Programming Project Report

Title:

Descent into the Infernal Abyss

Name: Syed Muhammad Hassaan Bin Ghayas

Project Title: Descent into the Infernal Abyss

Contents

Introduction and Description of Software Prototype Application

"Descent into the Infernal Abyss" is an immersive text-based horror adventure game that plunges players into a cursed dungeon. Players assume the role of a nameless wanderer trapped in the depths of the Eternal Void and must navigate through layers of terror to escape. The journey is fraught with cursed entities, shadowy hazards, and eerie encounters, testing the player's resilience and strategic planning.

The game begins with a haunting menu setting the adventure's tone, offering options to view the Infernal Guide, start the game, save/load progress, or exit. Upon entering the abyss, players are introduced to their character's core attributes: **Health**, **Sanity**, **Fear Level**, and **Tenacity**. These attributes are vital to survival, influencing how players interact with the world and respond to challenges.

The dungeon is structured as a series of interconnected levels, modelled as nodes in a **DungeonTree**, a binary tree representation. Each node corresponds to a dungeon level, such as the *Hall of Shadows*, *Pit of Despair*, or *Eternal Void*. These levels are sequentially linked, creating a harrowing path for players to traverse. The player's Fear Level influences progression through the dungeon. At each branch, the game evaluates the player's fear:

- **Low Fear Levels** unlock less hazardous paths (e.g., "Chamber of Echoes").
- **High Fear Levels** force players into perilous regions (e.g., "Pit of Despair").

This dynamic path selection adds unpredictability and forces players to manage their fear effectively. Every level is unique, hosting a variety of hazards, cursed entities, and items to discover. Entities like **Demon Guards** and **Lost Souls** present dynamic interactions, while hazards such as the **Wall of Shadows** and **Maw of Darkness** obstruct progress, requiring players to use their cursed skills and items strategically.

To interact with the dungeon's elements, the game employs a **Visitor Pattern** for dynamic interactions. For example, encountering a Lost Soul triggers text-based choices like offering a soul shard, fleeing, or confronting the entity. Each decision impacts the player's attributes and progress. **Hazards** like the **Wall of Shadows** demand skill checks, such as climbing requiring high strength, while items like **Soul Fragments** or **Potions** offer aid in dire moments. The randomness of fear events adds unpredictability, keeping players on edge as they navigate the dungeon.

One of the game's central mechanics is **soul shard collection**, a crucial progression aspect. Players must gather four soul shards scattered across the dungeon, stored in a **stack**. This data structure ensures that shards are inserted into the Eternal Void portal in reverse order of collection, emphasizing the player's memory and strategy. Without all four shards, the portal remains sealed, dooming the player to eternal torment.

The player's inventory is managed as a **doubly linked list**, including cursed items like **Hexblades** and **Dark Elixirs**. These items provide benefits and drawbacks, embodying the game's theme of "power at a cost." For instance, the Hexblade increases tenacity but raises the Fear Level. Additionally, cursed skills are stored in another doubly linked list, allowing players to enhance abilities like **Intellect**, **Survivor**, or **Brute Force**. These skills influence outcomes, such as negotiating with a Lost Soul or overcoming a hazard.

The **Fear Mechanic** introduces a dynamic difficulty element, with randomly triggered events like auditory jump scares or chilling descriptions. Fear increases unpredictably, reducing sanity recovery and heightening vulnerability to hazards. For instance, entering the Hall of Shadows might result in a sudden scream echoing through the abyss, increasing the Fear Level and diminishing the player's composure.

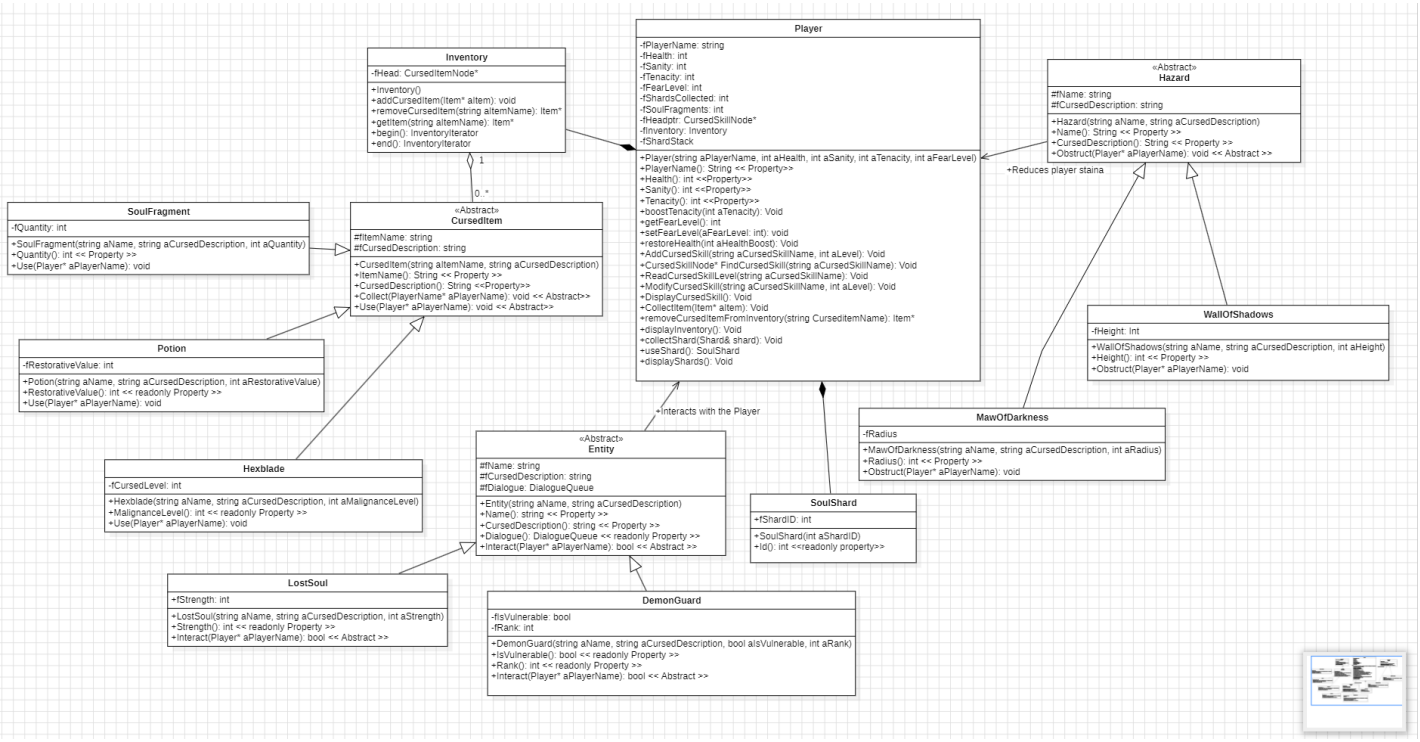
As players delve deeper, the game periodically offers choices to inspect attributes, view the inventory, or proceed. These decisions shape the narrative, allowing players to engage in strategic planning. For instance, using a Potion to restore sanity might be critical before facing a DemonGuard. The tension rises as players approach the *Eternal Void*, where the final test awaits: offering the collected soul shards to unlock the portal and escape.

The game's immersive narrative unfolds through descriptive dialogues and text-based interactions, complemented by **SFML sound integration**. Eerie background music and sudden jump scares amplify the atmosphere, immersing players in the horror. Combining rich gameplay mechanics, strategic decision-making, and dynamic storytelling creates a gripping experience that keeps players engaged until the final moment.

The end goal is clear yet challenging: survive the abyss, collect the shards, and unlock the portal. Success hinges on resource management, strategic thinking, and the player's ability to withstand the terrors of the cursed dungeon. *Descent into the Infernal Abyss* delivers a thrilling blend of horror and strategy, offering players a uniquely chilling adventure.

DESCENT INTO THE INFERNAL ABYSS

UML Diagram:



Game Menu:

```
C:\Users\kingh\source\repos
------------------------------------------------------------------------
                DESCENT INTO THE INFERNAL ABYSS
------------------------------------------------------------------------

The void whispers your name... Can you escape the horror?
++++++++++++++
GAME MENU
++++++++++++++
1. View the Infernal Guide
2. Enter the Abyss
3. Save Game
4. Load Game
5. Embrace the Darkness (Exit)
Enter your choice: 1
======================================================
                INFERNAL GUIDE
======================================================
|                                                    |
|   Welcome, wanderer. The abyss beckons.            |
|   Prepare yourself for the horrors ahead.          |
|                                                    |
======================================================

+--------------------------------------------------+
|                   OBJECTIVE                      |
+--------------------------------------------------+
| Traverse the cursed dungeon, gathering soul      |
| shards to unlock the Eternal Void. Escape        |
| its horrors... or become one with them.          |
+--------------------------------------------------+

+--------------------------------------------------+
|              CURSED DUNGEON                      |
+--------------------------------------------------+
| A labyrinth of despair filled with shadows,      |
| traps, and cursed entities. Each level           |
| offers danger and reward. Tread carefully.       |
+--------------------------------------------------+

+--------------------------------------------------+
|              SOUL SHARDS                         |
+--------------------------------------------------+
| Scattered remnants of tortured souls.            |
| Collect them to unlock the portal. Beware,       |
| they are heavily guarded by the abyss.           |
+--------------------------------------------------+

+--------------------------------------------------+
|       HEALTH, SANITY, AND FEAR                   |
+--------------------------------------------------+
| - **Health:** Stay alive by avoiding traps       |
|   and surviving encounters.                      |
| - **Sanity:** Hold onto your mind; losing        |
|   it will leave you vulnerable.                  |
```

CONCEPTS

1. Object-Oriented Programming:

a) Inheritance and Derived Classes

Inheritance plays a pivotal role in structuring the core gameplay mechanics of *Descent into the Infernal Abyss*. All dungeon elements, such as hazards and entities, are encapsulated in the Entity base class. This class defines shared properties like name and description and virtual functions for interaction, ensuring a consistent interface across all derived classes.

The derived classes, such as DemonGuard, LostSoul, and Hazard, extend the base Entity class, introducing specialized behaviors tailored to the gameplay. For example, the DemonGuard class adds properties like strength and a combat sequence, while the LostSoul class introduces sanity-draining effects. This structure enables seamless integration of new entity types without modifying the existing game logic.

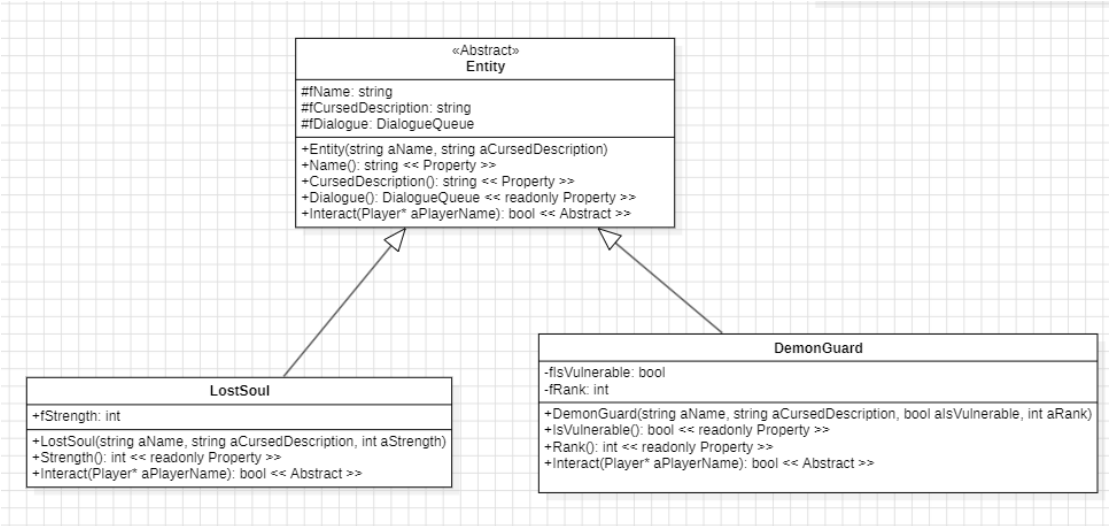
Application in Gameplay:

Inheritance simplifies the representation of the cursed dungeon's inhabitants. Whenever the player encounters an entity in the dungeon, the Entity pointer invokes the appropriate interaction dynamically, whether it is a combat trial with a DemonGuard or a sanity test with a LostSoul. This dynamic behavior is crucial to maintaining the unpredictability and replay ability of the game.

Alternative Structures and Justification:

An alternative to inheritance could be using separate, unrelated classes for each entity type. However, this approach would lead to code duplication and rigid logic, as every entity would require standalone implementation. In contrast, inheritance consolidates shared behaviors in the base class, enabling scalability and code reuse C++ Inhetorial. (*Cplusplus, 2023)*.

Implementation and Use Case:



Entity Hierarchy

The entity hierarchy centers on the abstract Entity class, which represents all cursed beings encountered in the dungeon. Derived classes include:

- **LostSoul**: An entity that drains the player's sanity, represented by its strength attribute.
- **DemonGuard**: A combat-focused entity with attributes such as isVulnerable and rank, introducing dynamic interactions based on the player's strategy.

The Interact() method in each derived class defines specific behaviors during encounters, emphasizing the use of polymorphism for dynamic gameplay interactions.

Implementation and Use Case

1. **Base Class Implementation:**

```cpp
#pragma once

#include <string>
#include "Player.h"
#include "DialogueQueue.h"

// Represents a base class for all entities in the game.
class Entity {
protected:
    std::string fName;              // Name of the entity.
    std::string fCurseDescription;  // Description of any curse or lore associated with the entity.
    DialogueQueue fDialogue;        // Queue of dialogues linked to this entity.

public:
    // Default constructor to initialize the entity with default values.
    Entity();

    // Constructor to initialize the entity with a specific name and curse description.
    Entity(const std::string& aName, const std::string& aCurseDescription);

    // Virtual destructor to ensure proper cleanup of resources in derived classes.
    virtual ~Entity();

    // Retrieves the name of the entity.
    std::string getName() const;

    // Retrieves the curse description or lore tied to the entity.
    std::string getCurseDescription() const;

    // Provides access to the dialogue queue associated with the entity.
    DialogueQueue& getDialogueQueue();

    // Defines interaction behavior when the entity engages with a player.
    // Must be implemented by derived classes.
    virtual bool interact(Player* aPlayer) = 0;
};
```

2. **Derived Class: DemonGuard**

Programming Project (Global Scope)

```h
#pragma once

#include <string>
#include "Entity.h"
#include "Visitor.h"
#include "DialogueQueue.h"

// Represents a Demon Guard entity in the game, derived from the base Entity class.
class DemonGuard : public Entity {
private:
    bool fIsVulnerable;  // Indicates whether the Demon Guard can be influenced or attacked.
    int fRank;           // Rank of the Demon Guard within its hierarchy.

public:
    // Default constructor to initialize the Demon Guard with default values.
    DemonGuard();

    // Constructor to initialize the Demon Guard with a name, curse description, vulnerability status, and rank.
    DemonGuard(const std::string& aName, const std::string& aCurseDescription, bool aIsVulnerable, int aRank);

    // Destructor to clean up resources.
    ~DemonGuard();

    // Checks if the Demon Guard is vulnerable to interactions or attacks.
    bool isVulnerable() const;

    // Retrieves the rank of the Demon Guard.
    int getRank() const;

    // Handles interaction logic when a player encounters the Demon Guard.
    bool interact(Player* aPlayer) override;
};
```

3. Derived Class: LostSoul

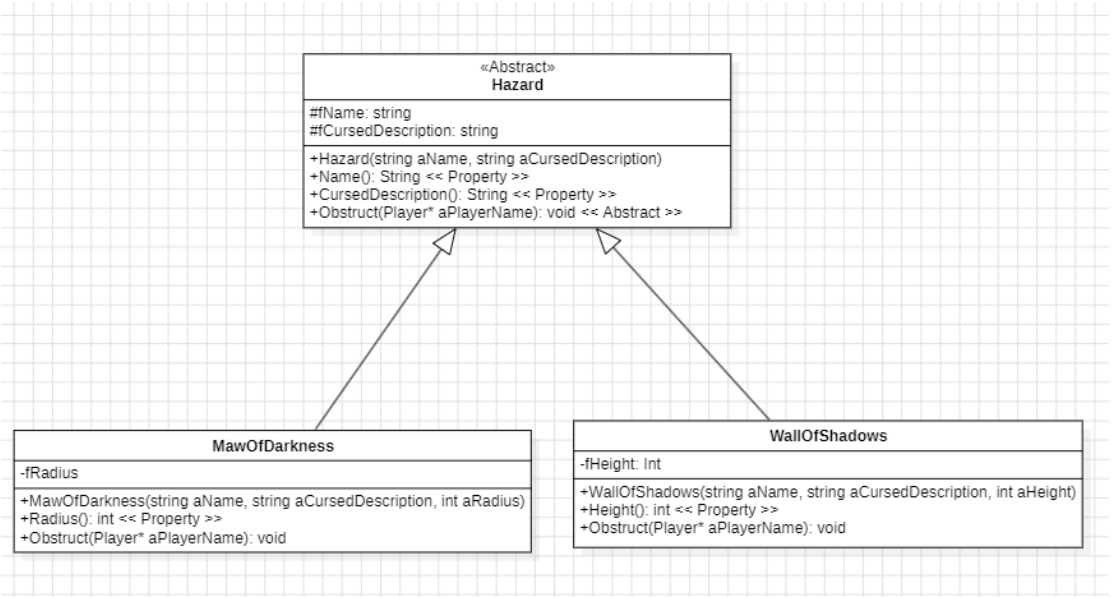
Programming Project (Global Scope)

```h
#pragma once

#include <iostream>
#include <string>
#include "Entity.h"
#include "DialogueQueue.h"
#include "Visitor.h"

// Represents a lost soul in the game, inheriting from the base Entity class.
class LostSoul : public Entity {
private:
    int fStrength;       // Strength or combat ability of the LostSoul.
    int fHauntLevel;     // Haunting level of the LostSoul (formerly stealth level).
    DialogueQueue fDialogue; // Dialogue associated with the LostSoul.

public:
    // Default constructor to initialize a generic LostSoul.
    LostSoul();

    // Constructor to initialize a LostSoul with specific attributes.
    LostSoul(const string& aName, const string& aCurseDescription, int aStrength, int aHauntLevel);

    // Destructor to clean up resources associated with the LostSoul.
    ~LostSoul();

    // Retrieves the strength of the LostSoul.
    int getStrength() const;

    // Overrides the interaction behavior to define how a LostSoul interacts with a player.
    bool interact(Player* aPlayer) override;
};
```

Testing and Output:

```
Currently at Maw of Darkness

A chill sweeps over you as you encounter the tormented presence of Mournful Wraith.
Mournful Wraith: Spare me, mortal... a shard of your soul... or suffer my eternal curse.
Choose your fate:
1: Offer a shard of your soul to ease the torment.
2: Flee in terror, hoping the soul won't catch you.
3: Stand firm against the soul's overwhelming presence.
Your response (1/2/3): 2
Your legs falter as the soul's icy grip clutches at you. You cannot escape.
The soul's wail haunts you... you feel your strength waning.
```

Other Implementations:

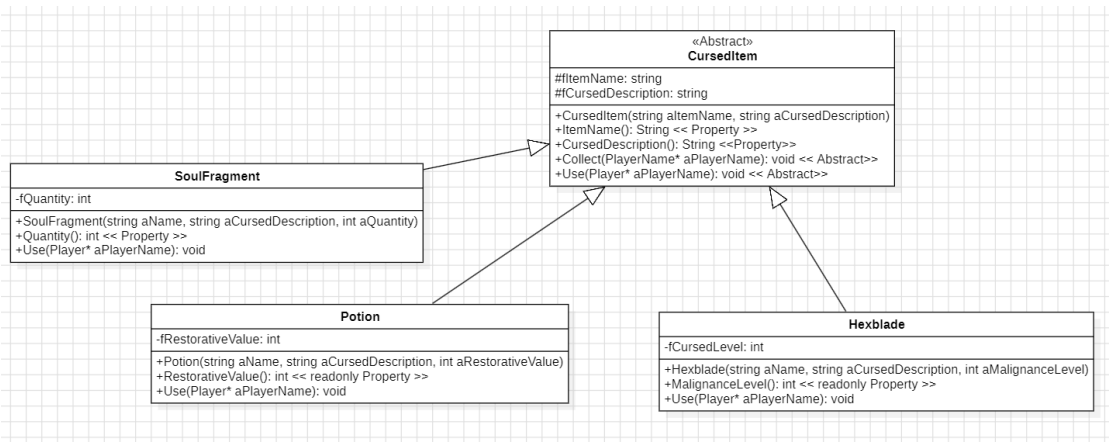


Hazard Hierarchy

The hazard hierarchy is modeled using an abstract base class Hazard, representing all potential obstacles in the dungeon. Derived classes include:

- **MawOfDarkness**: A hazard with a specific radius that drains the player's health and sanity.
- **WallOfShadows**: A towering obstacle requiring skill checks to bypass, with a specific height attribute.

This hierarchy ensures modularity, as new hazards can be added by extending the Hazard base class. The Obstruct() method in derived classes provides specific behavior for interacting with the player.



Cursed Item Hierarchy

The cursed item hierarchy is built around the abstract CursedItem class, representing all collectible items in the game. Derived classes include:

- **SoulFragment**: A collectible item required to unlock the final portal, with a quantity attribute.
- **Potion**: A restorative item that replenishes health or sanity, defined by its RestorativeValue.
- **Hexblade**: A weapon with a CursedLevel, offering high damage at the cost of increased fear.

This structure ensures flexibility in adding new items and provides distinct functionality for each type through the Use() method.

Troubleshooting Summary

1. **Issues:**
   There were no issues faced during inheritance as it was covered detailed in OOP and the recap of this in DSP.
2. **Resources Used:**
   No external references were required Lecture Slides and links provided on tutorial were significant help.

- 2023, Cplusplus.com, viewed 16 November 2024,<https://cplusplus.com/doc/tutorial/inheritance/>.

b) Polymorphism

Polymorphism is central to managing hazards within *Descent into the Infernal Abyss*. Hazards like the **Maw of Darkness** and **Wall of Shadows** create dynamic challenges for the player, affecting key attributes such as health, sanity, or progress. These hazards are modelled using an abstract base class Hazard, which defines common properties and behaviours. Derived classes override the Obstruct() method to implement specific hazard interactions.

The use of polymorphism allows the game to treat all hazards uniformly while maintaining the ability to invoke unique behaviours at runtime. For example:

- A **Maw of Darkness** drains the player's health and sanity based on its radius attribute.
- A **Wall of Shadows** challenges the player to overcome a skill-based obstacle, requiring a check against its height.

Application in Gameplay:

During dungeon traversal, the player encounters hazards at various levels. These hazards are stored as pointers to the base Hazard class. At runtime, the correct derived class behaviour is invoked dynamically through polymorphism. This ensures that the game logic remains clean and extensible, as new hazard types can be added without modifying existing code.

For instance:

- When encountering a **Maw of Darkness**, the Obstruct() method reduces the player's health and sanity.
- When encountering a **Wall of Shadows**, the Obstruct() method prompts the player to perform a skill check to bypass the obstacle.

This dynamic behaviour enhances gameplay by creating varied and unpredictable challenges for the player.

Alternative Structures and Justification

Without polymorphism, each hazard type would require explicit type-checking (e.g., if or switch statements) to determine the appropriate behaviour. However, this approach introduces significant complexity and tight coupling between components, making the code less maintainable. Polymorphism offers a cleaner, more extensible solution by enabling derived classes to implement their unique behaviours while adhering to a common interface defined in the base class.

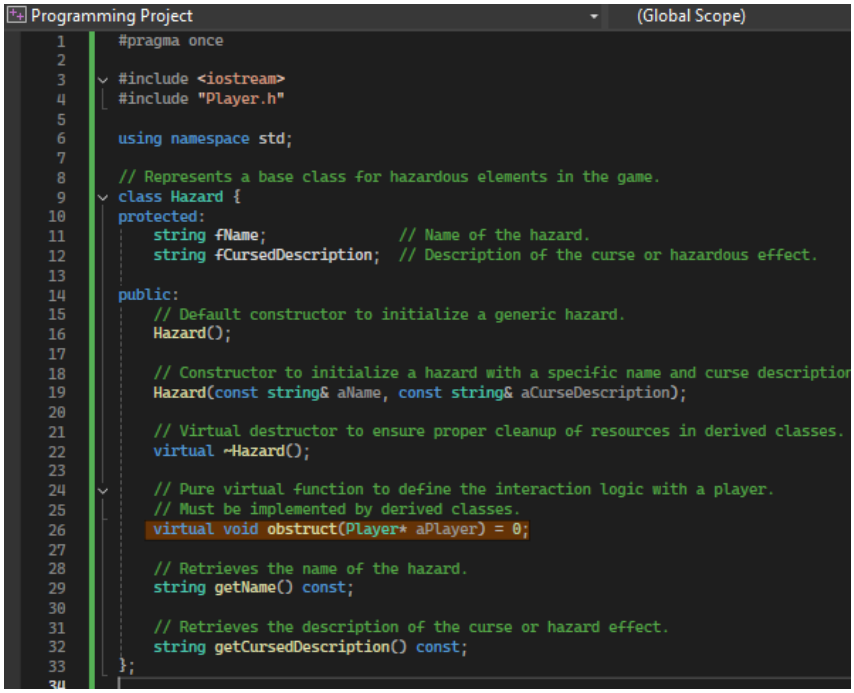
GeeksforGeeks highlights that polymorphism simplifies code design, improves maintainability, and makes the system extensible by avoiding rigid and error-prone type-checking constructs like if or switch statements. It ensures that behaviours are dynamically dispatched at runtime, which is particularly useful in applications like gaming, where diverse objects need to interact dynamically (GeeksforGeeks, 2017).

Diagram Representation:

Refer to **Hazard Hierarchy** pasted above.

