Aliza Samreen Agha (31562)

OOP S’25 Assignment 2

Report: Dungeon Escape(updated)

This report provides a detailed overview of the Dungeon Escape game's C++ code, covering its design, implementation details, and the use of various C++ features.

1. Use of Object-Oriented Programming (OOP) Principles

- **Encapsulation:** Achieved by bundling data (attributes) and methods (functions) that operate on the data within classes. Private members (e.g., Player::health, Room::name) are only accessible through public methods (e.g., Player::getHealth(), Room::getName()), ensuring data integrity and control over how objects are modified.
- **Abstraction:** The Character class serves as an abstract base class with a pure virtual function displayStatus(). This defines a common interface for all characters (Player and Enemy) without specifying the implementation details, which are left to derived classes. This allows for treating Player and Enemy objects generically as Character pointers or references.
- **Inheritance:** The Player and Enemy classes inherit from the Character base class. They reuse the name and health attributes and the displayStatus() method's interface, while adding their specific functionalities and data (e.g., Player::inventory, Enemy::description).
- **Polymorphism:** Demonstrated through the virtual ~Character() destructor and the displayStatus() method. Although displayStatus() is not explicitly called polymorphically in the GUI, its presence signifies the design choice for potential future polymorphic behavior. The operator<< for Player is an example of function overloading.

2. GUI Integration

The game integrates a Graphical User Interface (GUI) using the **SFML (Simple and Fast Multimedia Library)**.

- **GUI Class:** This central class manages the SFML window, handles events, updates UI elements, and draws everything to the screen.
- **Game States:** The GameState enum (NAME_INPUT, INSTRUCTIONS, PLAYING, GAME_OVER) is crucial for managing different screens and interactions within the

GUI. The GUI::draw and GUI::handleEvent methods use a switch statement based on gameState to render and react appropriately.

- **Event Handling:** The GUI::handleEvent method processes SFML events (e.g., window close, mouse clicks, text input) and translates them into game actions or state transitions.
- **Rendering:** The GUI::draw method clears the window and draws various SFML drawable objects (sf::Text, sf::RectangleShape) based on the current gameState and updated game data.
- **Player Stats Display:** On the GAME_OVER screen, the GUI::drawGameOver method now explicitly displays the player's name, health, moves, coins collected, enemies defeated, and sorted inventory using multiple sf::Text objects for clear presentation.

3. Design Choices

The game's design follows a clear separation of concerns, although some tight couplings exist for simplicity:

- **Modular Design:** The game is structured into distinct classes (Character, Player, Enemy, Treasure, Room, Dungeon, GameAssetManager, GUI), each responsible for a specific aspect of the game.
- **Game Loop (gameLoopWithGUI):** This function orchestrates the main game flow, handling event processing, game logic updates, and rendering. It acts as the bridge between the game's core logic (Player, Dungeon) and the GUI.
- **Asset Management (GameAssetManager):** A templated class designed to manage unique pointers to various game assets (specifically Room objects in this implementation). This promotes resource management and reduces redundancy.
- **Dungeon Structure:** The Dungeon class represents the game world, containing rooms and providing methods for navigation (advanceToNextRoom, backtrack) and accessing the current room.
- **Player-Centric Actions:** Player actions (fight, bypass, backtrack, quit) directly influence the player's state (health, moves, inventory) and the game's progression through the dungeon.

4. Data Structures Used

The code effectively utilizes several Standard Template Library (STL) containers:

- **std::vector<std::unique_ptr<T>> (in GameAssetManager):** Used to store dynamically allocated game assets (rooms). The use of std::unique_ptr ensures proper memory management for these assets.
- **std::list<std::string> (in Player::inventory):** Chosen for the player's inventory, likely due to its efficient insertion and deletion of elements, though sorting a std::list is also efficient.
- **std::queue<Enemy> (in Dungeon):** Used to hold a queue of enemies. While not directly used for in-game encounters (enemies are tied to rooms), it demonstrates understanding of queue data structures.
- **std::stack<const Room*> (in Dungeon):** Implements the backtracking mechanism, allowing the player to return to previously visited rooms in a LIFO (Last-In, First-Out) manner.

