Assessment Cover Sheet

This Assessment Cover Sheet is only to be attached to hard copy submission of assessments.

ASSESSMENT DETAILS

Unit title	Data Structures and Patterns	Tutorial /Lab Group		Office use only
Unit code	COS30008	Due date		
Name of lecturer/tutor				
Assignment title	Programming Project			Faculty or school date stamp

STUDENT(S) DETAILS

	Student Name(s)	Student ID Number(s)
(1)	Ryan Alistair Anak Allen	101215012
(2)		
(3)		
(4)		
(5)		
(6)		

DECLARATION AND STATEMENT OF AUTHORSHIP

1. I/we have not impersonated, or allowed myself/ourselves to be impersonated by any person for the purposes of this assessment.
2. This assessment is my/our original work and no part of it has been copied from any other source except where due acknowledgement is made.
3. No part of this assessment has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/tutor concerned.
4. I/we have not previously submitted this work for this or any other course/unit.
5. I/we give permission for my/our assessment response to be reproduced, communicated, compared and archived for plagiarism detection, benchmarking or educational purposes.

I/we understand that:

6. Plagiarism is the presentation of the work, idea or creation of another person as though it is your own. It is a form of cheating and is a very serious academic offence that may lead to exclusion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.

Student signature/s

I/we declare that I/we have read and understood the declaration and statement of authorship.

(1)	Ryan	(4)	
(2)		(5)	
(3)		(6)	

Further information relating to the penalties for plagiarism, which range from a formal caution to expulsion from the University is contained on the Current Students website at **https://www.swinburne.edu.my/current-students/manage-course/exams-results-assessment**

Copies of this form can be downloaded from the Student Forms web page at **https://www.swinburne.edu.my/current-students/manage-course/exams-results-assessment/how-to-submit-work.php**

1 Link to Video Demonstration

https://youtu.be/3IQUT0tILuQ?si=16bpA0OEREQi2cpJ

2 Introduction and Description of Software Prototype

2.1 Introduction

My software prototype is a Text-Based Role-Playing Game (RPG) developed using the C++ Programming Language and SFML (Simple and Fast Multimedia Library) as it's graphical interface. This game immerses players in a rogue-like adventure, challenging them with a dynamic world where their character progressively by:

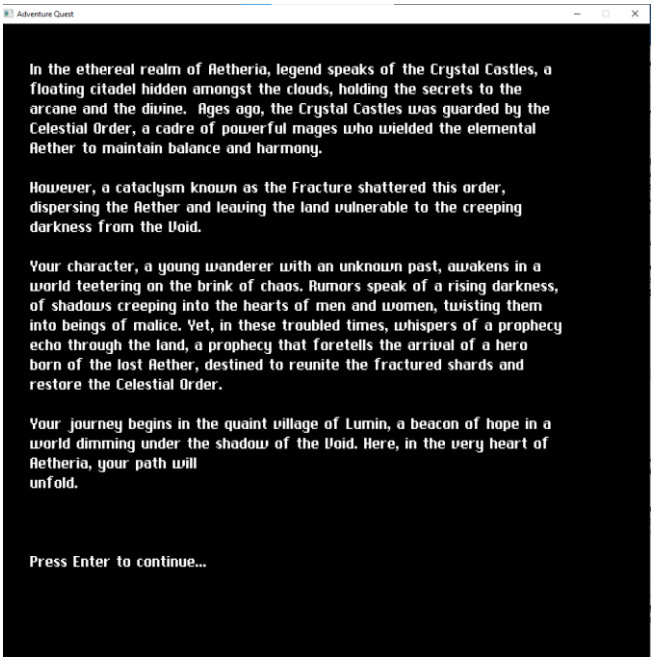
- **Gaining Experience and Leveling Up:** Players gain experience points (Exp) by "Traveling." During these travels, characters may encounter enemies or puzzles. Defeating enemies results in earning both experience and gold. On the other hand, successfully navigating puzzles rewards players with random items, which may include Weapons, Armor, or Potions of varying rarities. Each experience point earned contributes to leveling up the character, which in turn allows for the upgrading of character stats. As the character's level increases, the game's world and its enemies scale accordingly, maintaining a consistent level of challenge.

- **Acquiring Gold:** Gold is obtained primarily through defeating enemies. This gold serves as a currency in the game's shops, where players can purchase randomly generated items. These items come with stats dependent on the character's level to ensure gameplay balance. Players can buy Armor to enhance their DEFENSE, Weapons to increase DAMAGE, or Potions to either boost specific stats or restore health.

The core gameplay loop involves the player navigating through a world of increasingly difficult enemies, necessitating strategic use of level-up points for upgrading stats, acquiring better equipment, or obtaining helpful potions. The game concludes when the character's HP drops to

zero. However, players have the option to save their character progress and reload it to avoid losing their progress upon character death.

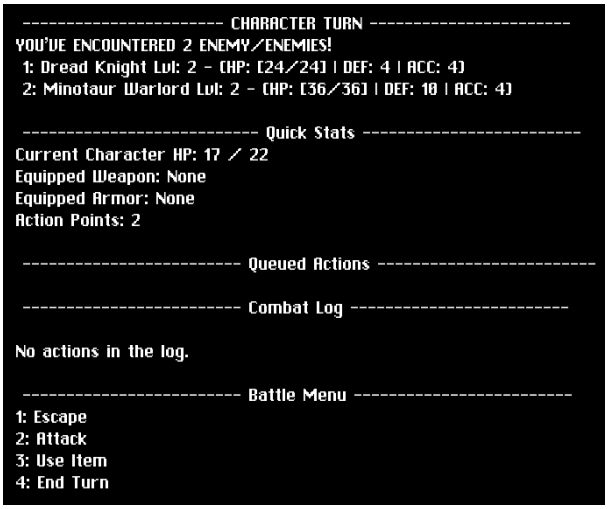
2.2 Description



After the initial screen, players are introduced to the game world, setting the stage for their immersive experience. The main hub of the game offers several functions:

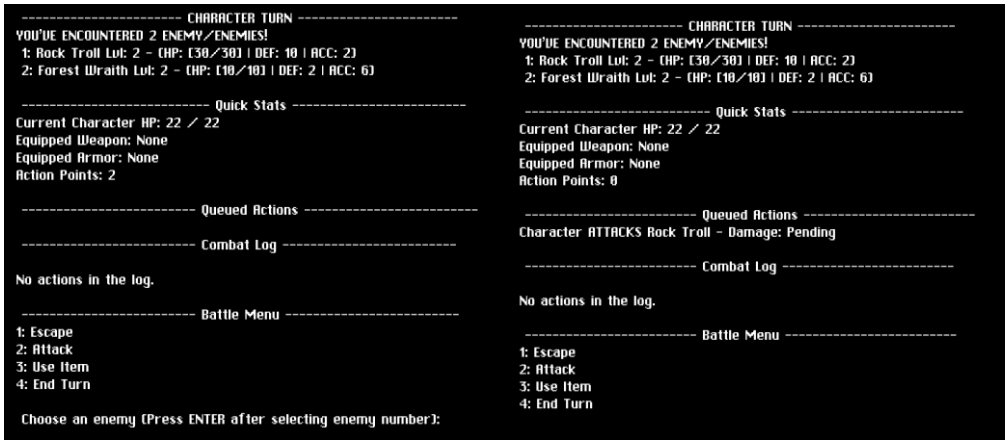
- **Travel:** Serving as the central gameplay mechanic, traveling randomly sorts players into either ENEMY encounters or PUZZLE encounters, each with distinct outcomes and challenges:

- **ENEMY encounter** – In an enemy encounter, a number of enemies will be randomly generated and displayed to the player, then the player will also be shown some basic stats like current HP, current equipped Weapon and Armor (if any), available action Points, Queued Actions and also a battle menu which gives the player the option to:



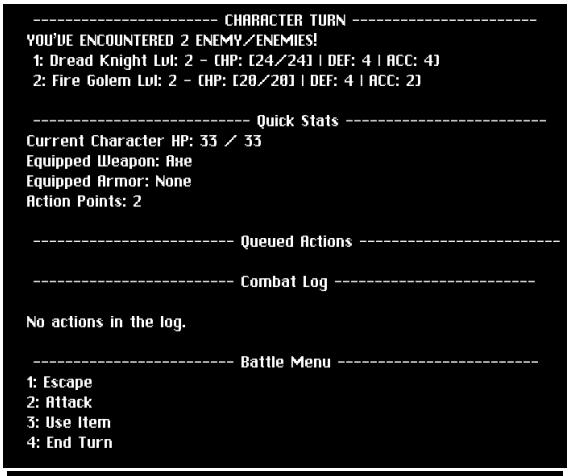
  - **Escape** – the player will go back to main menu where they can choose the other options if they so please

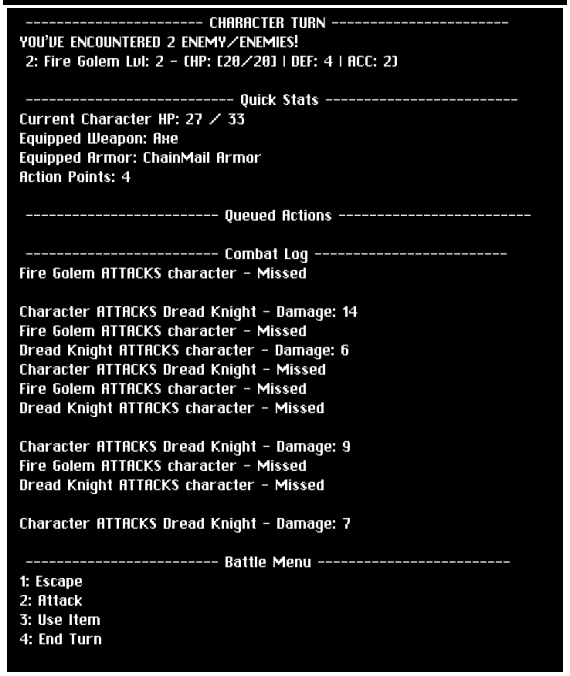
  - **Attack** – the player will be given the choice to target an enemy of their choice and this attack action will cost them action points. Once they selected the enemy, it will show up as one the Queued Actions specifically saying the name of the monster too. If there are no more action Points then the player can no longer choose to attack another monster.



  - **User Item** – the player can equip any weapon or armor or use a potion from their inventory with this choice and this will not cost any action points. Additionally, any dropped items from the enemies can also be instantly equipped by the player. Depending on which item they choose, a weapon can increase the player damage,

armor can increase the player defense and potion can increase certain stats that may give them an advantage or heal the character HP.

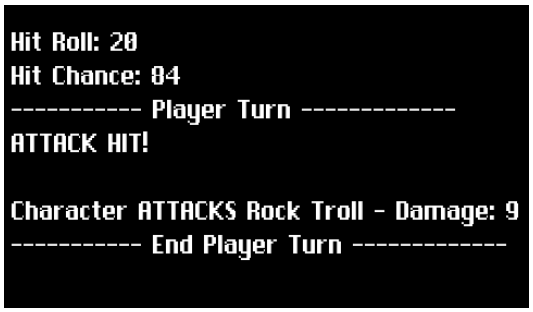




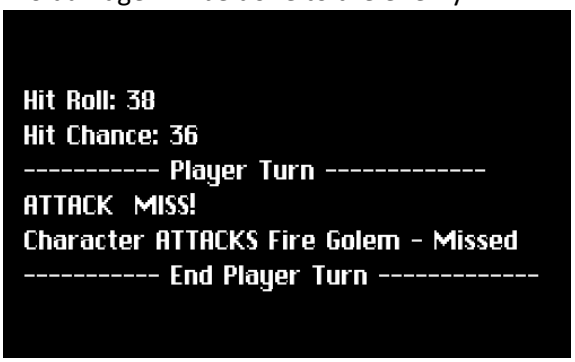
- **End Turn** – Once the player has used up their action points, they will have to choose this option to end the player turn to execute the Queued Actions. Then there will be a selectively random chance of hitting the enemy as this HIT or MISS calculation depends on the Character's intelligence against the Enemy's Defence and some random factor into it.

    - **If the Attack is a HIT** – a random amount of damage depending on a range of the characters damage will be done onto the enemy and in the next round this will be reflected in the enemy HP. If the Character kills the enemy, the enemy have a random chance of dropping a Weapon or Armor with random

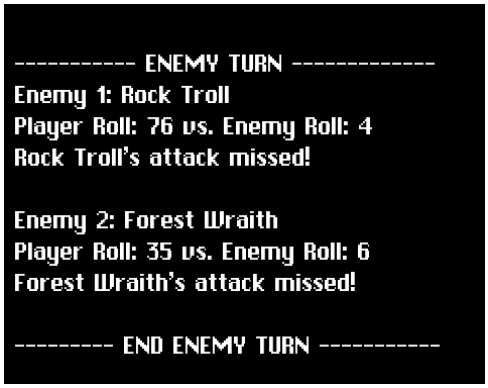
rarity. Additionally, the character will receive a certain amount of gold and exp depending on the character level.



- **If it is a MISS** – no damage will be done to the enemy.