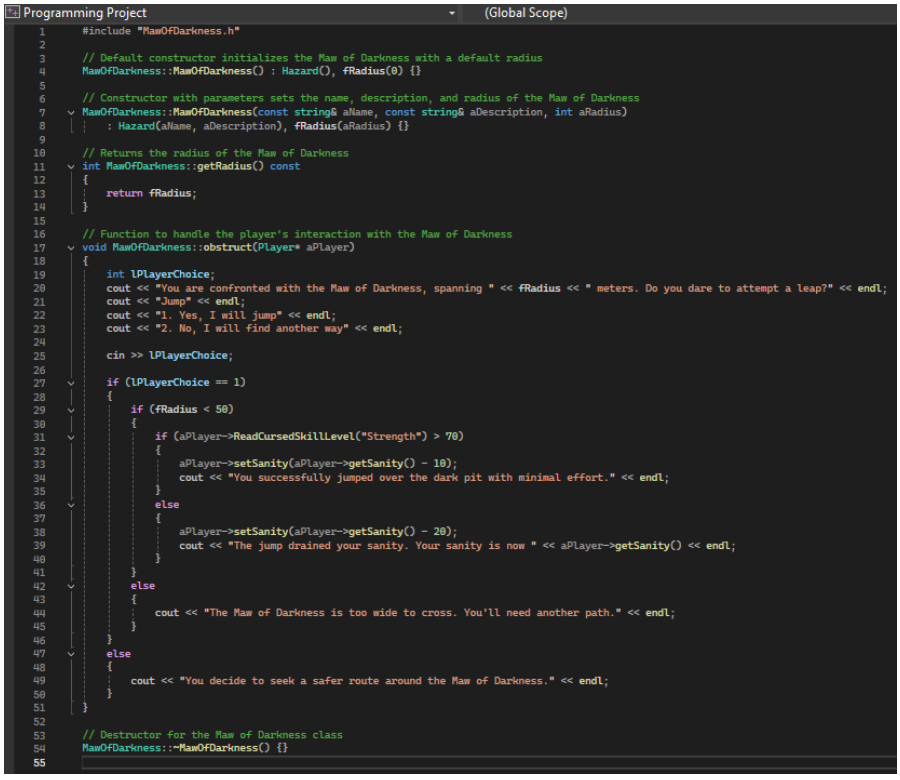
Implementation:

Base Class Implementation (Hazard)



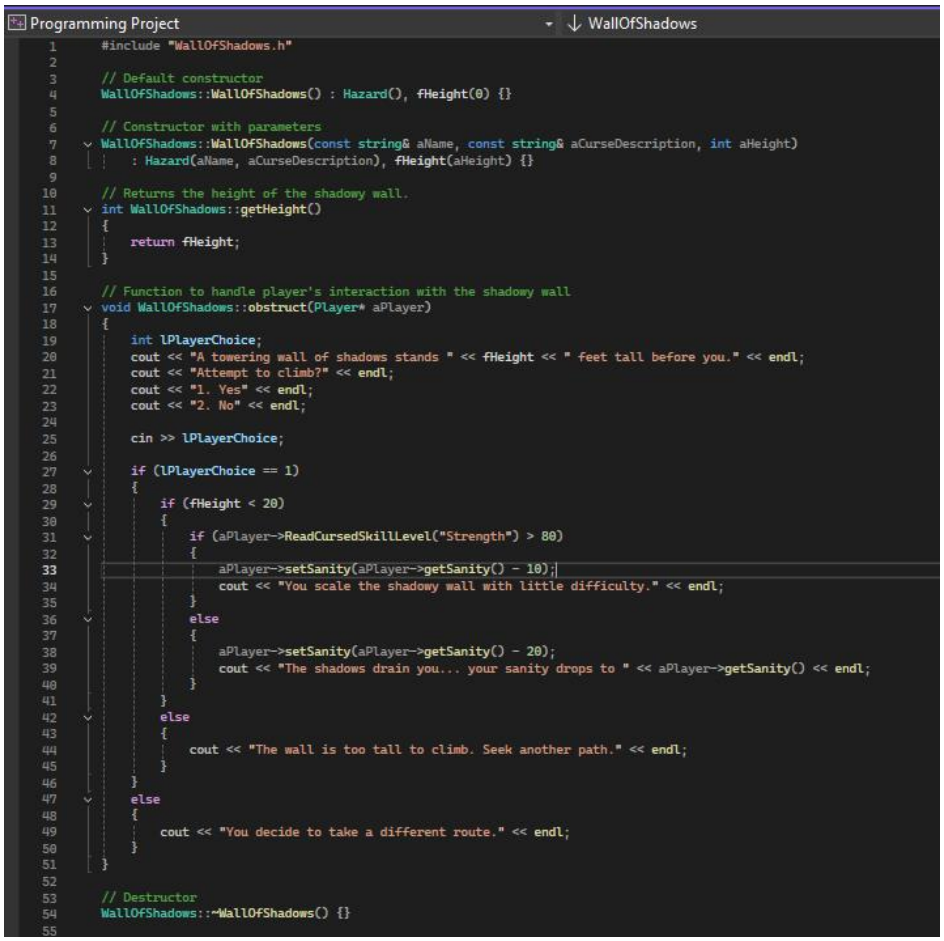

```cpp
#pragma once

#include <iostream>
#include "Player.h"

using namespace std;

// Represents a base class for hazardous elements in the game.
class Hazard {
protected:
    string fName;              // Name of the hazard.
    string fCursedDescription;  // Description of the curse or hazardous effect.

public:
    // Default constructor to initialize a generic hazard.
    Hazard();

    // Constructor to initialize a hazard with a specific name and curse description
    Hazard(const string& aName, const string& aCurseDescription);

    // Virtual destructor to ensure proper cleanup of resources in derived classes.
    virtual ~Hazard();

    // Pure virtual function to define the interaction logic with a player.
    // Must be implemented by derived classes.
    virtual void obstruct(Player* aPlayer) = 0;

    // Retrieves the name of the hazard.
    string getName() const;

    // Retrieves the description of the curse or hazard effect.
    string getCursedDescription() const;
};
```

Derived Class: MawOfDarkness




```cpp
#include "MawOfDarkness.h"

// Default constructor initializes the Maw of Darkness with a default radius
MawOfDarkness::MawOfDarkness() : Hazard(), fRadius(0) {}

// Constructor with parameters sets the name, description, and radius of the Maw of Darkness
MawOfDarkness::MawOfDarkness(const string& aName, const string& aDescription, int aRadius)
    : Hazard(aName, aDescription), fRadius(aRadius) {}

// Returns the radius of the Maw of Darkness
int MawOfDarkness::getRadius() const
{
    return fRadius;
}

// Function to handle the player's interaction with the Maw of Darkness
void MawOfDarkness::obstruct(Player* aPlayer)
{
    int lPlayerChoice;
    cout << "You are confronted with the Maw of Darkness, spanning " << fRadius << " meters. Do you dare to attempt a leap?" << endl;
    cout << "Jump" << endl;
    cout << "1. Yes, I will jump" << endl;
    cout << "2. No, I will find another way" << endl;

    cin >> lPlayerChoice;

    if (lPlayerChoice == 1)
    {
        if (fRadius < 50)
        {
            if (aPlayer->ReadCursedSkillLevel("Strength") > 70)
            {
                aPlayer->setSanity(aPlayer->getSanity() - 10);
                cout << "You successfully jumped over the dark pit with minimal effort." << endl;
            }
            else
            {
                aPlayer->setSanity(aPlayer->getSanity() - 20);
                cout << "The jump drained your sanity. Your sanity is now " << aPlayer->getSanity() << endl;
            }
        }
        else
        {
            cout << "The Maw of Darkness is too wide to cross. You'll need another path." << endl;
        }
    }
    else
    {
        cout << "You decide to seek a safer route around the Maw of Darkness." << endl;
    }
}

// Destructor for the Maw of Darkness class
MawOfDarkness::~MawOfDarkness() {}
```

Derived Class: WallOfShadows



Testing and Output:

```
You gather your resolve and move deeper into the abyss...

Currently at Desolate Chamber
A towering wall of shadows stands 36 feet tall before you.
Attempt to climb?
1. Yes
2. No
1
The wall is too tall to climb. Seek another path.
```

Troubleshooting Summary

1. **Issues:**
   Polymorphism was implemented smoothly as I have been using this concept frequently in my classwork and other projects. The clear understanding of virtual functions and runtime polymorphism gained during class exercises meant that I encountered no significant issues during implementation. As such, I did not

need to consult online resources or external materials for this part of the project.

2. **Resources Used:**

- GeeksforGeeks 2017, *C++ Polymorphism*, GeeksforGeeks, viewed 20 November 2024, <https://www.geeksforgeeks.org/cpp-polymorphism/>.

2. Composite Data Structures

a) Hash Tables

Hash tables play a critical role in the inventory management system of *Descent into the Infernal Abyss*. The **Potion** implemented using the std::unordered_map class from the C++ Standard Template Library (STL), efficiently organizes and retrieves potions. This data structure associates potion names (as keys) with their corresponding Potion objects, enabling constant-time average complexity for operations like insertion, retrieval, and deletion.

The hash table is especially useful in a game like *Descent into the Infernal Abyss*, where the player frequently interacts with potions during gameplay. Whether it's retrieving a health potion in a combat scenario or a sanity potion after a fear-inducing event, the hash table ensures these operations are fast and seamless.

I have used Hash for potions only while the rest are using array as I was more comfortable with using them.

Application in Gameplay

In *Descent into the Infernal Abyss*, the hash table is a core part of the inventory system, specifically used for managing potions. During gameplay, the player encounters and collects various potions, which are then stored in a hash table for efficient retrieval and management. The hash table enables the following operations:

1. **Potion Retrieval During Combat or Recovery:**
    - Players can instantly search for and use specific potions, such as health or sanity potions, to recover lost attributes during critical moments.
    - The hash table ensures near-instantaneous lookup, providing a seamless gameplay experience even as the inventory grows.
2. **Dynamic Inventory Updates:**
    - As potions are used or discarded, the hash table dynamically updates its entries, keeping the inventory organized without requiring manual effort.
3. **Player Decision-Making:**
    - The player can view their potion inventory at any time, with each potion's details (name and restorative value) displayed.

- This transparency helps players strategize about which potion to use based on their current situation.

Alternative Structures and Justification

While hash tables are optimal for managing potions in this game, alternative data structures were considered but deemed less suitable (GeeksforGeeks 2016):

1. **Arrays:**
   Arrays provide a simple way to store potion objects. However, searching for a potion by name requires a linear scan, resulting in O(n) time complexity for lookups. This approach would slow down gameplay, especially as the inventory grows.

2. **Linked Lists:**
   A linked list allows dynamic addition and removal of potions but suffers from the same inefficiency as arrays for lookups (O(n) complexity). Additionally, traversing a linked list to search for specific potions would be more error-prone and difficult to debug.

3. **Binary Search Trees (BSTs):**
   A BST provides O(log n) lookup time but requires maintaining the tree's balance to ensure efficient operations. The overhead of maintaining balance makes it less practical than hash tables for this application.

4. **Why Hash Tables?**
   Hash tables provide O(1) average time complexity for insertion, lookup, and deletion, making them ideal for a dynamic and fast-paced gameplay scenario. By associating potion names (keys) with potion objects (values), hash tables offer direct and efficient access to inventory items.

Implementation:

Programming Project (Global Scope)

```h
#pragma once
#include "CursedItem.h"
#include <unordered_map> // Include unordered_map for the hash table

// Represents a potion in the game, inheriting from the CursedItem class.
class Potion : public CursedItem {
private:
    int fRestorativeValue; // The restorative value of the potion.
    static std::unordered_map<std::string, int> fPotionEffects; // Hashtable to store potion effects

public:
    // Default constructor to initialize a generic potion.
    Potion();

    // Constructor to initialize a potion with a specific name, curse description, and restorative value.
    Potion(const std::string& aItemName, const std::string& aCurseDescription, int aRestorativeValue);

    // Destructor to clean up resources associated with the potion.
    virtual ~Potion();

    // Retrieves the restorative value of the potion.
    int getRestorativeValue() const;

    // Overrides the 'use' method from CursedItem to define how the potion interacts with a player.
    void use(Player* aPlayer) override;

    // Static function to add an effect to the hash table
    static void addPotionEffect(const std::string& effectName, int effectValue);

    // Static function to retrieve an effect from the hash table
    static int getPotionEffect(const std::string& effectName);
};

```

Programming Project (Global Scope)

```cpp
#include "Potion.h"
#include "Player.h"
#include <iostream>
#include <unordered_map> // Include unordered_map for hash table usage

// Initialize the static hash table
std::unordered_map<std::string, int> Potion::fPotionEffects;

// Default constructor
Potion::Potion() : CursedItem(), fRestorativeValue(0) {}

// Overloaded constructor
Potion::Potion(const std::string& aItemName, const std::string& aCurseDescription, int aRestorativeValue)
    : CursedItem(aItemName, aCurseDescription), fRestorativeValue(aRestorativeValue) {}

// Retrieves the restorative value of the potion.
int Potion::getRestorativeValue() const {
    return fRestorativeValue;
}

// Player uses potion to restore sanity
void Potion::use(Player* aPlayer) {
    int lBeforeBoost = aPlayer->getSanity();
    aPlayer->restoreSanity(fRestorativeValue);
    std::cout << "\n" << fItemName << " boosts " << aPlayer->getName() << "'s sanity from "
        << lBeforeBoost << " to " << aPlayer->getSanity() << std::endl;
    aPlayer->removeCursedItemFromInventory(fItemName);
}

// Static function to add an effect to the hash table
void Potion::addPotionEffect(const std::string& effectName, int effectValue) {
    fPotionEffects[effectName] = effectValue;
    std::cout << "Added effect: " << effectName << " with value: " << effectValue << std::endl;
}

// Static function to retrieve an effect from the hash table
int Potion::getPotionEffect(const std::string& effectName) {
    if (fPotionEffects.find(effectName) != fPotionEffects.end()) {
        return fPotionEffects[effectName];
    }
    else {
        std::cerr << "Effect " << effectName << " not found!" << std::endl;
        return 0;
    }
}

// Destructor
Potion::~Potion() {}

```

Testing and Output:

```
The Potion restores you. Health is now 100, and Fear is now 0.
```

Troubleshooting Summary

1. **Issue:**
   Implementing hash tables for managing potions in *Descent into the Infernal Abyss* was a bit challenging initially, as I was more familiar with using arrays for data storage. Transitioning to a hash-based structure required understanding how std::unordered_map works in C++ and how to integrate it into the existing inventory system.
   I decided to focus on using hashing for potions to improve their management efficiency while keeping other inventory items in arrays for simplicity. This mixed approach allowed me to test and compare the benefits of hash tables with traditional array-based storage.
   To implement hash tables for potions, I took assistance from online resources like GeeksforGeeks and generative AI tools like GPT. GPT provided detailed examples and explanations of how hash tables work and how to apply them in a dynamic inventory system.
2. **Resources:**

- GeeksforGeeks 2016, *unordered_map in C++ STL*, GeeksforGeeks, viewed 20 November 2024, <https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/>.
- *ChatGPT* 2024, Chatgpt.com, viewed 20 November 2024, <https://chatgpt.com/ >.

b) Singly Linked-List

In *Descent into the Infernal Abyss*, the **singly linked list (SLL)** is used to manage **cursed items** in the inventory system. Each cursed item, such as a hexblade or a potion, is represented as a node in the list. The SLL's simplicity allows efficient memory usage while maintaining dynamic inventory capabilities, as cursed items are frequently added, removed, or iterated over during gameplay.

Application in Gameplay

In *Descent into the Infernal Abyss*, the **singly linked list (SLL)** is utilized in the Inventory system to manage **cursed items** dynamically. This system allows players to collect, use, and discard cursed items during their journey through the dungeon.

The SLL structure supports the following gameplay features:

1. **Dynamic Item Addition:**
   - When a player collects a new cursed item, such as a Hexblade or a Potion of Clarity, it is added as a new node at the head of the list.
   - This ensures efficient insertion with O(1) time complexity.
2. **Efficient Item Removal:**

- If the player uses or discards an item, the removeCursedItem method removes the corresponding node and returns the associated item object.
- This allows players to manage their inventory effectively during gameplay, such as freeing up space for new items or removing outdated ones.

3. **Inventory Browsing:**
   - The displayItems method traverses the list and outputs the names and descriptions of all items in the inventory. This helps players make informed decisions about which items to use or prioritize.
4. **Item Retrieval:**
   - The getItem method allows players to search for and retrieve specific cursed items by name without removing them from the inventory. This is particularly useful for quick access during combat or puzzle-solving.

Diagram Representation:

Base Class Implementation for Singly Linked List:

Programming Project — CursedItemNode

```
#pragma once

#include "CursedItem.h"

class CursedItem; // Forward declaration of the CursedItem class to avoid circular dependencies.

// Represents a node in a linked list, containing a cursed item and a pointer to the next node.
class CursedItemNode {
public:
    CursedItem* fCursedItem;       // Pointer to a CursedItem object.
    CursedItemNode* fNext;         // Pointer to the next node in the linked list.

    // Default constructor to initialize an empty node.
    CursedItemNode();

    // Parameterized constructor to initialize a node with a cursed item and a pointer to the next node.
    CursedItemNode(CursedItem* aCursedItem, CursedItemNode* aNext);

    // Destructor to clean up resources associated with the node.
    ~CursedItemNode();
};

```

Programming Project (Global Scope)

```cpp
#include "CursedItemNode.h"

// Default constructor
CursedItemNode::CursedItemNode() : fCursedItem(nullptr), fNext(nullptr) {}

// Parameterized constructor
CursedItemNode::CursedItemNode(CursedItem* aCursedItem, CursedItemNode* aNext)
    : fCursedItem(aCursedItem), fNext(aNext) {}

// Destructor
CursedItemNode::~CursedItemNode() {
    // Safeguard against invalid pointer deletion
    if (fCursedItem) {
        delete fCursedItem; // Delete the dynamically allocated CursedItem
        fCursedItem = nullptr; // Set pointer to nullptr to avoid dangling pointers
    }
}
```

Programming Project (Global Scope)

```cpp
#include "CursedItemNode.h"

// Default constructor
CursedItemNode::CursedItemNode() : fCursedItem(nullptr), fNext(nullptr) {}

// Parameterized constructor
CursedItemNode::CursedItemNode(CursedItem* aCursedItem, CursedItemNode* aNext)
    : fCursedItem(aCursedItem), fNext(aNext) {}

// Destructor
CursedItemNode::~CursedItemNode() {
    // Safeguard against invalid pointer deletion
    if (fCursedItem) {
        delete fCursedItem; // Delete the dynamically allocated CursedItem
        fCursedItem = nullptr; // Set pointer to nullptr to avoid dangling pointers
    }
}
```


```cpp
#include "Inventory.h"

// Constructor for Inventory
Inventory::Inventory() : fHead(nullptr) {}

// Adds a new cursed item to the inventory by creating a new CursedItemNode and linking it at the beginning of the list.
void Inventory::addCursedItem(CursedItem* aItem) {
    fHead = new CursedItemNode(aItem, fHead); // New node becomes the new head of the list.
}

// Removes a cursed item from the inventory by name and returns a pointer to the item, if found.
CursedItem* Inventory::removeCursedItem(const std::string& aItemName) {
    CursedItemNode** current = &fHead;
    while (*current != nullptr) {
        if ((*current)->fCursedItem->getItemName() == aItemName) { // Check if current item matches the name.
            CursedItemNode* toDelete = *current;
            CursedItem* item = toDelete->fCursedItem; // Extract the item before deleting the node.
            *current = (*current)->fNext;
            toDelete->fCursedItem = nullptr;  // Avoid deleting the item itself.
            delete toDelete;
            return item; // Return the removed item.
        }
        current = &((*current)->fNext);
    }
    return nullptr; // Return null if the item was not found.
}

// Retrieves a cursed item by name from the inventory and returns it, or nullptr if not found.
CursedItem* Inventory::getItem(const std::string& aItemName) const {
    for (CursedItemNode* current = fHead; current != nullptr; current = current->fNext) {
        if (current->fCursedItem->getItemName() == aItemName) { // Check if current item matches the name.
            return current->fCursedItem; // Return the found item.
        }
    }
    return nullptr; // Return null if the item is not found.
}

// Returns an iterator positioned at the start of the inventory.
InventoryIterator Inventory::begin() const {
    return InventoryIterator(fHead);
}

// Returns an iterator representing the end of the inventory.
InventoryIterator Inventory::end() const {
    return InventoryIterator(nullptr); // End iterator is a null pointer.
}

// Destructor for Inventory. Iterates through the list and deletes all item nodes to free memory.
Inventory::~Inventory() {
    clear(); // Reuse the clear method to clean up memory
}

// Implementation of the clear method
void Inventory::clear() {
    CursedItemNode* current = fHead;
    while (current != nullptr) {
        CursedItemNode* toDelete = current;
        current = current->fNext;
        delete toDelete->fCursedItem; // Delete the item itself
        delete toDelete; // Delete the node
    }
    fHead = nullptr; // Reset the head pointer
}
```

Testing and Output:

```
Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice: 3
Cursed Item: Dark Elixir, Curse: A vial filled with a sinister liquid
Cursed Item: Blade of the Damned, Curse: A blade cursed by the ancients
Cursed Item: Soul Fragments, Curse: Fragments of tortured souls

Do you wish to use a cursed item from your inventory? (yes/no): yes
Enter the name of the item you want to use: Blade of the Damned
Blade of the Damned has increased your tenacity. Current tenacity: 76
```

Alternative Structures and Justification

Alternative data structures like arrays or doubly linked lists were considered but found less suitable for this application (GeeksforGeeks 2024):

1. **Arrays:**
    - Adding or removing a skill in an array would require shifting elements, resulting in O(n) time complexity.
    - Arrays also require predefined sizes, limiting flexibility when the player gains or loses skills dynamically.
2. **Doubly Linked Lists:**
    - While doubly linked lists provide bidirectional traversal, this feature is unnecessary for cursed skills, where forward traversal suffices.
    - Singly linked lists are simpler and more memory-efficient for this use case.

Why Singly Linked Lists?
Singly linked lists provide efficient O(1) insertion and removal at the beginning or end of the list. This makes them ideal for dynamically managing cursed skills, where frequent additions and deletions are expected during gameplay.

Troubleshooting Summary

1. **Problem: Null Pointer Dereference**
   Attempting to remove an item not in the list caused crashes due to null pointer dereference.

Solution:
Added checks to ensure the pointer was valid before accessing or deleting it.

2. **Problem: Memory Leaks**
   Dynamically allocated nodes were not freed correctly, leading to memory leaks during testing.

Solution:
Implemented a destructor in the Inventory class to free all nodes upon destruction.

3. **Resources Used:**

- GeeksforGeeks 2024, *Singly Linked List Tutorial*, GeeksforGeeks, viewed 24 November 2024, <https://www.geeksforgeeks.org/singly-linked-list-tutorial/>.
- A little help of GPT was also needed.

c) Doubly Linked List

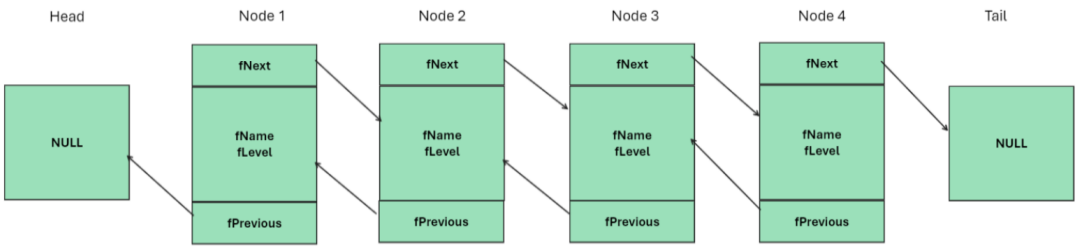
The **doubly linked list (DLL)** is used in *Descent into the Infernal Abyss* to manage the player's **cursed skills**. Each skill, such as *Brute Strength*, *Intellect*, or *Tenacity*, is stored in a node within the DLL. The DLL allows for efficient bidirectional traversal, making it easier to navigate, modify, and display the list of skills during gameplay. This flexibility is particularly useful when skills need to be upgraded, removed, or reordered based on the player's choices and the dynamic events in the dungeon.

Application in Gameplay

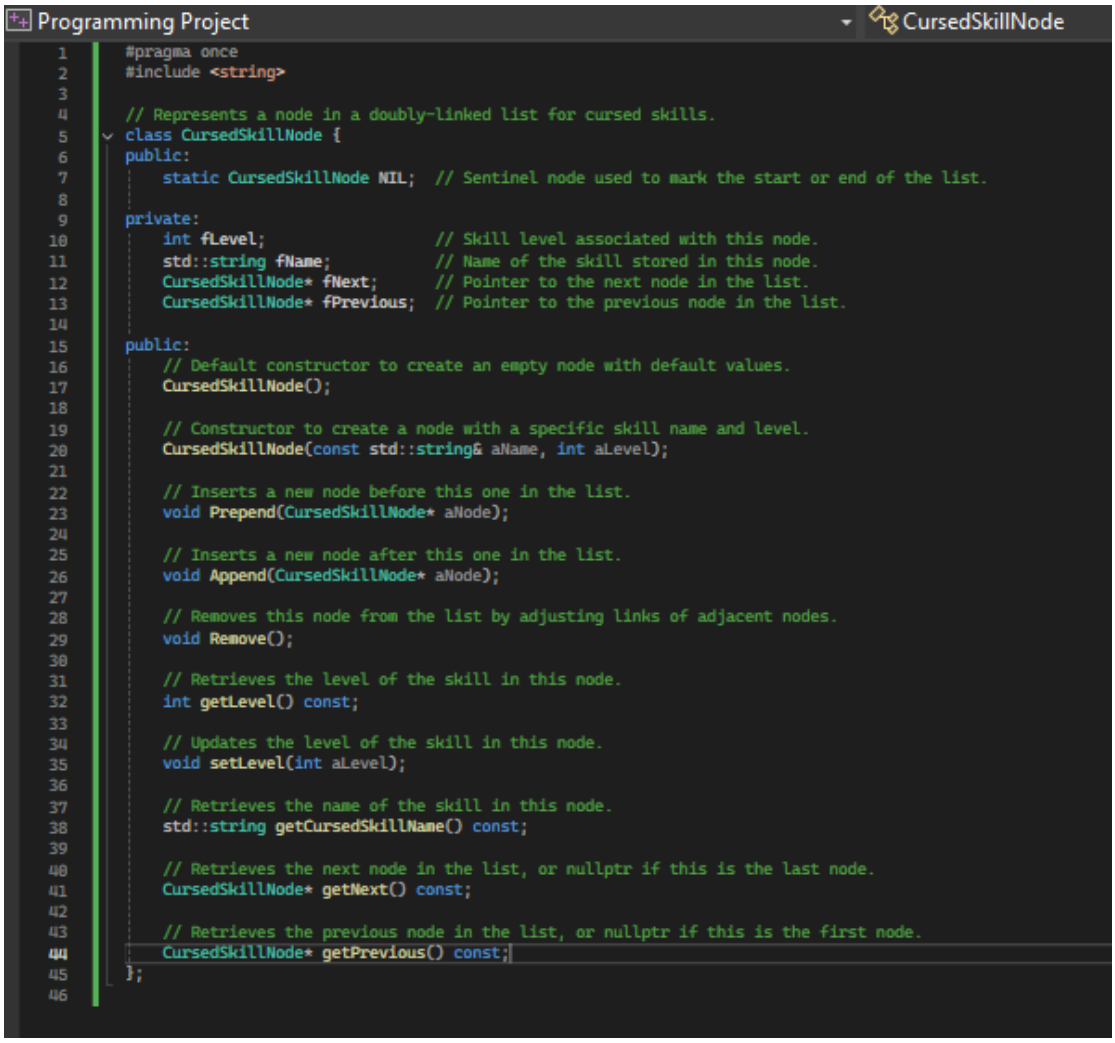
The DLL manages cursed skills dynamically. Here's how it integrates into the gameplay:

1. **Adding Skills:**
    - When the player acquires a new skill, it is added to the list via Append or Prepend.
    - Example: Acquiring *Brute Strength (Level 1)* adds it to the list.
2. **Upgrading Skills:**
    - Skill levels can be updated dynamically using setLevel, reflecting the player's progression.
3. **Removing Skills:**
    - Skills may be removed due to fear penalties or player choices. The Remove function adjusts links to ensure the list remains intact.
4. **Bidirectional Navigation:**
    - The DLL allows the game to traverse forwards or backwards through the skill list, which is useful for displaying or modifying skills.

