a key to make a selection.
Therefore, a vending machine must operate in synchronous Input mode - one action per command.

The key to making a more interactive, animated program (whether a game or not), is the asynchronicity of program input – instead of blocking the program and waiting for an input, a program shall continue with its course of action if no input is received, and only changes behaviour when an input is observed.

Definition: Asynchronous Input

- The program moves on to the next execution sequence if it sees no input to process. It will never stop and wait for input.
- Also known as **non-blocking** input mode because asynchronous input mode NEVER BLOCKS program progression.
- This mode is required when the program progression must remain independent of the presence of input.
  Inputs only change the course of program progression.



Real-Life Example

A music player app in repeating mode will continue playing music in the playlist even if you left it unattended.
Therefore, the music app in this case operates in asynchronous input mode - execution continues without commands; commands only change program behaviour.

In this project activity, you are to build a **Marquee Display Banner** with the following features:

- Upon startup, the program rolls a string of "`>><<`" across a 20x1 (Columns x Rows) marquee display area from right to left.
- When the display contents reach the leftmost display boundary, it must wrap around character-by-character to the rightmost boundary.
- For every alphabet / number pressed on the keyboard, it must be immediately inserted into the string between the "`>>`" (header) and "`<<`" (footer) symbols, and the newly entered character must roll along across the display with the existing contents between the header and footer.
- All subsequent inserted contents must be cascaded to the existing contents between the header and footer in the left-to-right sequential order.

- You may need to set an upper limit to the number of characters that can be inserted to the display string to prevent overwhelming the display area with too many characters. The header “>>” and footer “<<” must always be displayed.
- You may need to set up a key to shut down the program.
- Advanced Features (Above and Beyond Activities)
    - Implement a key that can toggle the rolling direction of the string movement.
    - Implement a marquee display that can roll a string of more than 20 characters across the 20x1 display window. Think about how.

Use the sample executable **PPA1.exe** for basic deliverable references. This sample executable comes with the PPA1 template code. No sample executables will be provided for above-and-beyond activities.

Note: MacUILib Library and Makefile no longer requires customization for your Operating System.

Marking Scheme

- **Basic Features [Total 25 marks]**
    - **[3 marks]** Correct initialization sequences to start up the program
    - **[4 marks]** Correct asynchronous input setup
    - **[2 marks]** Correct implementation of the static portion of the marquee display – Line 1, 2, and 4
    - Correct implementation of the Marquee Display – Line 3
        - **[2 marks]** Right-to-left rotation
        - **[4 marks]** Correct wraparound logic
        - **[3 marks]** Correct alphanumerical and punctuation character insertion to the string
        - **[2 marks]** No other ASCII characters allowed to be inserted to the string
        - **[2 marks]** Capping the total string size to avoid out-of-bound access
    - **[2 marks]** Correct implementation of non-alphanumerical shutdown command
    - **[1 mark]** Good choice of delay constant
- **Advanced Features – Complete either one [Total 5 marks]**
    - Correct implementation of a non-alphanumerical command to toggle the rolling direction of the marquee display
    - Correct implementation of a marquee display with 20-character window but a string size larger than 20 characters.

Recommended Workflow

Since this may be your first time working on developing a standalone animated program with asynchronous input mode, we have provided a recommended multi-iteration workflow to help you get familiar with developing a raw program from scratch.

Unless you are an experienced programmer, **DO NOT** attempt to develop this program in one shot – this is an extremely poor software development practice.

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.

