OOPS COURSE PROJECT

MAZE GAME RUNNER

CS23B1079, CS23B1077, CS23B2001

Overview

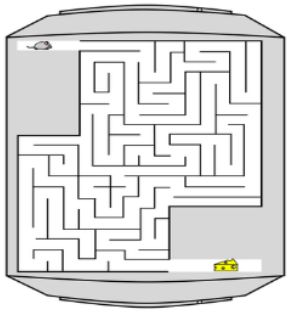


Figure 1: Maze Game Layout

This document provides an in-depth look into the Maze Runner game's C++ code, covering object-oriented programming principles such as encapsulation, classes and objects, abstraction, constructors and destructors, and more. It explains why certain techniques and features are used and discusses specific choices made in implementing this program.

Scalability with Modular Class Design
The code's modular class-based approach (with separate classes for Game, Maze, and Position) supports scalability.

Additional features, levels, or even new game mechanics can be introduced without significant restructuring. This object-oriented approach is ideal for future expansions, enabling the game to evolve with new content or functions.

Real-Time Feedback for Enhanced User Engagement
The use of real-time feedback, made possible with getch() from $< conio.h >$, enables the game to capture user input immediately. This feedback system allows players to see instant responses to their moves, improving the player experience.

Exploring Core Object-Oriented Concepts in the Code

1. Encapsulation and Abstraction

Encapsulation allows the code to securely enclose sensitive data within each class, providing access only through well-defined interfaces. For instance, the `Game, Maze`, and `Position` classes each maintain their data—such as the maze layout in grid, the player's position in `playerPos`, and the goal position in `endPos`—privately. Only methods within these classes can directly alter these members, ensuring security and modularity. This structure prevents unintended external access or modification, aligning with the core principles of encapsulation.
Abstraction simplifies complex game mechanics by focusing only on essential functions. For example, the Maze class abstracts details of grid management and movement mechanics, while Position abstracts coordinate handling, leaving the specifics out of the main game loop. This separation ensures each part of the code remains manageable and reusable, making the game structure easier to expand or modify.

2. Static Storage Class Usage

Tracking Total Games:
Without static, each Game object would have its own copy of the totalGames variable, and changes made by one instance would not reflect in other instances. Using static, the variable is stored in a single location, allowing all instances to track the total number of games played, regardless of how many Game objects are created. Static makes sure that the totalGames value is consistent across all Game objects.

3. Function Pointers in Game Controls

In this code, function pointers are used to add flexibility to the game controls, specifically in the Game class through the moveFunction pointer.
The moveFunction pointer can point to different functions that handle player movement, such as moving the player up (w), down (s), left (a), or right (d).
It allows the game to respond to player input dynamically, adapt to changes in movement behavior, and manage game mechanics with minimal changes to the core game logic. This approach makes the game more maintainable, extensible, and easier to update as new features are introduced.

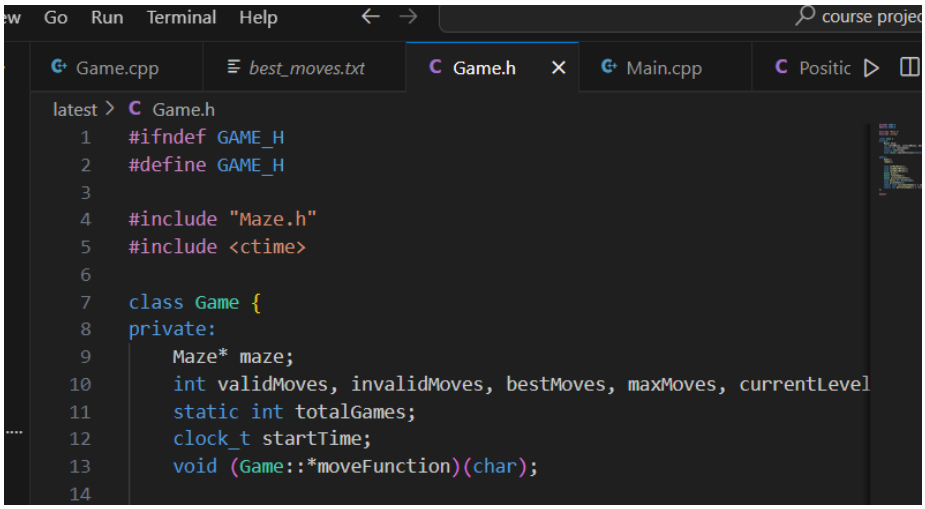


Figure 2: Function pointers

4. Operator Overloading

The Position class provides several overloaded operators to simplify and enhance its usage. These operators allow the class to behave more naturally in common operations such as addition, comparison, and input/output.

- Addition operator: This operator allows you to easily combine two positions, which is useful for moving a player or object in a game world.
- Equality operator: This operator allows easy comparison of positions to check if they are the same.
- Friend functions are used in the overloading of the << (output stream) and >> (input stream) operators. These operators are overloaded as friend functions to enable direct access to the private members (x and y) of the Position class. By making these functions friends, we allow them to perform input and output operations on the Position object, which would otherwise be inaccessible due to encapsulation.
  The << operator is used for printing the Position object to the output stream (such as std::cout), and the >> operator is used to read the coordinates from the input stream (such as std::cin).