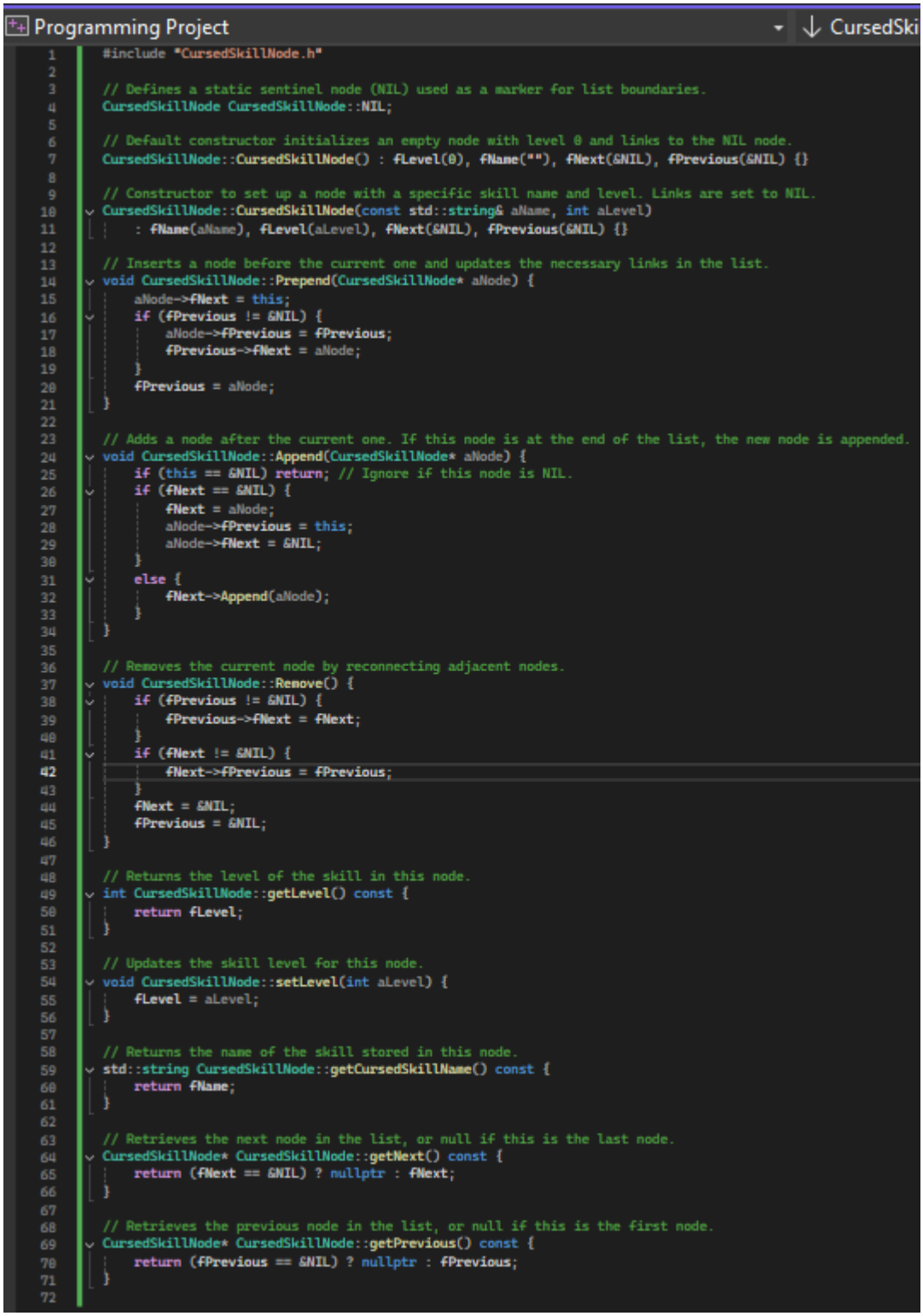
Diagram Representation:



Base Class Implementation Doubly Linked List:




```cpp
#pragma once
#include <string>

// Represents a node in a doubly-linked list for cursed skills.
class CursedSkillNode {
public:
    static CursedSkillNode NIL;  // Sentinel node used to mark the start or end of the list.

private:
    int fLevel;                  // Skill level associated with this node.
    std::string fName;           // Name of the skill stored in this node.
    CursedSkillNode* fNext;      // Pointer to the next node in the list.
    CursedSkillNode* fPrevious;  // Pointer to the previous node in the list.

public:
    // Default constructor to create an empty node with default values.
    CursedSkillNode();

    // Constructor to create a node with a specific skill name and level.
    CursedSkillNode(const std::string& aName, int aLevel);

    // Inserts a new node before this one in the list.
    void Prepend(CursedSkillNode* aNode);

    // Inserts a new node after this one in the list.
    void Append(CursedSkillNode* aNode);

    // Removes this node from the list by adjusting links of adjacent nodes.
    void Remove();

    // Retrieves the level of the skill in this node.
    int getLevel() const;

    // Updates the level of the skill in this node.
    void setLevel(int aLevel);

    // Retrieves the name of the skill in this node.
    std::string getCursedSkillName() const;

    // Retrieves the next node in the list, or nullptr if this is the last node.
    CursedSkillNode* getNext() const;

    // Retrieves the previous node in the list, or nullptr if this is the first node.
    CursedSkillNode* getPrevious() const;
};
```



Testing and Output:

```
Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice: 2
Health: 60
Sanity: 70
Tenacity: 60
Fear: 20

Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice: 4
Cursed Skill: Intellect, Level: 10
Cursed Skill: Survivor, Level: 90
Cursed Skill: Brute Force, Level: 50
Cursed Skill: Dark Power, Level: 49

Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice:
```

Alternative Structures and Justification

When implementing the cursed skills system, various data structures were considered. Each structure has its advantages and drawbacks, but a Doubly Linked List (DLL) was ultimately selected for its ability to handle the dynamic nature of skills in *Descent into the Infernal Abyss*. Alternatives were mentioned earlier in SLL.

Why Doubly Linked Lists?

The **cursed skills system** requires:

1. **Dynamic Skill Management:** Skills are frequently added, removed, and updated during gameplay, which DLLs handle efficiently.

2. **Bidirectional Traversal:** Forward and backward navigation is necessary for displaying and managing skills.
3. **Efficient Updates:** The DLL allows nodes to be updated or removed in O(1) time with direct access, which is essential for a responsive gameplay experience.

The DLL satisfies all these requirements, making it the optimal choice for managing cursed skills in *Descent into the Infernal Abyss*.

Troubleshooting Summary

Issues:

No issue faced as I have already did it in my problem set and tutorial class was detailed so it helped.

Reference:

- GeeksforGeeks 2024, *Doubly Linked List Tutorial*, GeeksforGeeks, viewed 24 November 2024, <https://www.geeksforgeeks.org/doubly-linked-list/>.

3. Abstract Data Type:

a) stack

A **stack** is a core abstract data type used in *Descent into the Infernal Abyss* to manage the player's **soul shards**, which are critical for completing the game. The stack is implemented using the **Last-In, First-Out (LIFO)** principle, where the most recently added shard is the first to be removed. This design is crucial for the gameplay mechanics, as players must collect and correctly insert the shards in reverse order to unlock the Eternal Void portal.

Application in Gameplay

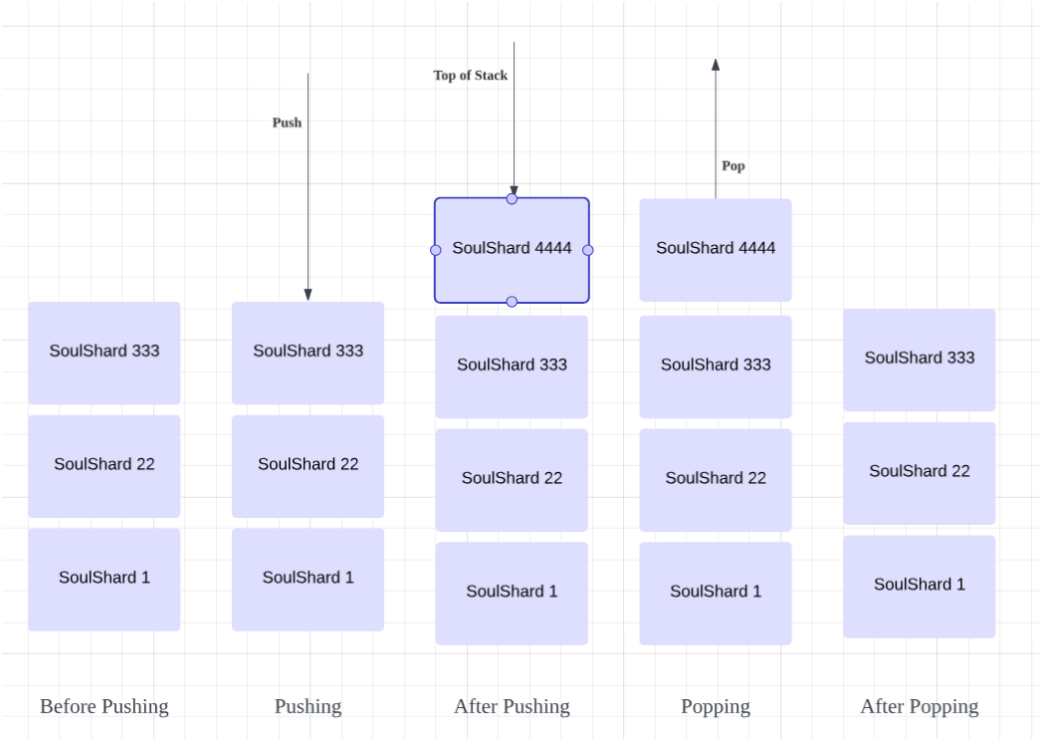
The stack is used to:

1. **Collect Soul Shards:**
    - As the player progresses through the dungeon, they collect **soul shards** that are pushed onto the stack.
    - The shards are added in the order they are discovered, reflecting the chronological sequence of collection.
2. **Portal Unlock Mechanic:**

- At the Eternal Void, the player must use the shards in **reverse order** of their collection to unlock the portal.
- This mirrors the LIFO behavior of the stack, where the most recently collected shard is the first to be removed (popped).

3. **Game Strategy:**
   - Players must strategically decide when to collect shards and ensure they use the stack correctly to avoid losing progress.

Diagram Representation



Base Class Implementation

Programming Project (Global Scope)

```
#pragma once

#include "SoulShard.h"
#include <iostream>

using namespace std;

// Implements a stack specifically for storing SoulShards, with a fixed maximum size.
class ShardStack {
private:
    static const int MAX_SIZE = 5; // Maximum number of SoulShards that can be stored in the stack.
    SoulShard fStack[MAX_SIZE];    // Array used to implement the stack.
    int fTopIndex;                 // Index indicating the top of the stack.

public:
    // Constructor to initialize an empty stack.
    ShardStack();

    // Destructor to clean up resources.
    ~ShardStack();

    // Stack operations.

    // Checks if the stack is empty.
    bool isEmpty() const;

    // Checks if the stack is full.
    bool isFull() const;

    // Retrieves the current number of SoulShards in the stack.
    int size() const;

    // Adds a SoulShard to the top of the stack.
    void push(const SoulShard& shard);

    // Removes and returns the SoulShard at the top of the stack.
    SoulShard pop();

    // Retrieves the SoulShard at the top of the stack without removing it.
    SoulShard peek() const;

    // Creates a snapshot of the stack's contents.
    SoulShard* getSnapshot() const;
};

```

Programming Project (Global Scope)

```cpp
#include "SoulShardStack.h"

// Constructor initializes the stack as empty.
ShardStack::ShardStack() : fTopIndex(-1) {}

// Returns true if the stack is empty.
bool ShardStack::isEmpty() const {
    return fTopIndex == -1;
}

// Returns true if the stack is full.
bool ShardStack::isFull() const {
    return fTopIndex == MAX_SIZE - 1;
}

// Returns the current size of the stack.
int ShardStack::size() const {
    return fTopIndex + 1;
}

// Adds a SoulShard to the top of the stack. Displays a warning if the stack is full.
void ShardStack::push(const SoulShard& shard) {
    if (isFull()) {
        cout << "You cannot stack more shards" << endl;
    }
    fStack[++fTopIndex] = shard;
}

// Removes and returns the top SoulShard from the stack. Displays a warning if the stack is empty.
SoulShard ShardStack::pop() {
    if (isEmpty())
    {
        cout << "No soul shards left to use." << endl;
    }
    return fStack[fTopIndex--];
}

// Returns the top SoulShard without removing it. Throws an error if the stack is empty.
SoulShard ShardStack::peek() const {
    if (isEmpty()) {
        throw std::runtime_error("No soul shards left to use.");
    }
    return fStack[fTopIndex];
}

// Returns a snapshot of the stack's current contents. Returns nullptr if the stack is empty.
SoulShard* ShardStack::getSnapshot() const {
    if (isEmpty()) {
        return nullptr;
    }
    int currentSize = size();
    SoulShard* snapshot = new SoulShard[currentSize];
    for (int i = 0; i < currentSize; ++i) {
        snapshot[i] = fStack[i];
    }
    return snapshot;
}


// Destructor
ShardStack::~ShardStack() {}

```

```
for (size_t i = 0; i < numberOfLevels;) {
    DungeonTree* currentLevel = levels[i];
    GameInteractionVisitor visitor(player);
    currentLevel->accept(visitor);

    // Trigger a random horror event
    triggerRandomEvent(player);

    if (i == numberOfLevels - 1) {
        if (currentLevel->level() == "Eternal Void") {
            if (player.getShardsCollected() == 4) {
                cout << "You stand before the Eternal Void. The air is thick with despair.\n";
                cout << "Four soul shards pulsate with dark energy, yearning to be offered to the portal.\n";
                int progress = 0;
                cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                while (player.getShardsCollected() != 0 && progress < 100) {
                    cout << "Offer a soul shard to the portal (type 'Offer' to proceed): ";
                    string command;
                    getline(cin, command);

                    if (command == "Offer") {
                        SoulShard usedShard = player.useShard();
                        progress += 25;
                        cout << "Shard ID " << usedShard.getShardID() << " consumed by the portal.\n";
                        cout << "The portal begins to awaken. Progress: " << progress << "%...\n";
                    }
                    else {
                        cout << "The portal rejects your hesitation. Type 'Offer' to proceed." << endl;
                    }
                }

                if (progress == 100) {
                    cout << "The portal bursts open, releasing a deafening roar!\n";
                    cout << "You step through the gateway, escaping the abyss at last. But its darkness will forever haunt you...\n";
                    cout << "Returning to the main menu...\n";
                    return; // Return to main menu after winning.
                }
            }
            else {
                cout << "You have reached the Eternal Void, but the shards elude you. The portal remains sealed.\n";
                cout << "The shadows close in, consuming your soul. Game Over.\n";
                cout << "Returning to the main menu...\n";
                return; // Return to main menu after losing.
            }
        }
    }
```

Testing and Output:

```
Currently at Eternal Void
You stand before the Eternal Void. The air is thick with despair.
Four soul shards pulsate with dark energy, yearning to be offered to the portal.
Offer a soul shard to the portal (type 'Offer' to proceed): Offer
Shard ID 4444 consumed by the portal.
The portal begins to awaken. Progress: 25%...
Offer a soul shard to the portal (type 'Offer' to proceed): Offer
Shard ID 333 consumed by the portal.
The portal begins to awaken. Progress: 50%...
Offer a soul shard to the portal (type 'Offer' to proceed): Offer
Shard ID 22 consumed by the portal.
The portal begins to awaken. Progress: 75%...
Offer a soul shard to the portal (type 'Offer' to proceed): Offer
Shard ID 1 consumed by the portal.
The portal begins to awaken. Progress: 100%...
The portal bursts open, releasing a deafening roar!
You step through the gateway, escaping the abyss at last. But its darkness will forever haunt you...
Returning to the main menu...
```

Alternative Structures and Justification

1. **Arrays:**
    - Arrays could store soul shards, but they do not inherently enforce the LIFO behavior.
    - Managing insertions and deletions would require additional logic, making arrays less efficient and harder to maintain.
2. **Queues:**

- Queues operate on a **First-In, First-Out (FIFO)** principle, which is the opposite of the stack's LIFO behavior.
- Using a queue would require reversing the order of the shards before inserting them, introducing unnecessary complexity.

Why Stack?

- A stack is the natural choice for this mechanic because:
  1. It enforces LIFO behavior, perfectly aligning with the shard insertion sequence.
  2. It is efficient, with O(1) complexity for both push (add) and pop (remove) operations.

Troubleshooting Summary

Issue:

Problem: Empty Stack Access

- During testing, attempting to pop from an empty stack caused undefined behavior.

Solution:
Added checks to ensure the stack is not empty before attempting a pop operation.

Reference:

- GeeksforGeeks 2015, *Stack in C++ STL*, GeeksforGeeks, viewed 24 November 2024, <https://www.geeksforgeeks.org/stack-in-cpp-stl/>.
- Gpt for errors and debug.

b) Queue

A **queue** is used in *Descent into the Infernal Abyss* to manage the **player's pending actions** or **event sequences**. The queue operates on the **First-In, First-Out (FIFO)** principle, ensuring that events are processed in the order they are added. This mechanism is critical for maintaining logical and chronological gameplay sequences, such as resolving hazard effects or processing player actions.

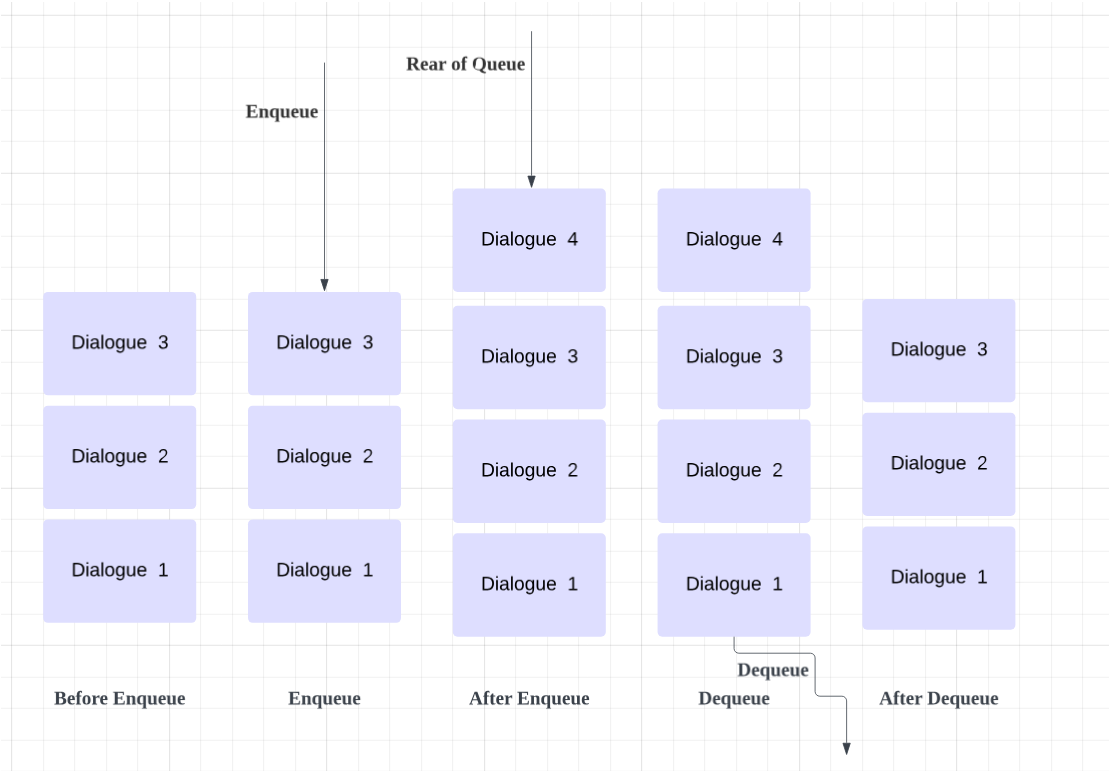
Application in Gameplay

The queue is used in *Descent into the Infernal Abyss* for managing interactions with entities and the Demon Lord encounter. It ensures that all interactions are processed in the correct order, following a First-In, First-Out (FIFO) behavior. This is crucial for maintaining a logical progression of events during gameplay.

Entity Interactions:

- All interactions with dungeon entities, such as Demon Guards, Lost Souls, and other cursed beings, are added to the queue.
- The queue ensures that entities encountered first are dealt with before moving on to subsequent interactions.
- The queue is critical during the final confrontation with the **Demon Lord**, where multiple phases or events may need to be resolved sequentially.

Diagram Representation



Base Class Implementation

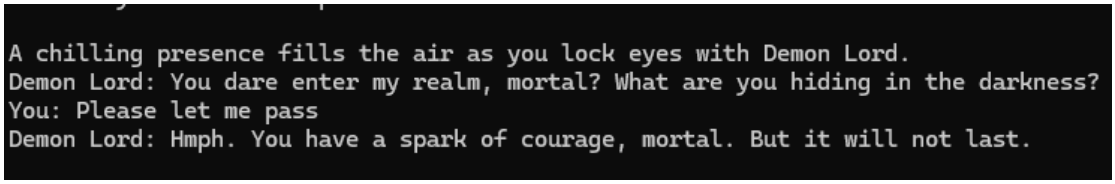
Programming Project

```h
#pragma once
#include <iostream>
#include "DialogueNode.h"

// Manages a queue of dialogue lines for game interactions.
class DialogueQueue {
private:
    DialogueNode* fHead; // Points to the first dialogue entry.
    DialogueNode* fTail; // Points to the last dialogue entry.
    int fCount;          // Tracks the number of entries in the queue.

public:
    // Constructs an empty dialogue queue.
    DialogueQueue();

    // Destructor clears all dialogue entries.
    ~DialogueQueue();

    // Adds a new line of dialogue to the queue.
    void enqueue(const std::string& aValue);

    // Removes and returns the dialogue at the front of the queue.
    std::string dequeue();

    // Returns the dialogue at the front without removing it.
    std::string peek() const;

    // Checks if the queue is empty.
    bool isEmpty() const;

    // Retrieves the number of dialogue entries in the queue.
    int size() const;
};
```

Programming Project

```cpp
#include "DialogueQueue.h"

// Initializes an empty dialogue queue.
DialogueQueue::DialogueQueue() : fHead(nullptr), fTail(nullptr), fCount(0) {}

// Adds a new dialogue line to the end of the queue.
void DialogueQueue::enqueue(const std::string& value) {
    DialogueNode* newNode = new DialogueNode(value);
    if (isEmpty()) {
        fHead = fTail = newNode;
    }
    else {
        fTail->fNext = newNode;
        fTail = newNode;
    }
    fCount++;
}

// Removes and returns the dialogue at the front of the queue.
// If the queue is empty, a warning is displayed.
std::string DialogueQueue::dequeue() {
    if (isEmpty()) {
        std::cout << "Queue is empty" << std::endl;
        return "";
    }
    DialogueNode* temp = fHead;
    std::string value = fHead->fData;
    fHead = fHead->fNext;
    delete temp;
    if (fHead == nullptr) {
        fTail = nullptr;
    }
    fCount--;
    return value;
}

// Retrieves the dialogue at the front without removing it.
// If the queue is empty, a warning is displayed.
std::string DialogueQueue::peek() const {
    if (isEmpty()) {
        std::cout << "Queue is empty" << std::endl;
        return "";
    }
    return fHead->fData;
}

// Checks whether the queue has no dialogue.
bool DialogueQueue::isEmpty() const {
    return fCount == 0;
}

// Returns the number of dialogue entries currently in the queue.
int DialogueQueue::size() const {
    return fCount;
}

// Destructor clears the queue by deleting all nodes.
DialogueQueue::~DialogueQueue() {
    while (!isEmpty()) {
        dequeue();
    }
}
```

Testing and Output:



Justification

Why Queue?

A queue is the optimal choice because:

- It enforces FIFO behavior, which aligns with the chronological processing of actions or events.
- It provides efficient O(1) complexity for both enqueue (add) and dequeue (remove) operations.
- It simplifies gameplay logic by maintaining the correct order of event processing.