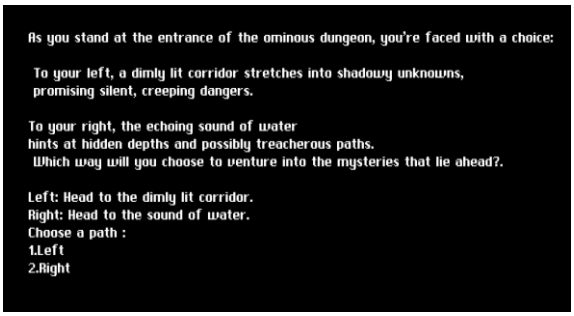


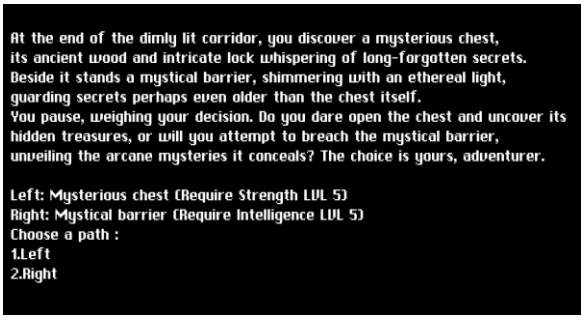
- **After Player turn** - After the player's turn, the enemies will have a chance opportunity to attack the player and this will also be either a HIT or MISS. If it is a HIT, the character will receive some damage and if the damage exceed the character's HP, the player will face the DEATH screen where they have the option to load an existing character or quit the game. If it is a miss, then the player will be unharmed.



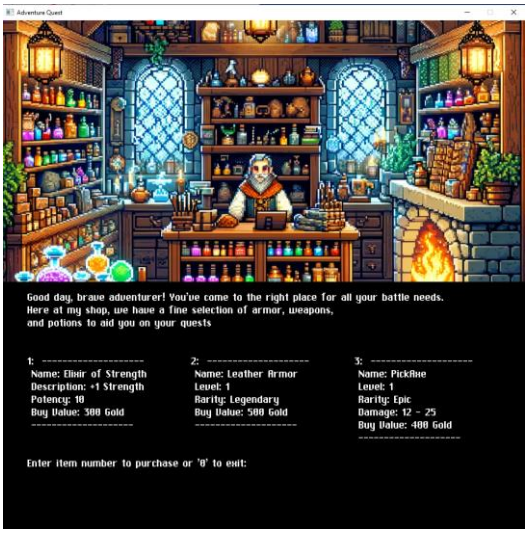
- **PUZZLE encounter** – In a puzzle encounter, the player has a chance to either enter a Dungeon Adventure Puzzle or Temple Adventure Puzzle. Both these Adventure Puzzle there will be decision points whereby the player has to choose on either one of two options. Depending on their choice, it will be event that checks for player stat and if successful they get a reward and

if unsuccessful the character will get damaged. The reward they get may involve getting a random Armor, Weapon or Potion added to their inventory, a random Character Stat Upgrade or some healing. So it is important for the player to upgrade their stats so as to pass through these skill checks.

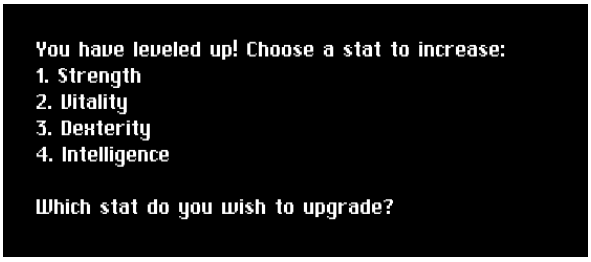




- **Shop** – When the player enters the shop, three random items will be generated and displayed to the player in which they can purchase using the gold in the inventory. These items will be one Armor, one Weapon, one random Potion each with different rarities depending on the character level. Of course this will involve gold so it is important the player travel to get gold to purchase anything.

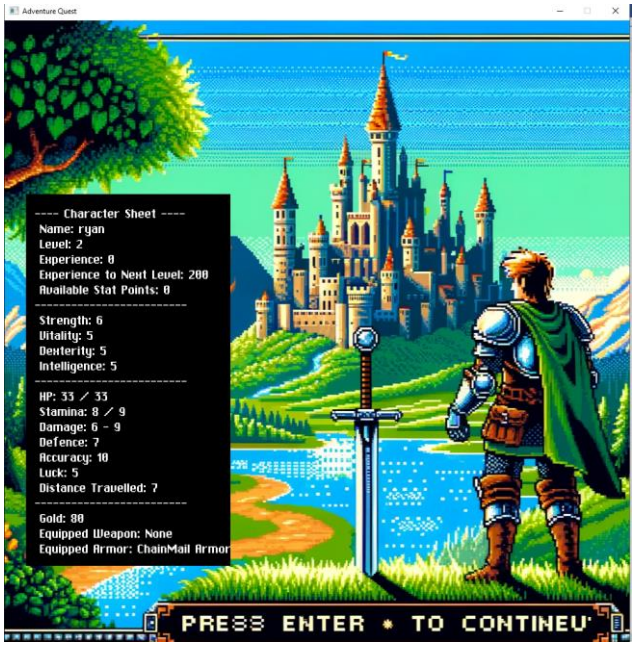


- **Level Up** – After the character returns from their travels, and they have received some exp, if the amount of exp they received is enough to reach the next level to level up, then a prompt will show up to show that the character Is ready to level up. In this window, the player will then be given the choice to upgrade one of the four important character stats (Strength, Vitality, Dexterity, Intelligence). This stat upgrade will be reflected in the show Character Info.



- **Rest** – In this choice, the character will come upon a resting healing area whereby for a price, the "Doctor" will heal the player. The price is calculated based on how much the player's HP is left.

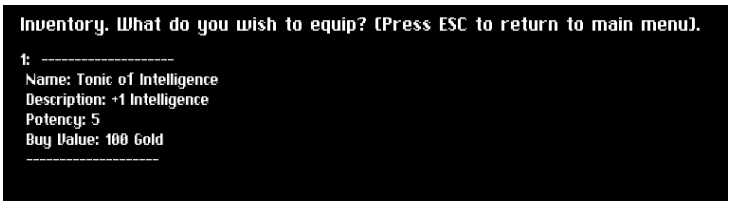
- **Show Character Info** – This choice allows the player to look at all the Character Stats which are:



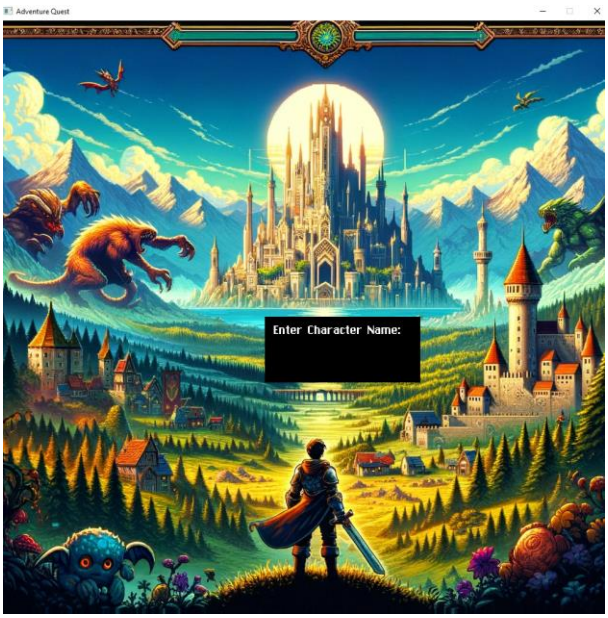
  - Name
  - Level
  - Experience (Exp)
  - Experience to Next Level – How much exp needed to level up
  - Available Stat Points – How many stat points available to be used to upgrade stats
  - Strength – Affects the character's damage range.
  - Vitality – Affects the character's max HP.

- Dexterity – Affects the characters hit or miss chance.
- Intelligence – Affects the character hit or miss chance and item drop rarity.

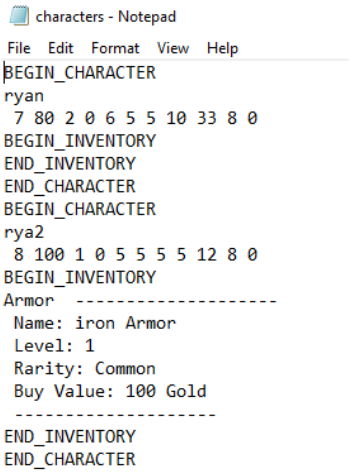
- **Show Inventory** – Shows all the Weapons, Armors or Potions available in the character's inventory and all of them are readily available to be equipped. Once equipped they will be removed from the list and the stats of the item will be the character stats. If the character already was equipped a weapon or armor, the previous armor or weapon is not destroyed but added back into the inventory.



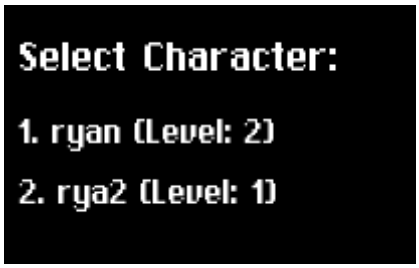
- **Create New Character** – This allows the player to create a whole new character with reset stats, level and inventory.



- **Save Character** – This allows the player to save the current character they are playing to a text file called "character.txt".



- **Select Character** – This will read from “character.txt” and obtain the values from the text file to be put into the character variables. This will then be reflected in the Show Character Info with all the saved character stats including their inventory.



- **Quit** – Quits the program.

Class Diagram of Program

5 Concepts

5.1 Inheritance and Derived Classes

5.1.1 Application

5.1.1.1 What area will you apply this concept to?

Inheritance and derived classes are applied in the area of item creation and management in the game. The Item class act as the parent class to the Weapon, Armor and Potion child classes. This includes different types of items like Weapons, Armor, and Potions.

5.1.1.2 What is the desired operation? Describe in detail.

The desired operation is to create a flexible and scalable item system where different types of items (Weapons, Armor, Potions) share common attributes and functionalities but also have their unique properties and behaviors. This system should allow for easy addition of new item types in the future.

5.1.2 Concept

5.1.2.1 How does it match this application's desired operation?

Inheritance and derived classes are perfectly suited for this application for several reasons:

- **Common Parent Class (Item):** A Parent class named Item is defined. It provides common properties and methods that are shared across all item types, such as name, level, buyValue, sellValue, rarity, and functions like clone() and toString().

- **Derived/Child Classes (Weapon, Armor, Potion):** Specific item types which are Weapon Armor, and Potion are derived from the Item class. They inherit the common properties and override or extend functionalities. For example, the Weapon class introduces specific attributes like damageMin and damageMax and overrides the clone() method to return a copy of the weapon object.

- **Factory Classes for Item Creation:** Factory classes like WeaponFactory, ArmorFactory, and PotionFactory are created, each deriving from a base ItemFactory class. They override the createItem method to instantiate their respective item types. This design pattern makes the item creation process more modular and extensible.

  In summary, the use of inheritance and derived classes in your game's item system allows for a structured, maintainable, and scalable approach to handling various types of items, each with their own set of characteristics and behaviors. This design pattern is critical in developing a complex game where multiple item types play a significant role. In this game, Items are randomly generated whenever, the puzzle encounter is successful, enemy encounter is successful, item shops are purchased so it is much easier to have one Base class that the derived classes inherit properties from rather than creating separate classes that may have similar properties which this then increases complexity of the game.

5.1.2.2 Are there any other alternative Structures can be used here? Why are they less preferred?

While inheritance and derived classes are a common approach in object-oriented programming for scenarios such as this, there are alternative structures that could be used. However, each comes with its own set of trade-offs. These are:

- **Composition over Inheritance**:
  Description: Instead of using inheritance the item system can be designed using composition, where an Item class contains a variety of components (like damage, defense, effects) that are combined differently for each item type.

  Disadvantages: While composition offers greater flexibility and can reduce complexity in some cases, it might overcomplicate scenarios where a clear hierarchical relationship exists like between Item and Weapon for example. This is where Inheritance provides a more straightforward and intuitive structure for such scenarios.

- **Interface Implementation:**
  Description: Using interfaces or abstract classes with only pure virtual functions in C++, where Weapon, Armor, and Potion classes implement an Item Interface.

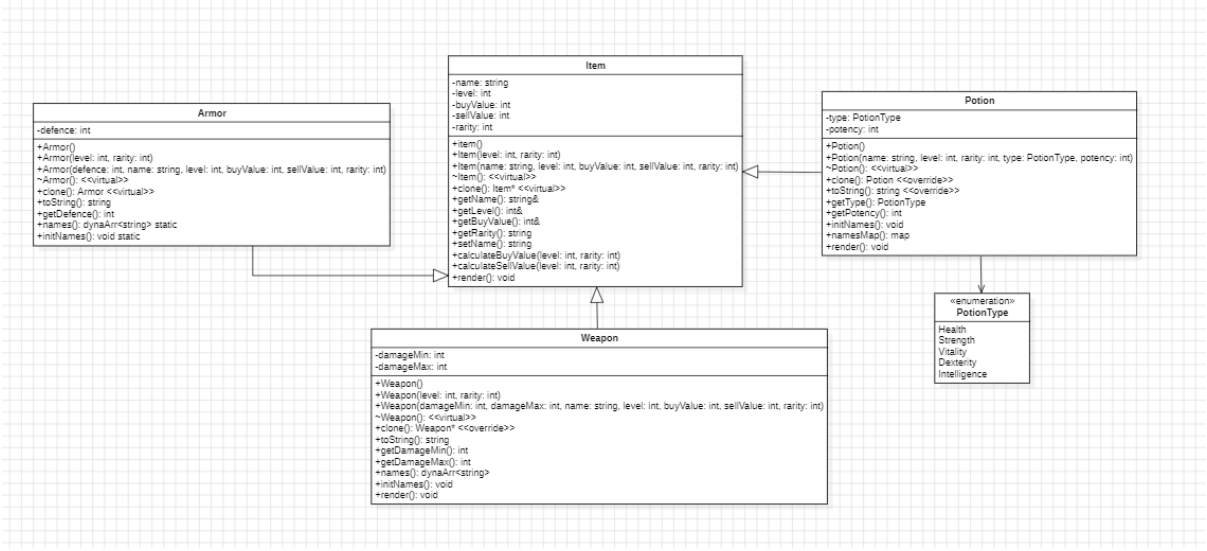
  Disadvantages: Interfaces ensure that all item types implement certain functions, but they don't provide shared implementation. This could lead to redundant code across different item types, which inheritance can avoid by allowing shared functionalities in the base class.