5. Dynamic Memory Allocation in Maze Class

In the Maze class, Dynamic Memory allocation is used to create and manage the maze grid and other dynamically allocated objects, such as the player's position and the maze's endpoint.

- Dynamic memory allows for creating a grid that matches the level's size at runtime. The grid can be resized and allocated based on the current game's level requirements.
- Since the player's position is subject to change throughout the game (as the player moves), dynamically allocating memory for playerPos ensures flexibility. The position is typically allocated dynamically, either directly or as a pointer.
- Like the player position, the endpoint's coordinates may also be subject to change between levels, which is why endPos is stored in dynamically allocated memory. The endpoint is allocated similarly to playerPos.

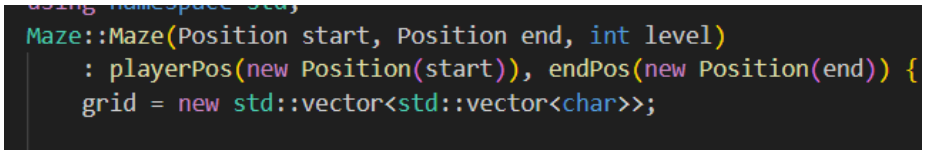


Figure 3: DMA

6. Cascaded Functions in the Game Class

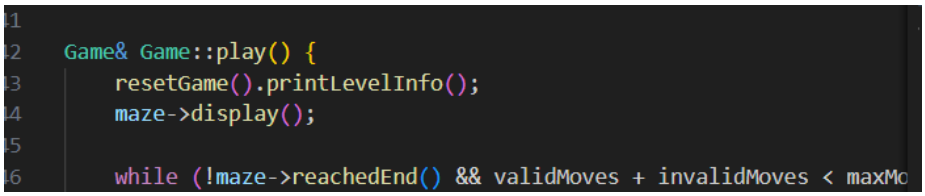


Figure 4: Cascading Functions

In the context of the Game class, methods such as play(), resetGame(), and printLevelInfo() return the current object (*this) after performing their respective operations. By doing so, they allow you to chain these methods together in one expression, which can make the code more concise and readable.

7. Function Composition in the Game Class

Function composition involves combining several functions to build more complex behaviors. In the Game class, the play() function is a prime example of function composition, as it integrates multiple smaller functions to construct the complete game loop. The play() method can call several functions in a sequence to handle various aspects of gameplay, such as:

- `resetGame()`: This function likely resets game variables, prepares the game state, or sets up the initial level.
- `printLevelInfo()`: This function probably prints or displays information related to the current level, such as the maze layout or available moves.
- `move()`This function handles the player's input and moves the character according to the rules of the game.

When you call play(), all these actions happen sequentially, one after the other, forming a coherent and smooth gameplay experience. The function composition ensures that the game logic is encapsulated within the play() method, reducing code duplication and improving modularity. You don't have to manually call each method separately; they are composed into a single call that encapsulates the entire flow of the game, making the code easier to manage.

This structured approach to composing functions ensures that different parts of the game mechanics (resetting the game, displaying information, and handling movement) are logically grouped together and executed in the correct order, which is crucial for maintaining a consistent and efficient game loop.

Description of BextMoves.txt file

The bestmoves.txt file serves as a persistent storage system for tracking the game's best moves, preserving high scores across sessions to motivate players by setting a benchmark for improvement.

By allowing the program to compare current moves with previously achieved best scores, it introduces a competitive edge, letting players challenge themselves to beat past records. Using a simple text file keeps data management efficient and lightweight, eliminating the need for complex databases and ensuring cross-platform compatibility.

Players and developers benefit from easy access for review and debugging, as the file's plain text format allows for quick checks and direct edits if needed. Within the code, bestmoves.txt is read at the start of each session to set the target moves and updated after a session if a new record is achieved, with error handling in place to initialize the file if missing, ensuring smooth game play regardless of file status.

Output Description

Throughout game play, the program provides detailed feedback to the player via print statements, creating a fully interactive and engaging experience. For

each move, the game displays key information such as the **player's current position within the maze, the total moves taken, and the current level**.

The program also tracks and displays **valid moves**, counting only the successful moves toward the total allowed moves for the level. After each successful move, the **time** elapsed since the start of the game session is printed, helping the player stay aware of the game's pace. **Game statistics, such as the total moves** taken across all levels, are displayed, as well as more immediate stats specific to the current level.

If the player exceeds the maximum moves for the level without reaching the end, the game outputs a message like **"You were lost in the maze"** and resets the level, prompting the player to try again. Upon successfully completing a level within the allowed moves, the player is **congratulated** and allowed to progress to the next level, with a message that acknowledges their accomplishment.

At the end of the game, if the player has achieved a record for minimal moves, the best moves are saved to **bestmoves.txt**, preserving the achievement for future reference. This approach keeps game play immersive by offering real-time feedback and a clear sense of progression.