Troubleshooting Summary

Issues:

The implementation of the queue was straightforward, as I had prior experience from working on custom data structures in earlier labs. However, integrating it into the main program to manage dynamic interactions, such as managing entity dialogues and player input, required additional research. To fully understand how to implement a responsive dialogue mechanism, I explored several online resources, including YouTube tutorials, to refine my approach. The referenced materials below were instrumental in helping me complete this feature.

References

- how 2014, *how do i store and get a queue of structure?*, Stack Overflow, viewed 24 November 2024, <https://stackoverflow.com/questions/25074741/how-do-i-store-and-get-a-queue-of-structure>.
- Data 2013, *Data structures: Introduction to Queues*, YouTube, viewed 24 November 2024, <https://www.youtube.com/watch?v=XuCbpw6Bj1U>.

c) Tree

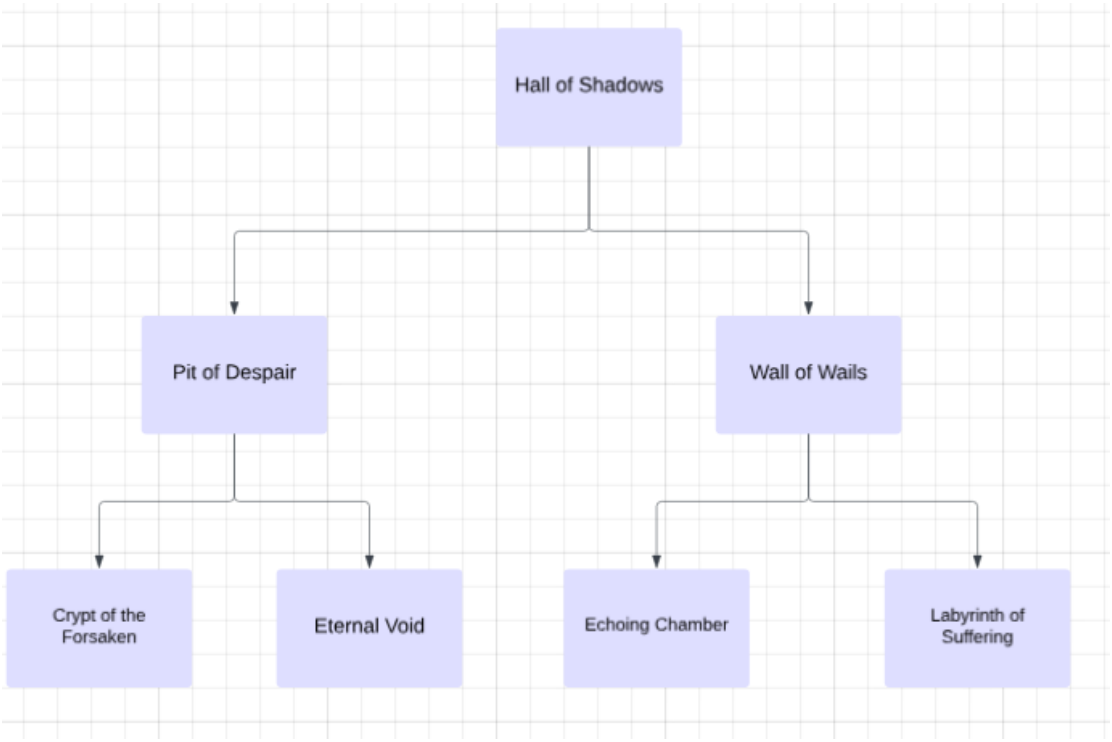
In *Descent into the Infernal Abyss*, a tree structure is used to represent the dungeon levels and their connections. Each node in the tree corresponds to a dungeon level, with child nodes representing the subsequent levels accessible from that node. This hierarchical representation allows players to explore the dungeon dynamically, with branches offering multiple paths or challenges.

Application in Gameplay

The binary tree is used in *Descent into the Infernal Abyss* to represent the dungeon structure, with each node corresponding to a dungeon level. The player's progression through the tree is influenced by their **fear threshold**, which determines the path they take at each branching point. This dynamic design enhances gameplay by integrating player attributes directly into dungeon navigation.

1. **Dungeon Design:**
   - The dungeon is structured as a **binary tree**, with:
     - The **root node** representing the starting point (e.g., *Hall of Shadows*).
     - **Left Child (fLeft)**: Represents one possible path or challenge (e.g., *Pit of Despair*).
     - **Right Child (fRight)**: Represents an alternate path or challenge (e.g., *Wall of Wails*).
   - Each level has unique challenges and hazards, dynamically branching based on player choices and attributes.
2. **Dynamic Progression with Fear Threshold:**
   - **Fear Threshold** plays a pivotal role in determining which path the player takes:
     - If the player's fear level exceeds a certain threshold, they are directed toward a more dangerous or challenging path (e.g., *Wall of Wails*).
     - If the fear level is below the threshold, they may take a safer route (e.g., *Pit of Despair*).
   - This mechanic adds an extra layer of strategy, as players must manage their fear levels carefully to influence their dungeon path.
3. **Replayability and Exploration:**
   - The branching paths encourage exploration, with different challenges and outcomes based on the path chosen.
   - Players can replay the game to experience alternate routes by deliberately managing their fear levels differently.
4. **Hierarchical Navigation:**
   - Players start at the **root node** and navigate to child nodes by resolving challenges at the current level. The fear threshold dynamically determines whether they progress to the left or right child.

Diagram Representation



Base Class Implementation

Programming Project — DungeonTree

```h
#pragma once

#include <string>
#include <stdexcept>
#include "DungeonEntity.h"
#include "Visitor.h"

// Represents a binary tree structure for managing dungeon entities.
class DungeonTree {
private:
    std::string fDungeonLevel;   // Name of the dungeon level at this node.
    DungeonEntity* fEntity;      // Pointer to the entity associated with this node.
    DungeonTree* fLeft;          // Pointer to the left child subtree.
    DungeonTree* fRight;         // Pointer to the right child subtree.
    int fFearThreshold;         // Minimum fear level required to enter this node.

    // Private constructor for initializing the NIL sentinel node.
    DungeonTree();

public:
    static DungeonTree NIL;      // Sentinel node representing an empty tree.

    // Constructs a tree node with a dungeon level and an optional entity.
    DungeonTree(const std::string& aDungeonLevel, DungeonEntity* aEntity = nullptr, int aFearThreshold = 0);

    // Destructor ensures proper cleanup of child nodes.
    ~DungeonTree();

    // Checks if this node is the NIL sentinel.
    bool isEmpty() const;

    // Returns the name of the dungeon level.
    const std::string& level() const;

    // Associates an entity with this node.
    void setEntity(DungeonEntity* aEntity);

    // Retrieves the entity associated with this node.
    DungeonEntity* getEntity() const;

    // Returns the left subtree.
    DungeonTree& left() const;

    // Returns the right subtree.
    DungeonTree& right() const;

    // Attaches a subtree as the left child.
    void attachLeft(DungeonTree* aBTree);

    // Attaches a subtree as the right child.
    void attachRight(DungeonTree* aBTree);

    // Detaches and returns the left subtree.
    DungeonTree* detachLeft();

    // Detaches and returns the right subtree.
    DungeonTree* detachRight();

    // Allows a visitor to interact with this node.
    void accept(Visitor& visitor);

    // Getters and setters for fear threshold
    int getFearThreshold() const;

    void setFearThreshold(int threshold);

};
```

Programming Project — DungeonTree

```
22      }
23
24      // Checks if this node is the NIL sentinel, indicating it's empty.
25      bool DungeonTree::isEmpty() const {
26          return this == &NIL;
27      }
28
29
30      // Returns the name of the dungeon level for this node.
31      // Throws an exception if the node is empty.
32      const std::string& DungeonTree::level() const {
33          if (isEmpty()) {
34              throw std::domain_error("Attempt to access the level of an empty node!");
35          }
36          return fDungeonLevel;
37      }
38
39      // Returns the left child of this node.
40      // Throws an exception if the node is empty.
41      DungeonTree& DungeonTree::left() const {
42          if (isEmpty()) {
43              throw std::domain_error("Attempt to access the left child of an empty node!");
44          }
45          return *fLeft;
46      }
47
48      // Returns the right child of this node.
49      // Throws an exception if the node is empty.
50      DungeonTree& DungeonTree::right() const {
51          if (isEmpty()) {
52              throw std::domain_error("Attempt to access the right child of an empty node!");
53          }
54          return *fRight;
55      }
56
57      // Attaches a subtree as the left child of this node.
58      // Throws an exception if the node is empty or already has a left child.
59      void DungeonTree::attachLeft(DungeonTree* aBTree) {
60          if (isEmpty()) {
61              throw std::domain_error("Cannot attach left to an empty node!");
62          }
63          if (fLeft != &NIL) {
64              throw std::domain_error("Left subtree is already occupied!");
65          }
66          fLeft = aBTree;
67      }
68
69      // Attaches a subtree as the right child of this node.
70      // Throws an exception if the node is empty or already has a right child.
71      void DungeonTree::attachRight(DungeonTree* aBTree) {
72          if (isEmpty()) {
73              throw std::domain_error("Cannot attach right to an empty node!");
```

```
// Navigates the dungeon.
static void navigateDungeon(DungeonTree* currentLevel, Player& player) {
    GameInteractionVisitor visitor(player);

    while (currentLevel != &DungeonTree::NIL) {
        cout << "Entering: " << currentLevel->level() << endl;

        if (player.getFearLevel() >= currentLevel->getFearThreshold()) {
            cout << "You brave the fears of this level.\n";
            currentLevel->accept(visitor);
            triggerRandomEvent(player);
        }
        else {
            cout << "Too fearful to proceed. Seeking less daunting path...\n";
            player.decreaseFear(5);
        }

        // Simplified pathfinding logic for clarity.
        if (!currentLevel->left().isEmpty() && player.getFearLevel() > currentLevel->left().getFearThreshold()) {
            currentLevel = &currentLevel->left();
        }
        else if (!currentLevel->right().isEmpty() && player.getFearLevel() > currentLevel->right().getFearThreshold()) {
            currentLevel = &currentLevel->right();
        }
        else {
            cout << "No path forward, turning back...\n";
            break;
        }
    }
}
```

Testing and Output:

```
You gather your resolve and move deeper into the abyss...

Currently at Echoing Chamber

A chilling presence fills the air as you lock eyes with Demon Lord.
Demon Lord: You dare enter my realm, mortal? What are you hiding in the darkness?
You: You will loose
Demon Lord: Hmph. You have a spark of courage, mortal. But it will not last.

Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice: 1

You gather your resolve and move deeper into the abyss...

Currently at Crypt of the Forsaken
You see a glowing soul shard with ID: 22
Would you like to collect the shard? (1 for Yes, 2 for No): 1
You collect the shard and add it to your collection.

Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice:
```

Alternative Structures and Justification

1. **Linked Lists:**
   - **Advantages:**
     Linked lists are simple and allow for dynamic growth.
   - **Disadvantages:**
     They do not inherently support hierarchical structures or branching, making them unsuitable for representing dungeon levels.

Why Tree?

- A tree structure provides a natural way to represent the hierarchical and branching nature of the dungeon. Its simplicity and efficiency make it ideal for navigating levels and managing progression.

Troubleshooting Summary

Issues:

I completed Tree structure in Problem set 4 and I struggled in PS4 but now I know most of the stuff and it was pretty easy. But instead of making game boring I implemented Fear threshold which will decide to which dungeon you will move.

Reference:

- Tutorial (Problem set 4)
- GeeksforGeeks 2023, *Tree Data Structure*, GeeksforGeeks, viewed 24 November 2024, <https://www.geeksforgeeks.org/tree-data-structure/>.

4. Design Patterns

a) Iterator

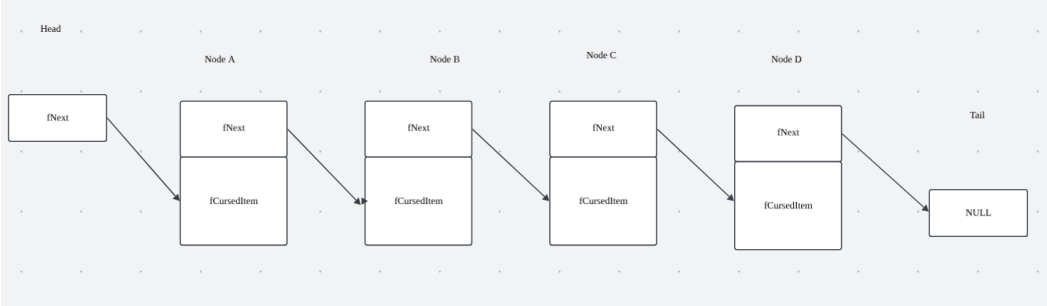
The **Iterator** design pattern is used in *Descent into the Infernal Abyss* to traverse the **dungeon levels** stored in the tree structure. This pattern allows the game to sequentially access nodes (dungeon levels) in a controlled and logical manner without exposing the underlying structure of the tree. By encapsulating the traversal logic in an iterator, the pattern provides a clean way to navigate the dungeon hierarchy during gameplay.

Application in Gameplay

The **Iterator Design Pattern** is used in *Descent into the Infernal Abyss* to traverse and interact with items in the player's inventory dynamically. This design pattern abstracts the underlying data structure (a singly linked list) and provides a consistent interface to iterate over inventory items without exposing the details of the linked list implementation. The iterator ensures seamless access to items, enabling gameplay mechanics such as item usage, inventory management, and displaying inventory contents.

1. **Inventory Navigation:**
    - The iterator provides a simple mechanism to traverse the inventory sequentially, allowing players to interact with each item without worrying about the linked list structure.
2. **Item Management:**
    - Items can be accessed and used directly through the iterator. For example, the player can consume a potion or equip a cursed item during combat or exploration.
3. **Dynamic Inventory Display:**
    - The iterator simplifies the process of displaying the inventory, ensuring that all items are accessed in order without skipping any.

Diagram Representation



Base Class Implementation

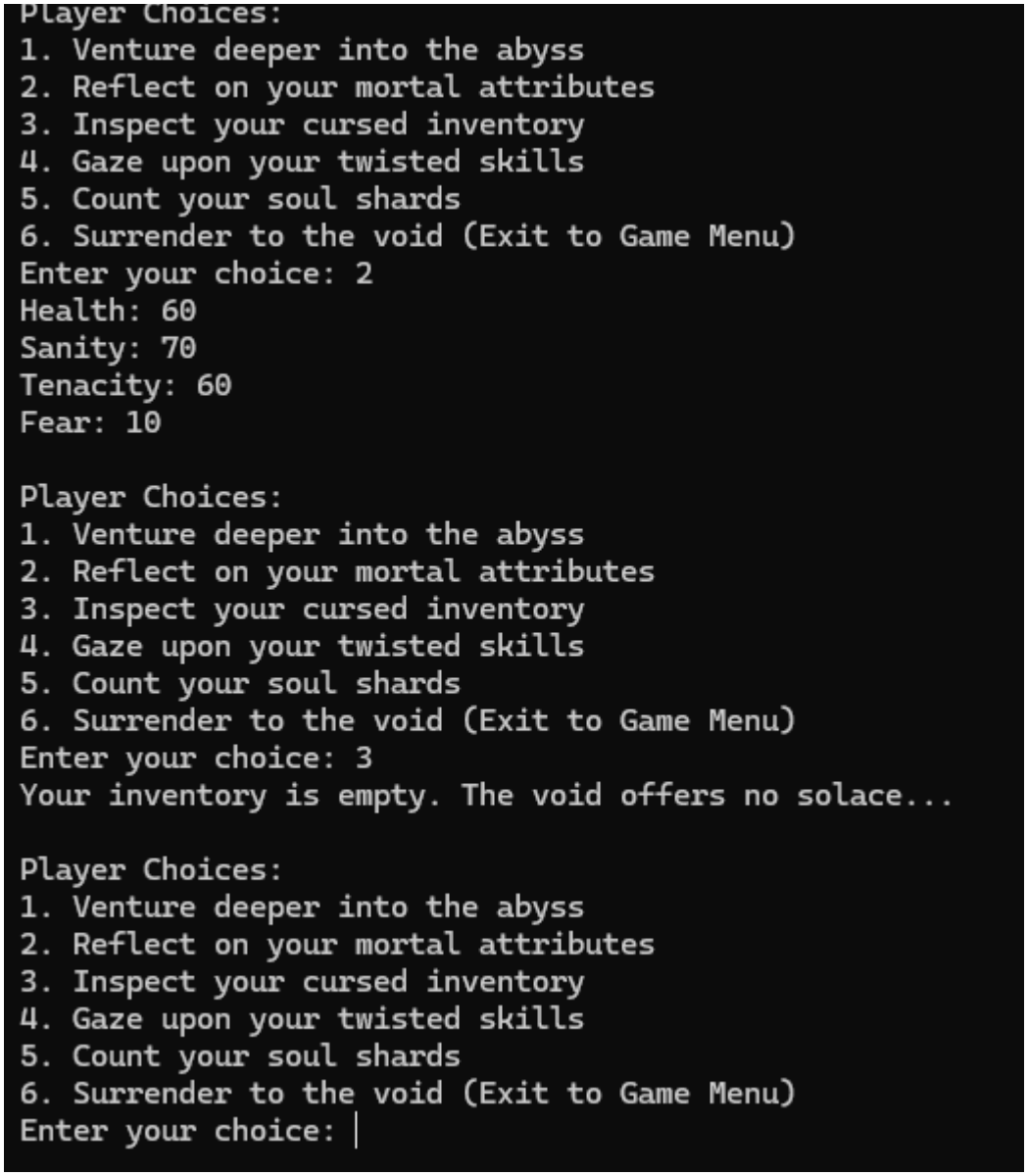
Programming Project (Global Scope)

```cpp
#pragma once

#include "CursedItemNode.h"

// Custom iterator for traversing the player's inventory of cursed items.
class InventoryIterator {
private:
    CursedItemNode* fCurrentNode; // Points to the current node in the inventory.

public:
    // Constructor initializes the iterator to start at a specific node.
    InventoryIterator(CursedItemNode* aStartNode);

    // Dereference operator retrieves the item at the current position.
    CursedItem* operator*() const;

    // Prefix increment operator moves to the next item.
    InventoryIterator& operator++();

    // Equality operator checks if two iterators point to the same node.
    bool operator==(const InventoryIterator& aOther) const;

    // Inequality operator checks if two iterators point to different nodes.
    bool operator!=(const InventoryIterator& aOther) const;
};
```

Programming Project InventoryIterator

```cpp
#include "InventoryIterator.h"

// Initializes the iterator to point to the given node in the inventory.
InventoryIterator::InventoryIterator(CursedItemNode* aStartNode) : fCurrentNode(aStartNode) {}

// Dereferences the iterator to access the current item.
CursedItem* InventoryIterator::operator*() const {
    return fCurrentNode->fCursedItem;
}

// Moves the iterator to the next item (prefix increment).
InventoryIterator& InventoryIterator::operator++() {
    fCurrentNode = fCurrentNode->fNext;
    return *this;
}

// Compares two iterators for equality.
bool InventoryIterator::operator==(const InventoryIterator& aOther) const {
    return fCurrentNode == aOther.fCurrentNode;
}

// Compares two iterators for inequality.
bool InventoryIterator::operator!=(const InventoryIterator& aOther) const {
    return fCurrentNode != aOther.fCurrentNode;
}
```

Testing and Output:



Alternative Structures and Justification

1. **Recursive Traversal:**

- **Advantages:**
  Recursive functions provide a direct way to traverse the tree.
- **Disadvantages:**
  Recursive traversal tightly couples the traversal logic to the tree structure, making it harder to extend or modify. Additionally, it can cause stack overflow for large trees.

2. **Manual Stack Management:**

- **Advantages:**
  Managing a stack explicitly provides flexibility for traversal order.

- **Disadvantages:**
  Without an iterator, managing the stack can clutter the code and reduce readability.

Why Iterator?

- The **Iterator** design pattern abstracts the traversal logic, making the code cleaner and more modular.
- It provides flexibility to implement different traversal methods (e.g., pre-order, in-order, breadth-first) without altering the tree structure.

Troubleshooting Summary

Issues:

The implementation of the iterator was straightforward as it followed the same logic I had used in earlier exercises. However, instead of using an array, I implemented the iterator with a singly linked list structure, which required slight modifications to the traversal logic. This version of the iterator was simpler than those I had used previously, as it did not include functionality like a decrement operator for backward traversal. Overall, no significant errors were encountered during the implementation process.

Reference:

- Tutorial and Problemset 2

b) Visitor

The **Visitor Design Pattern** is used in *Descent into the Infernal Abyss* to handle interactions with entities and objects in the dungeon. This pattern enables different types of interactions, such as combat, dialogue, or item collection, without modifying the entities themselves. The Visitor pattern decouples the operation logic from the entities, making it easier to extend or modify gameplay interactions.

Application in Gameplay

The **Visitor** pattern is applied to:

1. **Entity Interaction:**
   - Each entity in the dungeon (e.g., **Demon Guards**, **Lost Souls**) has a unique behavior when interacted with.
   - The Visitor pattern allows the game to apply specific actions to entities without altering their base class.
2. **Dynamic Event Handling:**
   - Visitors enable context-based interactions, such as combat with **Demon Guards** or sanity effects from **Lost Souls**.

Base Class Implementation

Programming Project (Global Scope)

```h
#pragma once

// Forward declarations of various game entities the visitor can interact with.
class DungeonTree;
class SoulShardEntity;
class EntityWrapper;
class CursedItemWrapper;
class HazardWrapper;

// Base class for the Visitor pattern, allowing different interactions with game entities.
class Visitor {
public:
    // Handles interaction with a dungeon node.
    virtual void Visit(DungeonTree& aNode) = 0;

    // Handles interaction with a soul shard entity.
    virtual void Visit(SoulShardEntity& aShardEntity) = 0;

    // Handles interaction with an entity wrapped for special behavior.
    virtual void Visit(EntityWrapper& aEntityWrapper) = 0;

    // Handles interaction with a cursed item wrapped for gameplay use.
    virtual void Visit(CursedItemWrapper& aCursedItemWrapper) = 0;

    // Handles interaction with a hazardous game element.
    virtual void Visit(HazardWrapper aHazardWrapper) = 0;

    // Virtual destructor to ensure proper cleanup in derived classes.
    virtual ~Visitor() = default;
};
```

```cpp
// Allows a visitor to interact with this node.
void DungeonTree::accept(Visitor& visitor) {
    visitor.Visit(*this);
}
```

```cpp
// Accepts a visitor to interact with this SoulShardEntity.
// The visitor's Visit method is called if a valid shard is present.
void SoulShardEntity::accept(Visitor& visitor) {
    if (fShard != nullptr) {
        visitor.Visit(*this);
    }
}
```

```cpp
// Allows a visitor to interact with this Entity.
void EntityWrapper::accept(Visitor& aVisitor) {
    if (fEntity != nullptr) {
        aVisitor.Visit(*this);
    }
}
```

```cpp

// Allows a visitor to interact with this cursed item.
void CursedItemWrapper::accept(Visitor& visitor) {
    if (fCursedItem != nullptr) {
        visitor.Visit(*this); // Let the visitor handle this cursed item.
    }
}

```

```cpp
// Allows a visitor to interact with this hazard.
void HazardWrapper::accept(Visitor& visitor) {
    if (fHazard != nullptr) {
        visitor.Visit(*this);
    }
}
```

Testing and Output:

```
Player Choices:
1. Venture deeper into the abyss
2. Reflect on your mortal attributes
3. Inspect your cursed inventory
4. Gaze upon your twisted skills
5. Count your soul shards
6. Surrender to the void (Exit to Game Menu)
Enter your choice: 1

You gather your resolve and move deeper into the abyss...

Currently at Hall of Shadows
You discover a Fragments of tortured souls: Soul Fragments
Would you like to collect the Soul Fragments? (1 for Yes, 2 for No): 1
The Soul Fragments is now in your inventory.
Collected: Soul Fragments
```

Justification

Why Visitor?

The Visitor pattern decouples interaction logic from the entity classes, making the system more extensible and maintainable. Adding new interactions requires only a new visitor class, without modifying existing entities.

Troubleshooting Summary

Issues:
There were some issues earlier with how to use it but YouTube videos were straightforward and error were solved with the help of GPT.

References:

- GeeksforGeeks 2017, *Visitor design pattern*, GeeksforGeeks, viewed 24 November 2024, <https://www.geeksforgeeks.org/visitor-design-pattern/>.
- Creating a Python C/C++ Wrapper 2018, *Creating a Python C/C++ Wrapper*, YouTube, viewed 24 November 2024, <https://www.youtube.com/watch?v=YX6J_8ejibc>.
- Visitor Design Pattern 2012, *Visitor Design Pattern*, YouTube, viewed 24 November 2024, <https://www.youtube.com/watch?v=pL4mOUDi54o>.

