CSL 210

OBJECT ORIENTED PROGRAMMING LAB

FINAL PROJECT REPORT

A 2D CONSOLE BASED MAZE GAME

SUBMITTED BY:

UMM-E-HABIBA IMRAN (048)

ZAINAB IDREES (051)

SUBMITTED TO:

SIR MUHAMMAD SIDDIQUE

MA'AM SAIMA JAWAD

INTRODUCTION

OVERVIEW OF THE PROJECT

The project aims to create an interactive and engaging 2D maze game. It addresses the challenge of combining logical problem-solving with entertainment and provides an innovative solution that enhances user engagement through dynamic gameplay. Developed using C++ and leveraging ANSI escape codes for graphical enhancements, the project is designed to offer a visually appealing and intellectually stimulating experience.

OBJECTIVES

The primary objectives of this project include:

- Developing a scalable and efficient maze-solving game with multiple difficulty levels.
- Enhancing user experience through visually enriched ASCII-based graphics and real-time feedback.
- Promoting logical thinking and strategic planning by challenging users with increasingly complex mazes.
- Ensuring robust error handling and cross-platform compatibility for a seamless gaming experience.

SCOPE

- Real-time navigation within dynamically loaded mazes.
- User-friendly interfaces with color-coded maze elements for enhanced visual clarity.
- Secure and efficient memory management to ensure smooth operation.
- Support for multiple difficulty levels: Easy, Medium, and Hard.

KEY COMPONENTS

GAME INITIALIZATION

- The maze is read from a file based on difficulty (easy.txt, medium.txt, hard.txt).
- Different characters represent the maze:
  - **P (Player):** Initial position.
  - **# (Walls):** Impassable obstacles.
  - **. (Pathways):** Movable spaces.
  - **E (Goal):** The endpoint to reach.
- A dynamically allocated 2D array (char**) stores the maze.
- The player's starting position (startX, startY) is set upon reading the maze file.

GAME CLASSES

The project targets casual gamers, students, and logic puzzle enthusiasts and is limited to text-based graphical environments. It will support single-player gameplay on systems capable of running C++ console applications. The following are the main classes included in our project

ABSTRACT BASE CLASS GAME

- Encapsulates player coordinates (x, y).
- Contains virtual methods like move to be overridden by subclasses.

DERIVED CLASS PLAYER

- Implements the move method to handle movement (w/a/s/d for directions).
- Provides methods like resetPosition for resetting the player's position after invalid moves.

CLASS MAZE

Handles:

- Maze loading from files.
- Validation of moves (validMove method).
- Interaction between player and maze.

Includes:

- movePlayer: Manages movement and validates moves.
- isGameComplete: Checks if the player reached the endpoint.
- restartGame: Resets player position and move count.
- Friend function to display the maze (operator<<).

USER INTERACTION

ANIMATED WELCOME MESSAGE

- Displays a creatively styled animated text using escape sequences for color.

HOME PAGE AND MENU

Options:

- View game rules and terms.
- Start the game.
- View the solution for a maze.

RULES AND TERMS

Provides an overview of game mechanics, rules, and terms for using the program.

DIFFICULTY SELECTION

- Allows the user to choose between Easy, Medium, or Hard difficulty levels.

GAME FEATURES

1. PLAYER MOVEMENT

- Players navigate the maze using W, A, S, and D keys to move up, left, down, and right, respectively.
- The player's position is updated dynamically based on input.
- Invalid moves (hitting a wall) reset the player to the starting position.

2. MAZE DIFFICULTY LEVELS

- Three difficulty levels are available: **Easy**, **Medium**, and **Hard**.
- Each level corresponds to a different maze structure, loaded from files (easy.txt, medium.txt, and hard.txt).

3. DYNAMIC MAZE DISPLAY

- The maze is displayed with color-coded elements:
    - **Green (P):** Player's current position.
    - **Red (#):** Walls.
    - **Blue (E):** Exit point.
    - **White (.) or Cyan:** Empty paths.
- The display updates after every move.

4. GOAL-ORIENTED GAMEPLAY

- The objective is to guide the player to the exit (E).
- Winning triggers a congratulatory message with stats like **total moves** and **time taken**.

5. CUSTOM WELCOME AND MENU

- Includes an animated welcome message.
- Options for:
    - Reading rules and terms and conditions (R).
    - Starting the game directly (M).
    - Viewing maze solutions (S).

6. ERROR HANDLING

- Invalid inputs during movement display a warning message without crashing the game.
- Missing or improperly formatted maze files trigger descriptive runtime errors.