Iteration 1 – Get Familiar with the Skeleton Code

- With the program loop flowchart at hand, read the skeleton code in PPA1.c to understand how each function fits into the flowchart. Lots of comments are provided in the skeleton code to help you learn the skeleton code.
- If you are wondering why the skeleton code is structured this way, it's for better code organization for sustainable development.
- To validate your understandings, here is what you should do before progressing onward:
    - In the Global Variable section, set up three integer variables – `a, b, result`.
    - Initialize these variables `{a = 3, b = 5, result = 0}` in the suitable routine
      Hint: …maybe `Initialize()`?
    - Compute the sum of `a` and `b`, and store it in `result`. Put this computation logic in the suitable routine.
      Hint: …maybe in `RunLogic()`?
    - Print the result using `MacUILib_printf()` in the suitable routine.
      Hint: …maybe `DrawScreen()`?
    - Compile and run the program (**`make`**, then **`./PPA`** or **`./PPA.exe`**). You should see the result being printed indefinitely all over the screen at an extremely fast rate. Think about why. Press **Ctrl+C** or **command+C** to terminate the program.
    - Time to get things under control. Try these:
        - Add some delay to the program loop. Use `MacUILib_Delay()` to add 0.1 second (100,000 microseconds) of delay per loop, and call this function in the suitable routine.
          Hint: …maybe `LoopDelay()`?
        - Compile and run the program again. You should now see the repeated contents being printed at a rate of 0.1 second. Again, think about why before you terminate the program.
        - Next, add `MacUILib_clearScreen()` in two places
            1) Right before you start the program loop, and
            2) Right before you print the results on the screen.
        - Compile and run the program again. You should now see only one line of results staying on the top left of the terminal. Again, think about why before you terminate the program.
    - Lastly, let's make things animated a bit.
        - Increment the value of the variable `a` by one for every program loop iteration.
        - Compile and run the program again. The calculation results will change every 0.1 second according to the new value in a. Think about why this works before terminating the program.

- The above work is typically referred to as "Bring-Up" works. When using a new tool set / IDE, an experienced programmer will always spare a minute do this to get familiar with the environment before committing to the actual work.

Iteration 2 – Collect Input Asynchronously

- We now need to get familiar with asynchronous input collection. The general algorithm is as follows:
  - Check whether there is an input – using `MacUILib_hasChar()`
  - If there is an input, acquire it using `MacUILib_getChar()` and save it in the program for further processing
- The first asynchronous input we need to implement is the shutdown command, so we can gracefully terminate the program without resorting to special terminal commands.
  - In the global variable section, create a character variable – `cmd`
  - In the appropriate routine, check whether there is an input for process. If yes, store it in `cmd`. Think about which routine should contain this logic. **Hint**: maybe in `GetInput`()?
  - Add a command-processing logic to the appropriate routine as follows:
    - If the input command is the character `'x'`, set the `exitFlag` to true, so the program will break out of the main program loop in the next loop iteration.
  - Compile and run the program. You can now press x to gracefully end the program. You can change the exit command to any ASCII character based on your project design.

- We have now completed the skeleton code setup for our project preparation activity. You should be familiar at this point:
  - How to decompose your desired program actions into corresponding routines in the program loop.
  - How to control the printed terminal contents and its refreshing rate.
  - How to invoke the asynchronous input to change the course of your program.

Iteration 3 – Print Static Contents

- Let's now get to the real work. **Delete the contents from Iteration 1 ONLY**. Keep the input collection and the exit command logic in your program.
- Then, do the following to set up the initial marquee display contents:
  - Create a string of 20 characters + 1 character for null termination (ASCII character integer value **0**) in global scope. This will be the string containing the marquee display content, or **Display String**.
    - In C, this is done by declaring a character array of 21 characters.
    - As a good design practice, you should #define a string size constant of 21, then declare the character array using this constant. You will thank yourself in later development stage.
    - Make sure to initialize all the elements in the character array to `"*COMPENG 2SH4 PPA1* "`

    In short, you can do so by:

    ```
    char displayString[21] = "*COMPENG 2SH4 PPA1* ";
    // for a 20-character string, you need 21 character spaces.
    // We will reveal why soon.
    ```

  - In the draw stage of the program, clear the screen before drawing the contents, then print on the screen:
    - Line 1: McMaster Marquee Display
    - Line 2: `====================` (20 equal signs)
    - Line 3: [PRINT THE **Display String** CONTENTS HERE]
    - Line 4: `====================` (20 equal signs)

    - For printing strings, use **`MacUILib_printf("%s", displayString);`**

  - Compile and run the program. Make sure you are satisfied with the result before moving on.

Iteration 4 – Roll Contents Across the Marquee Display

- **This section requires you to do some algorithm design and refinement yourself.**
  Only the general algorithm will be provided for your experiment.

