Choose Your Own Adventure Game In C++

By: William Newman

Thoughtful Selection of Variable, Function, and Class Names

Classes:

- AdventureGame, Wall, Inventory, DeskWall, BookcaseWall, DoorWall, WindowWall
- Reflects the purpose and nature of objects in the game (e.g., Wall represents different types of walls).

Functions:

- start(), displayRoom(), showCurrentWall(), openLobby(), getTimeRemaining()
- Descriptive names, making it clear what each function does.

Variables:

- currentWall, gameOver, inventory, startTime
- Variables are intuitive and self-descriptive, aiding in code readability.

Division of Code into Multiple Header and Source Files

Headers:

- AdventureGame.h, Wall.h, BookcaseWall.h, DeskWall.h, DoorWall.h, WindowWall.h, Inventory.h

Source Files:

- AdventureGame.cpp, BookcaseWall.cpp, DeskWall.cpp, DoorWall.cpp, WindowWall.cpp, Inventory.cpp

Rationale:

- Each class is separated into its own header and source file, making the code modular, maintainable, and easier to debug.

If Statements

Usage in AdventureGame::start() and AdventureGame::showCurrentWall()::

```
if (action == "l") {
    currentWall = (currentWall > 1) ? currentWall - 1 : 4; // Rotate left
    showCurrentWall();
} else if (action == "r") {
    currentWall = (currentWall < 4) ? currentWall + 1 : 1; // Rotate right
    showCurrentWall();
} else if (action == "w") {
    showCurrentWall();  // Show current wall
    if (currentWall == 1) {
        wall1->inspect();  // Inspect DeskWall
        if (!wall1->item.empty()) {
            inventory.addItem(wall1->item); // Add item to inventory
```

Purpose:

- Handles different game actions, checks user input validity, and controls the flow based on conditions (e.g., player choosing to open a door).

Loops: While, Do-While, and For

While Loop:

- Used in AdventureGame::start() to keep the game running while gameOver is false.

Do-While Loop:

- Ensures the player is ready before starting the game.

For Loop:

- Used in Inventory::display() to iterate through items in the inventory.

```
// Game loop runs until gameOver is true
while (!gameOver) {
    displayRoom();  // Display room status and options
    try {
        cin >> action;
        cin.ignore();
        if (action == "l" || action == "r" || action == "w" || action == "i" || action == "e") {
            // Valid actions
        } else {
            throw invalid_argument("Invalid action. Choose from the available options.");
        }
    } catch (const invalid_argument& e) {
        cout << e.what() << endl; // Handle invalid input
    }
```

```
    // Prompt the player to start the game
    do {
        cout << "Are you ready to begin the adventure? (yes/no): ";
        cin >> action;
        if (action == "yes") {
            ready = true;
        } else if (action == "no") {
            cout << "Take your time to prepare.\n";
        } else {
            cout << "Invalid input. Please enter 'yes' or 'no'.\n";
        }
    } while (!ready);
```

```
// Displays the inventory contents
void Inventory::display() const {
    cout << "\nInventory: ";
    if (items.empty()) { // Check if inventory is empty
        cout << "Empty";
    } else {
        for (size_t i = 0; i < items.size(); ++i) { // Loop through items
            cout << items[i];
            if (i != items.size() - 1) cout << " | "; // Add separator
        }
    }
    cout << "\n";
}
```

Functions

Examples of Functions in AdventureGame:

- void start() - Main game loop.
- void displayRoom() - Displays current room information.
- void showCurrentWall() - Displays ASCII art for the current wall.

Function Purpose:

- Modularizes the code for easier readability and reusability

Class Definitions

Class: AdventureGame

- Responsible for managing the game state and interactions.

Class: Wall and its subclasses (e.g., DeskWall, BookcaseWall)

- Encapsulates wall-specific functionality like description and items.

Class: Inventory

- Manages the player's inventory, adding and saving items.

Constructors and Destructors

Constructors:

- AdventureGame::AdventureGame() initializes the game state.
- Wall::Wall() initializes wall properties.
- BookcaseWall::BookcaseWall() and others initialize specific wall types.

Destructors:

- Wall::~Wall() is virtual, ensuring proper cleanup of derived classes.

Operator Overloading

Overloaded << Operator in Wall Class:

- Used to display wall information.

Example:

```
// Overload the stream insertion operator to print Wall details
ostream& operator<<(ostream& os, const Wall& wall) {
    os << "Description: " << wall.description << endl; // Print wall description
    if (!wall.item.empty()) { // Print the item if it exists
        os << "You find a " << wall.item << " on this wall.\n";
    }
    return os;
}
```

Object Composition

Composition in AdventureGame:

- AdventureGame contains objects like wall1, wall2, etc. (using unique_ptr for dynamic memory management).
- Represents "has-a" relationships, e.g., an AdventureGame "has-a" Wall.

Inheritance

Example:

- DeskWall, BookcaseWall, DoorWall, and WindowWall inherit from the Wall class.
- Polymorphism allows different wall types to share common behavior while maintaining specific functionality.