- **Prototype Pattern:**
  Description: Instead of creating subclasses for each item type, = a prototype pattern can be used where each item type is an instance of an Item class that is cloned to create new instances.

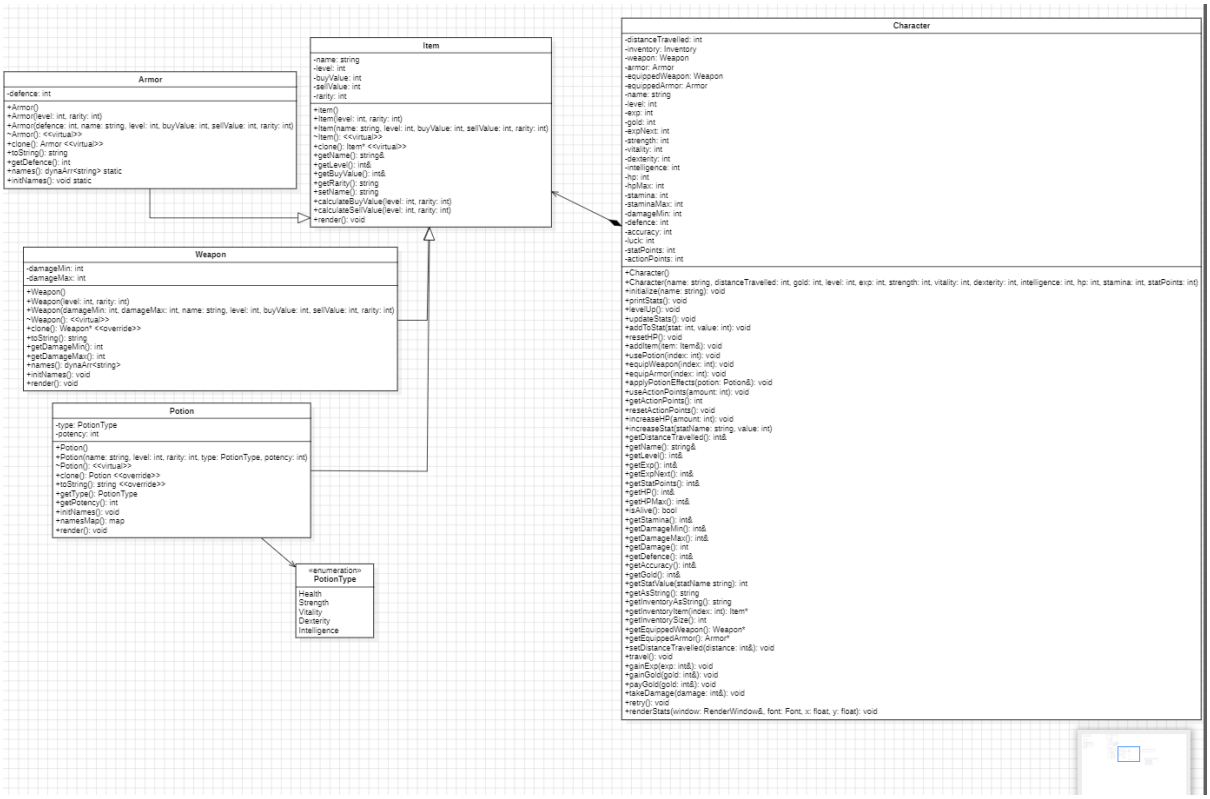
  Disadvantages: This pattern is useful when instantiation from a standard type is more efficient than subclassing, but it can be less clear and harder to manage when items have distinct behaviors or properties especially in the case between Weapon, Armor and Potion items which all three have very distinct properties when used by the character.

  In this game's context, inheritance and derived classes offer a straightforward and intuitive approach for representing different item types, ensuring code reusability and maintainability. The alternatives, while viable, might introduce unnecessary complexity or reduce the clarity of the relationships between different item types. Therefore, they are less preferred in this specific scenario.

5.1.2.3 Draw a diagram to represent how this Structure/Concept is used in this part of the program.



Explanation: Armor, Weapon and Potion classes are Inherited from Item Class.



Explanation: The Character class is associated with the Item Class. This means that the character object has access or “owns” the Item Objects. This is a “has-a” relationship, which means that Items are a part of the Character’s state but are separate, distinct objects. The Character Class owns these Items through its Inventory which is a collection of Items that is able to be accessed by the Character class to equip them such as when equipping armor, weapons or using potions and these items significantly impact the Character’s attributes and abilities within the game.

5.1.3 Implementation and Output

```cpp
class Item
{
public:
    Item();
    Item(int level, int rarity);
    Item(string name,
        int level,
        int buyValue,
        int sellValue,
        int rarity);

    virtual ~Item();

    inline string debugPrint() const { return this->name; }
    virtual Item* clone() const = 0;
    virtual string toString() const = 0;

    //Accessors
    inline const string& getName() const { return this->name; }
    inline const int& getLevel() const { return this->level; }
    inline const int& getBuyValue() const { return this->buyValue; }
    inline const int& getSellValue() const { return this->sellValue; }

    //Rarity of Items converted to string
    inline string getRarity() const {
        switch (this->rarity) {
        case COMMON: return "Common";
        case UNCOMMON: return "Uncommon";
        case RARE: return "Rare";
        case EPIC: return "Epic";
        case LEGENDARY: return "Legendary";
        default: return "Unknown";
        }
    }

    //Modifiers
    inline void setName(string name) { this->name = name; }

    //Static
    inline static int calculateBuyValue(int level, int rarity){ return 100 * level * (rarity + 1); }
    inline static int calculateSellValue(int level, int rarity) { return calculateBuyValue(level, rarity) / 2; };

    void render(sf::RenderWindow& window, sf::Font& font, float x, float y) const;
private:
    string name;
    int level;
    int buyValue;
    int sellValue;
    int rarity;
};
```

```cpp
#pragma once

#include "STLInclude.h"
#include "Item.h"


class Weapon:public Item
{
public:
    Weapon();
    Weapon(int level, int rarity);
    Weapon(int damageMin,
        int damageMax,
        string name,
        int level,
        int buyValue,
        int sellValue,
        int rarity);

    virtual ~Weapon();

    //Pure virtual
    virtual Weapon* clone() const override;


    //Functions
    string toString() const;

    //Accessors
    inline int getDamageMin() const { return this->damageMin; }
    inline int getDamageMax() const { return this->damageMax; }

    //Modifiers
    static dynaArr<string> names;
    static void initNames();

    void render(sf::RenderWindow& window, sf::Font& font, float x, float y) const;

private:
    int damageMin;
    int damageMax;
};

```

TextBasedGame (Global Scope)

```h
#pragma once
#include "STLInclude.h"
#include "Item.h"

class Armor : public Item
{
public:
    Armor();
    Armor(int level, int rarity);
    Armor(int defence,
        string name,
        int level,
        int buyValue,
        int sellValue,
        int rarity);

    virtual ~Armor();

    // Pure virtual
    virtual Armor* clone() const override;

    // Functions
    string toString() const;

    // Accessors
    inline int getDefence() const { return this->defence; }

    // Modifiers
    static dynaArr<string> names; //using Dynamic Array to store Armor names
    static void initNames(); //Initialise unique Armor names



private:
    int defence;
};
```

```h
#pragma once
#include "Item.h"

class Potion : public Item
{
public:
    enum class PotionType
    {
        Health = 0,
        Strength,
        Vitality,
        Dexterity,
        Intelligence
    };

    Potion();
    Potion(string name, int level, int rarity, PotionType type, int potency);

    virtual ~Potion();

    virtual Potion* clone() const override;
    virtual string toString() const override;

    //Accessors
    PotionType getType() const;
    int getPotency() const;

    static void initNames();

    static map<PotionType, string> namesMap;

    void render(sf::RenderWindow& window, sf::Font& font, float x, float y) const;
private:
    PotionType type;
    int potency;
};

```

Here, as an example of the Inheritance of the Item Class to its Derived Classes of Potion, Iron and Weapon, it can be seen as these Items are generated each of them have their own distinct properties but all them share the common properties of Item such as Name, Level, Buy Value. Not all of the common properties of Items are displayed here for easier user readibility.

5.1.4 Troubleshooting Summary

5.1.4.1 Did you run into any issues while implementing this concept?

During the implementation of inheritance, particularly in the development of the Item class and its subclasses (Weapon, Armor, Potion), a few key issues were encountered:

Ambiguity in Overridden Methods: Certain methods overridden in the subclasses were not behaving as expected, causing incorrect item behavior in the game.

Difficulty in Managing Constructors: Properly initializing subclass objects with unique attributes while still leveraging the base class constructor proved challenging.

5.1.4.2 Did you refer to any resources (tutorials/guides) to find the solution?

Online C++ Documentation and Tutorials: Helped me understand the syntax of a proper inheritance structure.

Stack Overflow Forums: Helped troubleshoot specific problems related to inheritance and object-oriented design as well as proper inheritance structure.

5.1.4.3 How did you solve the problem?

Method Overriding: Enhanced understanding of virtual functions and method overriding helped fix ambiguity in subclass methods. Specific attention was given to ensuring that virtual methods in the base class were correctly overridden in subclasses.

Constructor Initialization: Utilized member initializer lists in subclass constructors to properly initialize base class attributes. This approach ensured that subclass objects were initialized correctly, preserving inherited properties.

5.1.4.4 Cite your resources/references

GeeksforGeeks 2022, Inheritance in C++, viewed 12 November 2023, <https://www.geeksforgeeks.org/inheritance-in-c>.

TutorialsPoint 2022, C++ Inheritance, viewed 11 November 2023, <https://www.tutorialspoint.com/cplusplus/cpp_inheritance.htm>.

5.2 Polymorphism

5.2.1 Application

5.2.1.1 What area will you apply this concept to?

Polymorphism is applied in the decision-making process of the game specifically during the Puzzle Encounter. This is evident in the DecisionEffect class and its subclasses FindItemEffect, DamageHealthEffect, IncreaseStatEffect. These classes are used to apply different effects onto the active character based on the player's decisions during the puzzle encounter.

5.2.1.2 What is the desired operation? Describe in detail.

The Puzzle Encounter features different scenarios where players make decisions that lead to various outcomes. Each decision can have different effects, such as finding an item, taking damage, or increasing a character stat. The DecisionEffect base class provides a virtual function applyEffect which is overridden in each subclass to implement the specific effect of a decision. When a player makes a decision, the game uses these classes to apply the corresponding effect to the character. For example, FindItemEffect adds an item to the player's inventory, DamageHealthEffect reduces the player's health, and IncreaseStatEffect boosts a specific stat of the player. The use of polymorphism here allows the game to handle various decision effects in a flexible and scalable manner. Different types of effects can be added or modified easily by creating new subclasses of DecisionEffect, which makes the game's design more robust and maintainable.

5.2.2 Concept

5.2.2.1 How does it match this application's desired operation?

Polymorphism in the context of the DecisionEffect class and its subclasses perfectly matches the desired operation of handling various decision outcomes in the puzzle encounter because of its:

5.2.1.1 **Unified Interface with Diverse Implementations** – The DecisionEffect base class provides a common interface (applyEffect method) for all decision effects. Each subclass (FindItemEffect, DamageHealthEffect, IncreaseStatEffect) overrides this method to execute its specific effect. This design allows the game to treat all decision effects uniformly while enabling them to have distinct behaviors, aligning with the diverse nature of decision outcomes in the puzzle encounter.

5.2.1.2 **Flexibility and Scalability** - Using polymorphism makes the game's design more flexible and scalable. New types of decision effects can be added simply by creating additional subclasses of DecisionEffect without altering existing code.

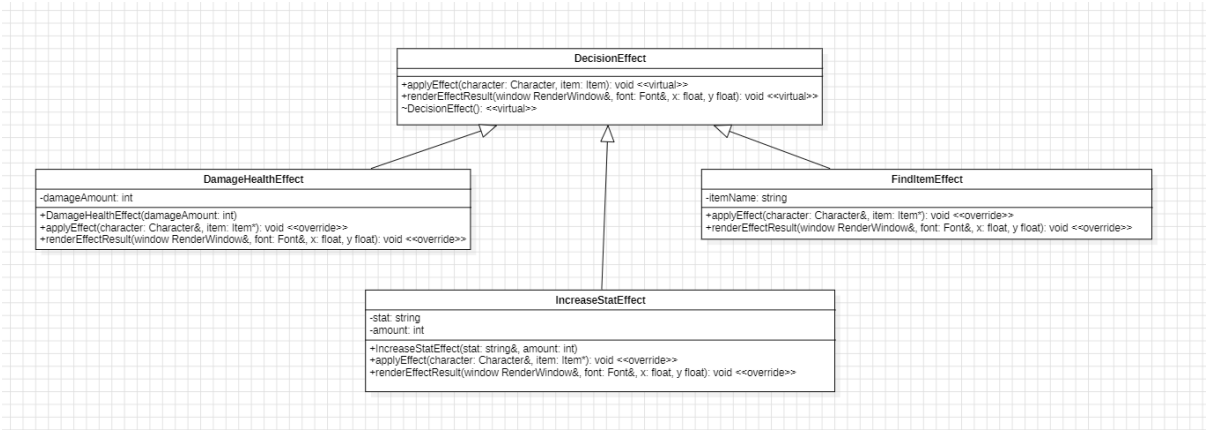
5.2.1.3 **Simplification of Game Logic** - Polymorphism simplifies the game logic related to applying decision effects. The game can handle any DecisionEffect object generically, delegating the responsibility of specific actions to the relevant subclasses.

5.2.2.2 Are there any other alternative Structures can be used here? Why are they less preferred?

- **Conditional Logic (If-Else)**
  Description: An alternative is to use a series of conditional statements for example if-else blocks to determine the type of effect and then apply it.