7. MOVE COUNT AND TIME TRACKING

- Tracks the number of moves taken by the player.
- Uses a timer to measure how long it takes to solve the maze.

8. REPLAY AND RESTART OPTIONS

- After completing a maze, players can:
    - Replay the same maze.
    - Return to the main menu to select a new maze.

9. VISUAL MAZE SOLUTIONS

- Provides a step-by-step solution for each maze difficulty.
- Displays the solved maze with the path highlighted in red (X).

10. ROBUST TERMINATION

- Players can quit or restart at any stage.
- Proper cleanup of dynamically allocated memory ensures stability.

CORE CONCEPTS USED

COMPOSITION

- **Definition**: Composition is when one class contains objects of other classes as members. This represents a "has-a" relationship.
- **Usage in the code**: The Maze class contains a Player object, representing a "has-a" relationship between the maze and the player. This means the Maze class doesn't just manipulate individual player data, but directly contains the player object within itself.

```
class Maze {
private:
    Player player;
```

OPERATOR OVERLOADING

- **Definition**: Operator overloading allows user-defined classes to define custom behavior.
- **Usage in the code**: In the Maze class, the << operator is overloaded to display the maze in a user-friendly format. This customizes how the maze is printed when the cout stream is used.

```
friend ostream& operator<<(ostream& out, const Maze& mazeObj) {
```

INHERITANCE

- **Definition**: Inheritance allows one class (child class) to inherit properties and methods from another class (parent class), promoting code reuse and hierarchy.
- **Usage in the code**: The Player class inherits from the Game class. The Player class is a specialized version of the Game class, gaining its properties and methods like x, y, and move() but overriding the move() method to provide specific movement logic.

```
class Player : public Game {
```

POLYMORPHISM

- **Definition**: Polymorphism allows objects of different classes to be treated as objects of a common base class. There are two types: compile-time and runtime.
- **Usage in the code**: The move() function in the Game class is a pure virtual function, meaning it must be overridden in derived classes like Player. This is an example of runtime polymorphism, where the actual function to call is determined at runtime.

```
Game(int startX = 0, int startY = 0) : x(0), y(0) {}

class Player : public Game {
    void move(char direction) override {}
```

EXCEPTION HANDLING

- **Definition**: It is a mechanism to handle runtime errors using try, catch, and throw blocks.
- **Usage in the code**: The Maze constructor, errors related to file opening and reading are caught with try, catch, and runtime exceptions are thrown for invalid difficulty levels. In the movePlayer() method, invalid move is caught, and the player is reset to the starting position.

```
catch (const runtime_error& e) {}
catch (const exception& e) {}
catch (...) {}
```

FILING

- **Definition**: Filing involves reading from and writing to files, such as saving data or loading configuration.
- **Usage in the code**: The Maze constructor reads the maze layout from a file based on the selected difficulty level (easy, medium, hard). If the file cannot be opened or an error occurs during reading, exceptions are thrown and caught.

```
ifstream file(filename);
if (!file.is_open()) {
```

FUTURE ENHANCEMENTS

1. GRAPHICAL USER INTERFACE (GUI)

Transition from a console-based game to a GUI-based game using frameworks like SFML, SDL, or Qt for C++. Provide more visually appealing graphics for walls, players, and the goal.

2. MAZE SOLVER AND HINT SYSTEM

Allow the player to request a hint, showing them the next step in the optimal path. Add an auto-solve feature to display the solution path step-by-step.

3. STORY MODE

Add a narrative or theme to the game, like escaping from a dungeon or exploring an ancient temple.

4. ONLINE FEATURES

Introduce an online multiplayer mode where players can solve mazes collaboratively or competitively. Host time-limited events with unique mazes and rewards to encourage replay ability and community engagement.