Header files used

In the maze game code, each header file serves a distinct purpose, ensuring smooth functionality and efficient handling of operations related to input, output, memory, time management, and game data storage. Here's an overview of the role each header plays in the game's mechanics:

$< conio.h >$ The ¡conio.h¿ header provides console input/output functions, particularly useful for games with real-time interaction. It includes functions like getch() for reading keystrokes instantly without requiring the user to press Enter, allowing smoother and more interactive game controls. This enhances user experience by allowing immediate player movement based on inputs like w, a, s, and d for directional control within the maze.

$< fstream >$ The ¡fstream¿ header enables file handling operations, allowing the game to read from and write to external files. This is especially important for storing and retrieving the game's best move records, which are saved in a text file (e.g., bestmoves.txt). Using ¡fstream¿, the game can track the player's best performances across sessions, load saved game statistics, and write updated scores, enhancing continuity and replay ability.

$< climits >$ provides constants for the limits of integer data types. These constants are helpful for initializing high scores or move counts with maximum

values, which can then be minimized as players achieve better scores. It ensures that initial values are optimally set, allowing for accurate comparisons as game progress is tracked.

< *ctime* > The ¡ctime¿ header supplies functions for time tracking, crucial for calculating the game's duration and logging timestamps. It allows the game to initialize, record, and display time-related data, such as the start time and total time taken to complete levels or the entire game. This adds an extra layer of challenge for players by allowing performance measurement over time.

< *vector* > The ¡vector¿ header introduces a dynamic array structure, which the game uses to store and manage data flexibly, such as the maze grid, valid moves, and player positions. Vectors can dynamically resize, making them ideal for handling collections of data that vary in size depending on the level or game play. This flexibility simplifies memory management and reduces the risk of overflow issues, especially in a complex game environment where different levels may require varying grid sizes or layouts.

Go Run Terminal Help course project

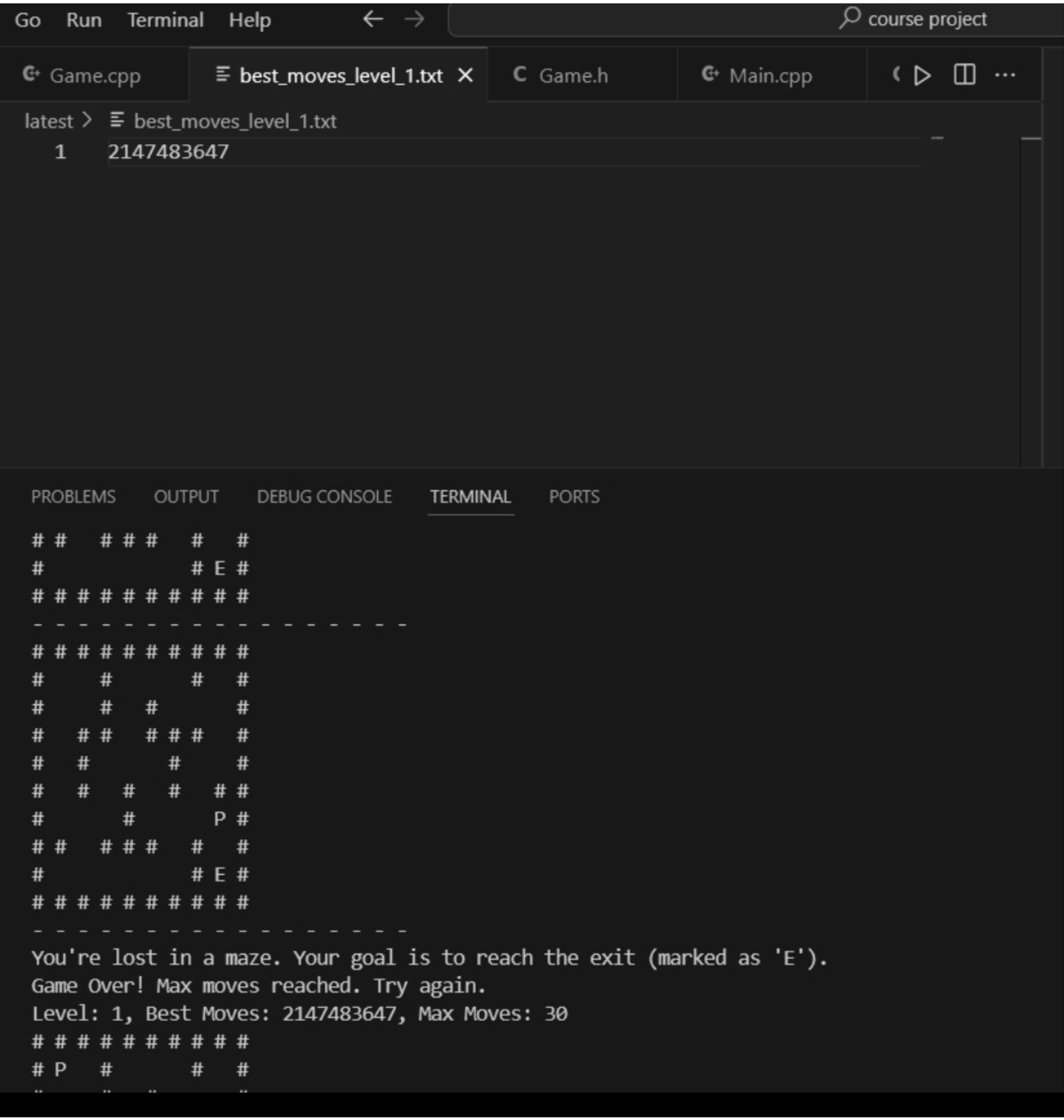
Game.cpp | best_moves_level_1.txt | Game.h | Main.cpp