  Disadvantages: This approach can lead to a rigid and less maintainable code structure, especially as the number of decision effects grows.

5.2.2.3 Diagram



Explanation: FindItemEffect, DamageHealthEffect, and IncreaseStatEffect are subclasses of DecisionEffect. These subclasses inherit the structure and behavior (methods) from DecisionEffect.

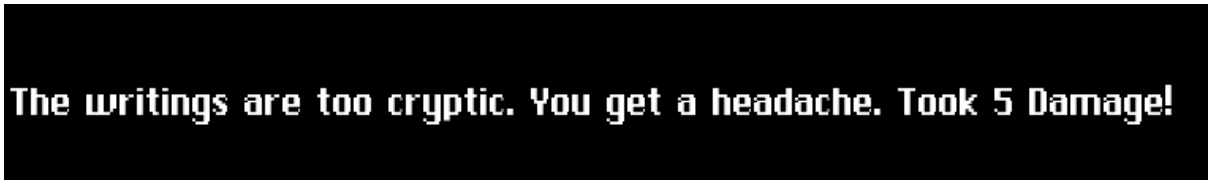
5.2.3 Implementation and Output

- Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

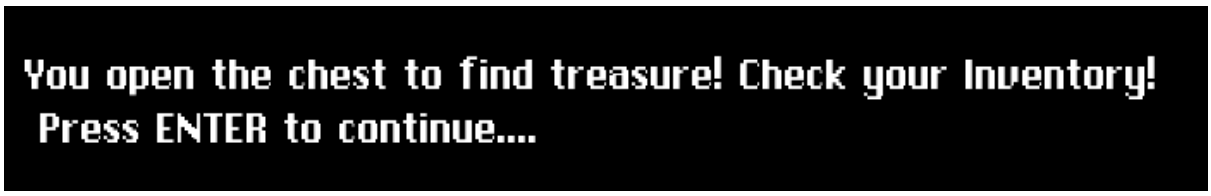
```cpp
class DecisionEffect {
public:
    virtual void applyEffect(Character& character, Item* item) = 0;
    virtual void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x, float y) = 0;
    virtual ~DecisionEffect() {}
};
```

```cpp
class FindItemEffect : public DecisionEffect {
    std::string itemName;

public:
    FindItemEffect() {}

    void applyEffect(Character& character, Item* item) override {
        if (item) {
            character.addItem(*item);
            itemName = item->getName();
            std::cout << "You found an item: " << itemName << "!" << std::endl;
        }
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x, float y) override {
        if (!itemName.empty()) {
            sf::Text text;
            text.setFont(font);
            text.setString("You found an item: " + itemName + "!");
            text.setCharacterSize(24);
            text.setFillColor(sf::Color::White);
            text.setPosition(x, y);
            window.draw(text);
        }
    }
};
```

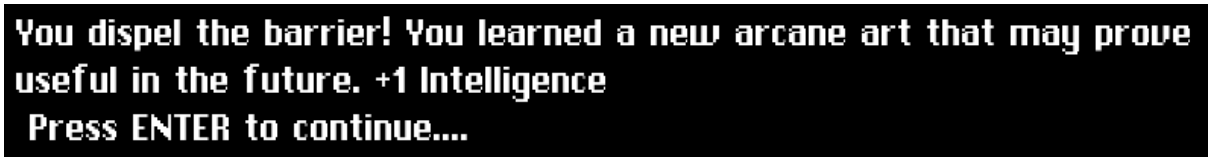
```cpp
class DamageHealthEffect : public DecisionEffect {
    int damageAmount;

public:
    DamageHealthEffect(int damageAmount) : damageAmount(damageAmount) {}

    void applyEffect(Character& character, Item* item) override {
        character.takeDamage(damageAmount);
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x, float y) override {
        sf::Text text;
        text.setFont(font);
        text.setString("You take " + std::to_string(damageAmount) + " damage!");
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setPosition(x, y);
        window.draw(text);
    }
};

class IncreaseStatEffect : public DecisionEffect {
    std::string stat;
    int amount;

public:
    IncreaseStatEffect(const std::string& stat, int amount) : stat(stat), amount(amount) {}

    void applyEffect(Character& character, Item* item) override {
        character.increaseStat(stat, amount);
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x, float y) override {
        // Render the result of the stat increase effect.
        sf::Text text;
        text.setFont(font);
        text.setString("Your " + stat + " has increased by " + std::to_string(amount) + "!");
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setPosition(x, y);
        window.draw(text);
    }
};
```



DamageHealthEffect



FindItemEffect



IncreaseStatEffect

5.2.4 Troubleshooting Summary

5.2.4.1 Did you run into any issues while implementing this concept?

Virtual Function Overriding: Challenges in ensuring that overridden methods in subclasses were functioning as intended. Some subclass methods were not executing due to incorrect override signatures.

5.2.4.2 Did you refer to any resources (tutorials/guides) to find the solution?

Stack Overflow Discussions: for troubleshooting specifically how to use virtual correctly.

5.2.4.3 How did you solve the problem?

Refining Virtual Functions: Reviewed and corrected the virtual function signatures in subclasses to align with the base class method signature. This ensured that the overridden methods were called as expected.

5.3 Dynamic Array

5.3.1 Application

5.3.1.1 What area will you apply this concept to?

5.3.1.1 **Game Class (Character Storage):** The Game class uses dynamic arrays to manage a collection of characters. This is essential for handling multiple character instances, each with unique attributes and states. It allows the game to dynamically add or remove characters based on player actions or game events.

5.3.1.2 **Weapon Class (Weapon Names Storage):** In the Weapon class, dynamic arrays store a variety of weapon names. This enables the game to offer a diverse range of weapon name which are not randomly generated so as not to suddenly generate random names.

5.3.1.3 Armor Class (Armor Names Storage): Similar to weapons, the Armor class employs dynamic arrays to store names of different armor types. This storage method allows for easy expansion and modification of armor options available to players, contributing to a more dynamic and engaging equipment system and most importantly consistency in naming structure.

5.3.1.2 What is the desired operation? Describe in detail.

- **Character Management**: Efficient handling of multiple character instances, including adding new characters, removing existing ones, or accessing specific characters for gameplay mechanics like combat or character stat upgrades.
- **Weapon Variety Management:** Allows the game to dynamically manage a variety of weapon names and types, which can be assigned different stats. This system supports scalability in terms of weapon diversity. This allows for easy addition of new Weapon Names and the possibility of implementing specific weapon attributes or bonuses in the future.
- **Armor Variety Management:** Similar to weapons, the dynamic array in the Armor class enables the game to handle various armor types, each with unique names. This enhances the player's experience by offering a range of equipment choices and allows for future addition and new implementation of specific armor attributes or bonuses.

5.3.2 Concept

5.3.2.1 How does it match this application's desired operation?

- **Game Class:** The dynamic array characters efficiently manages multiple character instances. This is crucial for operations like creating new characters, managing character levels, saving and loading character states, and selecting characters for gameplay. The dynamic array's ability to dynamically resize based on the game's needs makes it ideal for these operations.
- **Weapon Class**: The names dynamic array in the Weapon class stores a variety of weapon names. This supports the operation of managing weapon variety in the game. The array allows for easy addition and retrieval of weapon names, facilitating the introduction of new weapons and maintaining a consistent naming structure.
- **Armor Class:** Similar to the Weapon class, the Armor class uses a dynamic array for storing armor names. This aids in managing a diverse range of armor types, enhancing the player's experience by offering various equipment choices. The dynamic array's flexibility is key to adding new armor types and updating existing ones efficiently.

5.3.2.2 Are there any other alternative Structures can be used here? Why are they less preferred?

Several alternative data structures could be used, but they might have limitations compared to dynamic arrays:

- **Static Arrays:** While they offer simplicity, static arrays have a fixed size, which limits the scalability of the application. This is a significant drawback for a game that requires dynamic addition and removal of characters, weapons, and armors.
- **Linked Lists:** They offer dynamic size and ease in insertion/deletion operations. However, they may incur higher memory usage and slower access times compared to dynamic arrays, especially when accessing elements at random positions.

- **Hash Tables:** Useful for quick lookups, insertions, and deletions, but they are more complex to implement and manage. For tasks requiring ordered data (like your game might), hash tables are less efficient than dynamic arrays.
- **Trees (like Binary Search Trees):** They offer ordered storage and efficient searches, but they are more complex and might not offer the straightforward, contiguous memory access of dynamic arrays.

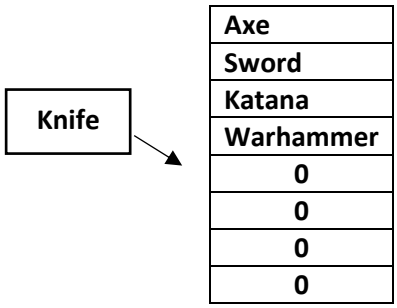
Dynamic arrays are preferred in for the application due to their balance of flexibility (dynamic resizing), efficient access time (direct indexing), and relative simplicity in implementation. These features make dynamic arrays particularly well-suited for game development scenarios where the number of elements (characters, weapons, armors) can change frequently, and fast access is crucial for a smooth gameplay experience.

5.3.2.3 Diagram

- **Initialization**: The dynamic array starts with a certain initial capacity. For example here a dynamic array of weapon names.

Axe
Sword
Katana
Warhammer

- **Adding Elements:** When a new element (like a character, weapon name, or armor name) is added, the dynamic array checks if there is enough space. If not, it increases its size (typically doubles) and allocates new memory to accommodate more elements.



5.3.3 Implementation and Output

5.3.3.1 Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
#include "Weapon.h"

dynaArr<string> Weapon::names;

void Weapon::initNames()
{
    Weapon::names.push("Combat Knife");
    Weapon::names.push("Great Sword");
    Weapon::names.push("Axe");
    Weapon::names.push("PickAxe");
    Weapon::names.push("WarHammer");
    Weapon::names.push("Katana");
}
```

Inside Weapon.cpp

```
dynaArr<string> Armor::names;

//Use Dynamic Array to store Armor names
void Armor::initNames()
{
    Armor::names.push("Leather Armor");
    Armor::names.push("iron Armor");
    Armor::names.push("Gold Armor");
    Armor::names.push("ChainMail Armor");
    Armor::names.push("Diamond Armor");
    Armor::names.push("Nether Armor");
}
```

Inside Armor.cpp