Meeting Assignment Requirements

A. At least 2 Horror Gameplay Elements/Mechanics

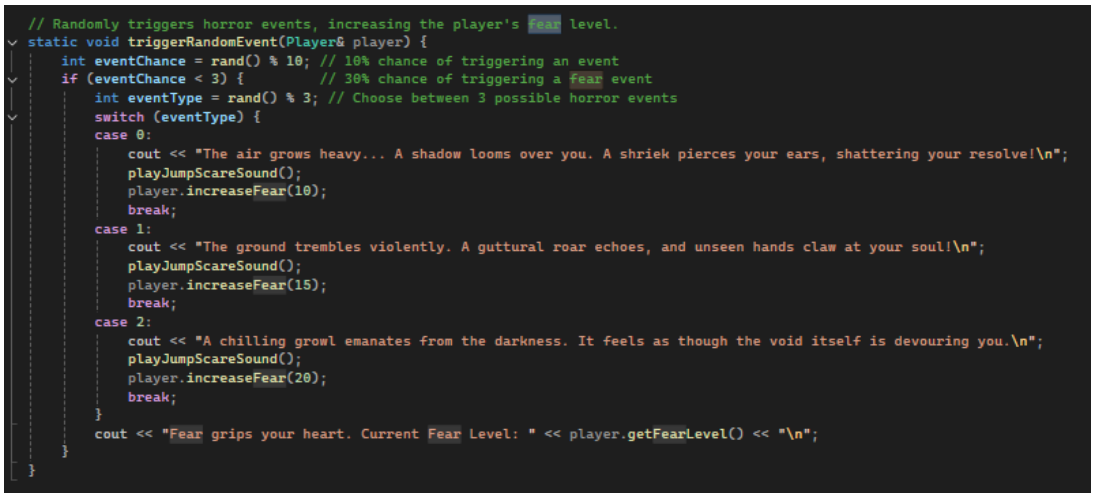
1. **Fear Threshold Mechanic:**
    - The game dynamically adjusts the player's dungeon path based on their **fear level**. High fear levels force the player onto more dangerous paths, while low fear levels provide safer routes.
    - **Gameplay Impact:**
        - High fear reduces the player's ability to recover sanity and increases the difficulty of encounters.
        - This mechanic adds psychological tension, as players must balance their fear levels while navigating through hazardous environments.
    - **Implementation:**
        - The player's fear level is tracked as an attribute in the Player class.
        - The DungeonTree navigation logic checks the fear threshold to determine which child node (path) to follow.

```cpp
// Navigates the dungeon.
static void navigateDungeon(DungeonTree* currentLevel, Player& player) {
    GameInteractionVisitor visitor(player);

    while (currentLevel != &DungeonTree::NIL) {
        cout << "Entering: " << currentLevel->level() << endl;

        if (player.getFearLevel() >= currentLevel->getFearThreshold()) {
            cout << "You brave the fears of this level.\n";
            currentLevel->accept(visitor);
            triggerRandomEvent(player);
        }
        else {
            cout << "Too fearful to proceed. Seeking less daunting path...\n";
            player.decreaseFear(5);
        }

        // Simplified pathfinding logic for clarity.
        if (!currentLevel->left().isEmpty() && player.getFearLevel() > currentLevel->left().getFearThreshold()) {
            currentLevel = &currentLevel->left();
        }
        else if (!currentLevel->right().isEmpty() && player.getFearLevel() > currentLevel->right().getFearThreshold()) {
            currentLevel = &currentLevel->right();
        }
        else {
            cout << "No path forward, turning back...\n";
            break;
        }
    }
}
```

2. **Randomized Horror Events:**
    - With a probability of triggering at each level, **random horror events** such as sudden noises, screen flashes, or jump scares increase the player's fear level.
    - **Gameplay Impact:**
        - Unpredictable events keep the player on edge, enhancing the horror atmosphere.

- Increased fear makes survival more challenging and heightens immersion.

- **Implementation:**
  - Random number generation determines whether an event is triggered at a given level.
  - Fear level increases dynamically based on the severity of the event.



```
// Randomly triggers horror events, increasing the player's fear level.
static void triggerRandomEvent(Player& player) {
    int eventChance = rand() % 10; // 10% chance of triggering an event
    if (eventChance < 3) {         // 30% chance of triggering a fear event
        int eventType = rand() % 3; // Choose between 3 possible horror events
        switch (eventType) {
        case 0:
            cout << "The air grows heavy... A shadow looms over you. A shriek pierces your ears, shattering your resolve!\n";
            playJumpScareSound();
            player.increaseFear(10);
            break;
        case 1:
            cout << "The ground trembles violently. A guttural roar echoes, and unseen hands claw at your soul!\n";
            playJumpScareSound();
            player.increaseFear(15);
            break;
        case 2:
            cout << "A chilling growl emanates from the darkness. It feels as though the void itself is devouring you.\n";
            playJumpScareSound();
            player.increaseFear(20);
            break;
        }
        cout << "Fear grips your heart. Current Fear Level: " << player.getFearLevel() << "\n";
    }
}
```

B. Use an Audio Library (SFML, DirectX, Vulkan, etc.)

The game incorporates sound effects and music using the **SFML** (Simple and Fast Multimedia Library). This audio integration enhances the horror experience by immersing the player in an eerie, tension-filled environment.

1. **Background Music:**
   - A looping track of atmospheric music (e.g., background_music.mp3) plays throughout the game to set a foreboding tone.
   - **Implementation:**
     - SFML's sf::Music class is used to load and play the music at 50% volume.
2. **Jump Scare Sound Effects:**
   - Specific sound effects (e.g., jump_scare.mp3) are triggered during random horror events to startle the player and increase their fear level.
   - **Implementation:**

- SFML's sf::Sound class plays sound effects dynamically during gameplay events.

```
#include <ctime>
#include <SFML/Audio.hpp>
#include "GameSaveManager.h"
```

```
// Starts the dungeon exploration and manages gameplay flow.
static void startGame(Player& player) {
    // Load and play background music
    sf::Music backgroundMusic;
    if (!backgroundMusic.openFromFile("assets/sounds/background_music.mp3")) {
        cout << "Error: Could not load background music!" << endl;
        return;
    }
    backgroundMusic.setVolume(50);
    backgroundMusic.setLoop(true);
    backgroundMusic.play();
```

C. Player Progress

File operations are implemented to save and load player progress, ensuring continuity between gameplay sessions. This feature enhances the user experience by allowing players to resume their journey without losing progress.

1. **Saving Player Progress:**
    - Player attributes (e.g., health, sanity, fear level, inventory items) are written to a file (savegame.txt) at specific points in the game.
    - **Implementation:**
        - File I/O functions (std::ofstream) are used to serialize player data into a save file.
2. **Loading Player Progress:**
    - When the game starts, the player can load a previously saved game by reading from the savegame.txt file.
    - **Implementation:**
        - File I/O functions (std::ifstream) deserialize data from the file and restore the player's state.

```cpp
// Main game loop for managing the menu and game flow.
int main() {
    Player player("Wanderer", 60, 70, 60); // Initialize player with starting attributes.
    int choice;
    while (true) {
        displayMenu();
        cin >> choice;
        switch (choice) {
        case 1:
            viewInstructions(); // Display instructions.
            break;
        case 2:
            startGame(player); // Begin the game.
            break;
        case 3: {
            // Save the game state to a file.
            if (GameSaveManager::saveGame(player, "game_save.txt")) {
                cout << "Game saved successfully!" << endl;
            } else {
                cout << "Failed to save the game." << endl;
            }
            break;
        }
        case 4: {
            // Load the game state from a file.
            if (GameSaveManager::loadGame(player, "game_save.txt")) {
                cout << "Game loaded successfully!" << endl;
            } else {
                cout << "Failed to load the game." << endl;
            }
            break;
        }
        case 5:
            cout << "Exiting game. Goodbye!\n";
            return 0; // Exit the program.
        default:
            cout << "Invalid choice. Please try again.\n";
        }
    }
    return 0;
}
```

Appendix:

CursedItem.h

```
#pragma once

#include <iostream>

class Player; // Forward declaration of the Player class to
avoid circular dependencies.

using namespace std;

// Represents a cursed item in the game with a name and a
description of its curse.
class CursedItem {
protected:
    string fItemName;           // Name of the cursed item.
    string fCurseDescription;   // Description of the curse.

public:
    // Default constructor to initialize an empty cursed item.
    CursedItem();

    // Overloaded constructor to initialize a cursed item with
a specific name and curse description.
    CursedItem(const string& aItemName, const string&
aCurseDescription);

    // Virtual destructor to allow proper cleanup in derived
classes.
    virtual ~CursedItem();

    // Retrieves the name of the cursed item.
    string getItemName() const;

    // Retrieves the description of the curse.
    string getCurseDescription() const;

    // Updates the name of the cursed item.
    void setItemName(const std::string& aItemName);

    // Updates the description of the curse.
    void setCurseDescription(const std::string&
aCurseDescription);

    // Defines the behavior when a player collects this cursed
```

```
item.
    void collect(Player* aPlayer);

    // Abstract method to specify how the item is used by a
player. Must be implemented by derived classes.
    virtual void use(Player* aPlayer) = 0;
};
```

CursedItem.cpp

```
#include "CursedItem.h"
#include "Player.h"

// Default constructor
CursedItem::CursedItem()
    : fItemName(""), fCurseDescription("") {}

// Overloaded constructor
CursedItem::CursedItem(const std::string& aItemName, const
std::string& aCurseDescription)
    : fItemName(aItemName),
fCurseDescription(aCurseDescription) {}

// Getter for item name
string CursedItem::getItemName() const {
    return fItemName;
}

// Getter for curse description
string CursedItem::getCurseDescription() const {
    return fCurseDescription;
}

// Set item name
void CursedItem::setItemName(const std::string& aItemName) {
    fItemName = aItemName;
}

// Set curse description
void CursedItem::setCurseDescription(const std::string&
aCurseDescription) {
    fCurseDescription = aCurseDescription;
}

// Adds the cursed item to the player's inventory
void CursedItem::collect(Player* aPlayer)
{
    aPlayer->CollectItem(this);
}
```

```
// Destructor
CursedItem::~CursedItem() {}
```

CursedItemNode.h

```
#pragma once

#include "CursedItem.h"

class CursedItem; // Forward declaration of the CursedItem
class to avoid circular dependencies.

// Represents a node in a linked list, containing a cursed
item and a pointer to the next node.
class CursedItemNode {
public:
    CursedItem* fCursedItem;        // Pointer to a CursedItem
object.
    CursedItemNode* fNext;          // Pointer to the next node
in the linked list.

    // Default constructor to initialize an empty node.
    CursedItemNode();

    // Parameterized constructor to initialize a node with a
cursed item and a pointer to the next node.
    CursedItemNode(CursedItem* aCursedItem, CursedItemNode*
aNext);

    // Destructor to clean up resources associated with the
node.
    ~CursedItemNode();
};
```

CursedItemNode.cpp

```
#include "CursedItemNode.h"

// Default constructor
CursedItemNode::CursedItemNode() : fCursedItem(nullptr),
fNext(nullptr) {}

// Parameterized constructor
CursedItemNode::CursedItemNode(CursedItem* aCursedItem,
CursedItemNode* aNext)
```

```
    : fCursedItem(aCursedItem), fNext(aNext) {}

// Destructor
CursedItemNode::~CursedItemNode() {
    // Safeguard against invalid pointer deletion
    if (fCursedItem) {
        delete fCursedItem; // Delete the dynamically
allocated CursedItem
        fCursedItem = nullptr; // Set pointer to nullptr to
avoid dangling pointers
    }
}
```

CursedItemWrapper.h

```
#pragma once

#include "DungeonEntity.h"
#include "CursedItem.h"

// Represents a wrapper for a CursedItem, making it compatible
with the DungeonEntity system.
class CursedItemWrapper : public DungeonEntity {
private:
    CursedItem* fCursedItem;  // Pointer to the wrapped
CursedItem object.

public:
    // Constructor to initialize the wrapper with a specific
CursedItem.
    explicit CursedItemWrapper(CursedItem* aCursedItem);

    // Destructor to clean up the wrapped CursedItem.
    ~CursedItemWrapper();

    // Retrieves the wrapped CursedItem object.
    CursedItem* getCursedItem() const;

    // Allows a Visitor object to interact with this
CursedItemWrapper.
    void accept(Visitor& visitor) override;
};
```

CursedItemWrapper.cpp

```
#include "CursedItemWrapper.h"
#include "Visitor.h"
```

```
// Constructor: sets up a CursedItemWrapper with a given
cursed item.
CursedItemWrapper::CursedItemWrapper(CursedItem* aCursedItem)
: fCursedItem(aCursedItem) {}

// Returns the cursed item inside this wrapper.
CursedItem* CursedItemWrapper::getCursedItem() const {
    return fCursedItem;
}

// Allows a visitor to interact with this cursed item.
void CursedItemWrapper::accept(Visitor& visitor) {
    if (fCursedItem != nullptr) {
        visitor.Visit(*this); // Let the visitor handle this
cursed item.
    }
}

// Destructor
CursedItemWrapper::~CursedItemWrapper() {}
```

CursedSkillNode.h

```
#pragma once
#include <string>

// Represents a node in a doubly-linked list for cursed
skills.
class CursedSkillNode {
public:
    static CursedSkillNode NIL;  // Sentinel node used to mark
the start or end of the list.

private:
    int fLevel;                  // Skill level associated
with this node.
    std::string fName;           // Name of the skill stored
in this node.
    CursedSkillNode* fNext;      // Pointer to the next node
in the list.
    CursedSkillNode* fPrevious;  // Pointer to the previous
node in the list.

public:
    // Default constructor to create an empty node with
```

```
default values.
    CursedSkillNode();

    // Constructor to create a node with a specific skill name
and level.
    CursedSkillNode(const std::string& aName, int aLevel);

    // Inserts a new node before this one in the list.
    void Prepend(CursedSkillNode* aNode);

    // Inserts a new node after this one in the list.
    void Append(CursedSkillNode* aNode);

    // Removes this node from the list by adjusting links of
adjacent nodes.
    void Remove();

    // Retrieves the level of the skill in this node.
    int getLevel() const;

    // Updates the level of the skill in this node.
    void setLevel(int aLevel);

    // Retrieves the name of the skill in this node.
    std::string getCursedSkillName() const;

    // Retrieves the next node in the list, or nullptr if this
is the last node.
    CursedSkillNode* getNext() const;

    // Retrieves the previous node in the list, or nullptr if
this is the first node.
    CursedSkillNode* getPrevious() const;
};
```

CursedSkillNode.cpp

```
#include "CursedSkillNode.h"

// Defines a static sentinel node (NIL) used as a marker for
list boundaries.
CursedSkillNode CursedSkillNode::NIL;

// Default constructor initializes an empty node with level 0
and links to the NIL node.
CursedSkillNode::CursedSkillNode() : fLevel(0), fName(""),
fNext(&NIL), fPrevious(&NIL) {}
```

```
// Constructor to set up a node with a specific skill name and
level. Links are set to NIL.
CursedSkillNode::CursedSkillNode(const std::string& aName, int
aLevel)
    : fName(aName), fLevel(aLevel), fNext(&NIL),
fPrevious(&NIL) {}

// Inserts a node before the current one and updates the
necessary links in the list.
void CursedSkillNode::Prepend(CursedSkillNode* aNode) {
    aNode->fNext = this;
    if (fPrevious != &NIL) {
        aNode->fPrevious = fPrevious;
        fPrevious->fNext = aNode;
    }
    fPrevious = aNode;
}

// Adds a node after the current one. If this node is at the
end of the list, the new node is appended.
void CursedSkillNode::Append(CursedSkillNode* aNode) {
    if (this == &NIL) return; // Ignore if this node is NIL.
    if (fNext == &NIL) {
        fNext = aNode;
        aNode->fPrevious = this;
        aNode->fNext = &NIL;
    }
    else {
        fNext->Append(aNode);
    }
}

// Removes the current node by reconnecting adjacent nodes.
void CursedSkillNode::Remove() {
    if (fPrevious != &NIL) {
        fPrevious->fNext = fNext;
    }
    if (fNext != &NIL) {
        fNext->fPrevious = fPrevious;
    }
}

// Returns the level of the skill in this node.
int CursedSkillNode::getLevel() const {
    return fLevel;
}

// Updates the skill level for this node.
```

```
void CursedSkillNode::setLevel(int aLevel) {
    fLevel = aLevel;
}

// Returns the name of the skill stored in this node.
std::string CursedSkillNode::getCursedSkillName() const {
    return fName;
}

// Retrieves the next node in the list, or null if this is the
last node.
CursedSkillNode* CursedSkillNode::getNext() const {
    return (fNext == &NIL) ? nullptr : fNext;
}

// Retrieves the previous node in the list, or null if this is
the first node.
CursedSkillNode* CursedSkillNode::getPrevious() const {
    return (fPrevious == &NIL) ? nullptr : fPrevious;
}
```

DemonGuard.h

```
#pragma once

#include <string>
#include "Entity.h"
#include "Visitor.h"
#include "DialogueQueue.h"

// Represents a Demon Guard entity in the game, derived from
the base Entity class.
class DemonGuard : public Entity {
private:
    bool fIsVulnerable;  // Indicates whether the Demon Guard
can be influenced or attacked.
    int fRank;           // Rank of the Demon Guard within its
hierarchy.

public:
    // Default constructor to initialize the Demon Guard with
default values.
    DemonGuard();

    // Constructor to initialize the Demon Guard with a name,
curse description, vulnerability status, and rank.
    DemonGuard(const std::string& aName, const std::string&
aCurseDescription, bool aIsVulnerable, int aRank);
```

```
    // Destructor to clean up resources.
    ~DemonGuard();

    // Checks if the Demon Guard is vulnerable to interactions
or attacks.
    bool isVulnerable() const;

    // Retrieves the rank of the Demon Guard.
    int getRank() const;

    // Handles interaction logic when a player encounters the
Demon Guard.
    bool interact(Player* aPlayer) override;
};
```

DemonGuard.cpp

```
#include "DemonGuard.h"

DemonGuard::DemonGuard()
    : Entity("", ""), fIsVulnerable(false), fRank(0) {}

DemonGuard::DemonGuard(const string& aName, const string&
aCurseDescription, bool aIsVulnerable, int aRank)
    : Entity(aName, aCurseDescription),
fIsVulnerable(aIsVulnerable), fRank(aRank) {}

bool DemonGuard::isVulnerable() const {
    return fIsVulnerable;
}

int DemonGuard::getRank() const {
    return fRank;
}

bool DemonGuard::interact(Player* aPlayer)
{
    fDialogue.enqueue("\nA chilling presence fills the air as
you lock eyes with " + this->fName + ".");

    if (fIsVulnerable) {
        // The DemonGuard is vulnerable, but it still exudes
menace.
        fDialogue.enqueue(this->fName + ": You dare enter my
realm, mortal? What are you hiding in the darkness?");
        fDialogue.enqueue("[PLAYER_INPUT_REQUIRED]");
    }
```

```
    else {
        // The DemonGuard is not vulnerable and is more
threatening.
        fDialogue.enqueue(this->fName + ": Leave now, or the
shadows will consume you.");
    }

    // Process the dialogue queue
    while (!fDialogue.isEmpty()) {
        string dialogueLine = fDialogue.dequeue();

        if (dialogueLine == "[PLAYER_INPUT_REQUIRED]") {
            // Process the player's interaction using the
"Fear" skill level.
            cout << "You: ";

cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
// Clear the input buffer
            string playerResponse;
            getline(cin, playerResponse);

            // Evaluate the player's fear level.
            if (aPlayer->getFearLevel() > 50) {
                fDialogue.enqueue(this->fName + ": Your fear
betrays you. You cannot hide from me, weakling.");
                fDialogue.enqueue(this->fName + ": Surrender
to the inevitable, mortal. Your soul is mine.");
            }
            else {
                fDialogue.enqueue(this->fName + ": Hmph. You
have a spark of courage, mortal. But it will not last.");
            }
        }
        else {
            // Display non-player input dialogue.
            cout << dialogueLine << std::endl;
        }
    }

    // Return whether there is more dialogue to process.
    return !fDialogue.isEmpty();
}

DemonGuard::~DemonGuard() {}
```

DialogueNode.h

```
#pragma once
#include <iostream>

using namespace std;

// A node for storing dialogue text and linking to the next
part of the conversation.
struct DialogueNode
{
    string fData;          // The dialogue text.
    DialogueNode* fNext;   // Pointer to the next dialogue
node.

    // Constructor initializes the dialogue text and next node
pointer.
    DialogueNode(const std::string& aValue, DialogueNode*
aNextDialogueNode = nullptr)
        : fData(aValue), fNext(aNextDialogueNode) {}
};
```

DialogueQueue.h

```
#pragma once
#include <iostream>
#include "DialogueNode.h"

// Manages a queue of dialogue lines for game interactions.
class DialogueQueue {
private:
    DialogueNode* fHead; // Points to the first dialogue
entry.
    DialogueNode* fTail; // Points to the last dialogue entry.
    int fCount;          // Tracks the number of entries in
the queue.

public:
    // Constructs an empty dialogue queue.
    DialogueQueue();

    // Destructor clears all dialogue entries.
    ~DialogueQueue();

    // Adds a new line of dialogue to the queue.
    void enqueue(const std::string& aValue);

    // Removes and returns the dialogue at the front of the
```

```
queue.
    std::string dequeue();

    // Returns the dialogue at the front without removing it.
    std::string peek() const;

    // Checks if the queue is empty.
    bool isEmpty() const;

    // Retrieves the number of dialogue entries in the queue.
    int size() const;
};
```

DialogueQueue.cpp

```
#include "DialogueQueue.h"

// Initializes an empty dialogue queue.
DialogueQueue::DialogueQueue() : fHead(nullptr),
fTail(nullptr), fCount(0) {}

// Adds a new dialogue line to the end of the queue.
void DialogueQueue::enqueue(const std::string& value) {
    DialogueNode* newNode = new DialogueNode(value);
    if (isEmpty()) {
        fHead = fTail = newNode;
    }
    else {
        fTail->fNext = newNode;
        fTail = newNode;
    }
    fCount++;
}

// Removes and returns the dialogue at the front of the queue.
// If the queue is empty, a warning is displayed.
std::string DialogueQueue::dequeue() {
    if (isEmpty()) {
        std::cout << "Queue is empty" << std::endl;
        return "";
    }
    DialogueNode* temp = fHead;
    std::string value = fHead->fData;
    fHead = fHead->fNext;
    delete temp;
    if (fHead == nullptr) {
        fTail = nullptr;
```

```
    }
    fCount--;
    return value;
}

// Retrieves the dialogue at the front without removing it.
// If the queue is empty, a warning is displayed.
std::string DialogueQueue::peek() const {
    if (isEmpty()) {
        std::cout << "Queue is empty" << std::endl;
        return "";
    }
    return fHead->fData;
}

// Checks whether the queue has no dialogue.
bool DialogueQueue::isEmpty() const {
    return fCount == 0;
}

// Returns the number of dialogue entries currently in the
queue.
int DialogueQueue::size() const {
    return fCount;
}

// Destructor clears the queue by deleting all nodes.
DialogueQueue::~DialogueQueue() {
    while (!isEmpty()) {
        dequeue();
    }
}
```

DungeonEntity.h

```
#pragma once

#include "Visitor.h"

// Abstract base class representing an entity in a dungeon
level.
class DungeonEntity {
public:
    // Allows a visitor to interact with the entity.
    // Must be implemented by derived classes to define
specific visitor interactions.
    virtual void accept(Visitor& visitor) = 0;
```

```
    // Virtual destructor to ensure proper cleanup of
resources in derived classes.
    virtual ~DungeonEntity() = default;
};
```

DungeonTree.h

```
#pragma once

#include <string>
#include <stdexcept>
#include "DungeonEntity.h"
#include "Visitor.h"

// Represents a binary tree structure for managing dungeon
entities.
class DungeonTree {
private:
    std::string fDungeonLevel;   // Name of the dungeon level
at this node.
    DungeonEntity* fEntity;      // Pointer to the entity
associated with this node.
    DungeonTree* fLeft;          // Pointer to the left child
subtree.
    DungeonTree* fRight;         // Pointer to the right child
subtree.
    int fFearThreshold;         // Minimum fear level required
to enter this node.

    // Private constructor for initializing the NIL sentinel
node.
    DungeonTree();

public:
    static DungeonTree NIL;      // Sentinel node representing
an empty tree.

    // Constructs a tree node with a dungeon level and an
optional entity.
    DungeonTree(const std::string& aDungeonLevel,
DungeonEntity* aEntity = nullptr, int aFearThreshold = 0);

    // Destructor ensures proper cleanup of child nodes.
    ~DungeonTree();

    // Checks if this node is the NIL sentinel.
    bool isEmpty() const;
```

```
    // Returns the name of the dungeon level.
    const std::string& level() const;

    // Associates an entity with this node.
    void setEntity(DungeonEntity* aEntity);

    // Retrieves the entity associated with this node.
    DungeonEntity* getEntity() const;

    // Returns the left subtree.
    DungeonTree& left() const;

    // Returns the right subtree.
    DungeonTree& right() const;

    // Attaches a subtree as the left child.
    void attachLeft(DungeonTree* aBTree);

    // Attaches a subtree as the right child.
    void attachRight(DungeonTree* aBTree);

    // Detaches and returns the left subtree.
    DungeonTree* detachLeft();

    // Detaches and returns the right subtree.
    DungeonTree* detachRight();

    // Allows a visitor to interact with this node.
    void accept(Visitor& visitor);

    // Getters and setters for fear threshold
    int getFearThreshold() const;

    void setFearThreshold(int threshold);

};
```

DungeonTree.cpp

```
#include "DungeonTree.h"

// Definition of the NIL sentinel node, used as a placeholder
for empty tree nodes.
DungeonTree DungeonTree::NIL = DungeonTree();

// Private constructor initializes the NIL node with default
```

```
values.
DungeonTree::DungeonTree()
    : fDungeonLevel("NIL"), fEntity(nullptr), fLeft(this),
fRight(this), fFearThreshold(0) {}

// Constructor initializes a tree node with a specific level
and optional entity.
DungeonTree::DungeonTree(const std::string& aDungeonLevel,
DungeonEntity* aEntity, int aFearThreshold)
    : fDungeonLevel(aDungeonLevel), fEntity(aEntity),
fLeft(&NIL), fRight(&NIL), fFearThreshold(aFearThreshold) {}

// Associates an entity with this tree node.
void DungeonTree::setEntity(DungeonEntity* aEntity) {
    fEntity = aEntity;
}

// Retrieves the entity associated with this node.
DungeonEntity* DungeonTree::getEntity() const {
    return fEntity;
}

// Checks if this node is the NIL sentinel, indicating it's
empty.
bool DungeonTree::isEmpty() const {
    return this == &NIL;
}


// Returns the name of the dungeon level for this node.
// Throws an exception if the node is empty.
const std::string& DungeonTree::level() const {
    if (isEmpty()) {
        throw std::domain_error("Attempt to access the level
of an empty node!");
    }
    return fDungeonLevel;
}

// Returns the left child of this node.
// Throws an exception if the node is empty.
DungeonTree& DungeonTree::left() const {
    if (isEmpty()) {
        throw std::domain_error("Attempt to access the left
child of an empty node!");
    }
    return *fLeft;
}
```

```
// Returns the right child of this node.
// Throws an exception if the node is empty.
DungeonTree& DungeonTree::right() const {
    if (isEmpty()) {
        throw std::domain_error("Attempt to access the right
child of an empty node!");
    }
    return *fRight;
}

// Attaches a subtree as the left child of this node.
// Throws an exception if the node is empty or already has a
left child.
void DungeonTree::attachLeft(DungeonTree* aBTree) {
    if (isEmpty()) {
        throw std::domain_error("Cannot attach left to an
empty node!");
    }
    if (fLeft != &NIL) {
        throw std::domain_error("Left subtree is already
occupied!");
    }
    fLeft = aBTree;
}

// Attaches a subtree as the right child of this node.
// Throws an exception if the node is empty or already has a
right child.
void DungeonTree::attachRight(DungeonTree* aBTree) {
    if (isEmpty()) {
        throw std::domain_error("Cannot attach right to an
empty node!");
    }
    if (fRight != &NIL) {
        throw std::domain_error("Right subtree is already
occupied!");
    }
    fRight = aBTree;
}
//Dictates whether the player can proceed down that path based
on their current fear level
int DungeonTree::getFearThreshold() const {
    return fFearThreshold;
}

void DungeonTree::setFearThreshold(int threshold) {
    fFearThreshold = threshold;
```

```
}

// Detaches and returns the left child of this node.
// Throws an exception if the node is empty.
DungeonTree* DungeonTree::detachLeft() {
    if (isEmpty()) {
        throw std::domain_error("Cannot detach left from an
empty node!");
    }
    DungeonTree* result = fLeft;
    fLeft = &NIL;
    return result;
}

// Detaches and returns the right child of this node.
// Throws an exception if the node is empty.
DungeonTree* DungeonTree::detachRight() {
    if (isEmpty()) {
        throw std::domain_error("Cannot detach right from an
empty node!");
    }
    DungeonTree* result = fRight;
    fRight = &NIL;
    return result;
}

// Allows a visitor to interact with this node.
void DungeonTree::accept(Visitor& visitor) {
    visitor.Visit(*this);
}

// Destructor clears child nodes unless this is the NIL
sentinel.
DungeonTree::~DungeonTree() {
    if (this != &NIL) {
        delete fLeft;
        delete fRight;
    }
}
```