latest > best_moves_level_1.txt

```
1
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\DELL\Contacts\Desktop\c++\course project\latest> ./a.exe

Welcome to the Maze Runner!


_____________________________

Warning: Failed to read best moves for Level 1. Setting to default.

Use 'W', 'A', 'S', 'D' keys to move up, left, down, and right, respectively.
Level: 1, Best Moves: 2147483647, Max Moves: 30
# # # # # # # # # #
# P   #       #   #
#     #   #       #
#   # #   # # #   #
#   #       #     #
#   #   #   #   # #
#       #         #
# #   # # #   #   #
#             # E #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - -
# # # # # # # # # #
#   P #       #   #
#     #   #       #
#   # #   # # #   #
#   #       #     #
```


Go Run Terminal Help
course project
Game.cpp
best_moves_level_1.txt
Game.h
Main.cpp
latest > best_moves_level_1.txt
1 2147483647
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
# # # # # # #
E
# # # # # # # # # # #
- - - - - - - - - - - - - - - - - -
# # # # # # # # # # #
# # # # #
# # # # #
# # # # # # #
# # # # # #
# # # # # # #
# # # # # # #
P
# # # # # # #
E
# # # # # # # # # # #
- - - - - - - - - - - - - - - - - -
You're lost in a maze. Your goal is to reach the exit (marked as 'E').
Game Over! Max moves reached. Try again.
Level: 1, Best Moves: 2147483647, Max Moves: 30
# # # # # # # # # # #
P # #

Go Run Terminal Help course project

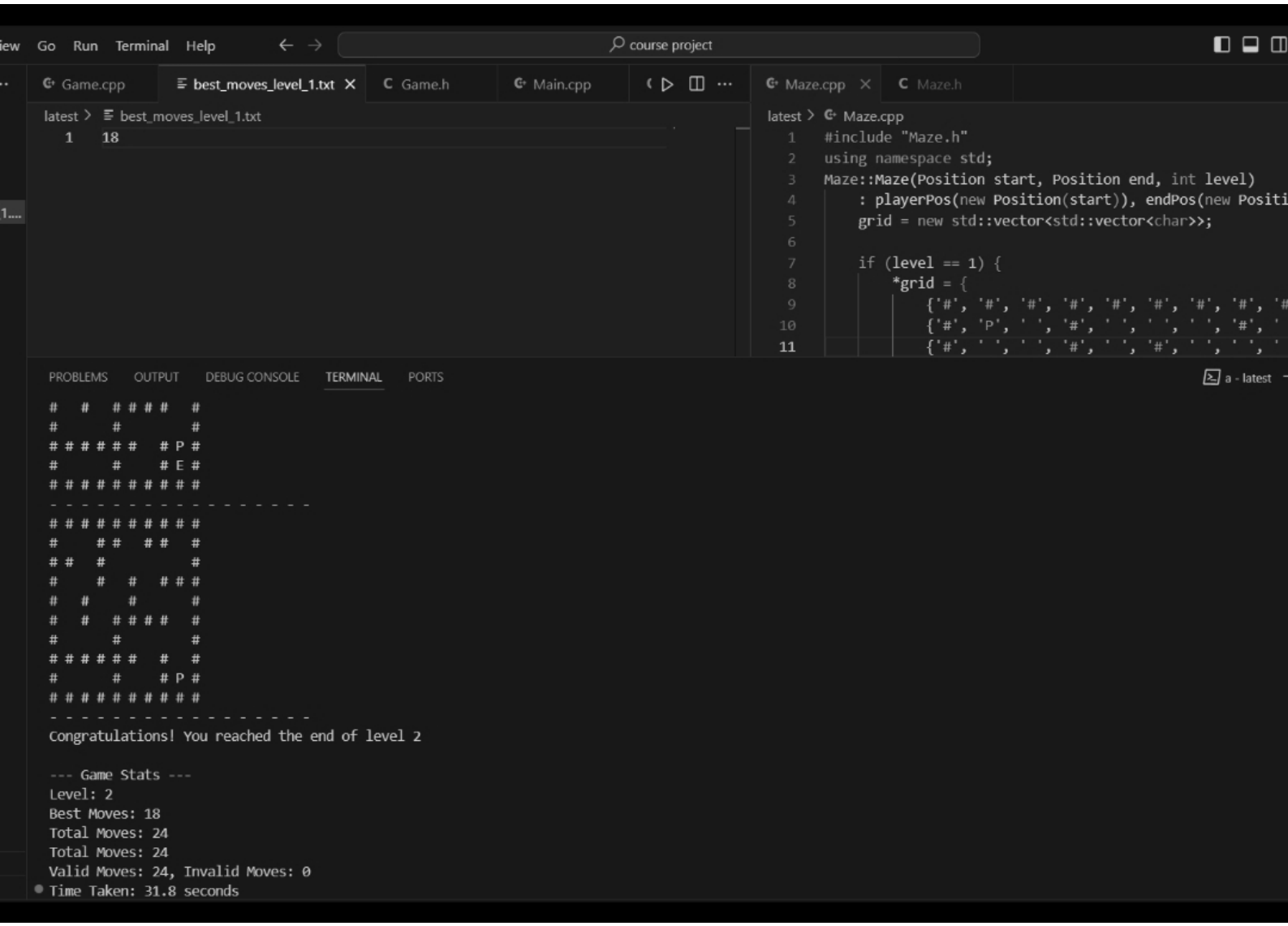
Game.cpp | best_moves_level_1.txt | Game.h | Main.cpp