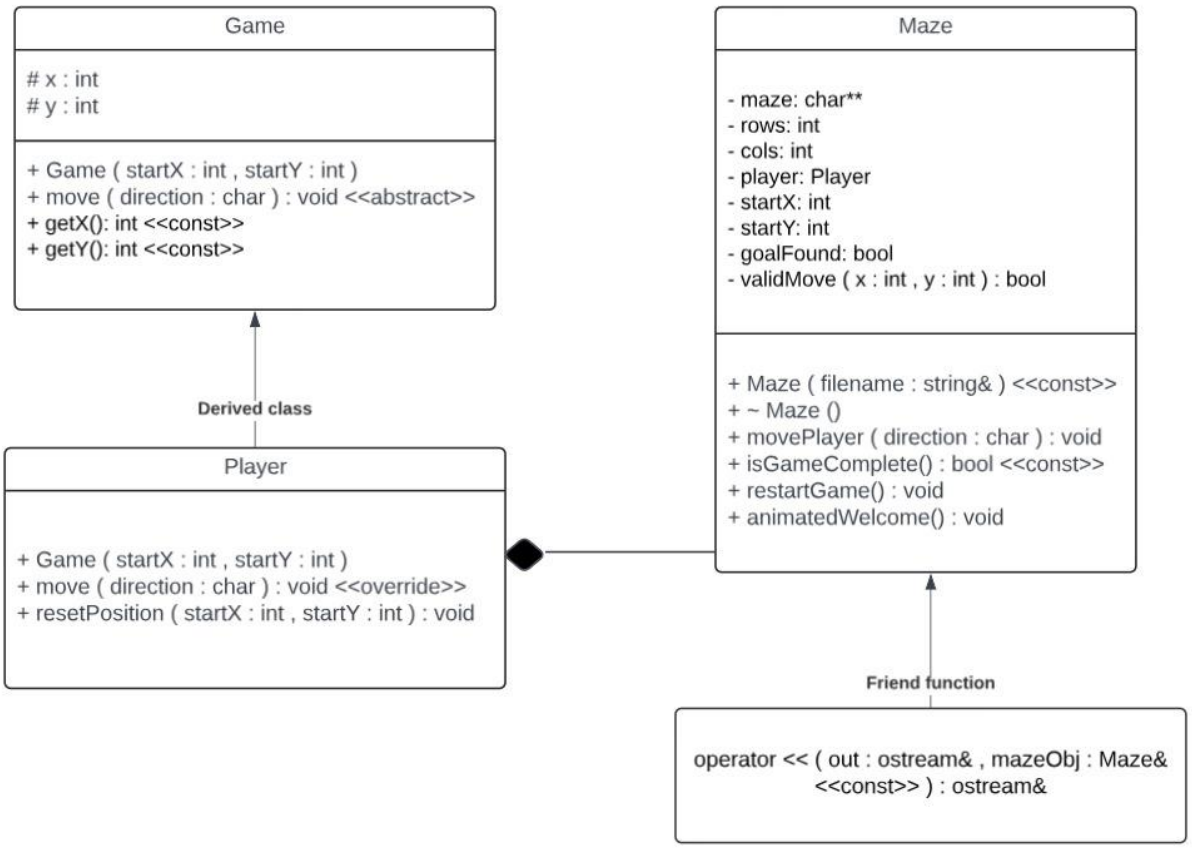
5. PLATFORM INDEPENDENCE:

Refactor the code to work seamlessly on different operating systems (e.g., Linux, macOS) by removing platform-specific dependencies like Windows.h.

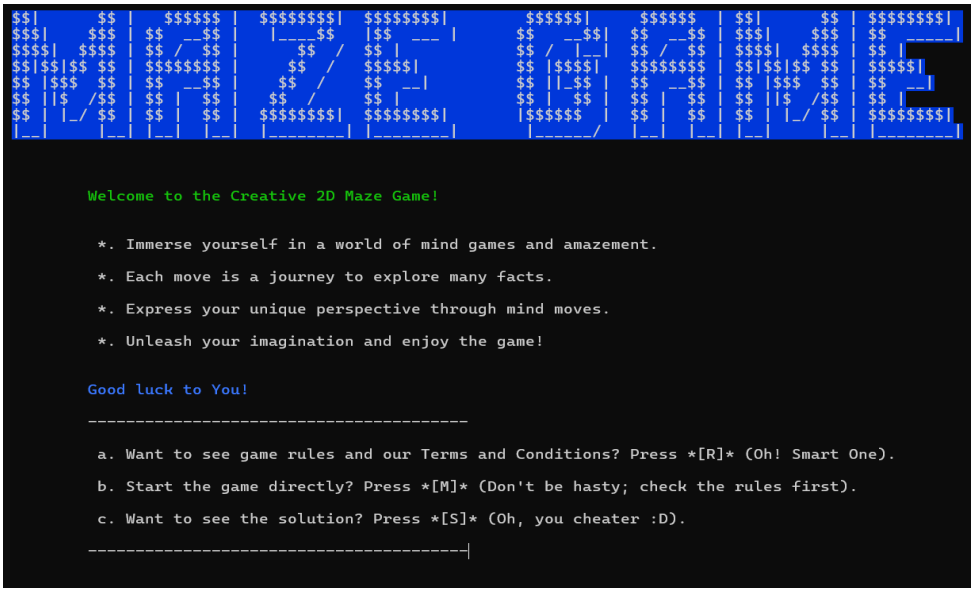
6. LEADERBOARDS AND ACHIEVEMENTS:

Add online leaderboards for players to compete globally. Introduce achievement badges for milestones (e.g., "Fastest Completion," "Fewest Moves").