Entity.h

```
#pragma once

#include <string>
#include "Player.h"
#include "DialogueQueue.h"
```

```
// Represents a base class for all entities in the game.
class Entity {
protected:
    std::string fName;              // Name of the entity.
    std::string fCurseDescription;  // Description of any
curse or lore associated with the entity.
    DialogueQueue fDialogue;        // Queue of dialogues
linked to this entity.

public:
    // Default constructor to initialize the entity with
default values.
    Entity();

    // Constructor to initialize the entity with a specific
name and curse description.
    Entity(const std::string& aName, const std::string&
aCurseDescription);

    // Virtual destructor to ensure proper cleanup of
resources in derived classes.
    virtual ~Entity();

    // Retrieves the name of the entity.
    std::string getName() const;

    // Retrieves the curse description or lore tied to the
entity.
    std::string getCurseDescription() const;

    // Provides access to the dialogue queue associated with
the entity.
    DialogueQueue& getDialogueQueue();

    // Defines interaction behavior when the entity engages
with a player.
    // Must be implemented by derived classes.
    virtual bool interact(Player* aPlayer) = 0;
};
```

Entity.cpp

```
#include "Entity.h"

// Default constructor initializes the entity with an empty
name and no curse description.
```

```
Entity::Entity() : fName(""), fCurseDescription("") {}

// Constructor initializes the entity with a given name and
curse description.
Entity::Entity(const std::string& aName, const std::string&
aCurseDescription)
    : fName(aName), fCurseDescription(aCurseDescription) {}

// Returns the name of the entity.
std::string Entity::getName() const {
    return fName;
}

// Retrieves the curse description associated with the entity.
std::string Entity::getCurseDescription() const {
    return fCurseDescription;
}

// Provides access to the entity's dialogue queue.
DialogueQueue& Entity::getDialogueQueue() {
    return fDialogue;
}

// Virtual destructor ensures proper cleanup of derived
classes.
Entity::~Entity() {}
```

EntityWrapper.h

```
#pragma once

#include "DungeonEntity.h"
#include "Entity.h"

// Wraps an Entity to make it usable as a DungeonEntity in the
game environment.
class EntityWrapper : public DungeonEntity {
private:
    Entity* fEntity;  // Pointer to the wrapped generic
Entity.

public:
    // Constructor to initialize the wrapper with a specific
Entity.
    explicit EntityWrapper(Entity* aEntity);

    // Destructor to clean up resources associated with the
```

```
wrapped Entity.
    ~EntityWrapper();

    // Retrieves the wrapped Entity object.
    Entity* getEntity() const;

    // Allows a Visitor object to interact with this
EntityWrapper.
    void accept(Visitor& aVisitor) override;
};

EntityWrapper.cpp
#include "EntityWrapper.h"
#include "Visitor.h"

// Constructor: sets up an EntityWrapper with a given Entity.
EntityWrapper::EntityWrapper(Entity* aEntity) :
fEntity(aEntity) {}

// Returns the Entity inside this wrapper.
Entity* EntityWrapper::getEntity() const
{
    return fEntity;
}

// Allows a visitor to interact with this Entity.
void EntityWrapper::accept(Visitor& aVisitor) {
    if (fEntity != nullptr) {
        aVisitor.Visit(*this);
    }
}

// Destructor
EntityWrapper::~EntityWrapper() {}
```

GameInteractionVisitor.h

```
#pragma once

#include "Visitor.h"
#include "DungeonTree.h"
#include "SoulShard.h"
#include "Player.h"
#include "SoulShardEntity.h"
#include "EntityWrapper.h"
#include "CursedItemWrapper.h"
#include "HazardWrapper.h"
```

```
#include <iostream>

// Implements the Visitor pattern to handle interactions
between the player and various game elements.
class GameInteractionVisitor : public Visitor {
private:
    Player& fPlayer; // Reference to the player involved in
the interactions.

public:
    // Constructor to initialize the visitor with a specific
player.
    explicit GameInteractionVisitor(Player& aPlayer);

    // Defines interaction behavior when visiting a
DungeonTree node.
    void Visit(DungeonTree& aNode) override;

    // Defines interaction behavior when visiting a Soul Shard
entity.
    void Visit(SoulShardEntity& aSoulShardEntity) override;

    // Defines interaction behavior when visiting a generic
entity wrapped in an EntityWrapper.
    void Visit(EntityWrapper& aEntityWrapper) override;

    // Defines interaction behavior when visiting a cursed
item wrapped in a CursedItemWrapper.
    void Visit(CursedItemWrapper& aCursedItemWrapper)
override;

    // Defines interaction behavior when visiting a hazard
wrapped in a HazardWrapper.
    void Visit(HazardWrapper aHazardWrapper) override;
};
```

GameInteractionVisitor.cpp

```
#include "GameInteractionVisitor.h"

// Constructor implementation
GameInteractionVisitor::GameInteractionVisitor(Player&
aPlayer) : fPlayer(aPlayer) {}

// Visit method for DungeonTree
void GameInteractionVisitor::Visit(DungeonTree& aNode) {
    std::cout << "\nCurrently at " << aNode.level() <<
std::endl;
```

```
    DungeonEntity* entity = aNode.getEntity();
    if (entity != nullptr) {
        entity->accept(*this);
    }
}

// Visit method for SoulShardEntity
void GameInteractionVisitor::Visit(SoulShardEntity&
aSoulShardEntity) {
    SoulShard* shard = aSoulShardEntity.getShard();
    if (shard != nullptr) {
        std::cout << "You see a glowing soul shard with ID: "
<< shard->getShardID() << std::endl;

        int response = 0;
        while (true) {  // Loop until a valid response is
given
            std::cout << "Would you like to collect the shard?
(1 for Yes, 2 for No): ";
            if (!(std::cin >> response)) {  // Input
validation for numeric input
                std::cin.clear();  // Clear the error flag

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');  // Discard invalid input
                std::cout << "Invalid input. Please enter 1
for Yes or 2 for No." << std::endl;
                continue;
            }

            if (response == 1) {
                std::cout << "You collect the shard and add it
to your collection." << std::endl;
                fPlayer.collectShard(*shard);
                break;  // Exit the loop on a valid response
            }
            else if (response == 2) {
                std::cout << "You leave the shard behind,
feeling a sense of unease." << std::endl;
                break;  // Exit the loop on a valid response
            }
            else {
                std::cout << "Invalid choice. Please enter 1
for Yes or 2 for No." << std::endl;
            }
        }
    }
```

```
}


// Visit method for EntityWrapper
void GameInteractionVisitor::Visit(EntityWrapper&
aEntityWrapper) {
    Entity* entity = aEntityWrapper.getEntity();
    if (entity != nullptr) {
        entity->interact(&fPlayer);
    }
}

// Visit method for CursedItemWrapper
void GameInteractionVisitor::Visit(CursedItemWrapper&
aCursedItemWrapper) {
    CursedItem* item = aCursedItemWrapper.getCursedItem();
    if (item != nullptr) {
        std::cout << "You discover a " << item-
>getCurseDescription() << ": " << item->getItemName() <<
std::endl;

        int response = 0;
        while (true) {  // Loop until a valid response is
given
            std::cout << "Would you like to collect the " <<
item->getItemName() << "? (1 for Yes, 2 for No): ";
            if (!(std::cin >> response)) {  // Input
validation for numeric input
                std::cin.clear();  // Clear the error flag

std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');  // Discard invalid input
                std::cout << "Invalid input. Please enter 1
for Yes or 2 for No." << std::endl;
                continue;
            }

            if (response == 1) {
                std::cout << "The " << item->getItemName() <<
" is now in your inventory." << std::endl;
                item->collect(&fPlayer);
                break;  // Exit the loop on a valid response
            }
            else if (response == 2) {
                std::cout << "You leave the " << item-
>getItemName() << " behind, but a dark shadow lingers in the
air." << std::endl;
                break;  // Exit the loop on a valid response
```

```
            }
            else {
                std::cout << "Invalid choice. Please enter 1
for Yes or 2 for No." << std::endl;
            }
        }
    }
}


// Visit method for HazardWrapper
void GameInteractionVisitor::Visit(HazardWrapper
aHazardWrapper) {
    Hazard* hazard = aHazardWrapper.getHazard();
    if (hazard != nullptr) {
        hazard->obstruct(&fPlayer);
    }
}
```

GameSaveManager.h

```
#pragma once
#include "Player.h"
#include <string>

// GameSaveManager handles saving and loading player progress.
class GameSaveManager {
public:
    // Saves the player state to a file.
    static bool saveGame(const Player& player, const
std::string& filename);

    // Loads the player state from a file.
    static bool loadGame(Player& player, const std::string&
filename);
};
```

GameSaveManager.cpp

```
#include <fstream>
#include <iostream>
#include <sstream>
#include "GameSaveManager.h"
#include "Player.h"
#include "Potion.h"
```

```
#include "Hexblade.h"
#include "SoulFragment.h"

bool GameSaveManager::saveGame(const Player& player, const
std::string& filename) {
    std::ofstream outFile(filename);
    if (!outFile.is_open()) {
        std::cerr << "Error: Could not open file for saving: "
<< filename << std::endl;
        return false;
    }

    // Save player stats
    outFile << player.getName() << '\n';
    outFile << player.getHealth() << '\n';
    outFile << player.getSanity() << '\n';
    outFile << player.getTenacity() << '\n';
    outFile << player.getFearLevel() << '\n';

    // Save inventory with item type
    const Inventory& inventory = player.getInventory();
    InventoryIterator it = inventory.begin();
    while (it != inventory.end()) {
        CursedItem* item = *it;
        if (item) {
            if (dynamic_cast<Potion*>(item)) {
                outFile << "ITEM Potion|" << item-
>getItemName() << '|' << item->getCurseDescription() << '\n';
            }
            else if (dynamic_cast<Hexblade*>(item)) {
                outFile << "ITEM Hexblade|" << item-
>getItemName() << '|' << item->getCurseDescription() << '\n';
            }
            else if (dynamic_cast<SoulFragment*>(item)) {
                outFile << "ITEM SoulFragment|" << item-
>getItemName() << '|' << item->getCurseDescription() << '\n';
            }
        }
        ++it;
    }

    // Save cursed skills
    CursedSkillNode* currentSkill =
player.FindCursedSkill(""); // Start from the head
    while (currentSkill != nullptr) {
        outFile << "SKILL " << currentSkill-
>getCursedSkillName() << '|' << currentSkill->getLevel() <<
'\n';
```

```
        currentSkill = currentSkill->getNext();
    }

    outFile.close();
    return true;
}

bool GameSaveManager::loadGame(Player& player, const
std::string& filename) {
    std::ifstream inFile(filename);
    if (!inFile.is_open()) {
        std::cerr << "Error: Could not open file for loading:
" << filename << std::endl;
        return false;
    }

    std::string line;
    std::getline(inFile, line);
    player.setName(line);

    int health, sanity, tenacity, fear;
    inFile >> health >> sanity >> tenacity >> fear;
    inFile.ignore(); // Consume the newline character

    player.setHealth(health);
    player.setSanity(sanity);
    player.setTenacity(tenacity);
    player.increaseFear(fear - player.getFearLevel()); //
Adjust fear level if needed

    // Clear existing inventory and cursed skills
    const_cast<Inventory&>(player.getInventory()).clear(); //
Ensure `clear` is defined in Inventory
    player.DisplayCursedSkill();

    // Load inventory and skills
    while (std::getline(inFile, line)) {
        if (line.find("ITEM ") == 0) { // Check if the line
starts with "ITEM "
            std::istringstream itemStream(line.substr(5)); //
Remove "ITEM "
            std::string itemType, itemName, curseDescription;
            std::getline(itemStream, itemType, '|');
            std::getline(itemStream, itemName, '|');
            std::getline(itemStream, curseDescription);

            if (itemType == "Potion") {
                player.CollectItem(new Potion(itemName,
```

```
curseDescription, 10));
            }
            else if (itemType == "Hexblade") {
                player.CollectItem(new Hexblade(itemName,
curseDescription, 5));
            }
            else if (itemType == "SoulFragment") {
                player.CollectItem(new SoulFragment(itemName,
curseDescription, 3));
            }
            else {
                std::cerr << "Unknown item type: " << itemType
<< std::endl;
            }
        }
        else if (line.find("SKILL ") == 0) { // Check if the
line starts with "SKILL "
            std::istringstream skillStream(line.substr(6)); //
Remove "SKILL "
            std::string skillName;
            int level;
            std::getline(skillStream, skillName, '|');
            skillStream >> level;

            player.AddCursedSkill(skillName, level);
        }
    }

    inFile.close();
    return true;
}
```

Hazard.h

```
#pragma once

#include <iostream>
#include "Player.h"

using namespace std;

// Represents a base class for hazardous elements in the game.
class Hazard {
protected:
    string fName;               // Name of the hazard.
    string fCursedDescription;  // Description of the curse or
hazardous effect.
```

```
public:
    // Default constructor to initialize a generic hazard.
    Hazard();

    // Constructor to initialize a hazard with a specific name
and curse description.
    Hazard(const string& aName, const string&
aCurseDescription);

    // Virtual destructor to ensure proper cleanup of
resources in derived classes.
    virtual ~Hazard();

    // Pure virtual function to define the interaction logic
with a player.
    // Must be implemented by derived classes.
    virtual void obstruct(Player* aPlayer) = 0;

    // Retrieves the name of the hazard.
    string getName() const;

    // Retrieves the description of the curse or hazard
effect.
    string getCursedDescription() const;
};
```

Hazard.cpp

```
#include "Hazard.h"

// Default constructor
Hazard::Hazard() : fName(""), fCursedDescription("") {}

// Overloaded constructor
Hazard::Hazard(const string& aName, const string&
aDescription)
    : fName(aName), fCursedDescription(aDescription) {}

// Getter method for the hazard's name
string Hazard::getName() const {
    return fName;
}

// Getter method for the hazard's description
string Hazard::getCursedDescription() const {
    return fCursedDescription;
```

```
}

// Destructor
Hazard::~Hazard() {}
```

HazardWrapper.h

```
#pragma once

#include "DungeonEntity.h"
#include "Hazard.h"

// Wraps a Hazard object to make it compatible with the
DungeonEntity system.
class HazardWrapper : public DungeonEntity {
private:
    Hazard* fHazard;  // Pointer to the wrapped Hazard object.

public:
    // Constructor to initialize the wrapper with a specific
Hazard.
    explicit HazardWrapper(Hazard* aHazard);

    // Destructor to clean up resources associated with the
wrapped Hazard.
    ~HazardWrapper();

    // Retrieves the wrapped Hazard object.
    Hazard* getHazard() const;

    // Allows a Visitor object to interact with this
HazardWrapper.
    void accept(Visitor& visitor) override;
};
```

HazardWrapper.cpp

```
#include "HazardWrapper.h"
#include "Visitor.h"

// Constructor: sets up a HazardWrapper with a given Hazard.
HazardWrapper::HazardWrapper(Hazard* aHazard) :
fHazard(aHazard) {}

// Returns the Hazard inside this wrapper.
Hazard* HazardWrapper::getHazard() const
```

```
{
    return fHazard;
}

// Allows a visitor to interact with this hazard.
void HazardWrapper::accept(Visitor& visitor) {
    if (fHazard != nullptr) {
        visitor.Visit(*this);
    }
}

// Destructor
HazardWrapper::~HazardWrapper() {}
```

Hexblade.h

```
#pragma once

#include "CursedItem.h"

// Represents a cursed weapon in the game, inheriting from the
CursedItem class.
class Hexblade : public CursedItem {
private:
    int fCursedLevel;  // Level of the curse associated with
the Hexblade.

public:
    // Default constructor to initialize a generic Hexblade.
    Hexblade();

    // Parameterized constructor to initialize a Hexblade with
specific attributes.
    Hexblade(const string& aItemName, const string&
aCurseDescription, int aCursedLevel);

    // Virtual destructor to ensure proper cleanup of
resources.
    virtual ~Hexblade();

    // Retrieves the cursed level of the Hexblade.
    int getCursedLevel() const;

    // Overrides the 'use' method from the CursedItem class to
define how a player uses the Hexblade.
    void use(Player* aPlayer) override;
};
```

Hexblade.cpp

```
#include "Hexblade.h"
#include "Player.h"

// Default constructor
Hexblade::Hexblade() : CursedItem(), fCursedLevel(0) {}

// Overloaded constructor
Hexblade::Hexblade(const string& aItemName, const string&
aCurseDescription, int aCursedLevel)
    : CursedItem(aItemName, aCurseDescription),
fCursedLevel(aCursedLevel) {}

// Retrieves the cursed level of the hexblade.
int Hexblade::getCursedLevel() const {
    return fCursedLevel;
}

// Uses the hexblade to empower the player.
void Hexblade::use(Player* aPlayer)
{
    aPlayer->boostTenacity(fCursedLevel);
    cout << fItemName << " has increased your tenacity. " <<
"Current tenacity: " << aPlayer->getTenacity() << endl;
}

// Destructor for Hexblade
Hexblade::~Hexblade() {}
```

Inventory.h

```
#pragma once

#include "InventoryIterator.h"

// Forward declarations to avoid circular dependencies.
class CursedItemNode;
class CursedItem;
class InventoryIterator;

// Represents a collection of cursed items carried by a
player.
class Inventory {
private:
```

```
    CursedItemNode* fHead;  // Pointer to the first node in
the linked list of cursed items.

public:
    // Default constructor to initialize an empty inventory.
    Inventory();

    // Destructor to clear and deallocate the inventory.
    ~Inventory();

    // Adds a new cursed item to the inventory.
    void addCursedItem(CursedItem* aItem);

    // Removes and returns a cursed item by its name.
    CursedItem* removeCursedItem(const std::string&
aItemName);

    // Retrieves a cursed item by its name without removing
it.
    CursedItem* getItem(const std::string& aItemName) const;

    // Returns an iterator pointing to the beginning of the
inventory.
    InventoryIterator begin() const;

    // Returns an iterator representing the end of the
inventory.
    InventoryIterator end() const;

    // Clears all items from the inventory.
    void clear();
};
```

Inventory.cpp

```
#include "Inventory.h"

// Constructor for Inventory
Inventory::Inventory() : fHead(nullptr) {}

// Adds a new cursed item to the inventory by creating a new
CursedItemNode and linking it at the beginning of the list.
void Inventory::addCursedItem(CursedItem* aItem) {
    fHead = new CursedItemNode(aItem, fHead); // New node
becomes the new head of the list.
}
```

```
// Removes a cursed item from the inventory by name and
returns a pointer to the item, if found.
CursedItem* Inventory::removeCursedItem(const std::string&
aItemName) {
    CursedItemNode** current = &fHead;
    while (*current != nullptr) {
        if ((*current)->fCursedItem->getItemName() ==
aItemName) { // Check if current item matches the name.
            CursedItemNode* toDelete = *current;
            CursedItem* item = toDelete->fCursedItem; //
Extract the item before deleting the node.
            *current = (*current)->fNext;
            toDelete->fCursedItem = nullptr;  // Avoid
deleting the item itself.
            delete toDelete;
            return item; // Return the removed item.
        }
        current = &((*current)->fNext);
    }
    return nullptr; // Return null if the item was not found.
}

// Retrieves a cursed item by name from the inventory and
returns it, or nullptr if not found.
CursedItem* Inventory::getItem(const std::string& aItemName)
const {
    for (CursedItemNode* current = fHead; current != nullptr;
current = current->fNext) {
        if (current->fCursedItem->getItemName() == aItemName)
{ // Check if current item matches the name.
            return current->fCursedItem; // Return the found
item.
        }
    }
    return nullptr; // Return null if the item is not found.
}

// Returns an iterator positioned at the start of the
inventory.
InventoryIterator Inventory::begin() const {
    return InventoryIterator(fHead);
}

// Returns an iterator representing the end of the inventory.
InventoryIterator Inventory::end() const {
    return InventoryIterator(nullptr); // End iterator is a
null pointer.
}
```

```
// Destructor for Inventory. Iterates through the list and
deletes all item nodes to free memory.
Inventory::~Inventory() {
    clear(); // Reuse the clear method to clean up memory
}

// Implementation of the clear method
void Inventory::clear() {
    CursedItemNode* current = fHead;
    while (current != nullptr) {
        CursedItemNode* toDelete = current;
        current = current->fNext;
        delete toDelete->fCursedItem; // Delete the item
itself
        delete toDelete; // Delete the node
    }
    fHead = nullptr; // Reset the head pointer
}
```

InventoryIterator.h

```
#pragma once

#include "CursedItemNode.h"

// Custom iterator for traversing the player's inventory of
cursed items.
class InventoryIterator {
private:
    CursedItemNode* fCurrentNode; // Points to the current
node in the inventory.

public:
    // Constructor initializes the iterator to start at a
specific node.
    InventoryIterator(CursedItemNode* aStartNode);

    // Dereference operator retrieves the item at the current
position.
    CursedItem* operator*() const;

    // Prefix increment operator moves to the next item.
    InventoryIterator& operator++();

    // Equality operator checks if two iterators point to the
same node.
    bool operator==(const InventoryIterator& aOther) const;
```

```
    // Inequality operator checks if two iterators point to
different nodes.
    bool operator!=(const InventoryIterator& aOther) const;
};
```

InventoryIterator.cpp

```
#include "InventoryIterator.h"

// Initializes the iterator to point to the given node in the
inventory.
InventoryIterator::InventoryIterator(CursedItemNode*
aStartNode) : fCurrentNode(aStartNode) {}

// Dereferences the iterator to access the current item.
CursedItem* InventoryIterator::operator*() const {
    return fCurrentNode->fCursedItem;
}

// Moves the iterator to the next item (prefix increment).
InventoryIterator& InventoryIterator::operator++() {
    fCurrentNode = fCurrentNode->fNext;
    return *this;
}

// Compares two iterators for equality.
bool InventoryIterator::operator==(const InventoryIterator&
aOther) const {
    return fCurrentNode == aOther.fCurrentNode;
}

// Compares two iterators for inequality.
bool InventoryIterator::operator!=(const InventoryIterator&
aOther) const {
    return fCurrentNode != aOther.fCurrentNode;
}
```

LostSoul.h

```
#pragma once

#include <iostream>
#include <string>
#include "Entity.h"
#include "DialogueQueue.h"
```

```h
#include "Visitor.h"

// Represents a lost soul in the game, inheriting from the
base Entity class.
class LostSoul : public Entity {
private:
    int fStrength;       // Strength or combat ability of the
LostSoul.
    int fHauntLevel;     // Haunting level of the LostSoul
(formerly stealth level).
    DialogueQueue fDialogue; // Dialogue associated with the
LostSoul.

public:
    // Default constructor to initialize a generic LostSoul.
    LostSoul();

    // Constructor to initialize a LostSoul with specific
attributes.
    LostSoul(const string& aName, const string&
aCurseDescription, int aStrength, int aHauntLevel);

    // Destructor to clean up resources associated with the
LostSoul.
    ~LostSoul();

    // Retrieves the strength of the LostSoul.
    int getStrength() const;

    // Overrides the interaction behavior to define how a
LostSoul interacts with a player.
    bool interact(Player* aPlayer) override;
};
```