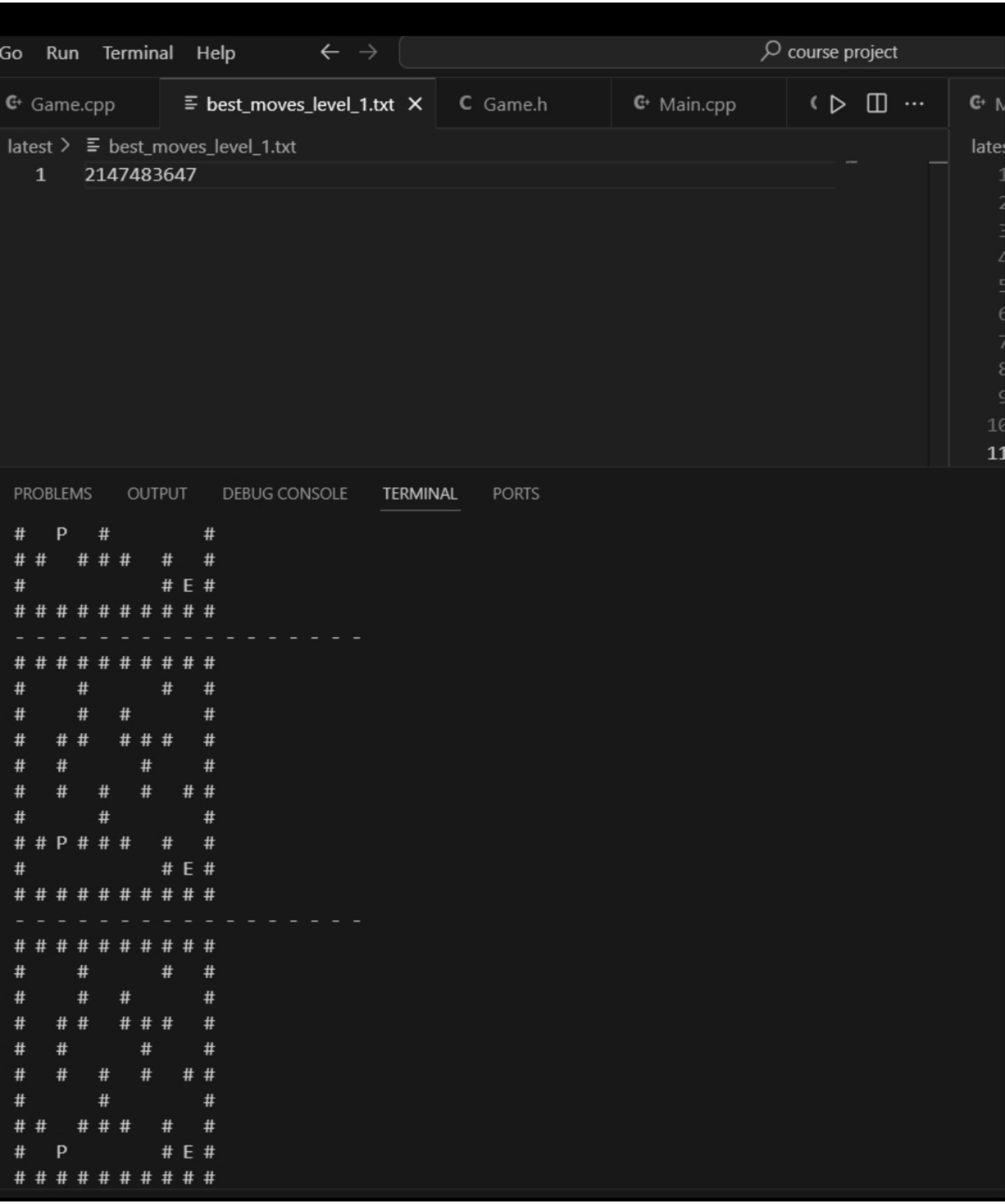
latest > best_moves_level_1.txt

```
1   20
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
# #   # # #   #   #
#           # P #
# # # # # # # # #
- - - - - - - - - - - - - - - - - -
Best moves saved: 20 for Level 1
Congratulations! You reached the end of level 1

--- Game Stats ---
Level: 1
Best Moves: 20
Total Moves: 20
Valid Moves: 20, Invalid Moves: 0
Time Taken: 36.464 seconds
Level: 2, Best Moves: 20, Max Moves: 30
# # # # # # # # # #
# P   # #   # #   #
# #   #         #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #         #
# # # # # #   #   #
#       #     # E #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - - -
```