```
private:
    int choice;
    bool playing;
    int activeCharacter;
    dynaArr<Character*> characters;
    std::string fileName;
    SinglyLinkedNode<Enemy>* enemies;
```

Inside Game.h

5.3.4 Troubleshooting Summary

5.3.4.1 Did you run into any issues while implementing this concept?

- While implementing dynamic arrays in the Game, Weapon, and Armor classes, I encountered a significant challenge in memory management. The primary issue was managing the resizing of dynamic arrays effectively, ensuring no memory leaks and avoiding excessive reallocation that could impact the game's performance.

5.3.4.2 Did you refer to any resources (tutorials/guides) to find the solution?

- **GeeksforGeeks - How do Dynamic Arrays Work?** :This article explains the working of dynamic arrays, which automatically grow when an insertion is made, and there is no more space for the new item. It specifically mentions examples like vector in C++ and ArrayList in Java, which are commonly used implementations of dynamic arrays

5.3.4.3 How did you solve the problem?

- The key to solving this issue was the implementation of copy constructors and assignment operators to manage deep copies of the dynamic arrays. I made sure that proper memory allocation and deallocation techniques were employed every time a dynamic array was resized. This approach significantly enhanced the performance and reliability of the dynamic array usage in the game.

5.3.4.4 Cite your resources/references

- GeeksforGeeks 2023, How do Dynamic arrays work?, viewed 24 November 2023, <https://www.geeksforgeeks.org/how-do-dynamic-arrays-work>

5.4 Singly Linked-List

5.4.1 Application

5.4.1.1 What area will you apply this concept to?

- The Shop class uses a singly linked list to store the items available for sale. Each node in the list contains an item, and the list structure allows for dynamic management of the shop's inventory.

5.4.1.2 What is the desired operation? Describe in detail.

- The primary operations include adding new items to the inventory, displaying items for sale, and handling purchases. The singly linked list facilitates these operations by providing efficient item insertion and traversal capabilities.

5.4.2 Concept

5.4.2.1 How does it match this application's desired operation?

- **Efficient Item Insertion:** Adding new items to the shop's inventory is straightforward with a singly linked list. Since items can be added at the beginning, middle, or end, the list provides flexibility and speed, essential for a game where inventory can change frequently.

- **Traversal for Display:** The singly linked list enables efficient traversal when displaying items for sale. This linear structure makes it simple to iterate through the list and display each item in order.

- **Handling Purchases and Removals:** When players purchase or remove items from the shop, the singly linked list can efficiently handle these deletions. Unlike arrays, where deletion might involve shifting elements, the singly linked list can remove items with minimal overhead.

5.4.2.2 Are there any other alternative Structures can be used here? Why are they less preferred?

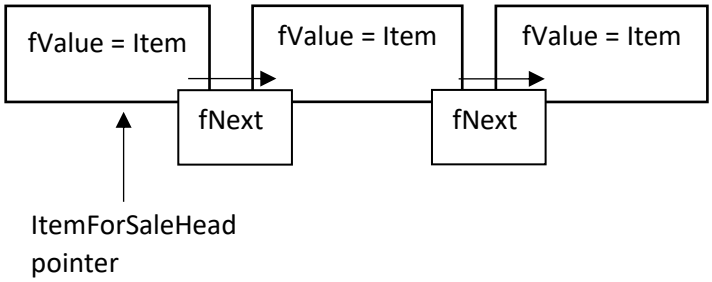
- **Dynamic Arrays:** While dynamic arrays offer the benefit of direct access to elements, they are not as efficient as singly linked lists for operations involving frequent insertions and deletions. Each time the array resizes, it requires reallocation and copying of elements, which can be costly in terms of performance, especially when the array size is large.

- **Doubly Linked Lists:** These provide bidirectional traversal, which is more than what is needed for the Shop class. The additional memory required for storing two pointers (next and previous) for each item is unnecessary overhead when only forward traversal is required.

- **Stacks and Queues:** These data structures could be used for specific types of inventory management but are limited by their LIFO (Last In, First Out) and FIFO (First In, First Out) nature, respectively. They don't provide the flexibility needed for the various types of operations (like specific item access and removal) required in the Shop class.

In conclusion, the singly linked list is the most suitable data structure for the Shop class due to its efficiency in handling dynamic inventory changes and the nature of operations required in a game environment. Other structures, while viable, offer either excessive functionality with additional overhead or lack the necessary flexibility.

Diagram



- **Nodes:** Each rectangle or block represents a node in the linked list, symbolizing an item for sale in the shop.
- **Pointers**: Arrows between the nodes show the 'next' pointers. Each arrow points from one node to the following node in the list, indicating the sequence of items.
- **Head Pointer:** At the beginning of the list is the 'head' called ItemForSaleHead pointer, pointing to the first item in the shop's inventory.

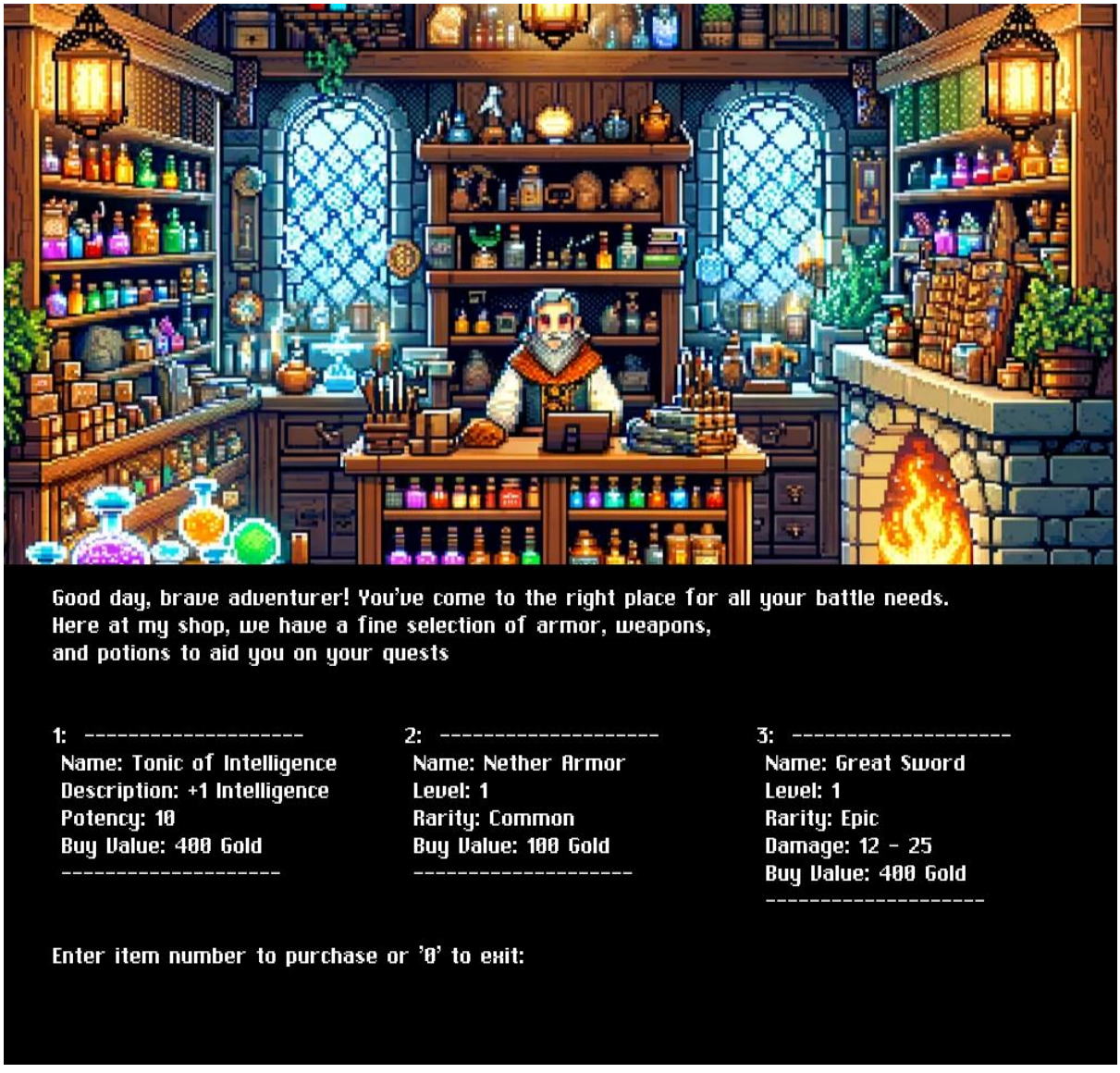
5.4.3 Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
template<class DataType>  <T> Provide sample template arguments for IntelliSense
class SinglyLinkedNode {
public:
    typedef SinglyLinkedNode<DataType> Node;

private:
    DataType fValue;
    Node* fNext;

public:
    // Corrected constructor
    SinglyLinkedNode(const DataType& aValue, SinglyLinkedNode* aNext)
        : fValue(aValue), fNext(aNext) {}

    // Methods to manipulate the list
    void append(Node* aNode);
    void remove();
    int size() const;
    void setNext(Node* nextNode);
    DataType* get(int index);

    // Accessors
    DataType& getValue() { return fValue; }
    Node* getNext() const { return fNext; }

    // Check if the node is the end of the list
    bool isEnd() const { return fNext == nullptr; }
};
```

```
class Shop {
public:
    Shop(int playerLevel);
    ~Shop();

    void openShop(Character& character, sf::RenderWindow& window, sf::Font& font);
    void displayItemsForSale(sf::RenderWindow& window, sf::Font& font) const;
    void purchaseItem(Character& character, int index, sf::RenderWindow& window, sf::Font& font);
    int getItemsCount() const;

private:
    SinglyLinkedNode<Item*>* itemsForSaleHead;
    int playerLevel;

    void generateItemsForSale();
    void clearShop();
    int calculateItemPrice(const Item* item) const;

    sf::Texture ShopTexture;
    sf::Sprite ShopSprite;
};
```

```cpp
void Shop::displayItemsForSale(sf::RenderWindow& window, sf::Font& font) const {
    int index = 1;
    float xPos = 50.0f; // Starting y-position for displaying items

    for (SinglyLinkedNode<Item*>* current = itemsForSaleHead; current != nullptr; current = current->getNext()) {
        int price = calculateItemPrice(current->getValue());
        sf::Text itemText;
        itemText.setFont(font);
        itemText.setString(std::to_string(index) + ": " + current->getValue()->toString());
        itemText.setCharacterSize(24);
        itemText.setFillColor(sf::Color::White);
        itemText.setPosition(xPos, 650.f); // Adjusted position for better visibility

        window.draw(itemText);
        xPos += 320.0f; // Increased space between lines for clarity
        index++;
    }
}
```



5.4.4 Troubleshooting Summary

5.4.4.1 Did you run into any issues while implementing this concept?

- During the implementation of the singly linked list in the Shop class, I faced challenges with memory management, particularly in ensuring that there were no memory leaks during the deletion of nodes (items).

5.4.4.2 Did you refer to any resources (tutorials/guides) to find the solution?

- **Simplilearn** - **How to Implement A Singly Linked List in Data Structures.** This resource provides a comprehensive guide on creating nodes in singly linked lists using classes or structures and linking them using the next pointer. It also explains the operations that can be performed on a singly linked list, such as insertion and deletion at various positions.

5.4.4.3 How did you solve the problem?

- The solution involved implementing a robust destructor for the singly linked list that carefully deallocates memory for each node when the list is destroyed or when an item is removed from the list.

5.4.4.4 Cite your resources/references

- "Simplilearn 2023, How to Implement A Singly Linked List in Data Structures, viewed 24 November 2023, <https://www.simplilearn.com/tutorials/data-structure-tutorial/singly-linked-list.>

5.5 Doubly Linked-List

5.5.1 Application

5.5.1.1 What area will you apply this concept to?

The doubly linked list concept is applied to the Inventory class in the programming project. This class manages a collection of items like weapons, armor and potions in the character's inventory system.

5.5.1.2 What is the desired operation? Describe in detail.

- **Dynamic Inventory Management:** The doubly linked list allows for efficient and flexible management of the inventory, accommodating a varying number of items.
- **Efficient Item Insertion and Removal:** Items can be added or removed from both the beginning and end of the inventory, or any specific position, thanks to the list's bidirectional nature.
- **Easy Navigation:** The inventory can be navigated both forwards and backwards, which is useful for displaying items in the inventory screen or for sorting and rearranging items.
- **Memory Management:** Effective handling of dynamic memory allocation and deallocation for inventory items, ensuring efficient use of resources and preventing memory leaks.

5.5.2 Concept

5.5.2.1 How does it match this application's desired operation?

The doubly linked list in the Inventory class matches the desired operation in several ways:

- **Bidirectional Traversal**: Allows for easy navigation through the inventory, crucial for operations like displaying items or sorting.

- **Efficient Insertions and Deletions**: Items can be added or removed from anywhere in the list without needing to shift other elements, unlike in an array.

Are there any other alternative Structures can be used here? Why are they less preferred?
Yes, but they are less preferred because of the following reasons:

- **Arrays:** They are less flexible for operations that involve frequent insertions and deletions, especially in the middle, as these operations can be time-consuming due to the need to shift elements.
- **Singly Linked Lists:** While they can be used for inventory management, they lack the ease of backward traversal offered by doubly linked lists, making some operations less efficient.

5.5.3 Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
template<class DataType>
class DoublyLinkedNode {
public:
    typedef DoublyLinkedNode<DataType> Node;

    private:
    DataType fValue;
    Node* fNext;
    Node* fPrevious;

public:
    // Constructor for node with a value
    DoublyLinkedNode(const DataType& aValue) : fValue(aValue), fNext(nullptr), fPrevious(nullptr) {}

    // Methods to manipulate the list
    void prepend(Node* aNode);
    void append(Node* aNode);
    void remove();

    // Accessors
    const DataType& getValue() const { return fValue; }
    Node* getNext() const { return fNext; }
    Node* getPrevious() const { return fPrevious; }

    // Check if the node is the end of the list
    bool isEnd() const { return fNext == nullptr; }
};
```

```cpp
class Inventory
{
public:
    Inventory();
    ~Inventory();

    //copy constructor
    Inventory(const Inventory& obj);

    void addItem(const Item& item);
    void removeItem(int index);

    // Accessors
    Item* getItem(int index) const;
    int size() const;
    void render(sf::RenderWindow& window, sf::Font& font, float x, float y) const;

private:
    //Using Doubly Linked List
    DoublyLinkedNode<Item*>* head;
    DoublyLinkedNode<Item*>* tail;
    int itemCount;
};
```

```cpp
Inventory::Inventory(const Inventory& obj) : Inventory() {
    if (obj.head == nullptr) {
        return; // Nothing to copy
    }

    DoublyLinkedNode<Item*>* current = obj.head;
    while (current != nullptr) {
        Item* currentItem = current->getValue();
        if (currentItem != nullptr) {
            addItem(*currentItem);
        }
        else {
            // Handle the error: currentItem is null
        }
        current = current->getNext();
    }
}
```

```cpp
void Inventory::addItem(const Item& item)
{
    DoublyLinkedNode<Item*>* newNode = new DoublyLinkedNode<Item*>(item.clone());
    if (tail != nullptr)
    {
        tail->append(newNode);

    }
    else
    {
        head = newNode;
    }

    tail = newNode;
    itemCount++;
}
```

```cpp
void Inventory::removeItem(int index) {
    if (index < 0 || index >= itemCount)
    {
        throw out_of_range("Index out of bounds");
    }

    DoublyLinkedNode<Item*>* current = head;
    for (int i = 0; i < index; ++i)
    {
        current = current->getNext();
    }

    if (current == head)
    {
        head = current->getNext();
    }

    if (current == tail)
    {
        tail = current->getPrevious();
    }

    current->remove();
    delete current->getValue();
    delete current;
    itemCount--;
}


Item* Inventory::getItem(int index) const {
    if (index < 0 || index >= itemCount)
    {
        throw out_of_range("Index out of bounds");
    }