LostSoul.cpp

```cpp
#include "LostSoul.h"

// Default Constructor
LostSoul::LostSoul() : Entity("", ""), fStrength(0),
fHauntLevel(0) {}

// Overloaded Constructor
LostSoul::LostSoul(const string& aName, const string&
aCurseDescription, int aStrength, int aHauntLevel)
    : Entity(aName, aCurseDescription), fStrength(aStrength),
fHauntLevel(aHauntLevel) {}
```

```
int LostSoul::getStrength() const {
    return fStrength;
}

bool LostSoul::interact(Player* aPlayer)
{
    fDialogue.enqueue("\nA chill sweeps over you as you
encounter the tormented presence of " + this->fName + ".");
    fDialogue.enqueue(this->fName + ": Spare me, mortal... a
shard of your soul... or suffer my eternal curse.");

    fDialogue.enqueue("[PLAYER_INPUT_REQUIRED]");

    while (!fDialogue.isEmpty()) {
        std::string dialogueLine = fDialogue.dequeue();

        if (dialogueLine == "[PLAYER_INPUT_REQUIRED]") {
            // Provide the player with multiple options to
choose from.
            std::cout << "Choose your fate: " << std::endl;
            std::cout << "1: Offer a shard of your soul to
ease the torment." << std::endl;
            std::cout << "2: Flee in terror, hoping the soul
won't catch you." << std::endl;
            std::cout << "3: Stand firm against the soul's
overwhelming presence." << std::endl;
            std::cout << "Your response (1/2/3): ";
            int choice;
            std::cin >> choice;

            if (choice == 1) {
                fDialogue.enqueue("You offer a shard of your
soul, watching in horror as it absorbs your essence.");
                fDialogue.enqueue(this->fName + ": Peace...
for now. But remember, I will always haunt you, mortal.");
            }
            else if (choice == 2) {
                if (aPlayer->ReadCursedSkillLevel("Courage") >
50) {
                    fDialogue.enqueue("You flee, heart racing,
as the wailing soul fades into the darkness.");
                    fDialogue.enqueue("You feel a cold breath
on your neck... but escape... for now.");
                }
                else {
                    fDialogue.enqueue("Your legs falter as the
soul's icy grip clutches at you. You cannot escape.");
```

```
                    fDialogue.enqueue("The soul's wail haunts
you... you feel your strength waning.");
                    aPlayer->setHealth(aPlayer->getHealth() -
15); // Reduce health for failing to escape
                }
            }
            else if (choice == 3) {
                if (aPlayer->getTenacity() > this-
>getStrength()) {
                    fDialogue.enqueue("Your willpower burns
like a beacon, driving the lost soul away.");
                    fDialogue.enqueue(this->fName + ": You are
strong, mortal... but I will return.");
                }
                else {
                    fDialogue.enqueue("The soul's presence is
too powerful. It overwhelms you, draining your life force.");
                    fDialogue.enqueue("You feel your strength
fade as the soul feeds on your fear...");
                    aPlayer->setHealth(aPlayer->getHealth() -
20); // Drain health if failed
                }
            }
            else {
                fDialogue.enqueue("The lost soul's voice
echoes in your mind, growing louder. You must choose.");
            }
        }
        else {
            std::cout << dialogueLine << std::endl;
        }
    }
    return fDialogue.isEmpty();
}

LostSoul::~LostSoul() {}
```

MawOfDarkness.h

```
#pragma once

#include "Hazard.h"

// Represents a deadly trap called the "Maw of Darkness,"
inheriting from the Hazard class.
class MawOfDarkness : public Hazard {
private:
```

```
    int fRadius; // The radius of the Maw of Darkness.

public:
    // Default constructor to initialize a generic Maw of
Darkness.
    MawOfDarkness();

    // Constructor to initialize the Maw of Darkness with
specific attributes.
    MawOfDarkness(const string& aName, const string&
aDescription, int aRadius);

    // Destructor to clean up resources associated with the
Maw of Darkness.
    ~MawOfDarkness();

    // Retrieves the radius of the Maw of Darkness.
    int getRadius() const;

    // Overrides the obstruct method from the Hazard class to
define how the Maw of Darkness affects a player.
    void obstruct(Player* aPlayer) override;
};
```

MawOfDarkness.cpp

```
#include "MawOfDarkness.h"

// Default constructor initializes the Maw of Darkness with a
default radius
MawOfDarkness::MawOfDarkness() : Hazard(), fRadius(0) {}

// Constructor with parameters sets the name, description, and
radius of the Maw of Darkness
MawOfDarkness::MawOfDarkness(const string& aName, const
string& aDescription, int aRadius)
    : Hazard(aName, aDescription), fRadius(aRadius) {}

// Returns the radius of the Maw of Darkness
int MawOfDarkness::getRadius() const
{
    return fRadius;
}

// Function to handle the player's interaction with the Maw of
Darkness
void MawOfDarkness::obstruct(Player* aPlayer)
{
```

```
    int lPlayerChoice;
    cout << "You are confronted with the Maw of Darkness,
spanning " << fRadius << " meters. Do you dare to attempt a
leap?" << endl;
    cout << "Jump" << endl;
    cout << "1. Yes, I will jump" << endl;
    cout << "2. No, I will find another way" << endl;

    cin >> lPlayerChoice;

    if (lPlayerChoice == 1)
    {
        if (fRadius < 50)
        {
            if (aPlayer->ReadCursedSkillLevel("Strength") >
70)
            {
                aPlayer->setSanity(aPlayer->getSanity() - 10);
                cout << "You successfully jumped over the dark
pit with minimal effort." << endl;
            }
            else
            {
                aPlayer->setSanity(aPlayer->getSanity() - 20);
                cout << "The jump drained your sanity. Your
sanity is now " << aPlayer->getSanity() << endl;
            }
        }
        else
        {
            cout << "The Maw of Darkness is too wide to cross.
You'll need another path." << endl;
        }
    }
    else
    {
        cout << "You decide to seek a safer route around the
Maw of Darkness." << endl;
    }
}

// Destructor for the Maw of Darkness class
MawOfDarkness::~MawOfDarkness() {}
```

Player.h

```
#pragma once

#include <iostream>
#include <string>
#include "CursedSkillNode.h"
#include "Inventory.h"
#include "SoulShard.h"
#include "SoulShardStack.h"

// Represents a player character in the game, managing
attributes, inventory, skills, and collected shards.
class Player {
private:
    std::string fPlayer;         // Player's name.
    int fHealth;                 // Current health level.
    int fSanity;                 // Current sanity level.
    int fTenacity;               // Player's tenacity level.
    int fShardsCollected;        // Number of shards collected.
    int fFearLevel;              // Current fear level.
    CursedSkillNode* fHeadptr;   // Pointer to the first cursed
skill node.
    Inventory fInventory;        // Player's inventory of
items.
    ShardStack fShardStack;      // Stack of soul shards
collected by the player.

public:
    // Constructors and Destructor
    Player(); // Default constructor.
    Player(const std::string& aPlayer, int aHealth, int
aSanity, int aTenacity); // Overloaded constructor.
    ~Player(); // Destructor to clean up resources.

    // Setters
    void setName(const std::string& newName);          // Sets
the player's name.
    void setHealth(int newHealth);                     // Sets
the player's health.
    void setFearLevel(int newFear);                    // Sets
the player's fear.
    void setSanity(int newSanity);                     // Sets
the player's sanity.
    void setTenacity(int aTenacity);                   // Sets
the player's tenacity.

    // Getters
    std::string getName() const;                       // Gets
```

```
the player's name.
    int getHealth() const;                              // Gets
the player's health.
    int getSanity() const;                              // Gets
the player's sanity.
    int getTenacity() const;                            // Gets
the player's tenacity.
    int getShardsCollected() const;                     // Gets
the number of shards collected.
    int getFearLevel() const;                           // Gets
the player's fear level.
    const Inventory& getInventory() const;              // Gets
the player's inventory.

    // Fear Management
    void increaseFear(int amount);                      //
Increases the player's fear level.
    void decreaseFear(int amount);                      //
Decreases the player's fear level.

    // Display and Update Player Attributes
    void PrintDetails() const;                          //
Displays the player's current details.
    void boostTenacity(int aTenacity);                  //
Enhances the player's tenacity.
    void restoreHealth(int aHealthBoost);               //
Restores the player's health.
    void restoreSanity(int aSanityBoost);               //
Restores the player's sanity.

    // Cursed Skill Management
    void AddCursedSkill(const std::string& aCursedSkillName,
int aLevel); // Adds a new cursed skill.
    CursedSkillNode* FindCursedSkill(const std::string&
aCursedSkillName) const; // Finds a specific cursed skill.
    int ReadCursedSkillLevel(const std::string&
aCursedSkillName) const; // Retrieves the level of a specific
cursed skill.
    void ModifyCursedSkill(const std::string&
aCursedSkillName, int aLevel) const; // Modifies the level of
a cursed skill.
    void DisplayCursedSkill() const;                // Displays
all cursed skills.

    // Inventory Management
    void CollectItem(CursedItem* aItem);            // Adds a
cursed item to the inventory.
    CursedItem* removeCursedItemFromInventory(const
```

```
std::string& CursedItemName); // Removes a cursed item from
the inventory.
    void displayInventory() const;              // Displays
the contents of the inventory.

    // Soul Shard Management
    void collectShard(const SoulShard& shard);  // Collects a
soul shard.
    SoulShard useShard();                       // Uses a
collected soul shard.
    void displayShards() const;                 // Displays
all collected soul shards.

    // Serialization and Deserialization
    std::string serializeInventory() const;     // Serializes
the inventory into a string format.
    std::string serializeCursedSkills() const;  // Serializes
cursed skills into a string format.
    void deserializeInventory(const std::string&
inventoryData); // Loads inventory from a serialized string.
    void deserializeCursedSkills(const std::string&
skillData);  // Loads cursed skills from a serialized string.
};
```

Player.cpp

```
#include "Player.h"
#include "Potion.h"
#include "Hexblade.h"
#include "SoulFragment.h"
#include <sstream>

// Default constructor initializes player attributes to
default values.
Player::Player()
    : fPlayer(""), fHealth(100), fSanity(100), fTenacity(0),
fShardsCollected(0), fFearLevel(0), fHeadptr(nullptr),
fInventory() {}

// Constructor allows initializing player attributes with
custom values.
Player::Player(const std::string& aPlayerName, int aHealth,
int aSanity, int aTenacity)
    : fPlayer(aPlayerName), fHealth(aHealth),
fSanity(aSanity), fTenacity(aTenacity), fShardsCollected(0),
```

```
fFearLevel(0), fInventory() {
    fHeadptr = nullptr;
}

// Returns the player's name.
std::string Player::getName() const { return fPlayer; }

// Returns the player's health.
int Player::getHealth() const { return fHealth; }

// Returns the player's sanity.
int Player::getSanity() const { return fSanity; }

// Returns the player's tenacity.
int Player::getTenacity() const { return fTenacity; }

// Returns the number of shards collected.
int Player::getShardsCollected() const { return
fShardStack.size(); }

// Returns the player's inventory.
const Inventory& Player::getInventory() const { return
fInventory; }

// Returns the player's fear level.
int Player::getFearLevel() const { return fFearLevel; }

// Updates the player's name.
void Player::setName(const std::string& aPlayerName) { fPlayer
= aPlayerName; }

// Updates the player's health.
void Player::setHealth(int aHealth) { fHealth = aHealth; }

// Updates the player's fear.
void Player::setFearLevel(int newFear) { fFearLevel = newFear;
}

// Updates the player's sanity.
void Player::setSanity(int aSanity) { fSanity = aSanity; }

// Updates the player's tenacity.
void Player::setTenacity(int aTenacity) { fTenacity =
aTenacity; }

// Increases the player's fear level, capped at 100.
void Player::increaseFear(int amount) {
    fFearLevel += amount;
```

```
    if (fFearLevel > 100) fFearLevel = 100;
}

// Decreases the player's fear level, ensuring it doesn't go
below 0.
void Player::decreaseFear(int amount) {
    fFearLevel -= amount;
    if (fFearLevel < 0) fFearLevel = 0;
}

// Displays the player's current attributes.
void Player::PrintDetails() const {
    std::cout << "Health: " << fHealth << std::endl;
    std::cout << "Sanity: " << fSanity << std::endl;
    std::cout << "Tenacity: " << fTenacity << std::endl;
    std::cout << "Fear: " << fFearLevel << std::endl;
}

// Increases the player's tenacity, capped at 100.
void Player::boostTenacity(int aTenacity) {
    if (getTenacity() != 100) {
        setTenacity(getTenacity() + aTenacity);
    }
}

// Restores the player's health, capped at 100.
void Player::restoreHealth(int aHealthBoost) {
    fHealth += aHealthBoost;
    if (fHealth > 100) fHealth = 100;
}

// Restores the player's sanity, capped at 100.
void Player::restoreSanity(int aSanityBoost) {
    fSanity += aSanityBoost;
    if (fSanity > 100) fSanity = 100;
}

// Adds a cursed skill to the player's skill list.
void Player::AddCursedSkill(const std::string&
aCursedSkillName, int aLevel) {
    CursedSkillNode* newNode = new
CursedSkillNode(aCursedSkillName, aLevel);
    if (!fHeadptr) {
        fHeadptr = newNode;
    }
    else {
        CursedSkillNode* current = fHeadptr;
        while (current->getNext() && current->getNext() !=
```

```
&CursedSkillNode::NIL) {
            current = current->getNext();
        }
        current->Append(newNode);
    }
}

// Serializes the player's inventory into a string format.
std::string Player::serializeInventory() const {
    std::ostringstream oss;
    for (InventoryIterator it = fInventory.begin(); it !=
fInventory.end(); ++it) {
        CursedItem* item = *it;
        if (item) {
            oss << item->getItemName() << "|" << item-
>getCurseDescription() << '\n';
        }
    }
    return oss.str();
}

// Serializes the player's cursed skills into a string format.
std::string Player::serializeCursedSkills() const {
    std::ostringstream oss;
    CursedSkillNode* currentSkill = fHeadptr;
    while (currentSkill) {
        oss << currentSkill->getCursedSkillName() << "|" <<
currentSkill->getLevel() << '\n';
        currentSkill = currentSkill->getNext();
    }
    return oss.str();
}

// Deserializes inventory data from a string.
void Player::deserializeInventory(const std::string&
inventoryData) {
    std::istringstream iss(inventoryData);
    std::string line;
    while (std::getline(iss, line)) {
        size_t separator = line.find('|');
        if (separator != std::string::npos) {
            std::string itemName = line.substr(0, separator);
            std::string curseDescription =
line.substr(separator + 1);
            if (itemName.find("Potion") != std::string::npos)
{
                CollectItem(new Potion(itemName,
curseDescription, 10));
```

```
            }
            else if (itemName.find("Hexblade") !=
std::string::npos) {
                CollectItem(new Hexblade(itemName,
curseDescription, 5));
            }
            else if (itemName.find("SoulFragment") !=
std::string::npos) {
                CollectItem(new SoulFragment(itemName,
curseDescription, 3));
            }
        }
    }
}

// Deserializes cursed skills data from a string.
void Player::deserializeCursedSkills(const std::string&
skillData) {
    std::istringstream iss(skillData);
    std::string line;
    while (std::getline(iss, line)) {
        size_t separator = line.find('|');
        if (separator != std::string::npos) {
            std::string skillName = line.substr(0, separator);
            int level = std::stoi(line.substr(separator + 1));
            AddCursedSkill(skillName, level);
        }
    }
}

// Finds a cursed skill node by its name.
CursedSkillNode* Player::FindCursedSkill(const std::string&
aCursedSkillName) const {
    CursedSkillNode* current = fHeadptr;
    while (current) {
        if (current->getCursedSkillName() == aCursedSkillName)
{
            return current;
        }
        current = current->getNext();
    }
    return nullptr;
}

// Reads the level of a specific cursed skill by its name.
int Player::ReadCursedSkillLevel(const std::string&
aCursedSkillName) const {
    CursedSkillNode* skill =
```

```
FindCursedSkill(aCursedSkillName);
    return skill ? skill->getLevel() : 0;
}

// Modifies the level of an existing cursed skill.
void Player::ModifyCursedSkill(const std::string&
aCursedSkillName, int aLevel) const {
    CursedSkillNode* skill =
FindCursedSkill(aCursedSkillName);
    if (skill) {
        skill->setLevel(aLevel);
    }
    else {
        std::cout << "No skill found for modification." <<
std::endl;
    }
}

// Displays all cursed skills the player possesses.
void Player::DisplayCursedSkill() const {
    CursedSkillNode* current = fHeadptr;
    while (current) {
        std::cout << "Cursed Skill: " << current-
>getCursedSkillName()
            << ", Level: " << current->getLevel() <<
std::endl;
        current = current->getNext();
    }
}

// Adds an item to the player's inventory.
void Player::CollectItem(CursedItem* item) {
    fInventory.addCursedItem(item); // Assuming fInventory is
the player's inventory
    std::cout << "Collected: " << item->getItemName() <<
std::endl;
}


// Removes an item from the player's inventory by name.
CursedItem* Player::removeCursedItemFromInventory(const
std::string& CursedItemName) {
    return fInventory.removeCursedItem(CursedItemName);
}

// Displays all items in the player's inventory.
void Player::displayInventory() const {
    for (InventoryIterator it = fInventory.begin(); it !=
```

```
fInventory.end(); ++it) {
        CursedItem* currentItem = *it;
        if (currentItem) {
            std::cout << "Cursed Item: " << currentItem-
>getItemName()
                << ", Curse: " << currentItem-
>getCurseDescription() << std::endl;
        }
        else {
            std::cout << "Your inventory is empty." <<
std::endl;
        }
    }
}

// Collects a soul shard and adds it to the stack.
void Player::collectShard(const SoulShard& shard) {
    fShardStack.push(shard);
}

// Removes and returns the top shard from the stack.
SoulShard Player::useShard() {
    if (fShardStack.isEmpty()) {
        std::cout << "No soul shards available to use." <<
std::endl;
    }
    return fShardStack.pop();
}

// Displays all collected soul shards.
void Player::displayShards() const {
    SoulShard* shardsSnapshot = fShardStack.getSnapshot();
    if (!shardsSnapshot) {
        std::cout << "No shards collected." << std::endl;
        return;
    }
    int size = fShardStack.size();
    for (int i = 0; i < size; ++i) {
        std::cout << "Shard ID: " <<
shardsSnapshot[i].getShardID() << std::endl;
    }
    delete[] shardsSnapshot;
}

// Destructor to clean up cursed skills.
Player::~Player() {
    CursedSkillNode* current = fHeadptr;
    while (current) {
```

```
        CursedSkillNode* next = current->getNext();
        delete current;
        current = next;
    }
}
```

Potion.h

```
#pragma once
#include "CursedItem.h"
#include <unordered_map> // Include unordered_map for the hash
table

// Represents a potion in the game, inheriting from the
CursedItem class.
class Potion : public CursedItem {
private:
    int fRestorativeValue; // The restorative value of the
potion.
    static std::unordered_map<std::string, int>
fPotionEffects; // Hashtable to store potion effects

public:
    // Default constructor to initialize a generic potion.
    Potion();

    // Constructor to initialize a potion with a specific
name, curse description, and restorative value.
    Potion(const std::string& aItemName, const std::string&
aCurseDescription, int aRestorativeValue);

    // Destructor to clean up resources associated with the
potion.
    virtual ~Potion();

    // Retrieves the restorative value of the potion.
    int getRestorativeValue() const;

    // Overrides the 'use' method from CursedItem to define
how the potion interacts with a player.
    void use(Player* aPlayer) override;

    // Static function to add an effect to the hash table
    static void addPotionEffect(const std::string& effectName,
int effectValue);
```

```
    // Static function to retrieve an effect from the hash
table
    static int getPotionEffect(const std::string& effectName);
};
```

Potion.cpp

```
#include "Potion.h"
#include "Player.h"
#include <iostream>
#include <unordered_map> // Include unordered_map for hash
table usage

// Initialize the static hash table
std::unordered_map<std::string, int> Potion::fPotionEffects;

// Default constructor
Potion::Potion() : CursedItem(), fRestorativeValue(0) {}

// Overloaded constructor
Potion::Potion(const std::string& aItemName, const
std::string& aCurseDescription, int aRestorativeValue)
    : CursedItem(aItemName, aCurseDescription),
fRestorativeValue(aRestorativeValue) {}

// Retrieves the restorative value of the potion.
int Potion::getRestorativeValue() const {
    return fRestorativeValue;
}

// Player uses potion to restore sanity
void Potion::use(Player* aPlayer) {
    int lBeforeBoost = aPlayer->getSanity();
    aPlayer->restoreSanity(fRestorativeValue);
    std::cout << "\n" << fItemName << " boosts " << aPlayer-
>getName() << "'s sanity from "
        << lBeforeBoost << " to " << aPlayer->getSanity() <<
std::endl;
    aPlayer->removeCursedItemFromInventory(fItemName);
}

// Static function to add an effect to the hash table
void Potion::addPotionEffect(const std::string& effectName,
int effectValue) {
```

```
    fPotionEffects[effectName] = effectValue;
    std::cout << "Added effect: " << effectName << " with
value: " << effectValue << std::endl;
}

// Static function to retrieve an effect from the hash table
int Potion::getPotionEffect(const std::string& effectName) {
    if (fPotionEffects.find(effectName) !=
fPotionEffects.end()) {
        return fPotionEffects[effectName];
    }
    else {
        std::cerr << "Effect " << effectName << " not found!"
<< std::endl;
        return 0;
    }
}

// Destructor
Potion::~Potion() {}
```

SoulFragment.h

```
#pragma once

#include "CursedItem.h"

// Represents soul fragments in the game, inheriting from the
CursedItem class.
class SoulFragment : public CursedItem {
private:
    int fQuantity; // The quantity of soul fragments.

public:
    // Default constructor to initialize a generic
SoulFragment.
    SoulFragment();

    // Constructor to initialize a SoulFragment with specific
attributes.
    SoulFragment(const string& aItemName, const string&
aCurseDescription, int aQuantity);

    // Destructor to clean up resources associated with the
SoulFragment.
```

```
    virtual ~SoulFragment();

    // Retrieves the quantity of soul fragments.
    int getQuantity() const;

    // Sets the quantity of soul fragments.
    void setQuantity(int aQuantity);

    // Overrides the 'use' method from CursedItem to define
how a player can use soul fragments.
    void use(Player* aPlayer) override;
};
```

SoulFragment.cpp

```
#include "SoulFragment.h"
#include "Player.h"

// Default constructor
SoulFragment::SoulFragment() : CursedItem(), fQuantity(0) {}

// Overloaded constructor
SoulFragment::SoulFragment(const string& aItemName, const
string& aCurseDescription, int aQuantity)
    : CursedItem(aItemName, aCurseDescription),
fQuantity(aQuantity) {}

// Retrieves the quantity of soul fragments.
int SoulFragment :: getQuantity() const {
    return fQuantity;
}

// Used by player to appease entities
void SoulFragment :: use(Player* aPlayer)
{
    cout << "\nYou offer " << fQuantity << " " << fItemName <<
" as an appeasement." << endl;
    aPlayer->removeCursedItemFromInventory(fItemName);
}

// Destructor
SoulFragment::~SoulFragment() {}
```

SoulShard.h

```
#pragma once

#include "DungeonEntity.h"

// Represents a soul shard in the game, uniquely identified by
an ID.
class SoulShard {
private:
    int fShardID; // Unique identifier for the soul shard.

public:
    // Default constructor to initialize a generic SoulShard.
    SoulShard();

    // Constructor to initialize a SoulShard with a specific
ID.
    SoulShard(int aShardID);

    // Destructor to clean up resources associated with the
SoulShard.
    ~SoulShard();

    // Retrieves the unique ID of the soul shard.
    int getShardID() const;
};
```

SoulShard.cpp

```
#include "SoulShard.h"

// Default Constructor
SoulShard::SoulShard() : fShardID(0) {}

// Overloaded Constructor
SoulShard::SoulShard(int aShardID) : fShardID(aShardID) {}

// Getter for shard ID
int SoulShard::getShardID() const
{
    return fShardID;
}

// Destructor
SoulShard::~SoulShard() {}
```

SoulShardEntity.h

```
#pragma once

#include "DungeonEntity.h"
#include "SoulShard.h"

// Represents an entity in the game that holds a soul shard.
class SoulShardEntity : public DungeonEntity {
private:
    SoulShard* fShard; // Pointer to the associated SoulShard
object.

public:
    // Constructor to initialize the entity with a specific
SoulShard.
    explicit SoulShardEntity(SoulShard* aShard);

    // Destructor to clean up resources associated with the
SoulShardEntity.
    ~SoulShardEntity();

    // Retrieves the SoulShard associated with this entity.
    SoulShard* getShard() const;

    // Overrides the accept method to allow a Visitor to
interact with this entity.
    void accept(Visitor& visitor) override;
};
```

SoulShardEntity.cpp

```
#include "SoulShardEntity.h"
#include "Visitor.h"

// Constructor
SoulShardEntity::SoulShardEntity(SoulShard* aShard) :
fShard(aShard) {}

// Returns the SoulShard pointer associated with this entity.
SoulShard* SoulShardEntity::getShard() const {
    return fShard;
}
```

```
// Accepts a visitor to interact with this SoulShardEntity.
// The visitor's Visit method is called if a valid shard is
present.
void SoulShardEntity::accept(Visitor& visitor) {
    if (fShard != nullptr) {
        visitor.Visit(*this);
    }
}

// Destructor
SoulShardEntity::~SoulShardEntity() {}
```

SoulShardStack.h

```
#pragma once

#include "SoulShard.h"
#include <iostream>

using namespace std;

// Implements a stack specifically for storing SoulShards,
with a fixed maximum size.
class ShardStack {
private:
    static const int MAX_SIZE = 5; // Maximum number of
SoulShards that can be stored in the stack.
    SoulShard fStack[MAX_SIZE];    // Array used to implement
the stack.
    int fTopIndex;                 // Index indicating the top
of the stack.

public:
    // Constructor to initialize an empty stack.
    ShardStack();

    // Destructor to clean up resources.
    ~ShardStack();

    // Stack operations.

    // Checks if the stack is empty.
    bool isEmpty() const;

    // Checks if the stack is full.
    bool isFull() const;
```

```
    // Retrieves the current number of SoulShards in the
stack.
    int size() const;

    // Adds a SoulShard to the top of the stack.
    void push(const SoulShard& shard);

    // Removes and returns the SoulShard at the top of the
stack.
    SoulShard pop();

    // Retrieves the SoulShard at the top of the stack without
removing it.
    SoulShard peek() const;

    // Creates a snapshot of the stack's contents.
    SoulShard* getSnapshot() const;
};
```