UML CLASS DIAGRAM

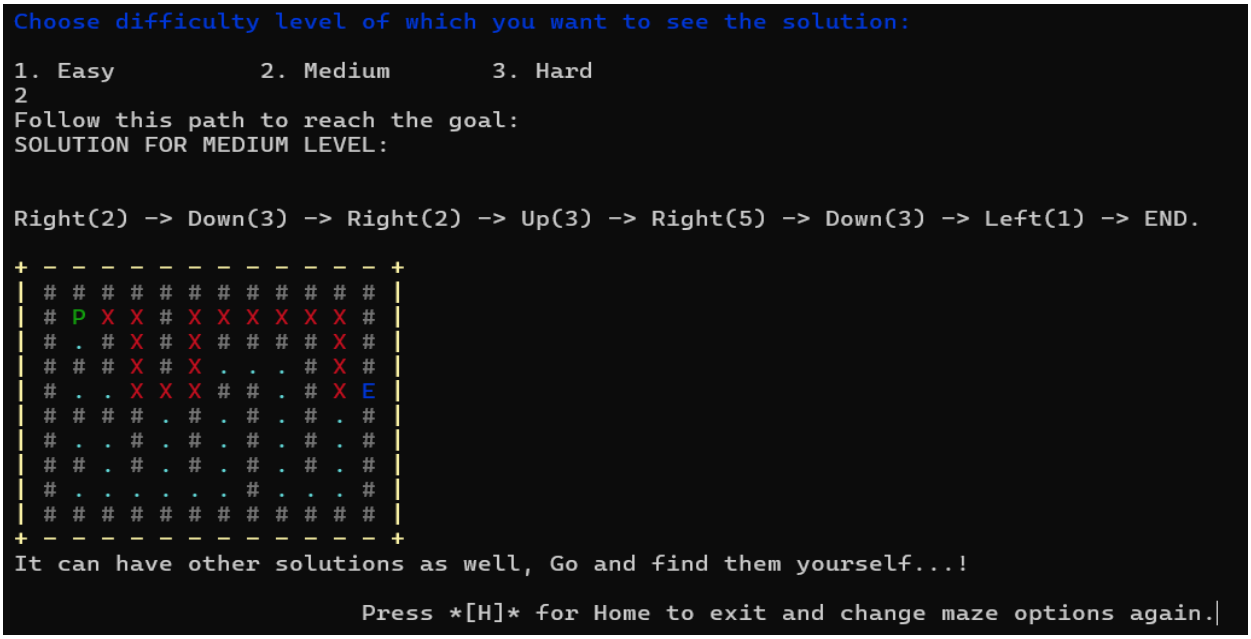


SAMPLE OUTPUTS



HOME PAGE

SOLUTION



GAME MODE

WINNING MODE

```
Congratulations! You completed the maze!

Total Moves: 19

Time Taken: 8 seconds

Press *[Y]* to replay


Press *[H]* for Home to exit and change maze options again.

Press *[ANY OTHER KEY]* to exit
```

CONCLUSION

This code demonstrates a well-structured implementation of a **2D Maze Game** using **object-oriented programming (OOP)** concepts in C++. It showcases a blend of functionalities and features that provide an engaging and dynamic user experience. Below are the key points summarizing the conclusion:

OBJECT-ORIENTED DESIGN:

The use of classes such as Game, Player, and Maze ensures modularity, code reuse, and easy scalability. Proper use of inheritance and polymorphism is evident, especially in the move function, which is overridden in the Player class.

ERROR HANDLING:

The program handles potential errors gracefully, such as invalid file input, missing goals in the maze, or invalid moves by the player.

ENHANCED USER INTERFACE:

ASCII art, colorful terminal output, and motivational messages make the game more immersive and engaging for the player.

DYNAMIC GAMEPLAY:

The player can explore the maze, restart the game upon completion, and navigate using simple keypresses (W, A, S, D).

SOLUTION DISPLAY:

A feature is provided to display a sample solution path, enhancing the game's accessibility for beginners.