Proiect de Programare / C++

-= Sistem gestiune & interacțiune magazin =-

Student 1: Rareș Gherasă

Student 2: Marius Dînșorean

I. Project Description

The project implements an interactive quiz game inspired by "Who Wants to Be a Millionaire".

Student 1: Responsible for game management functionalities in quiz_app.exe. Tasks include:

- Loading questions from a file and presenting them to the player.
- Implementing the 50/50 lifeline to eliminate two incorrect options.
- Tracking player scores and updating the leaderboard.
- Handling game flow (question progression, correct/incorrect answers).

Student 2: Responsible for administrative functionalities in manage_app.exe. Tasks include:

- Managing the question bank (adding, deleting, modifying questions).
- Viewing and managing the leaderboard (displaying scores, clearing entries).
- Viewing player history based on player names.
- Ensuring data persistence through file operations.

II. Data Structures Used by the Team

The project uses the following C++ classes:

- **Question**:
  - **Attributes**:
    - `std::string questionText`: The text of the quiz question.
    - `std::vector<std::string> options`: A vector of four answer options (e.g., "A. Option1", "B. Option2", etc.).
    - `int correctAnswer`: Index of the correct answer (0–3).
    - `bool used`: Flag indicating if the question has been used in the current game session.
  - **Purpose**: Represents a single quiz question with its options and state.
  - **Relationship**: Used in composition within the `Game` class, as a collection of questions is integral to the game's functionality.
- **Player**:
  - **Attributes**:
    - `std::string name`: The player's name.
    - `float score`: The player's score (number of correct answers).
  - **Purpose**: Stores information about a player's performance in a game session.
  - **Relationship**: Contained within the `Leaderboard` class, establishing a composition relationship.
- **Leaderboard**:
  - **Attributes**:
    - `std::vector<Player> players`: A collection of `Player` objects representing leaderboard entries.
  - **Purpose**: Manages the leaderboard, allowing score additions, sorted display, and player history lookup.
  - **Relationship**: Contains a collection of `Player` objects (composition), as players are integral to the leaderboard's functionality.
- **Game**:
  - **Attributes**:
    - `std::vector<Question> questions`: A collection of questions loaded from the question file.
    - `Leaderboard leaderboard`: The leaderboard for storing player scores.
    - `std::mt19937 rng`: Random number generator for selecting questions randomly.
  - **Purpose**: Orchestrates the game, managing question loading, player interactions, lifeline usage, and leaderboard updates.

- **Relationships**:
    - **Composition**: Contains a `std::vector<Question>` to manage all questions.
    - **Composition**: Contains a `Leaderboard` object to manage player scores and history.

III. Structure of Files Used for Communication

The application uses the following file for data persistence:

- **questions.txt**:
    - **Purpose**: Stores the question bank for the quiz.
    - **Format**:
    - <question>,<optionA>,<optionB>,<optionC>,<optionD>,<correctAnswerIndex>
    - **Example**:
    - What is the capital of Brazil?,Rio de Janeiro,Brasilia,Sao Paulo,Salvador,1
    - Which planet is known as the Red Planet?,Venus,Mars,Jupiter,Saturn,1
    - **Usage**:
        - The `Game` class reads this file to load questions at startup.
        - If the file is not found, the application creates it with default questions.
- **Note on Leaderboard Storage**: The leaderboard is currently stored in memory within the `Leaderboard` class. For future extensions, a `leaderboard.txt` file could be implemented with the following format:
    - **Format**:
    - <number of entries>
    - <player_name1> <score1>
    - <player_name2> <score2>
    - **Example**:
    - 2
    - John 10.0
    - Alice 8.0
    - **Usage**:
        - `project.exe` could append player scores after each game session.
        - Administrative functions could read or clear this file for leaderboard management.
    - This is not implemented in the current version to keep the scope focused, but the `Leaderboard` class is designed to support such an extension.
- **Note on History Storage**: Player history is also stored in memory within the `Leaderboard` class. A `history.txt` file could be added with a similar format to `leaderboard.txt` to persist player session data, but this is not currently implemented.

IV. Commands Implemented by the Application

The application, `project.exe`, exposes the following command-line interface to initialize the game, adhering to the requirement for command-line argument usage:

- **Command**:
- ./project.exe [question_file]
    - **Description**: Starts the quiz game, loading questions from the specified `question_file`. If no file is provided, it defaults to `questions.txt`.
    - **Example**:
    - ./project.exe custom_questions.txt
    - Loads questions from `custom_questions.txt` and displays:
    - Using question file: custom_questions.txt
    - ./project.exe
    - Uses the default `questions.txt` and displays:
    - No question file specified. Using default: questions.txt
    - **Output**: Initializes the game and presents the main menu for further interaction.

V. Project File Structure

The project is organized into multiple files to satisfy the modularization requirement:

- **main.cpp**: Entry point; processes command-line arguments and initializes the `Game` class.
- **Game.hpp / Game.cpp**: Defines the `Game` class, which manages game logic, question loading, player interactions, and menu navigation.
- **Question.hpp / Question.cpp**: Defines the `Question` class for managing individual quiz questions and their state.
- **Player.hpp / Player.cpp**: Defines the `Player` class for storing player data (name and score).
- **Leaderboard.hpp / Leaderboard.cpp**: Defines the `Leaderboard` class for managing and displaying player scores and history.
- **Utils.hpp / Utils.cpp**: Contains utility functions, including input validation, screen clearing, and ANSI color macros for terminal output.