Go Run Terminal Help
course project
Game.cpp
best_moves_level_1.txt
Game.h
Main.cpp
Maze.cpp
Maze.h
latest > best_moves_level_1.txt
1 18
latest > Maze.cpp
1 #include "Maze.h"
2 using namespace std;
3 Maze::Maze(Position start, Position end, int level)
4 : playerPos(new Position(start)), endPos(new Positi
5 grid = new std::vector<std::vector<char>>;
6
7 if (level == 1) {
8 *grid = {
9 {'#', '#', '#', '#', '#', '#', '#', '#', '#
10 {'#', 'P', ' ', '#', ' ', ' ', ' ', '#', '
11 {'#', ' ', ' ', '#', ' ', '#', ' ', ' ', '
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
a - latest
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
- - - - - - - - - - - - - - - - - - -
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # #
- - - - - - - - - - - - - - - - - - -
Congratulations! You reached the end of level 2
--- Game Stats ---
Level: 2
Best Moves: 18
Total Moves: 24
Total Moves: 24
Valid Moves: 24, Invalid Moves: 0
Time Taken: 31.8 seconds


Go Run Terminal Help
course project
Game.cpp
best_moves_level_1.txt
Game.h
Main.cpp
latest > best_moves_level_1.txt
1 2147483647
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
P #
# # # # # # #
E
# # # # # # # # # # #
- - - - - - - - - - - - - - - - -
# # # # # # # # # # #
# # # #
# # # #
# # # # # #
# # # #
# # # # # #
# # # #
P # # # # #
E
# # # # # # # # # # #
- - - - - - - - - - - - - - - - -
# # # # # # # # # # #
# # # #
# # # #
# # # # # #
# # # #
# # # # # #
# # # #
# # # # # # # #
P # E
# # # # # # # # # # #

Detailed Inferences for Inheritance

1. Abstracting Core Functionality in the Base Class (Maze)

The ‘Maze‘ class encapsulates all essential maze-related operations (grid initialization, player movement, path checking, etc.) in a single place. This ensures the program has a foundational implementation of a maze that other specialized mazes can extend or modify.

- Avoids duplicating functionality across multiple classes (e.g., ‘AdvancedMaze‘ and potential future derived classes).
- Centralized logic makes maintenance easier; changes to the core functionality affect all derived classes automatically.
- Keeps the base class focused on "what every maze should do," ensuring clear boundaries between generic and specialized behaviors.

2. Customization in Derived Class (AdvancedMaze)

The derived class ‘AdvancedMaze‘ is introduced to add unique properties (like ‘mazeName‘) and override certain behaviors (e.g., ‘display‘, ‘isWithinBounds‘) of the base class.

- Instead of modifying the ‘Maze‘ class, which could disrupt its intended generic behavior, inheritance is used to create a specialized version of the maze.
- Separation of concerns: Customizations in ‘AdvancedMaze‘ do not clutter the logic of the ‘Maze‘ class.
- Future-proofing: Developers can easily add other derived classes (e.g., ‘TimedMaze‘ or ‘PuzzleMaze‘) without altering the base class or ‘AdvancedMaze‘.

3. Reuse of the Base Class

The ‘AdvancedMaze‘ constructor explicitly calls the ‘Maze‘ constructor to initialize the ‘playerPos‘ and ‘endPos‘ fields in the base class. This prevents redundancy in the initialization logic for these attributes, as they are already handled in the base class.

4. Overriding Virtual Functions

Overriding the ‘display()‘ and ‘isWithinBounds()‘ methods in ‘AdvancedMaze‘ allows the program to provide specialized behavior while adhering to the same interface defined by the base class.

- ‘AdvancedMaze::display()‘ adds the maze name to the display output while retaining the core grid display functionality from the base class.
- ‘AdvancedMaze::isWithinBounds()‘ includes additional logging to inform the user about which class is handling the bounds check.

This supports polymorphism by ensuring that all maze types share the same interface, enabling flexible gameplay mechanics.

5. Polymorphic Usage in the Game

The ‘Maze* maze‘ pointer in the game allows the program to work with any maze type (e.g., ‘Maze‘ or ‘AdvancedMaze‘) dynamically.

- A base-class pointer (‘Maze*‘) is used instead of directly instantiating a ‘Maze‘ or ‘AdvancedMaze‘ object. This design allows the game to introduce new maze types in the future without changing existing gameplay code.

Inferences for Polymorphism

1. Dynamic Method Binding Through Virtual Functions

By declaring member functions such as ‘display()‘ and ‘isWithinBounds()‘ as virtual, the program ensures that the version of the function executed is determined at runtime, based on the actual object type being referenced (e.g., ‘Maze‘ or ‘AdvancedMaze‘).

- The program can call methods on a base class pointer (‘Maze* maze‘) and invoke the appropriate derived class implementation (‘AdvancedMaze‘’s overridden methods).

2. Use of Virtual Destructors

The destructor is explicitly marked ‘virtual‘, even if no direct resource cleanup is required in this example, to future-proof the design.