5. Memory Management

The code demonstrates modern C++ memory management practices primarily through **Smart Pointers**:

- **std::unique_ptr<T>:** Extensively used within the GameAssetManager class to manage Room objects. std::unique_ptr ensures that memory allocated for rooms is automatically deallocated when they go out of scope or the GameAssetManager is destroyed, preventing memory leaks. Ownership of assets is clearly managed.
- **RAII (Resource Acquisition Is Initialization):** Implicitly followed by using std::unique_ptr and SFML objects (like sf::RenderWindow, sf::Font), whose destructors handle resource cleanup automatically.

6. Explanation of Code

There are 8 classes in the provided code, each serving a distinct purpose in the game's architecture:

1. **Character (Abstract Base Class)**
   - **Constructor: Character(string n, int h)**
     - Use: Initializes the basic attributes of any character: name and health.
   - **Destructor: virtual ~Character()**
     - Use: A virtual destructor ensuring proper cleanup and deallocation of memory for objects of derived classes**.**

- **displayStatus(): virtual void displayStatus() const = 0**
    - Use: A pure virtual function that must be implemented by derived classes to display the character's status (e.g., name and health).
- **getName(): string getName() const**
    - Use: Returns the name of the character.
- **getHealth(): int getHealth() const**
    - Use: Returns the current health points of the character.
- **takeDamage(int damage): void takeDamage(int damage)**
    - Use: Reduces the character's health by a specified damage amount, ensuring health does not drop below zero.

2. **Player (Derived from Character)**
    - **Constructor: Player(string n)**
        - Use: Initializes a new player with a given name, default health (100), starting moves (10), zero coins, and no enemies defeated.
    - **heal(int amount): void heal(int amount)**
        - Use: Increases the player's health by amount, up to a maximum of 100.
    - **addToInventory(const T& item) (templated): template <typename T> void addToInventory(const T &item)**
        - Use: Adds an item (of any type T that can be streamed to a string) to the player's inventory list.
    - **addCoins(int amount): void addCoins(int amount)**
        - Use: Increases the player's coins count.
    - **useMove(): void useMove()**
        - Use: Decrements the player's available moves by one, ensuring moves don't go below zero.
    - **incrementEnemiesDefeated(): void incrementEnemiesDefeated()**
        - Use: Increments the counter for enemies defeated by the player.

   - **getMoves(): int getMoves() const**
     - Use: Returns the number of moves the player has remaining.
   - **getCoins(): int getCoins() const**
     - Use: Returns the total number of coins the player has collected.
   - **getEnemiesDefeated(): int getEnemiesDefeated() const**
     - Use: Returns the total count of enemies the player has defeated.
   - **getInventory(): list<string> getInventory() const**
     - Use: Returns a copy of the player's inventory list.
   - **sortInventory(): void sortInventory()**
     - Use: Sorts the items in the player's inventory alphabetically using a case-insensitive comparison (implemented with a lambda).
   - **displayStatus(): void displayStatus() const override**
     - Use: Overrides the base class method to display the player's name and health to the console.
   - **operator<<(ostream& os, const Player& player): ostream &operator<<(ostream &os, const Player &player)**
     - Use: An overloaded stream insertion operator that provides a comprehensive, formatted output of all player statistics to an output stream.

3. **Enemy (Derived from Character)**
   - **Constructor: Enemy(string n, string desc, int hp)**
     - Use: Initializes an enemy with a name, a description, and their health points (which also represents the health required to defeat them).
   - **getDescription(): string getDescription() const**
     - Use: Returns a descriptive string about the enemy.
   - **displayStatus(): void displayStatus() const override**
     - Use: Overrides the base class method to display the enemy's name and the health required to win against them to the console.