SoulShardStack.cpp

```
#include "SoulShardStack.h"

// Constructor initializes the stack as empty.
ShardStack::ShardStack() : fTopIndex(-1) {}

// Returns true if the stack is empty.
bool ShardStack::isEmpty() const {
    return fTopIndex == -1;
}

// Returns true if the stack is full.
bool ShardStack::isFull() const {
    return fTopIndex == MAX_SIZE - 1;
}

// Returns the current size of the stack.
int ShardStack::size() const {
    return fTopIndex + 1;
}

// Adds a SoulShard to the top of the stack. Displays a
warning if the stack is full.
void ShardStack::push(const SoulShard& shard) {
    if (isFull()) {
        cout << "You cannot stack more shards" << endl;
    }
```

```
    fStack[++fTopIndex] = shard;
}

// Removes and returns the top SoulShard from the stack.
Displays a warning if the stack is empty.
SoulShard ShardStack::pop() {
    if (isEmpty())
    {
        cout << "No soul shards left to use." << endl;
    }
    return fStack[fTopIndex--];
}

// Returns the top SoulShard without removing it. Throws an
error if the stack is empty.
SoulShard ShardStack::peek() const {
    if (isEmpty()) {
        throw std::runtime_error("No soul shards left to
use.");
    }
    return fStack[fTopIndex];
}

// Returns a snapshot of the stack's current contents. Returns
nullptr if the stack is empty.
SoulShard* ShardStack::getSnapshot() const {
    if (isEmpty()) {
        return nullptr;
    }
    int currentSize = size();
    SoulShard* snapshot = new SoulShard[currentSize];
    for (int i = 0; i < currentSize; ++i) {
        snapshot[i] = fStack[i];
    }
    return snapshot;
}


// Destructor
ShardStack::~ShardStack() {}
```

Visitor.h

```
#pragma once

// Forward declarations of various game entities the visitor
can interact with.
```

```
class DungeonTree;
class SoulShardEntity;
class EntityWrapper;
class CursedItemWrapper;
class HazardWrapper;

// Base class for the Visitor pattern, allowing different
interactions with game entities.
class Visitor {
public:
    // Handles interaction with a dungeon node.
    virtual void Visit(DungeonTree& aNode) = 0;

    // Handles interaction with a soul shard entity.
    virtual void Visit(SoulShardEntity& aShardEntity) = 0;

    // Handles interaction with an entity wrapped for special
behavior.
    virtual void Visit(EntityWrapper& aEntityWrapper) = 0;

    // Handles interaction with a cursed item wrapped for
gameplay use.
    virtual void Visit(CursedItemWrapper& aCursedItemWrapper)
= 0;

    // Handles interaction with a hazardous game element.
    virtual void Visit(HazardWrapper aHazardWrapper) = 0;

    // Virtual destructor to ensure proper cleanup in derived
classes.
    virtual ~Visitor() = default;
};
```

WallOfShadows.h

```
#pragma once

#include "Hazard.h"

// Represents a shadowy, supernatural barrier in the game,
inheriting from the Hazard class.
class WallOfShadows : public Hazard {
private:
    int fHeight; // The height of the shadowy wall.

public:
    // Default constructor to initialize a generic Wall of
```

```
Shadows.
    WallOfShadows();

    // Constructor to initialize a Wall of Shadows with
specific attributes.
    WallOfShadows(const string& aName, const string&
aCurseDescription, int aHeight);

    // Destructor to clean up resources associated with the
Wall of Shadows.
    ~WallOfShadows();

    // Retrieves the height of the wall.
    int getHeight();

    // Overrides the obstruct method from Hazard to define how
the wall affects a player.
    void obstruct(Player* aPlayer) override;
};
```

WallOfShadows.cpp

```
#include "WallOfShadows.h"

// Default constructor
WallOfShadows::WallOfShadows() : Hazard(), fHeight(0) {}

// Constructor with parameters
WallOfShadows::WallOfShadows(const string& aName, const
string& aCurseDescription, int aHeight)
    : Hazard(aName, aCurseDescription), fHeight(aHeight) {}

// Returns the height of the shadowy wall.
int WallOfShadows::getHeight()
{
    return fHeight;
}

// Function to handle player's interaction with the shadowy
wall
void WallOfShadows::obstruct(Player* aPlayer)
{
    int lPlayerChoice;
    cout << "A towering wall of shadows stands " << fHeight <<
" feet tall before you." << endl;
    cout << "Attempt to climb?" << endl;
    cout << "1. Yes" << endl;
```

```
    cout << "2. No" << endl;

    cin >> lPlayerChoice;

    if (lPlayerChoice == 1)
    {
        if (fHeight < 20)
        {
            if (aPlayer->ReadCursedSkillLevel("Strength") >
80)
            {
                aPlayer->setSanity(aPlayer->getSanity() - 10);
                cout << "You scale the shadowy wall with
little difficulty." << endl;
            }
            else
            {
                aPlayer->setSanity(aPlayer->getSanity() - 20);
                cout << "The shadows drain you... your sanity
drops to " << aPlayer->getSanity() << endl;
            }
        }
        else
        {
            cout << "The wall is too tall to climb. Seek
another path." << endl;
        }
    }
    else
    {
        cout << "You decide to take a different route." <<
endl;
    }
}

// Destructor
WallOfShadows::~WallOfShadows() {}
```

Main.cpp

```
#include <iostream>
#include "Player.h"
#include "DungeonTree.h"
#include "SoulShardEntity.h"
#include "EntityWrapper.h"
#include "HazardWrapper.h"
#include "CursedItemWrapper.h"
```

```cpp
#include "DemonGuard.h"
#include "LostSoul.h"
#include "WallOfShadows.h"
#include "MawOfDarkness.h"
#include "SoulFragment.h"
#include "Hexblade.h"
#include "Potion.h"
#include "SoulShard.h"
#include "GameInteractionVisitor.h"
#include "Visitor.h"
#include <cstdlib>
#include <ctime>
#include <SFML/Audio.hpp>
#include "GameSaveManager.h"

using namespace std;

// Displays the main game menu to the player.
void displayMenu() {
    cout << "--------------------------------------------------
--------------------\n";
    cout << "               DESCENT INTO THE INFERNAL ABYSS
\n";
    cout << "--------------------------------------------------
--------------------\n\n";
    cout << "The void whispers your name... Can you escape the
horror?\n";
    cout << "++++++++++++\n";
    cout << "GAME MENU\n";
    cout << "++++++++++++\n";
    cout << "1. View the Infernal Guide\n";
    cout << "2. Enter the Abyss\n";
    cout << "3. Save Game\n";
    cout << "4. Load Game\n";
    cout << "5. Embrace the Darkness (Exit)\n";
    cout << "Enter your choice: ";
}

// Displays in-game player options during dungeon traversal.
static void displayPlayerOptions() {
    cout << "\nPlayer Choices:\n";
    cout << "1. Venture deeper into the abyss\n";
    cout << "2. Reflect on your mortal attributes\n";
    cout << "3. Inspect your cursed inventory\n";
    cout << "4. Gaze upon your twisted skills\n";
    cout << "5. Count your soul shards\n";
    cout << "6. Surrender to the void (Exit to Game Menu)\n";
    cout << "Enter your choice: ";
```

```
}

// Finds the next empty dungeon level starting from a specific
index.
static int findNextEmptyDungeonLevel(DungeonTree* levels[],
int numberOfLevels, int start) {
    for (int i = start; i < numberOfLevels; ++i) {
        if (levels[i]->getEntity() == nullptr) {
            return i;
        }
    }
    return -1; // No empty level found.
}

// Plays a jump scare sound effect to heighten tension.
static void playJumpScareSound() {
    sf::SoundBuffer buffer;
    if (!buffer.loadFromFile("assets/sounds/jump_scare.mp3"))
{
        cerr << "Error: Could not load jump scare sound file."
<< endl;
        return;
    }
    sf::Sound sound;
    sound.setBuffer(buffer);
    sound.setVolume(10);
    sound.play();

    // Ensures the sound plays completely before continuing.
    sf::sleep(sf::milliseconds(1500)); // Adjust duration if
needed.
}


// Randomly triggers horror events, increasing the player's
fear level.
static void triggerRandomEvent(Player& player) {
    int eventChance = rand() % 10; // 10% chance of triggering
an event
    if (eventChance < 3) {          // 30% chance of triggering
a fear event
        int eventType = rand() % 3; // Choose between 3
possible horror events
        switch (eventType) {
        case 0:
            cout << "The air grows heavy... A shadow looms
over you. A shriek pierces your ears, shattering your
resolve!\n";
```

```
            playJumpScareSound();
            player.increaseFear(10);
            break;
        case 1:
            cout << "The ground trembles violently. A guttural
roar echoes, and unseen hands claw at your soul!\n";
            playJumpScareSound();
            player.increaseFear(15);
            break;
        case 2:
            cout << "A chilling growl emanates from the
darkness. It feels as though the void itself is devouring
you.\n";
            playJumpScareSound();
            player.increaseFear(20);
            break;
        }
        cout << "Fear grips your heart. Current Fear Level: "
<< player.getFearLevel() << "\n";
    }
}


// Starts the dungeon exploration and manages gameplay flow.
static void startGame(Player& player) {
    // Load and play background music
    sf::Music backgroundMusic;
    if
(!backgroundMusic.openFromFile("assets/sounds/background_music
.mp3")) {
        cout << "Error: Could not load background music!" <<
endl;
        return;
    }
    backgroundMusic.setVolume(50);
    backgroundMusic.setLoop(true);
    backgroundMusic.play();

    // Initialize player's starting skills.
    player.AddCursedSkill("Intellect", 10);
    player.AddCursedSkill("Survivor", 90);
    player.AddCursedSkill("Brute Force", 50);
    player.AddCursedSkill("Dark Power", 49);

    // Display initial player stats and skills.
    cout << "The abyss calls...\n";
    cout << "Your cursed attributes:\n";
    cout << "Health: " << player.getHealth() << "\n";
```

```
    cout << "Sanity: " << player.getSanity() << "\n";
    cout << "Tenacity: " << player.getTenacity() << "\n";
    cout << "Fear: " << player.getFearLevel() << "\n";
    cout << "Intellect : " <<
player.ReadCursedSkillLevel("Intellect") << endl;
    cout << "Survivor : " <<
player.ReadCursedSkillLevel("Survivor") << endl;
    cout << "Brute Force : " <<
player.ReadCursedSkillLevel("Brute Force") << endl;
    cout << "Dark Power : " <<
player.ReadCursedSkillLevel("Dark Power") << endl;
    cout << "Best of luck...\n";


    // Create and initialize dungeon levels.
    DungeonTree* root = new DungeonTree("Maw of Darkness");
    DungeonTree* levels[] = {
        root,
        new DungeonTree("Hall of Shadows"),
        new DungeonTree("Pit of Despair"),
        new DungeonTree("Echoing Chamber"),
        new DungeonTree("Crypt of the Forsaken"),
        new DungeonTree("The Abyssal Gate"),
        new DungeonTree("Labyrinth of Suffering"),
        new DungeonTree("Wall of Wails"),
        new DungeonTree("Doomed Path"),
        new DungeonTree("Temple of the Lost"),
        new DungeonTree("Corridor of Silence"),
        new DungeonTree("Desolate Chamber"),
        new DungeonTree("Chamber of Curses"),
        new DungeonTree("Veil of Sorrow"),
        new DungeonTree("Eternal Void")
    };

    const int numberOfLevels = sizeof(levels) /
sizeof(levels[0]);

    // Link dungeon levels in sequence.

    for (int i = 0; i < numberOfLevels - 1; ++i) {
        levels[i]->attachRight(levels[i + 1]);
    }


    // Populate dungeon with shards, entities, hazards, and
items.
    SoulShard* shard1 = new SoulShard(1);
    SoulShard* shard2 = new SoulShard(22);
```

```
    SoulShard* shard3 = new SoulShard(333);
    SoulShard* shard4 = new SoulShard(4444);
    SoulShardEntity* shardEntity1 = new
SoulShardEntity(shard1);
    SoulShardEntity* shardEntity2 = new
SoulShardEntity(shard2);
    SoulShardEntity* shardEntity3 = new
SoulShardEntity(shard3);
    SoulShardEntity* shardEntity4 = new
SoulShardEntity(shard4);

    DemonGuard* guardA = new DemonGuard("Demon Lord",
"Guarding the void", true, 1);
    DemonGuard* guardB = new DemonGuard("Sentinel", "Watching
the shadows", false, 3);
    LostSoul* soulA = new LostSoul("Trapped Soul", "Wandering
aimlessly", 40, 5);
    LostSoul* soulB = new LostSoul("Mournful Wraith", "Weeping
silently", 89, 3);
    EntityWrapper* entityWrapperGuardA = new
EntityWrapper(guardA);
    EntityWrapper* entityWrapperGuardB = new
EntityWrapper(guardB);
    EntityWrapper* entityWrapperSoulA = new
EntityWrapper(soulA);
    EntityWrapper* entityWrapperSoulB = new
EntityWrapper(soulB);

    Hazard* maw = new MawOfDarkness("Abyssal Chasm", "A
bottomless pit of despair", 14);
    Hazard* wall = new WallOfShadows("Shadowy Wall", "A wall
that obscures reality", 5);
    Hazard* wall2 = new WallOfShadows("Veil of Shadows", "An
impenetrable darkness", 36);
    HazardWrapper* hazardWrapperMaw = new HazardWrapper(maw);
    HazardWrapper* hazardWrapperWall = new
HazardWrapper(wall);
    HazardWrapper* hazardWrapperWall2 = new
HazardWrapper(wall2);

    CursedItem* soulFragments = new SoulFragment("Soul
Fragments", "Fragments of tortured souls", 6);
    CursedItem* hexblade = new Hexblade("Blade of the Damned",
"A blade cursed by the ancients", 16);
    CursedItem* potion = new Potion("Dark Elixir", "A vial
filled with a sinister liquid", 15);
    CursedItemWrapper* itemWrapperSoulFragments = new
CursedItemWrapper(soulFragments);
```

```
    CursedItemWrapper* itemWrapperHexblade = new
CursedItemWrapper(hexblade);
    CursedItemWrapper* itemWrapperPotion = new
CursedItemWrapper(potion);

    SoulShardEntity* shardEntities[] = { shardEntity1,
shardEntity2, shardEntity3, shardEntity4 };
    EntityWrapper* entityWrappers[] = { entityWrapperGuardA,
entityWrapperGuardB, entityWrapperSoulA, entityWrapperSoulB };
    HazardWrapper* hazardWrappers[] = { hazardWrapperMaw,
hazardWrapperWall, hazardWrapperWall2 };
    CursedItemWrapper* itemWrappers[] = {
itemWrapperSoulFragments, itemWrapperHexblade,
itemWrapperPotion };

    const int numberOfShards = sizeof(shardEntities) /
sizeof(shardEntities[0]);
    const int numberOfEntities = sizeof(entityWrappers) /
sizeof(entityWrappers[0]);
    const int numberOfHazards = sizeof(hazardWrappers) /
sizeof(hazardWrappers[0]);
    const int numberOfItems = sizeof(itemWrappers) /
sizeof(itemWrappers[0]);

    srand(static_cast<unsigned int>(time(NULL)));

    int levelIndex = 0;
    string targetLevelName = "Eternal Void";

    for (int i = 0; i < numberOfEntities; ++i) {
        int randomIndex = rand() % numberOfLevels;
        if (levels[randomIndex]->level() != targetLevelName &&
levels[randomIndex]->getEntity() == nullptr) {
            levels[randomIndex]->setEntity(entityWrappers[i]);
        }
    }

    for (int i = 0; i < numberOfShards; ++i) {
        levelIndex = findNextEmptyDungeonLevel(levels,
numberOfLevels, levelIndex);
        if (levelIndex != -1 && levels[levelIndex]->level() !=
targetLevelName) {
            levels[levelIndex]->setEntity(shardEntities[i]);
            levelIndex += numberOfLevels / numberOfShards;
        }
    }

    for (int i = 0; i < numberOfHazards + numberOfItems; ++i)
```

```
{
        int randomIndex = rand() % numberOfLevels;
        if (levels[randomIndex]->level() != targetLevelName &&
levels[randomIndex]->getEntity() == nullptr) {
            if (i < numberOfHazards) {
                levels[randomIndex]-
>setEntity(hazardWrappers[i]);
            }
            else {
                levels[randomIndex]->setEntity(itemWrappers[i
- numberOfHazards]);
            }
        }
    }

    for (size_t i = 0; i < numberOfLevels;) {
        DungeonTree* currentLevel = levels[i];
        GameInteractionVisitor visitor(player);
        currentLevel->accept(visitor);

        // Trigger a random horror event
        triggerRandomEvent(player);

        if (i == numberOfLevels - 1) {
            if (currentLevel->level() == "Eternal Void") {
                if (player.getShardsCollected() == 4) {
                    cout << "You stand before the Eternal
Void. The air is thick with despair.\n";
                    cout << "Four soul shards pulsate with
dark energy, yearning to be offered to the portal.\n";
                    int progress = 0;

cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                    while (player.getShardsCollected() != 0 &&
progress < 100) {
                        cout << "Offer a soul shard to the
portal (type 'Offer' to proceed): ";
                        string command;
                        getline(cin, command);

                        if (command == "Offer") {
                            SoulShard usedShard =
player.useShard();
                            progress += 25;
                            cout << "Shard ID " <<
usedShard.getShardID() << " consumed by the portal.\n";
                            cout << "The portal begins to
awaken. Progress: " << progress << "%...\n";
```

```
                        }
                        else {
                            cout << "The portal rejects your
hesitation. Type 'Offer' to proceed." << endl;
                        }
                    }

                    if (progress == 100) {
                        cout << "The portal bursts open,
releasing a deafening roar!\n";
                        cout << "You step through the gateway,
escaping the abyss at last. But its darkness will forever
haunt you...\n";
                        cout << "Returning to the main
menu...\n";
                        return; // Return to main menu after
winning.
                    }
                }
                else {
                    cout << "You have reached the Eternal
Void, but the shards elude you. The portal remains sealed.\n";
                    cout << "The shadows close in, consuming
your soul. Game Over.\n";
                    cout << "Returning to the main menu...\n";
                    return; // Return to main menu after
losing.
                }
            }
        }


        bool inventoryIsEmpty = true;
        for (InventoryIterator it =
player.getInventory().begin(); it !=
player.getInventory().end(); ++it) {
            if (*it != nullptr) {
                inventoryIsEmpty = false;
                break;
            }
        }

        bool playerDecisionMade = false;
        while (!playerDecisionMade) {
            displayPlayerOptions();
            int choice;
            cin >> choice;
```

```
            switch (choice) {
            case 1:
                cout << "\nYou gather your resolve and move
deeper into the abyss...\n";
                i++;
                playerDecisionMade = true;
                break;
            case 2:
                player.PrintDetails();
                break;
            case 3:
                player.displayInventory();
                if (!inventoryIsEmpty) {
                    cout << "\nDo you wish to use a cursed
item from your inventory? (yes/no): ";
                    string useItemResponse;
                    cin >> useItemResponse;

                    if (useItemResponse == "yes") {
                        cout << "Enter the name of the item
you want to use: ";
                        string itemName;
                        cin.ignore();
                        getline(cin, itemName);

                        bool itemFound = false;
                        for (InventoryIterator it =
player.getInventory().begin(); it !=
player.getInventory().end(); ++it) {
                            CursedItem* currentItem = *it;
                            if (currentItem && currentItem-
>getItemName() == itemName) {
                                currentItem->use(&player);
                                itemFound = true;
                                break;
                            }
                        }

                        if (!itemFound) {
                            cout << "The item '" << itemName
<< "' does not exist in your inventory. The void mocks your
efforts.\n";
                        }
                    }
                    else {
                        cout << "You choose to keep your items
and prepare for what lies ahead...\n";
                    }
```

```
                }
                else {
                    cout << "Your inventory is empty. The void
offers no solace...\n";
                }
                break;
            case 4:
                player.DisplayCursedSkill();
                break;
            case 5:
                player.displayShards();
                break;
            case 6:
                cout << "You succumb to the abyss,
surrendering your fate to the darkness. Goodbye.\n";
                return;
            default:
                cout << "The void does not understand your
choice. Please try again...\n";
                break;
            }
        }

        if (i >= numberOfLevels) {
            cout << "You have reached the end of your journey.
The abyss stares back...\n";
            break;
        }
    }

    for (size_t i = 0; i < numberOfLevels; ++i) {
        delete levels[i];
    }
    delete root;
    backgroundMusic.stop();
    cout << "The game has ended. The abyss remembers your
soul...\n";

}

// Displays an immersive guide for the player with a game-like
UI.
static void viewInstructions() {
    cout <<
"================================================\n";
    cout << "                INFERNAL GUIDE
\n";
    cout <<
```

```
"================================================\n";
    cout << "|
|\n";
    cout << "|   Welcome, wanderer. The abyss beckons.
|\n";
    cout << "|   Prepare yourself for the horrors ahead.
|\n";
    cout << "|
|\n";
    cout <<
"================================================\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|                OBJECTIVE
|\n";
    cout << "+-----------------------------------------
+\n";
    cout << "| Traverse the cursed dungeon, gathering soul
|\n";
    cout << "| shards to unlock the Eternal Void. Escape
|\n";
    cout << "| its horrors... or become one with them.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|             CURSED DUNGEON
|\n";
    cout << "+-----------------------------------------
+\n";
    cout << "| A labyrinth of despair filled with shadows,
|\n";
    cout << "| traps, and cursed entities. Each level
|\n";
    cout << "| offers danger and reward. Tread carefully.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|              SOUL SHARDS
|\n";
    cout << "+-----------------------------------------
+\n";
```

```
    cout << "| Scattered remnants of tortured souls.
|\n";
    cout << "| Collect them to unlock the portal. Beware,
|\n";
    cout << "| they are heavily guarded by the abyss.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|       HEALTH, SANITY, AND FEAR
|\n";
    cout << "+-----------------------------------------
+\n";
    cout << "| - **Health:** Stay alive by avoiding traps
|\n";
    cout << "|   and surviving encounters.
|\n";
    cout << "| - **Sanity:** Hold onto your mind; losing
|\n";
    cout << "|   it will leave you vulnerable.
|\n";
    cout << "| - **Fear:** The horrors of the abyss will
|\n";
    cout << "|   grip your heart. Manage it wisely.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|              INVENTORY
|\n";
    cout << "+-----------------------------------------
+\n";
    cout << "| Cursed items can aid or harm you. Some have
|\n";
    cout << "| unique powers, but beware of their curses.
|\n";
    cout << "| Use them at the right time to survive.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|            CURSED SKILLS
```

```
|\n";
    cout << "+-----------------------------------------------
+\n";
    cout << "| Power comes at a price. Enhance your
|\n";
    cout << "| abilities strategically, but remember:
|\n";
    cout << "| the abyss never gives without taking.
|\n";
    cout << "+-----------------------------------------------
+\n\n";

    cout << "+-----------------------------------------------
+\n";
    cout << "|         HORRORS AND ENTITIES
|\n";
    cout << "+-----------------------------------------------
+\n";
    cout << "| The dungeon is filled with demons, lost
|\n";
    cout << "| souls, and abominations. Some guard shards,
|\n";
    cout << "| others seek to destroy you. Approach with
|\n";
    cout << "| caution... or run.
|\n";
    cout << "+-----------------------------------------------
+\n\n";

    cout << "+-----------------------------------------------
+\n";
    cout << "|              HAZARDS
|\n";
    cout << "+-----------------------------------------------
+\n";
    cout << "| Spikes, pits, and traps await the unwary.
|\n";
    cout << "| Study the environment and tread carefully.
|\n";
    cout << "+-----------------------------------------------
+\n\n";

    cout << "+-----------------------------------------------
+\n";
    cout << "|            THE ETERNAL VOID
|\n";
    cout << "+-----------------------------------------------
+\n";
```

```
    cout << "| The ultimate challenge lies ahead. Gather
|\n";
    cout << "| all soul shards to awaken the portal. Only
|\n";
    cout << "| then can you hope to escape. Beware... the
|\n";
    cout << "| void is always watching.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout << "+-----------------------------------------
+\n";
    cout << "|             SURVIVAL TIPS
|\n";
    cout << "+-----------------------------------------
+\n";
    cout << "| - Explore thoroughly: Items and secrets
|\n";
    cout << "|   are hidden everywhere.
|\n";
    cout << "| - Balance your stats: Don't let health,
|\n";
    cout << "|   sanity, or fear drop too low.
|\n";
    cout << "| - Use items wisely: Some curses may help or
|\n";
    cout << "|   hinder your progress.
|\n";
    cout << "| - Plan ahead: Every decision has weight.
|\n";
    cout << "| - Above all... keep moving forward.
|\n";
    cout << "+-----------------------------------------
+\n\n";

    cout <<
"=====================================================\n";
    cout << " May the shadows guide your path. Survive, if you
can.\n";
    cout <<
"=====================================================\n\n";
}

// Navigates the dungeon.
static void navigateDungeon(DungeonTree* currentLevel, Player&
player) {
    GameInteractionVisitor visitor(player);
```

```
    while (currentLevel != &DungeonTree::NIL) {
        cout << "Entering: " << currentLevel->level() << endl;

        if (player.getFearLevel() >= currentLevel-
>getFearThreshold()) {
            cout << "You brave the fears of this level.\n";
            currentLevel->accept(visitor);
            triggerRandomEvent(player);
        }
        else {
            cout << "Too fearful to proceed. Seeking less
daunting path...\n";
            player.decreaseFear(5);
        }

        // Simplified pathfinding logic for clarity.
        if (!currentLevel->left().isEmpty() &&
player.getFearLevel() > currentLevel-
>left().getFearThreshold()) {
            currentLevel = &currentLevel->left();
        }
        else if (!currentLevel->right().isEmpty() &&
player.getFearLevel() > currentLevel-
>right().getFearThreshold()) {
            currentLevel = &currentLevel->right();
        }
        else {
            cout << "No path forward, turning back...\n";
            break;
        }
    }
}


// Main game loop for managing the menu and game flow.
int main() {
    Player player("Wanderer", 60, 70, 60); // Initialize
player with starting attributes.
    int choice;
    while (true) {
        displayMenu();
        cin >> choice;
        switch (choice) {
        case 1:
            viewInstructions(); // Display instructions.
            break;
        case 2:
```

```
            startGame(player); // Begin the game.
            break;
        case 3: {
            // Save the game state to a file.
            if (GameSaveManager::saveGame(player,
"game_save.txt")) {
                cout << "Game saved successfully!" << endl;
            } else {
                cout << "Failed to save the game." << endl;
            }
            break;
        }
        case 4: {
            // Load the game state from a file.
            if (GameSaveManager::loadGame(player,
"game_save.txt")) {
                cout << "Game loaded successfully!" << endl;
            } else {
                cout << "Failed to load the game." << endl;
            }
            break;
        }
        case 5:
            cout << "Exiting game. Goodbye!\n";
            return 0; // Exit the program.
        default:
            cout << "Invalid choice. Please try again.\n";
        }
    }
    return 0;
}
```