Why Polymorphism Works Well in This Program

- **Dynamic Behavior:** The program dynamically determines which ‘display‘ or ‘isWithinBounds‘ implementation to use, based on the maze type.
- **Extensibility:** New maze types can be added without altering the game logic, supporting future enhancements.
- **Encapsulation and Reusability:** Maze-specific logic is encapsulated within each class, maintaining a clean and modular design. The base class provides reusable functionality, while derived classes focus only on customizations.

Description of the Nickname Attribute

The ‘nickname‘ attribute is a specific addition to enhance user interactivity and personalize the player's experience.

- By allowing the user to set a nickname, the program creates a more engaging and interactive experience. Instead of referring to the player generically, the game can address them using their chosen identifier, fostering a sense of ownership and immersion.

Integration with Output: The nickname is incorporated into messages displayed during the game. For instance: "All the best, my dear PlayerName! Let's begin the adventure."

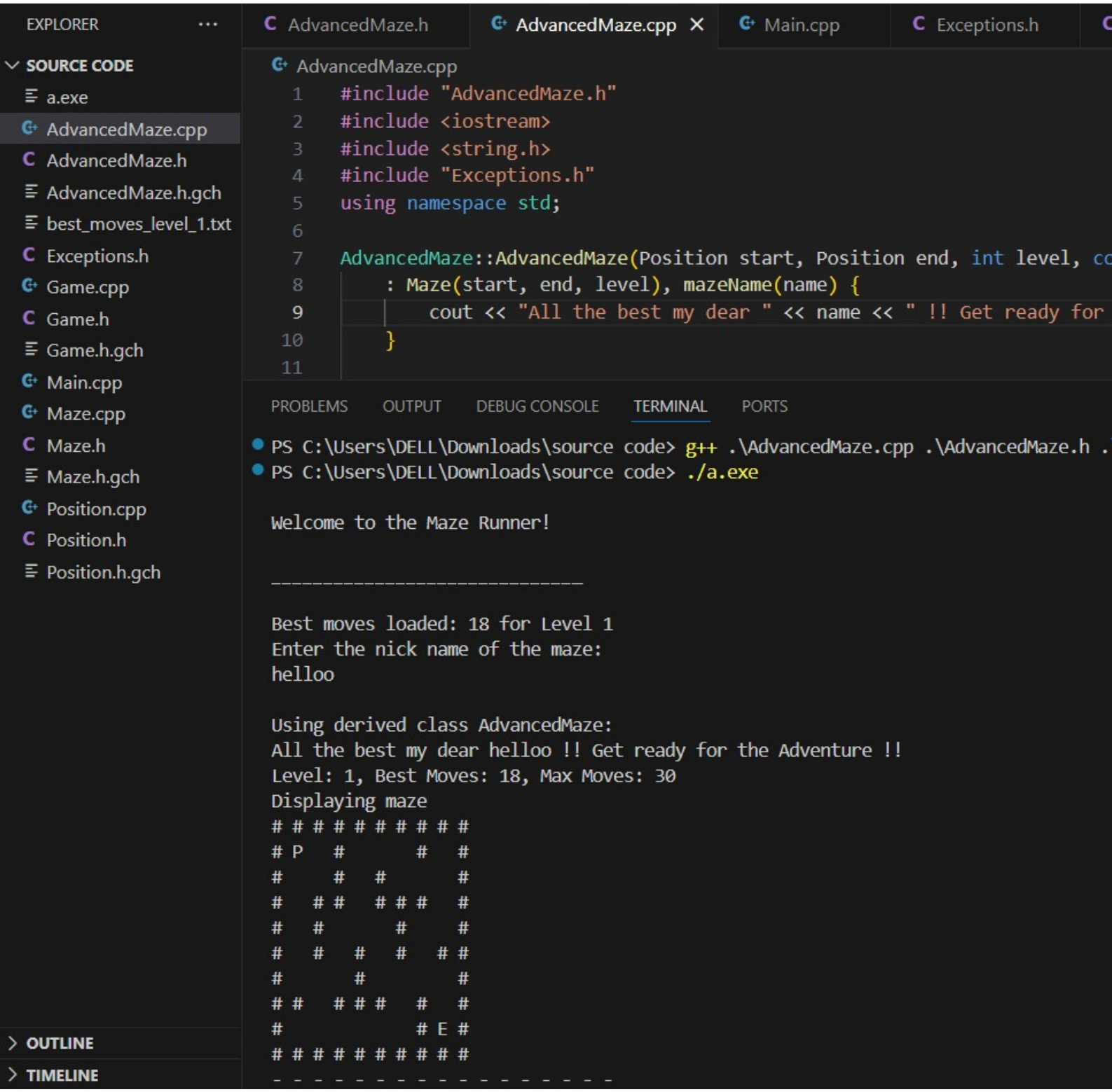
Exception Handling in the Code

How It's Achieved

- **Custom Exception Classes:** Four custom exception classes (‘MemoryAllocationException‘, ‘FileIOException‘, ‘InvalidMoveException‘, ‘OutOfBoundsException‘) derived from ‘std::exception‘ provide specific error messages.
- **Throwing Exceptions:**
  - ‘MemoryAllocationException‘ is thrown in the ‘Maze‘ constructor if memory allocation fails.
  - ‘FileIOException‘ is thrown during file operations for best moves.
  - ‘OutOfBoundsException‘ is thrown if a move attempts to go outside the grid.
  - ‘InvalidMoveException‘ is thrown when the player inputs an unsupported direction.
- **Catching Exceptions:** The ‘main‘ function uses a try-catch block to catch and handle exceptions, ensuring robust error reporting and recovery.
- **Code Specifics:**
  - The ‘Maze::isWithinBounds()‘ function explicitly throws an exception for invalid positions.
  - ‘Game::loadBestMoves()‘ and ‘saveBestMoves()‘ handle file I/O exceptions gracefully, ensuring the game continues despite failures.

```h
#ifndef ADVANCED_MAZE_H
#define ADVANCED_MAZE_H

#include "Maze.h"
#include <string>

class AdvancedMaze : public Maze {
private:
    std::string mazeName;

public:
    AdvancedMaze(Position start, Position end, int level, const std::string& name);

    void setMazeName(const std::string& name);
    std::string getMazeName() const;


    void display() const override;
    bool isWithinBounds(const Position& pos) const override;
};

#endif
```

```cpp
class MemoryAllocationException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Memory allocation failed.";
    }
};

class FileIOException : public std::exception {
public:
    const char* what() const noexcept override {
        return "File input/output error.";
    }
};

class InvalidMoveException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Invalid move input.";
    }
};

class OutOfBoundsException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Position is out of maze bounds.";
    }
};
```


EXPLORER
AdvancedMaze.h
AdvancedMaze.cpp
Main.cpp
Exceptions.h
SOURCE CODE
a.exe