4. **Treasure**
   - **Constructor: Treasure(string i1, string i2, string k)**
     - Use: Initializes a treasure object with two items and a key that can be collected by the player.
   - **getItem1(): string getItem1() const**
     - Use: Returns the first item contained within the treasure.
   - **getItem2(): string getItem2() const**
     - Use: Returns the second item contained within the treasure.
   - **getKey(): string getKey() const**
     - Use: Returns the key associated with the treasure.
5. **Room**
   - **Constructor: Room(string n, Enemy e, Treasure t, string c)**
     - Use: Initializes a room with a unique name, an Enemy present, a Treasure to be found, and a specific challenge associated with the room.
   - **getName(): string getName() const**
     - Use: Returns the name of the room.
   - **getEnemy(): const Enemy& getEnemy() const**
     - Use: Returns a constant reference to the Enemy object residing in the room.
   - **getTreasure(): const Treasure& getTreasure() const**
     - Use: Returns a constant reference to the Treasure object found in the room.
   - **getChallenge(): string getChallenge() const**
     - Use: Returns the string describing the challenge of the room.
6. **GameAssetManager<T> (Templated Class)**
   - **addAsset(unique_ptr<T> asset): void addAsset(unique_ptr<T> asset)**

        - Use: Adds a new asset to the manager's internal collection. It takes ownership of the std::unique_ptr to ensure proper memory management.
    - **getAsset(size_t index): const T* getAsset(size_t index) const**
        - Use: Retrieves a constant pointer to the asset at the specified index. It includes error handling (out_of_range) to prevent invalid access.
    - **getAssetCount(): size_t getAssetCount() const**
        - Use: Returns the total number of assets currently managed.

7. **Dungeon**
    - **Constructor: Dungeon()**
        - Use: Initializes the dungeon by creating and populating the GameAssetManager with predefined Room objects and filling the enemyQueue.
    - **getRules(): string getRules() const**
        - Use: Returns a multi-line string containing the game's rules and objectives.
    - **getCurrentRoom(): const Room* getCurrentRoom() const**
        - Use: Returns a constant pointer to the room the player is currently in.
    - **advanceToNextRoom(): const Room* advanceToNextRoom()**
        - Use: Moves the player to the next room in the dungeon sequence. It pushes the current room onto a stack to enable backtracking.
    - **backtrack(): const Room* backtrack()**
        - Use: Allows the player to return to the previously visited room by popping from the room stack.
    - **displayRanking(const Player& player): void displayRanking(const Player& player) const**
        - Use: Displays the final game over ranking and the player's complete statistics to the console.

8. **GUI**

- **Constructor: GUI()**
  - Use: Initializes the SFML rendering window, attempts to load the necessary font, and sets up all static UI elements (buttons, text fields, panels).
- **isOpen(): bool isOpen() const**
  - Use: Checks if the SFML window is currently open.
- **close(): void close()**
  - Use: Closes the SFML rendering window.
- **pollEvent(sf::Event& event): bool pollEvent(sf::Event &event)**
  - Use: Retrieves pending SFML events from the event queue (e.g., mouse clicks, key presses).
- **getPlayerName(): string getPlayerName() const**
  - Use: Returns the player's name that was entered via the GUI.
- **handleEvent(const sf::Event& event, GameState& gameState, int& choice): void handleEvent(const sf::Event &event, GameState &gameState, int &choice)**
  - Use: Processes a given SFML event and updates the gameState or the player's choice based on the event type and current game state.
- **update(GameState gameState, const Player& player, const Room* room, const string& message): void update(GameState gameState, const Player &player, const Room *room, const string &message)**
  - Use: Updates dynamic UI elements like button hover states and calls updateStatus to refresh player and room information.
- **draw(GameState gameState, const string& rules, const string& gameOverMessage, const Player& player): void draw(GameState gameState, const string &rules, const string &gameOverMessage, const Player &player)**
  - Use: Clears the window and draws all appropriate UI elements for the current gameState, including rules, game over messages, and player stats.

- **setupUI() (private): void setupUI()**
    - Use: A private helper method called by the constructor to set up initial positions, sizes, fonts, and strings for all GUI components.
- **updateStatus(const Player& player, const Room* room, const string& message) (private): void updateStatus(const Player &player, const Room *room, const string &message)**
    - Use: A private helper method to refresh the text strings displayed in the game's status panel (player health, moves, current room, enemy, inventory, etc.).
- **drawGameOver(const string& message, const Player& player) (private): void drawGameOver(const string &message, const Player &player)**
    - Use: A private helper method dedicated to rendering the game over screen, which now includes the final player statistics.
- **centerOrigin(sf::Text& text) (private): void centerOrigin(sf::Text &text)**
    - Use: A private utility function to center the origin of an sf::Text object, simplifying positioning.