    DoublyLinkedNode<Item*>* current = head;
    for (int i = 0; i < index; ++i)
    {
        current = current->getNext();
    }

    return current->getValue();
}
```

//test output

5.5.4 Troubleshooting Summary

5.5.4.1 Did you run into any issues while implementing this concept?

- **Circular References:** Encountered an issue where the tail node's next pointer was inadvertently set to the head node, creating a circular reference.

5.5.4.2 Did you refer to any resources (tutorials/guides) to find the solution?

- I referred to GeeksforGeeks – Insertion in Doubly Circular Linked List, specifically for guidance on handling insertions in doubly circular linked lists, which helped me understand and fix the circular reference issue in the doubly linked list.

5.5.4.3 How did you solve the problem?

- **Circular References:** Added checks to ensure that the tail node's next pointer was always set to null and similarly for the head node's previous pointer.

Cite your resources/references

GeeksforGeeks 2023, Insertion in doubly circular linked list, viewed 24 November 2023, <https://www.geeksforgeeks.org/insertion-in-doubly-circular-linked-list/>.

5.6 Dynamic Stack

5.6.1 Application

5.6.1.1 What area will you apply this concept to?

The dynamic stack is applied to manage the combat log during an enemy encounter in the event class. This combat log tracks and stores character and enemy ATTACK action and displays them in the order that the final attack information from the previous round is displayed first.

5.6.1.2 What is the desired operation? Describe in detail.

- **Event Recording**: As attack actions occur either by player or enemy, they are 'pushed' onto the stack.
- **Real-Time Event Retrieval**: The most recent combat events are accessed first, following the Last-In-First-Out (LIFO) principle. This is crucial for displaying the latest actions in the combat log and this combat log will be updated before the next round starts to be displayed properly.
- **Dynamic Capacity**: The stack grows or shrinks automatically, ensuring it can accommodate fluctuating numbers of combat events without wasting memory. This is because firstly, as the character levels up, their action points or ability to attack an enemy increases so there might more than one instance of attack action by the player. Secondly, as enemies are generated randomly, more enemies means there will be more attack action required to be stacked.

5.6.2 Concept

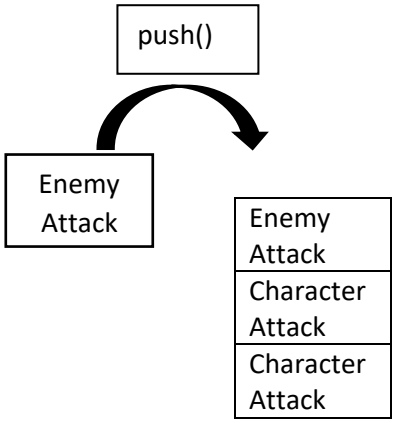
5.6.2.1 How does it match this application's desired operation?

- **Last-In-First-Out (LIFO) Principle**: Aligns perfectly with the need to access the most recent combat events first. This ensures that players can see the latest actions without scrolling through the entire log.
- **Dynamic Resizing:** Adapts to the varying flow of combat events, expanding and contracting as needed, which is essential in a dynamic combat environment.

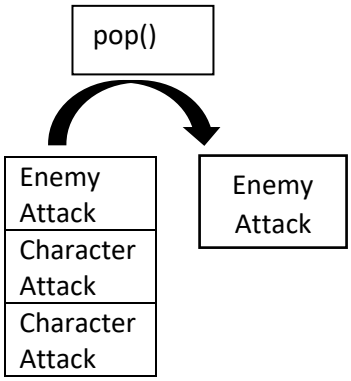
5.6.2.2 Are there any other alternative Structures can be used here? Why are they less preferred?

- **Queues (FIFO):** While they can store events, their First-In-First-Out nature makes them unsuitable for scenarios where the most recent event is prioritized.
- **Arrays:** Fixed-size or dynamic arrays could store events but lack the inherent LIFO structure and efficient real-time access that stacks provide.

5.6.2.3 Diagram



After an attack has been executed, it is recorded and added or “pushed” onto the stack. Each of the attack action are SinglyLinkedNodes.



Before the beginning of the next round, the combat log will pop the Action recorded in the previous round starting from the lates action which will be the enemy attack.

5.6.3 Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
template<class T>  <T> Provide sample template arguments for IntelliSense
class dynaStack
{
private:
    SinglyLinkedNode<T>* top;

public:
    dynaStack():top(nullptr){}

    ~dynaStack()
    {
        clear();
    }

    //Add an element to top of the stack
    void push(const T& value)
    {
        top = new SinglyLinkedNode<T>(value, top);
    }

    //Return the top element from the stack without removing it
    T peek() const
    {
        if (isEmpty())
        {
            throw out_of_range("Stack is empty");
        }

        return top->getValue();
    }

    //Remove and return the top element from the stack
    T pop()
    {
        if (isEmpty())
        {
            throw out_of_range("Stack is empty");
        }

        T value = top->getValue();
        SinglyLinkedNode<T>* oldTop = top;
        top = top->getNext();
        delete oldTop;
        return value;
    }
```

```
    //Check if the stack is empty
    bool isEmpty() const
    {
        return top == nullptr;
    }

    //Clear the stack
    void clear()
    {
        while (!isEmpty())
        {
            pop();
        }
    }

    // Get the top node of the stack for traversal without modification
    SinglyLinkedNode<T>* getTopNode() const {
        return top;
    }
```

```cpp
void Event::displayCombatLog(std::stringstream& ss, Character& character) {

    ss << "\n ------------------------ Combat Log ------------------------ \n";
    if (combatLog.isEmpty()) {
        ss << "\nNo actions in the log.\n";

    }

    // Temporary stack to hold actions while displaying them
    dynaStack<Action> tempStack;

    // Display and transfer actions to the temporary stack
    while (!combatLog.isEmpty()) {
        Action action = combatLog.pop();
        ss << action.message << endl;  // Concatenate the custom message
        tempStack.push(action);
    }

    // Transfer actions back to the original stack
    while (!tempStack.isEmpty()) {
        combatLog.push(tempStack.pop());
    }
}
```

```
 ------------------------ Combat Log ------------------------
Dread Knight ATTACKS character - Missed
Forest Wraith ATTACKS character - Missed
Forest Wraith ATTACKS character - Damage: 5
Dread Knight ATTACKS character - Damage: 4
Character ATTACKS Dread Knight - Missed

 ------------------------ Battle Menu ------------------------
```

5.6.4 Troubleshooting Summary

5.6.4.1 Did you run into any issues while implementing this concept?

- **Stack Overflow**: I encountered an issue where the stack overflowed due to continuous pushing of combat events without proper checks or limits.

5.3.1.1 Did you refer to any resources (tutorials/guides) to find the solution?

No references needed as I understood the need of a check and found the solution while debugging.

How did you solve the problem?

- **Stack Overflow:** Implemented a check to prevent pushing new events onto the stack once a certain limit was reached, thereby preventing overflow.

Queue

Application

What area will you apply this concept to?

The Queue is applied to queue the Attack action of the character. This is because there is an action point system whereby the number of attacks the player can do to the enemy is limited by this and the attack action is not executed immediately, only when the player ends their turn. This will be seen under Queued Actions in the implementation.

What is the desired operation? Describe in detail.

- **Sequential Action Processing:** The queue would store combat actions in the order they are initiated. This ensures that actions are processed in a First-In-First-Out (FIFO) manner.
- **Combat Flow Management:** The queue helps in managing the flow of combat by queuing actions and processing them sequentially, reflecting the turn-based nature of combat which is what I intend for the combat system to be.

Concept

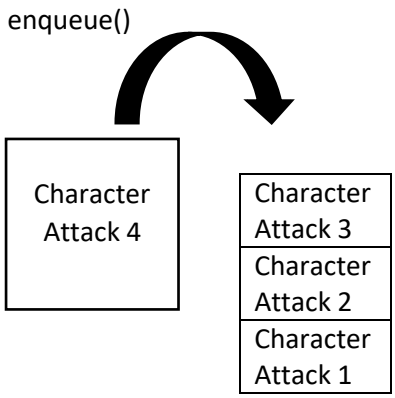
How does it match this application's desired operation?

- **FIFO (First in First out) Structure**: Aligns with the need for actions to be executed in the order they were initiated, maintaining the integrity of the turn-based combat system.
- **Dynamic Sizing**: Adapts to the varying number of combat actions, expanding and contracting as required. This is because as the player increases level, their number of action points also increases allowing them to queue even more attacks towards the enemy.

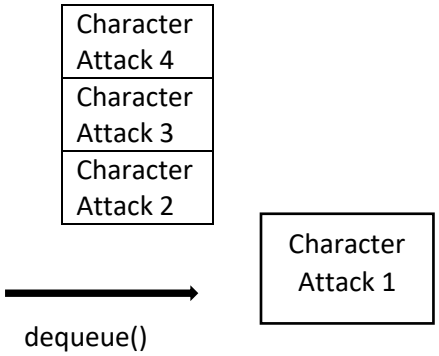
Are there any other alternative Structures can be used here? Why are they less preferred?

- **Stacks (Last In Last out):** Less suitable for turn-based actions as they would process the most recent action first, disrupting the combat flow.
- **Lists:** While they can offer similar functionality, managing actions in a FIFO manner is more complex with lists compared to queues.

Diagram



During a player's turn they can queue a number of attacks onto different enemies depending on their number of action points.



Once the player is satisfied with their queued actions, they will end their turn and this will execute the actions and dequeue in the order of First In First Out so the Character Attack 1 will dequeue first.

Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
template <class T>
class Queue
{
private:
    SinglyLinkedNode<T>* front;
    SinglyLinkedNode<T>* rear;
    int size;

public:
    //Constructor
    Queue(): front(nullptr), rear(nullptr), size(0){}

    //Destructor
    ~Queue()
    {
        while (!isEmpty())
        {
            dequeue();
        }
    }

    // Copy constructor
    Queue(const Queue<T>& other) : front(nullptr), rear(nullptr), size(0) {
        SinglyLinkedNode<T>* current = other.front;
        while (current != nullptr) {
            this->enqueue(current->getValue());
            current = current->getNext();
        }
    }

    // Copy assignment operator
    Queue<T>& operator=(const Queue<T>& other) {
        if (this != &other) { // Protect against self-assignment
            // Clear the current queue
            while (!this->isEmpty()) {
                this->dequeue();
            }

            // Copy elements from the other queue
            SinglyLinkedNode<T>* current = other.front;
            while (current != nullptr) {
                this->enqueue(current->getValue());
                current = current->getNext();
            }
        }
        return *this;
    }
```

```
    //Dequeue
    T dequeue()
    {
        if (isEmpty())
        {
            throw out_of_range("Queue is empty");
        }

        T value = front->getValue();
        SinglyLinkedNode<T>* toRemove = front;
        front = front->getNext();
        delete toRemove;
        size--;

        if (isEmpty())
        {
            rear = nullptr;
        }

        return value;
    }


    //Check if the queue is empty
    bool isEmpty() const
    {
        return front == nullptr;
    }

    //Get the size of the the queue
    int getSize() const
    {
        return size;
    }

    //Peek at the front of the queue
    T& peek() const
    {
        if (isEmpty())
        {
            throw out_of_range("Queue is empty");
        }

        return front->getValue();
    }
};
```

```cpp
void Event::queueAttackAction(Character& character, Enemy* targetedEnemy) {
    if (!targetedEnemy || !targetedEnemy->isAlive()) {
        // Handle the case where the targeted enemy is invalid or already defeated
        std::cout << "Cannot attack. Target enemy is invalid or already defeated." << std::endl;
        return;
    }

    // Create an attack action targeting the selected enemy
    Action attackAction(ActionType::ATTACK, &character, targetedEnemy, -1, "Attack action queued");

    // Add the attack action to the queue
    this->actionQueue.enqueue(attackAction);

    // Deduct action points required for the attack
    character.useActionPoints(2);
}
```

```cpp
void Event::displayQueuedActions(std::stringstream& ss, Character& character) {
    Queue<Action> tempQueue = this->actionQueue;

    ss << "\n ------------------------- Queued Actions ------------------------- \n";
    while (!tempQueue.isEmpty()) {
        Action currentAction = tempQueue.dequeue();
        ss << actionToString(currentAction, character) << "\n";
    }
}
```

```
 ---------------------- CHARACTER TURN ----------------------
YOU'VE ENCOUNTERED 4 ENEMY/ENEMIES!
 1: Dread Knight Lvl: 2 - [HP: [24/24] | DEF: 4 | ACC: 4]
 2: Forest Wraith Lvl: 2 - [HP: [10/10] | DEF: 2 | ACC: 6]
 3: Forest Wraith Lvl: 1 - [HP: [5/5] | DEF: 1 | ACC: 3]
 4: Dread Knight Lvl: 2 - [HP: [24/24] | DEF: 4 | ACC: 4]

 --------------------------- Quick Stats ---------------------------
Current Character HP: 17 / 22
Equipped Weapon: None
Equipped Armor: None
Action Points: 0

 ------------------------- Queued Actions -------------------------
Character ATTACKS Dread Knight - Damage: Pending

 ------------------------- Combat Log -------------------------

No actions in the log.