AdvancedMaze.cpp
AdvancedMaze.h
AdvancedMaze.h.gch
best_moves_level_1.txt
Exceptions.h
Game.cpp
Game.h
Game.h.gch
Main.cpp
Maze.cpp
Maze.h
Maze.h.gch
Position.cpp
Position.h
Position.h.gch
OUTLINE
TIMELINE
AdvancedMaze.cpp
#include "AdvancedMaze.h"
#include <iostream>
#include <string.h>
#include "Exceptions.h"
using namespace std;
AdvancedMaze::AdvancedMaze(Position start, Position end, int level, co
: Maze(start, end, level), mazeName(name) {
cout << "All the best my dear " << name << " !! Get ready for
}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS C:\Users\DELL\Downloads\source code> g++ .\AdvancedMaze.cpp .\AdvancedMaze.h .
PS C:\Users\DELL\Downloads\source code> ./a.exe
Welcome to the Maze Runner!
Best moves loaded: 18 for Level 1
Enter the nick name of the maze:
helloo
Using derived class AdvancedMaze:
All the best my dear helloo !! Get ready for the Adventure !!
Level: 1, Best Moves: 18, Max Moves: 30
Displaying maze
# # # # # # # # # #
P # #
# # # #
# # # # # #
# # # # #
# # # # # #
# # # #
# # # # # #
E
# # # # # # # # # #

```
# #   # # #   # P #
#           # E #
# # # # # # # # #
- - - - - - - - - - - - - - - - - -
Displaying maze
# # # # # # # # # #
#     #       #   #
#     #   #       #
#   # #   # # #   #
#   #       #     #
#   #   #   #   # #
#       #         #
# #   # # #   #   #
#           # P #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - -
Congratulations! You reached the end of level 1

--- Game Stats ---
Level: 1
Best Moves: 18
Total Moves: 18
Valid Moves: 18, Invalid Moves: 0
Time Taken: 9.683 seconds
Level: 2, Best Moves: 18, Max Moves: 30
Displaying maze
```

```
# # # # # # # # # #
#     # #   # # P #
# #   #           #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #         #
# # # # # #   #   #
#       #     # E #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - -
You're lost in a maze. Your goal is to reach the exit (marked as 'E').
Game Over! Max moves reached. Try again.
Level: 2, Best Moves: 18, Max Moves: 30
Displaying maze
# # # # # # # # # #
# P   # #   # #   #
# #   #           #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #         #
# # # # # #   #   #
#       #     # E #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - -
```

```
# # # # # # # # # #
- - - - - - - - - - - - - - - - - -
Displaying maze
# # # # # # # # # # #
#     # #   # #   #
# #   #           #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #       P #
# # # # # #   #   #
#       #     # E #
# # # # # # # # # # #
- - - - - - - - - - - - - - - - - -
Displaying maze
# # # # # # # # # # #
#     # #   # #   #
# #   #           #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #           #
# # # # # #   # P #
#       #     # E #
# # # # # # # # # # #
- - - - - - - - - - - - - - - - - -
```

```
- - - - - - - - - - - - - - - - - - - - -
Displaying maze
# # # # # # # # # # #
#     # #   # #   #
# #   #           #
#     #   #   # # #
#   #     #       #
#   #   # # # #   #
#       #         #
# # # # # #   #   #
#       #     # P #
# # # # # # # # # #
- - - - - - - - - - - - - - - - - - - - -
Congratulations! You reached the end of level 2

--- Game Stats ---
Level: 2
Best Moves: 18
Total Moves: 27
Valid Moves: 24, Invalid Moves: 3
Time Taken: 16.179 seconds
```