The game initializes in main() by creating a GUI object and a Dungeon object.

1. **Name Input:** The game starts in the NAME_INPUT state. The GUI handles text input, allowing the player to enter their name. Once entered, the Player object is created with the provided name.
2. **Instructions:** The game transitions to the INSTRUCTIONS state, displaying game rules. A "Start Game" button allows the player to proceed.
3. **Game Loop (gameLoopWithGUI):** This is the heart of the game, running continuously as long as the GUI window is open.
    - **Event Polling:** It constantly checks for user input events (mouse clicks, keyboard presses, window close).
    - **Game Logic:** Based on the current GameState and player actions (choice), the game logic updates:

        - **Room Navigation:** dungeon.advanceToNextRoom() moves the player forward, pushing the previous room onto a stack. dungeon.backtrack() pops rooms from the stack to move back.
        - **Combat (Fight):** If the player fights, their health is checked against the enemy's. Success leads to health reduction, treasure, coins, and enemies defeated. Failure results in damage and fleeing.
        - **Bypass:** The player takes minor damage but moves to the next room.
        - **Quit:** The game immediately transitions to GAME_OVER.
        - **Win/Loss Conditions:** The game continuously checks if the player has run out of rooms (win), run out of health (lose), or run out of moves (lose).
    - **GUI Update (gui.update):** This function updates the text strings for player status, room information, and any messages. It also handles button hover effects.
    - **GUI Drawing (gui.draw):** This function clears the screen and redraws all relevant UI elements based on the current GameState. For PLAYING, it shows status panels and action buttons. For GAME_OVER, it displays the game over message and detailed player statistics.
4. **Game Over:** When the game ends, the GAME_OVER state is set. The GUI displays the final message and the player's stats. The window waits for a key press or mouse click to close.
5. **Exit:** After the GUI window closes, final player statistics are also printed to the console (as a fallback/confirmation), and the program terminates.

7. Specific C++ Features

- **Sorting Algorithms:**
    - The Player::sortInventory() method uses std::list::sort(). This is an efficient, built-in sorting algorithm for std::list that directly modifies the list in place.
    - A lambda expression is provided as a custom comparison predicate to ensure case-insensitive sorting of inventory items.
- **Templates: Generic containers or functions:**

  - **GameAssetManager<T>:** This is a class template. It allows the GameAssetManager to store and manage std::unique_ptr to *any* type T, making it reusable for different game assets (e.g., Room, Enemy if needed) without rewriting the management logic.
  - **Player::addToInventory<T>(const T& item):** This is a templated member function. It allows the player to add items of various types (T) to their inventory. The item is converted to a std::string using stringstream before being added to the std::list<std::string> inventory.
- **Exception Handling:**
  - The code uses try-catch blocks to gracefully handle potential runtime errors.
  - **std::out_of_range:** Used in GameAssetManager::getAsset and within the Dungeon class (when retrieving rooms or populating enemy queue) to catch invalid index accesses. This prevents the program from crashing and allows for error logging.
  - **std::runtime_error:** Used in the GUI constructor to catch errors during font loading (font.loadFromFile). If the font cannot be loaded, an error message is printed, and the GUI can still attempt to run (though text won't display correctly).
- **STL Containers:**
  - std::vector (for GameAssetManager::assets)
  - std::list (for Player::inventory)
  - std::queue (for Dungeon::enemyQueue)
  - std::stack (for Dungeon::roomStack)
  - std::string (for various textual data)
  - std::stringstream (for formatting text, e.g., inventory display)
- **Lambdas:**
  - A lambda function is used in Player::sortInventory():

C++

```
inventory.sort([](const string &a, const string &b) {
  string lowerA, lowerB;
```

```
    transform(a.begin(), a.end(), back_inserter(lowerA), ::tolower);
    transform(b.begin(), b.end(), back_inserter(lowerB), ::tolower);
    return lowerA < lowerB;
});
```

This lambda defines a custom comparison logic for sorting strings in a case-insensitive manner, which is passed directly to std::list::sort().