# Assignment Report: The Greatest Adventurer

**Filbert Hamijoyo - 122040012**

## 1 Introduction

In this assignment, I developed a console-based game using the C programming language and threads (`pthread`) for multithreading. The game's objective is for the player to navigate through a map, collect all the gold shards ($), and avoid colliding with moving walls (`=`). The player is represented by the character `0` and can move in four directions using the keyboard. The game runs in a terminal environment and demonstrates the use of multithreading, synchronization, and real-time input handling.

## 2 Design

The program's design focuses on modularity, concurrency, and efficient resource management. Below is an overview of the key design components:

### 2.1 Data Structures

#### 2.1.1 Wall Structure

Defined by the `Wall` struct, each wall has properties:

- **rows**: The row position on the map.
- **columns**: The starting column position.
- **directions**: Movement direction (1 for right, -1 for left).

#### 2.1.2 GoldShard Structure

Defined by the `GoldShard` struct, each shard has properties:

- **rows**: The row position on the map.
- **columns**: The column position.
- **directions**: Movement direction.
- **isthere**: Status indicating if the shard is still available.

### 2.2 Global Variables

- **Map Representation**: A 2D character array `map[ROW][COLUMN + 1]` represents the game map.
- **Player Position**: Variables `player_x` and `player_y` track the player's position.
- **Game State**: Variable `game` controls the main game loop.
- **Mutex**: `pthread_mutex_t map_mutex` ensures synchronized access to shared resources.
- **Walls and Shards Arrays**: Arrays `walls[WALLS]` and `shards[SHARDS]` store wall and shard objects.

### 2.3 Multithreading

The program uses three threads to handle different aspects concurrently:

#### 2.3.1 Input Thread (`handle_input`)

Manages real-time user input without blocking the main thread. It captures keyboard events to move the player or quit the game. Synchronization is achieved using `pthread_mutex_lock` and `pthread_mutex_unlock` when updating shared resources.

#### 2.3.2 Walls Movement Thread (`walls_movement`)

Moves walls horizontally across the map. Walls reverse direction upon hitting map boundaries. Collision detection with the player results in game termination.

#### 2.3.3 Gold Shards Movement Thread (`goldshards_movement`)

Moves gold shards horizontally. When the player collects a shard, it's marked as collected. Collecting all shards results in winning the game.

### 2.4 Synchronization

- **Mutex Locks**: Used to prevent race conditions when threads access or modify shared data (e.g., the game map and player position).
- **Thread Safety**: All shared resources are accessed within mutex-protected sections to ensure data integrity.

### 2.5 Map Initialization and Display

- **Map Borders**: The map is initialized with borders using `HORI_LINE`, `VERT_LINE`, and `CORNER` characters.
- **Dynamic Updates**: The main thread refreshes the map display in real-time, reflecting the current positions of the player, walls, and shards.

### 2.6 Collision Detection

- **Walls and Player**: If a wall moves into the player's position, the game ends with a loss message.
- **Shards and Player**: If the player moves into a shard's position, the shard is collected. Collecting all shards ends the game with a win message.

## 3 Environment and Execution

### 3.1 Running Environment

- **Linux Version**: Ubuntu 22.04.5 LTS
- **Linux Kernel Version**: 5.10.195
- **GCC Version**: gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0

### 3.2 Compilation and Execution

1. **Compilation**:

   ```
   g++ hw2.cpp -lpthread
   ```

   Ensure that the `pthread` library is linked during compilation.

2. **Execution**:

   ```
   ./a.out
   ```

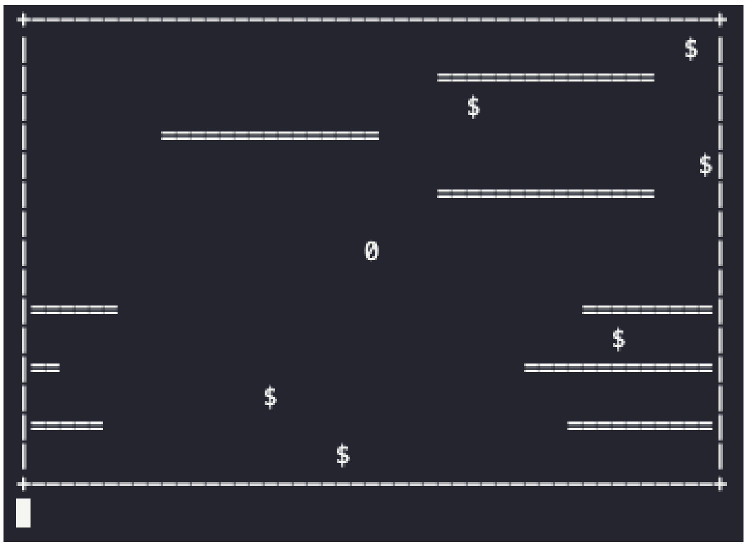
### 3.3 Controls

- **Move Up**: Press `W` or `w`.
- **Move Down**: Press `S` or `s`.
- **Move Left**: Press `A` or `a`.
- **Move Right**: Press `D` or `d`.
- **Quit Game**: Press `Q` or `q`.

### 3.4 Demonstration of Proper Execution

Upon running the program:

- The game map displays with the player character `O` at the center.
- Walls (`=`) and gold shards (`$`) start moving horizontally randomly either to the right or to the left.



```
+-----------------------------------------------+
|                                             $ |
|                            ===============    |
|                              $                |
|         ===============                       |
|                                              $|
|                            ===============    |
|                                               |
|                       0                       |
|                                               |
|======                                =========|
|                                        $      |
|==                                =============|
|                $                              |
|=====                                ==========|
|                     $                         |
+-----------------------------------------------+

```

Figure 1: Game with obstacles and player.

- The player can move around using the `W`, `A`, `S`, `D` keys.
- Collecting all gold shards results in a victory message:

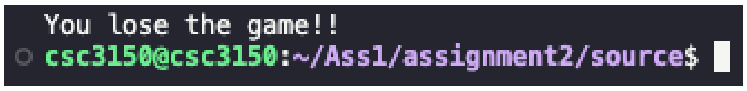
  `You win the game!!`

```
You win the game!!
csc3150@csc3150:~/Ass1/assignment2/source$
```

Figure 2: Winning the game.

- Colliding with a wall results in a loss message:

  `You lose the game!!`

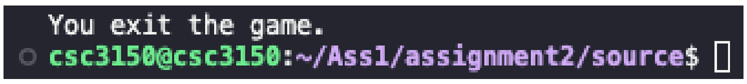


```
 You lose the game!!
○ csc3150@csc3150:~/Ass1/assignment2/source$ █
```

Figure 3: Losing the game.

- Exiting the game manually displays:

  `You exit the game.`



```
 You exit the game.
○ csc3150@csc3150:~/Ass1/assignment2/source$ ▯
```

Figure 4: Exiting the game.

The game's real-time updates and responsiveness demonstrate successful multithreading and synchronization.

## 4 Conclusion

This assignment provided me with hands-on experience in multithreading and synchronization in C using threads. By developing a real-time, interactive game, I learned how to manage concurrent threads, protect shared resources with mutexes, and handle asynchronous user input. The project reinforced the importance of thread safety and proper synchronization mechanisms to prevent race conditions. Overall, this assignment enhanced my understanding of concurrent programming concepts and their practical applications in software development.