Exceptions

Usage in AdventureGame::start():

- Catches invalid actions using try-catch blocks to handle exceptions.

```
// Game loop runs until gameOver is true
while (!gameOver) {
    displayRoom();  // Display room status and options
    try {
        cin >> action;
        cin.ignore();
        if (action == "l" || action == "r" || action == "w" || action == "i" || action == "e") {
            // Valid actions
        } else {
            throw invalid_argument("Invalid action. Choose from the available options.");
        }
    } catch (const invalid_argument& e) {
        cout << e.what() << endl; // Handle invalid input
    }
```

File I/O

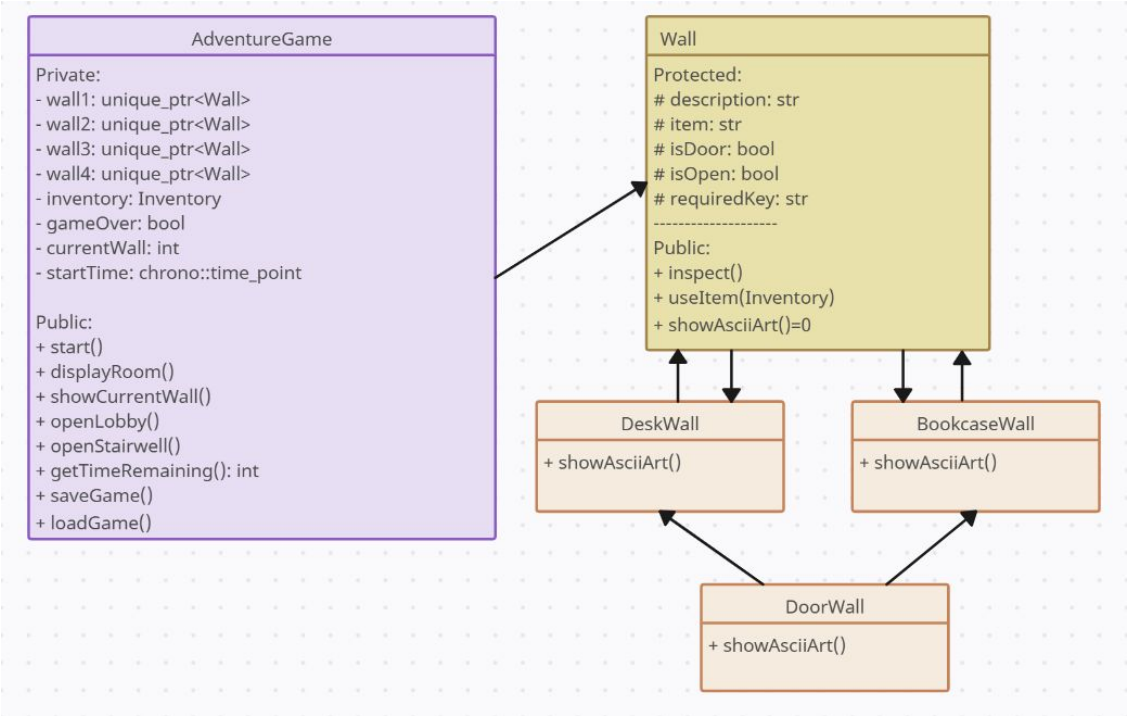
Saving and Loading Inventory:

- Inventory::saveToFile("inventory.txt") saves inventory to a file.
- Inventory::loadFromFile("inventory.txt") loads inventory from a file.

```
// Save inventory to file
void AdventureGame::saveGame() {
    inventory.saveToFile("inventory.txt");
}

// Load inventory from file
void AdventureGame::loadGame() {
    inventory.loadFromFile("inventory.txt");
}
```

UML Diagram


AdventureGame
Private:
- wall1: unique_ptr<Wall>
- wall2: unique_ptr<Wall>
- wall3: unique_ptr<Wall>
- wall4: unique_ptr<Wall>
- inventory: Inventory
- gameOver: bool
- currentWall: int
- startTime: chrono::time_point
Public:
+ start()
+ displayRoom()
+ showCurrentWall()
+ openLobby()
+ openStairwell()
+ getTimeRemaining(): int
+ saveGame()
+ loadGame()
Wall
Protected:
description: str
item: str
isDoor: bool
isOpen: bool
requiredKey: str

Public:
+ inspect()
+ useItem(Inventory)
+ showAsciiArt()=0
DeskWall
+ showAsciiArt()
BookcaseWall
+ showAsciiArt()
DoorWall
+ showAsciiArt()

C++ Command/Concept Not Covered in Class

make_unique (C++11):

- Used to create unique_ptr for automatic memory management.

Custom Definition to be compatible with C++11:

```cpp
#include <memory> // For std::unique_ptr and make_unique

// Define make_unique for C++11 compatibility
template <typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

User Interaction Flow

- Players can choose actions such as:
    - l: Turn left
    - r: Turn right
    - w: Inspect the current wall
    - i: View inventory
    - e: Open a door (if available)

- Interaction based on game state and the player's inventory (e.g., requiring keys to open doors).

Game Loop and Time Management

Game Loop:

- Continuously prompts the player for actions and updates the game state until the game ends (either by leaveing or running out of time).

Time Management:

- The game has a 3-minute countdown (represented as getTimeRemaining()).

- If the time runs out, the game ends with a message: "Time's up! You didn't leave in time."

Conclusion

- Questions?