- Animation happens with static pictures being swapped at a fast rate. We now have all the elements to achieve our very first screen animation program. Our program prints contents and refreshes the display at 0.1s rate. All we need to do now is to update Line 3 contents (mentioned in Iteration 3) for every new draw on the screen – typically referred to as a ***"frame"***.
- The following algorithm will ensure to print the string contents in every frame in a shifted way, thus giving you an illusion that the string is rolling across the screen.
  - Concept: Given a string `char str[4] = {a, b, c, d}`, printing character-by-character on screen, then:
    - If printed from index 0 printed contents will be "`abcd`"
    - If printed from index 1 printed contents will be "`bcda`"
    - If printed from index 2 printed contents will be "`cdab`"
    - If printed from index 3 printed contents will be "`dabc`"
    - The above rolling action will only work if we keep the index cycling between 0-3. This is done by deploying the wraparound calculation – when access index = size of the string, index access should wraparound to 0.
    - Then, if we update the starting index in every program loop iteration, and print using wraparound algorithm, our marquee display will start running!
  - Given the above concept, you should do the following:
    - Declare another integer global variable to track the printing start position.
    - Initialize starting position to zero.
    - In program logic, increment the starting position for every iteration. If the updated position exceeds the display string size, wrap it around to zero.
    - In drawing stage, print line 1, 2, and 4 as is. For line 3:
      - Print character-by-character from the current starting position for 20 characters.
        (**Hint**: maybe using a loop?)
      - When the access index is beyond the string size, wrap it around – again, think about how you can do it. There is a very handy arithmetic operator introduced in the class that can elegantly solve this problem.

  - Compile and run the program. Do not feel surprised if the rolling marquee display is buggy.
    - Debugging Tips – Code Walkthrough
      - Carefully walk through your code and track how indices are updated in every iteration.
      - Pay particular attention to the wraparound algorithm. Do the calculation on paper to make sure your algorithm actually makes sense.
    - Debugging Tips – Debugging Message
      - To validate your on-paper calculations, you can optionally print out the values of the critical variables (for example, printing start position) after line 4.
      - This is a common quick-and-dirty debugging technique used in the software industry. Not the best practice but serves the purpose here.

  - We will introduce a more formal debugging process later in the course. For now, the above tips serve the purpose.

  - Once you have gotten the message correctly rolling across the marquee display, you are ready to move on to the next iteration.

Iteration 5 – Modifying String Contents on the Fly

- **This section requires you to do some algorithm design and refinement yourself.** **DO NOT use string.h library.**
  Only the general algorithm will be provided for your experiment.

- In order to dynamically add characters to the marquee display string, we now have to make some modifications to the code in the previous iteration.
  - For the display string
    - Initialize all characters to `NULL` character (ASCII value 0) in the declaration stage (optional)
    - In the initialization stage, initialize the first four characters of the string to "`>><<`", and the remaining characters to `' '` (space character)
  - Then, modify the main logic with the following new features.
    - Change the exit command to another non-alphanumerical ASCII character (not alphabets or numbers).
    - Whenever an alphanumerical character is encountered, add it to the display string. The insertion position must be after the last character and before the footer "`<<`".
      Think about how you can do this without using string library.
    - You will need to set a cap on the maximum number of characters allowed in the string, so you don't run into out-of-bound access problem. (**Hint**: use the String Size preprocessor constant)
    - After processing the command, you probably need to reset the command.
      Think about why and how.
  - Compile and run the program. Use the aforementioned debugging techniques to fine-tune your program behaviour until your program works as intended.

  - Congratulations! You have completed the first standalone animated program from scratch.

Additional Iterations – Advanced Features (Above and Beyond Activities – You only need to do one to receive marks though you are welcome to try both)

- **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 these additional features.

- **Above-and-Beyond Feature 1 – Toggle Rotation Direction**
  - Think about how you can rotate the string in the opposite direction (left to right).
  - **Watch Out!** The wraparound condition is different for left-to-right movement.
  - You may want to set up an integer constant for direction control and use a non-alphanumerical ASCII input character to toggle the direction.

- **Above-and-Beyond Feature 2 – Roll Longer Strings Across the Display**
  - This requires you to isolate the display string size (>20 characters) with the display window size (20 characters)
  - When operating on the string itself in the main logic, it's the display string size that matters.
  - When drawing on line 3, both sizes matter. Think carefully which size constant you need to use for every action you want to implement.