 ------------------------- Battle Menu -------------------------
1: Escape
2: Attack
3: Use Item
4: End Turn
```

Troubleshooting Summary

Did you run into any issues while implementing this concept?

No issues. The Queue implementation was rather straightforward.

Binary Tree

Application

What area will you apply this concept to?

The binary tree concept is applied to manage the logic of puzzle encounters in the game. Each node of the tree represents a decision point in the adventure, leading to different outcomes and further decisions.

What is the desired operation? Describe in detail.

The binary tree is used to structure a series of decisions and outcomes in a puzzle adventure. Each decision leads to a different branch of the tree, with outcomes that affect the game's storyline or the character's attributes. The operation involves traversing this tree based on the player's choices, ultimately leading to different endings or consequences within the game.

Concept

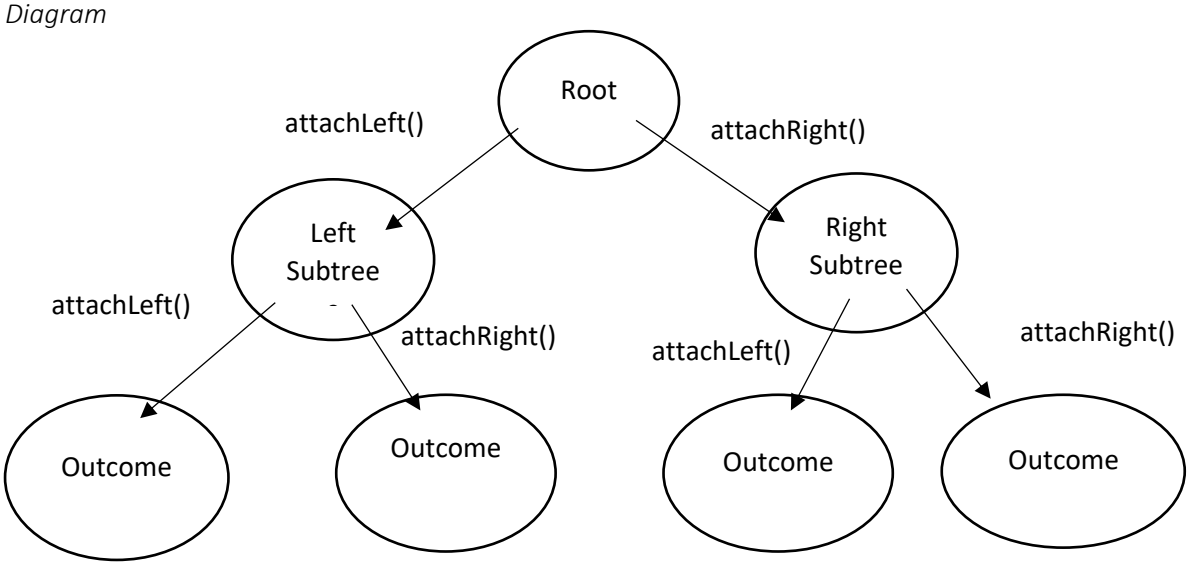
How does it match this application's desired operation?

The binary tree structure is ideal for this application because it efficiently represents the branching nature of decisions in the game. Each node in the tree corresponds to a decision point, and the branches represent the possible paths a player can take. This aligns perfectly with the concept of an adventure game where each choice can lead to a different storyline or outcome.

Are there any other alternative Structures can be used here? Why are they less preferred?

- Linked Lists: While linked lists could be used to represent a sequence of decisions, they lack the branching structure of a binary tree, making them less suitable for representing multiple decision paths.
- Arrays: Arrays are not dynamic and would be inefficient in representing complex, branching decision paths due to their fixed size and linear nature.

Diagram



This is how the Temple Adventures and Dungeon Adventures are structured in the game.

Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```
template <class T>
class BTree {
public:
    static std::unique_ptr<BTree<T>> NIL; // Sentinel

    // Constructors
    BTree(); // Empty BTree
    BTree(T aKey); // Root with 2 subtrees
    ~BTree() = default;

    // Member functions
    bool isEmpty() const;
    const T& key() const;
    std::unique_ptr<BTree<T>>& left();
    std::unique_ptr<BTree<T>>& right();
    void attachLeft(std::unique_ptr<BTree<T>> aBTree);
    void attachRight(std::unique_ptr<BTree<T>> aBTree);
    std::unique_ptr<BTree<T>> detachLeft();
    std::unique_ptr<BTree<T>> detachRight();

private:
    std::unique_ptr<T> fKey;
    std::unique_ptr<BTree<T>> fLeft;
    std::unique_ptr<BTree<T>> fRight;
};
```

```
}
unique_ptr<BTree<Decision>> Event::createTempleAdventure(Character& character) {
    // Root of the temple adventure decision tree
    auto rootDecision = DecisionFactory::createDecision(
        "As you approach the entrance of the ancient temple, \nshrouded in the mists of time, you are presented with a pivotal choi
        false, 0, "", "", "", nullptr, nullptr, nullptr);
    auto templeAdventure = make_unique<BTree<Decision>>(move(*rootDecision));

    // Left subtree: Overgrown passage
    auto leftPathDecision = DecisionFactory::createDecision(
        "You go through the overgrown passage and you found \na hidden alcove and a suspicious shiny door in your way.\n\nLeft: Hid
        false, 0, "", "", "", nullptr, nullptr, nullptr);
    auto leftPath = make_unique<BTree<Decision>>(move(*leftPathDecision));

    // Left-Left: Hidden alcove
    auto hiddenAlcoveDecision = DecisionFactory::createDecision(
        "You found a hidden alcove filled with ancient texts. \nThese would require some knowledge of the old arts to decipher (Int
        "You decipher the texts and gain knowledge! +1 Intelligence", "The writings are too cryptic. You get a headache. Took 5 Dam
        new IncreaseStatEffect("Intelligence", 1), new DamageHealthEffect(5), nullptr
    );
    leftPath->attachLeft(make_unique<BTree<Decision>>(move(*hiddenAlcoveDecision)));

    // Left-Right: Trapped floor
    auto trappedFloorDecision = DecisionFactory::createDecision(
        "Oh no, the door closed behind you and you find yourself in a room with hidden floor traps. (Dexterity LVL 10)", true, 10,
        "You navigate the traps safely! +1 Dexterity", "You trigger a trap but escape narrowly. You sprained an ankle. Took 5 Damag
        new IncreaseStatEffect("Dexterity", 1), new DamageHealthEffect(5), nullptr
    );
    leftPath->attachRight(make_unique<BTree<Decision>>(move(*trappedFloorDecision)));

    // Attach left subtree to root
    templeAdventure->attachLeft(move(leftPath));

    // Right subtree: Echoing hall
    auto rightPathDecision = DecisionFactory::createDecision(
        "You arrive at the Echoing hall and found a Guarded Relic \non your left and a Mysterious Statue on your right.\nLeft: Guar
        false, 0, "", "", "", nullptr, nullptr, nullptr
    );
    auto rightPath = make_unique<BTree<Decision>>(move(*rightPathDecision));

    // Right-Left: Guarded relic
    auto guardedRelicDecision = DecisionFactory::createDecision(
        "A relic is suddenly guarded by spectral forces. (Require Strength Lvl 15)", true, 15, "Strength",
        "You fend off the guardians and claim the relic! +1 Strength", "The spectral forces overpower you, but you escape. Barely.
        new IncreaseStatEffect("Strength", 1), new DamageHealthEffect(5), nullptr
    );
    rightPath->attachLeft(make_unique<BTree<Decision>>(move(*guardedRelicDecision)));

    // Right-Right: Mysterious statue
    auto mysteriousStatueDecision = DecisionFactory::createDecision(
        "A statue with a riddle. (Require Intelligence Lvl 14)", true, 14, "Intelligence",
        "You solve the riddle and find a hidden treasure! +1 Intelligence", "The riddle baffles you, until you get a migraine. Took
        new IncreaseStatEffect("Intelligence", 1), new DamageHealthEffect(5), nullptr
    );
    rightPath->attachRight(make_unique<BTree<Decision>>(move(*mysteriousStatueDecision)));

    // Attach right subtree to root
    templeAdventure->attachRight(move(rightPath));

    return templeAdventure;
}
```

```cpp
unique_ptr<BTree<Decision>> Event::createDungeonAdventure(Character& character) {
    // Root of the dungeon adventure decision tree
    auto rootDecision = DecisionFactory::createDecision(
        "As you stand at the entrance of the ominous dungeon, you're faced with a choice:\n\n To your left, a dimly
        false, 0, "", "", "", nullptr, nullptr, nullptr);

    auto dungeonAdventure = make_unique<BTree<Decision>>(move(*rootDecision));

    // Left subtree: Dimly lit corridor
    auto leftSubtreeDecision = DecisionFactory::createDecision(
        "At the end of the dimly lit corridor, you discover a mysterious chest, \nits ancient wood and intricate lo
        false, 0, "", "", "", nullptr, nullptr, nullptr);
    auto leftSubtree = make_unique<BTree<Decision>>(move(*leftSubtreeDecision));

    // Generate an item
    unique_ptr<Item> foundItem = generateRandomItem(character.getLevel());

    // Left-Left: Mysterious chest
    auto chestDecision = DecisionFactory::createDecision(
        "A heavy chest requires strength to open. I hope you are strong enough adventurer.",
        true, 5, "Strength",
        "You open the chest to find treasure! Check your Inventory!\n Press ENTER to continue....",
        "The chest won't budge. You sprained an ankle. Took 5 Damage! \n Press ENTER to continue....",
        new FindItemEffect(),
        new DamageHealthEffect(5),
        foundItem.release());
    leftSubtree->attachLeft(make_unique<BTree<Decision>>(move(*chestDecision)));

    // Left-Right: Mystical barrier
    auto barrierDecision = DecisionFactory::createDecision(
        "An arcane barrier blocks your path.", true, 5, "Intelligence",
        "You dispel the barrier! You learned a new arcane art that may prove \nuseful in the future. +1 Intelligenc
        new IncreaseStatEffect("Intelligence", 5), new DamageHealthEffect(5), nullptr);
    leftSubtree->attachRight(make_unique<BTree<Decision>>(move(*barrierDecision)));

    // Attach left subtree to root
    dungeonAdventure->attachLeft(move(leftSubtree));

    // Right subtree: Sound of water
    auto rightSubtreeDecision = DecisionFactory::createDecision(
        "Sound of water.\nLeft: Suspicious water spring.\nRight: Narrow bridge (Require Dexterity Level 5).\nChoose
        false, 0, "", "", "", nullptr, nullptr, nullptr);
    auto rightSubtree = make_unique<BTree<Decision>>(move(*rightSubtreeDecision));

    // Right-Left: Healing spring
    auto healingSpringDecision = DecisionFactory::createDecision(
        "You found a suspicious spring. It looks quite tempting to drink from it",
        true, 1, "Intelligence",
        "You feel rejuvenated! Vitality Increased. \n Press ENTER to continue....",
        "You feel rejuvenated! Vitality Increased. \n Press ENTER to continue....",
        new IncreaseStatEffect("Vitality", 1), nullptr, nullptr);
    rightSubtree->attachLeft(make_unique<BTree<Decision>>(move(*healingSpringDecision)));

    // Right-Right: Narrow bridge
    auto narrowBridgeDecision = DecisionFactory::createDecision(
        "A precarious bridge requires dexterity to cross.",
        true, 8, "Dexterity",
        "You skillfully cross the bridge! You were surprised on how you did that. +1 Dexterity \n Press ENTER to co
        "You tripped and fell on your face but manage to cross. Took 5 Damage! \n Press ENTER to continue....",
        new IncreaseStatEffect("Dexterity", 1), new DamageHealthEffect(5), nullptr);
    rightSubtree->attachRight(make_unique<BTree<Decision>>(move(*narrowBridgeDecision)));

    // Attach right subtree to root
    dungeonAdventure->attachRight(move(rightSubtree));

    return dungeonAdventure;
}
unique_ptr<BTree<Decision>> Event::createTempleAdventure(Character& character) {
```

As you approach the entrance of the ancient temple,
shrouded in the mists of time, you are presented with a pivotal choice.
To your left, an overgrown passage, its path concealed by the relentless grasp of nature,
suggests secrets hidden and reclaimed by the earth.

To your right, an echoing hall, its vastness reverberating with the whispers of ages past,
beckons you towards the unknown grandeur and potential perils within.

You can only choose one path.
Will it be the tangled mysteries of the overgrown passage
or the daunting expanse of the echoing hall?.

Left: Head to the overgrown passage.
Right: Head to echoing hall.

Choose a path :
1.Left
2.Right

You go through the overgrown passage and you found
a hidden alcove and a suspicious shiny door in your way.

Left: Hidden alcove.
Right: Suspicious Shiny Door
Choose a path :
1.Left
2.Right

The writings are too cryptic. You get a headache. Took 5 Damage!

Troubleshooting Summary

Did you run into any issues while implementing this concept?

The main issue in implementing this binary tree could be ensuring that each decision leads to the correct subsequent decision and that the end of each path is handled correctly especially on the effects that should happened to the character.

Did you refer to any resources (tutorials/guides) to find the solution?

None.

How did you solve the problem?

I introduced a DecisionEffect class to handle the outcome effects.

Iterator

Application

What area will you apply this concept to?

The Iterator pattern is applied to navigate through a collection of items available for sale in a shop within the game.

What is the desired operation? Describe in detail.

The desired operation involves iterating over the collection of items to find and process a specific item chosen for purchase. The Iterator provides a way to access items sequentially without exposing the underlying data structure (linked list).

Concept

How does it match this application's desired operation?

The Iterator pattern matches the application's desired operation by providing a straightforward way to traverse the collection of items. It abstracts the details of the data structure, allowing the Shop class to focus on higher-level functionalities like displaying items for sale and processing purchases.

Are there any other alternative Structures can be used here? Why are they less preferred?

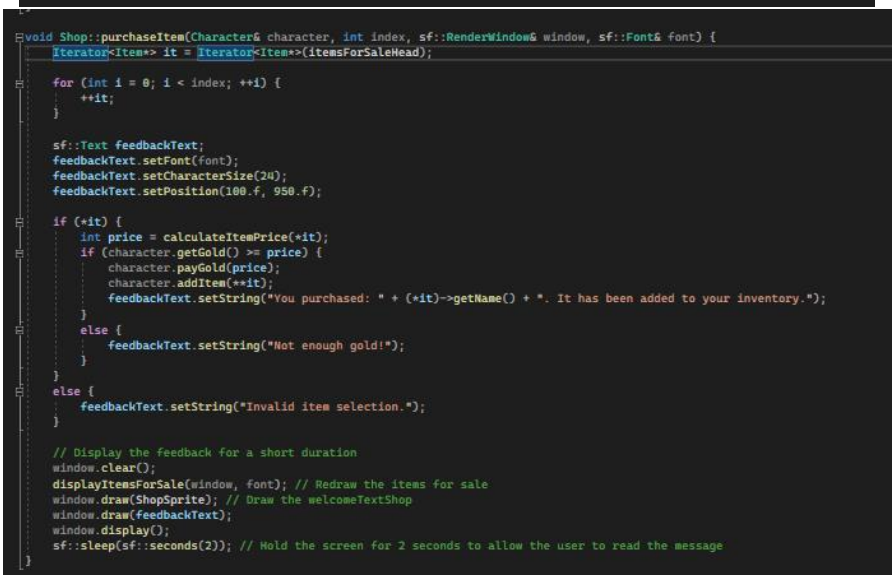
Diagram

- **Direct Access through Linked List:** Directly accessing items in a linked list is less preferred because it would require the Shop class to manage the details of list traversal, increasing coupling and reducing modularity.
- **Array-Based Approach:** Using an array might simplify access but lacks the dynamic resizing and efficient insertion/deletion at arbitrary locations provided by a linked list.

Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

```cpp
#include "SinglyLinkedNode.h"

template <class DataType>
class Iterator
{
private:
    SinglyLinkedNode<DataType>* currentNode;

public:
    //Constructor
    Iterator(SinglyLinkedNode<DataType>* node = nullptr): currentNode(node){}

    //Dereference operator
    DataType& operator*()
    {
        return currentNode->getValue();
    }

    //Prefix incremenet operator
    Iterator& operator++()
    {
        if (currentNode)
        {
            currentNode = currentNode->getNext();
        }
        return *this;
    }

    //Postfix increment operator
    Iterator operator++ (int)
    {
        Iterator temp = *this;
        ++(*this);
        return this;
    }

    //Equality check operator
    bool operator==(const Iterator& other) const
    {
        return currentNode == other.currentNode;
    }

    bool operator!=(const Iterator& other) const
    {
        return !(*this == other);
    }
};
```



```cpp
void Shop::purchaseItem(Character& character, int index, sf::RenderWindow& window, sf::Font& font) {
    Iterator<Item*> it = Iterator<Item*>(itemsForSaleHead);

    for (int i = 0; i < index; ++i) {
        ++it;
    }

    sf::Text feedbackText;
    feedbackText.setFont(font);
    feedbackText.setCharacterSize(24);
    feedbackText.setPosition(100.f, 950.f);

    if (*it) {
        int price = calculateItemPrice(*it);
        if (character.getGold() >= price) {
            character.payGold(price);
            character.addItem(**it);
            feedbackText.setString("You purchased: " + (*it)->getName() + ". It has been added to your inventory.");
        }
        else {
            feedbackText.setString("Not enough gold!");
        }
    }
    else {
        feedbackText.setString("Invalid item selection.");
    }

    // Display the feedback for a short duration
    window.clear();
    displayItemsForSale(window, font); // Redraw the items for sale
    window.draw(ShopSprite); // Draw the welcomeTextShop
    window.draw(feedbackText);
    window.display();
    sf::sleep(sf::seconds(2)); // Hold the screen for 2 seconds to allow the user to read the message
}
```

Troubleshooting Summary

Did you run into any issues while implementing this concept?

No issues

Factory

Application

What area will you apply this concept to?

The Factory pattern is applied in the game for creating different types of items dynamically, such as weapons, armor, and potions. Each type of item has different properties and behaviors.

What is the desired operation? Describe in detail.

The desired operation is to instantiate various item types based on the game's requirements at runtime. The Factory pattern allows for the creation of items without specifying the exact class of the object that will be created.

Concept

How does it match this application's desired operation?

The Factory pattern matches the application's desired operation by providing a flexible and scalable way to create objects. By using a Factory class, the game can instantiate different types of items (weapons, armor, potions) without being tightly coupled to the concrete classes of these items.

Are there any other alternative Structures can be used here? Why are they less preferred?

Direct Instantiation: Creating items directly without a Factory would tightly couple the game's code to specific item classes, making it less flexible and harder to maintain or extend with new item types.

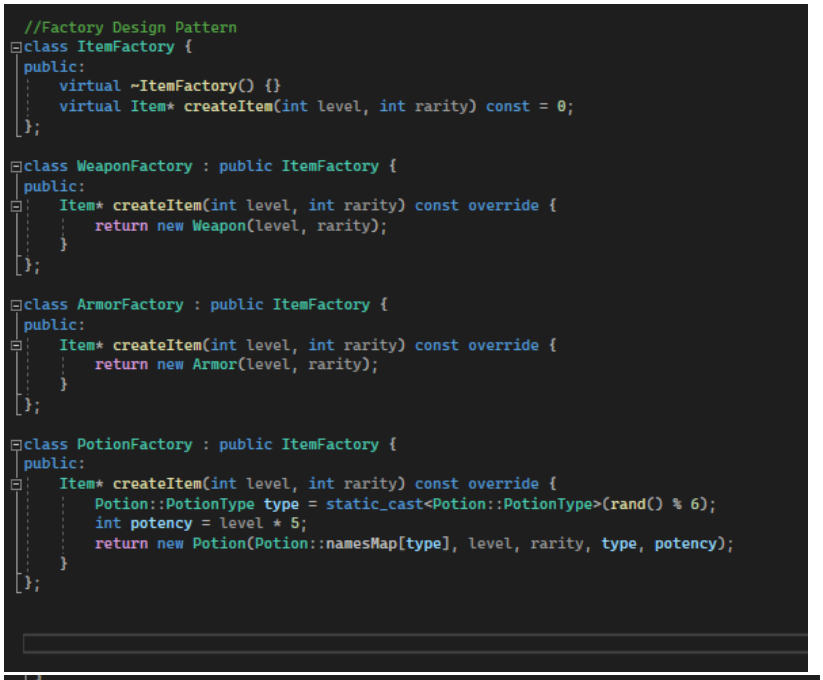
Prototype Pattern: While this could be used for cloning existing items, it wouldn't be as effective for creating new items with specific properties like level and rarity, which the Factory pattern can handle more efficiently.

Diagram

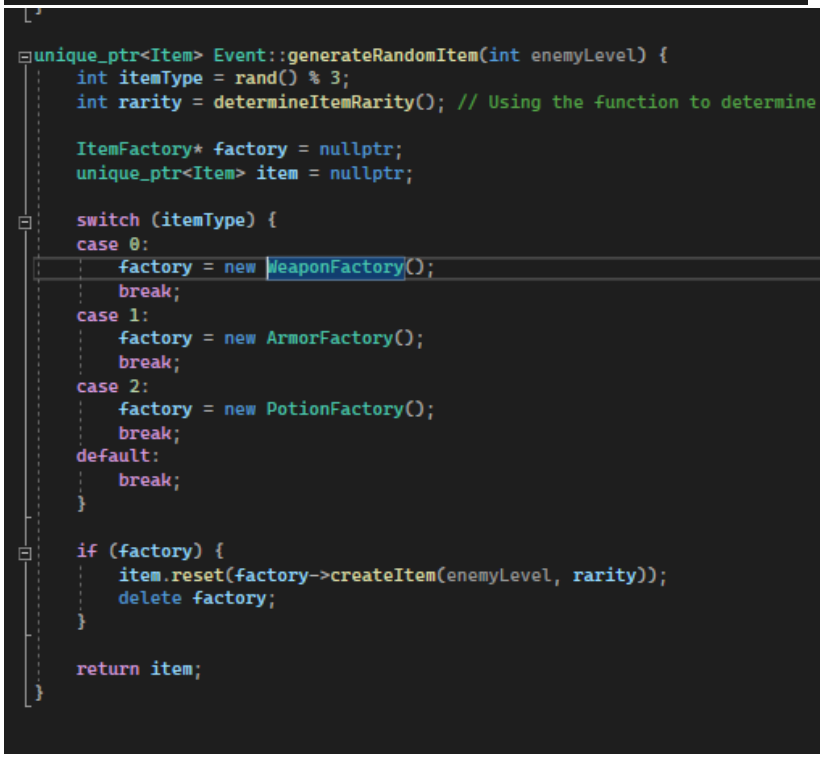
Implementation and Output

Include a screen-capture of this code implementation. Ensure that your code has comments to show how it works, step-by-step.

If the subsystem does not have an output, add temporary testing code to show that it works properly via Command Prompt output.



```
//Factory Design Pattern
class ItemFactory {
public:
    virtual ~ItemFactory() {}
    virtual Item* createItem(int level, int rarity) const = 0;
};

class WeaponFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        return new Weapon(level, rarity);
    }
};

class ArmorFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        return new Armor(level, rarity);
    }
};

class PotionFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        Potion::PotionType type = static_cast<Potion::PotionType>(rand() % 6);
        int potency = level * 5;
        return new Potion(Potion::namesMap[type], level, rarity, type, potency);
    }
};
```



```
unique_ptr<Item> Event::generateRandomItem(int enemyLevel) {
    int itemType = rand() % 3;
    int rarity = determineItemRarity(); // Using the function to determine

    ItemFactory* factory = nullptr;
    unique_ptr<Item> item = nullptr;

    switch (itemType) {
    case 0:
        factory = new WeaponFactory();
        break;
    case 1:
        factory = new ArmorFactory();
        break;
    case 2:
        factory = new PotionFactory();
        break;
    default:
        break;
    }

    if (factory) {
        item.reset(factory->createItem(enemyLevel, rarity));
        delete factory;
    }

    return item;
}
```

Troubleshooting Summary

Did you run into any issues while implementing this concept?

None

//Action.h

```
#pragma once

#include "STLInclude.h"
#include "Enemy.h"

//forward declaration Character
class Character;


enum class ActionType
{
	ATTACK, //Attack by Character
	ENEMY_ATTACK //Attack by Enemy
};

struct Action {
    ActionType type;
    Character* attacker;
    Enemy* targetEnemy;
    int damage;
    string message;

    // Updated constructor
    Action(ActionType type, Character* attacker, Enemy* targetEnemy, int damage,
string message = "")
        : type(type), attacker(attacker), targetEnemy(targetEnemy),
damage(damage), message(move(message)) {}
};
```

//Armor.h

```
#pragma once
#include "STLInclude.h"
#include "Item.h"

class Armor : public Item
{
public:
    Armor();
    Armor(int level, int rarity);
    Armor(int defence,
        string name,
        int level,
        int buyValue,
        int sellValue,
        int rarity);

    virtual ~Armor();

    // Pure virtual
    virtual Armor* clone() const override;

    // Functions
    string toString() const;

    // Accessors
    inline int getDefence() const { return this->defence; }

    // Modifiers
    static dynaArr<string> names; //using Dynamic Array to store Armor names
    static void initNames(); //Initialise unique Armor names
```

```
private:
    int defence;
};
```

//Armor.cpp

```
// Armor.cpp
#include "Armor.h"


dynaArr<string> Armor::names;

//Use Dynamic Array to store Armor names
void Armor::initNames()
{
	Armor::names.push("Leather Armor");
	Armor::names.push("iron Armor");
	Armor::names.push("Gold Armor");
	Armor::names.push("ChainMail Armor");
	Armor::names.push("Diamond Armor");
	Armor::names.push("Nether Armor");
}

//Defaule Constructor
Armor::Armor()
	:Item("NONE", 0, 0, 0, COMMON) {
	this->defence = 0;
}

//Constructor
Armor::Armor(int level, int rarity)
	:Item(Armor::names[rand() % Armor::names.size()],
		level,
		calculateBuyValue(level, rarity),
		calculateSellValue(level, rarity),
		rarity)
{
	this->defence = rand() % (level + (rarity + 1));
	this->defence += (rarity + 1) * 5;
	if (rarity == 3)
		this->defence += level * 5;
	else if (rarity == 4)
		this->defence += level * 10;
}


Armor::Armor(int defence, string name, int level, int buyValue, int sellValue,
int rarity)
	: Item(name, level, buyValue, sellValue, rarity) {
	this->defence = defence;
}

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

//Pure virtual
```

```cpp
Armor* Armor::clone() const
{
    return new Armor(*this);
}

//Convert and save private attributes in string
string Armor::toString() const
{
      string str =
            " -------------------- \n Name: " + this->getName()
            + " \n Level: " + to_string(this->getLevel())
            + " \n Rarity: " + this->getRarity()
            + " \n Buy Value: " + to_string(this->getBuyValue()) + " Gold"
            + " \n -------------------- ";


      return str;
}
```

//Btree.h

```cpp
#pragma once
#include <memory>
#include <stdexcept>

template <class T>
class BTree {
public:
    static std::unique_ptr<BTree<T>> NIL; // Sentinel

    // Constructors
    BTree(); // Empty BTree
    BTree(T aKey); // Root with 2 subtrees
    ~BTree() = default;

    // Member functions
    bool isEmpty() const;
    const T& key() const;
    std::unique_ptr<BTree<T>>& left();
    std::unique_ptr<BTree<T>>& right();
    void attachLeft(std::unique_ptr<BTree<T>> aBTree);
    void attachRight(std::unique_ptr<BTree<T>> aBTree);
    std::unique_ptr<BTree<T>> detachLeft();
    std::unique_ptr<BTree<T>> detachRight();

private:
    std::unique_ptr<T> fKey;
    std::unique_ptr<BTree<T>> fLeft;
    std::unique_ptr<BTree<T>> fRight;
};

// Static member initialization
template<class T>
std::unique_ptr<BTree<T>> BTree<T>::NIL = nullptr;

// Implementation of the BTree methods
template <class T>
BTree<T>::BTree() : fKey(nullptr), fLeft(nullptr), fRight(nullptr) {}

//Constructor for Root with 2 subtrees
template<class T>
```

```cpp
BTree<T>::BTree(T aKey) : fKey(std::make_unique<T>(std::move(aKey))),
fLeft(nullptr), fRight(nullptr) {}

//Check if Binary Tree is empty
template<class T>
bool BTree<T>::isEmpty() const {
    return fKey == nullptr;
}

//Return smart pointer
template <class T>
const T& BTree<T>::key() const {
    if (isEmpty()) throw std::logic_error("Empty tree has no key");
    return *fKey;
}

//Return left subtree
template<class T>
std::unique_ptr<BTree<T>>& BTree<T>::left() {
    if (isEmpty()) throw std::logic_error("Empty tree has no left child");
    return fLeft;
}

//Return right subtree
template<class T>
std::unique_ptr<BTree<T>>& BTree<T>::right() {
    if (isEmpty()) throw std::logic_error("Empty tree has no right child");
    return fRight;
}

//Attach to the left subtree
template<class T>
void BTree<T>::attachLeft(std::unique_ptr<BTree<T>> aBTree) {
    if (!isEmpty() && !fLeft) {
        fLeft = std::move(aBTree);
    }
    else {
        throw std::logic_error("Tree already has a left child or is empty");
    }
}

//Attach to the right subtree
template<class T>
void BTree<T>::attachRight(std::unique_ptr<BTree<T>> aBTree) {
    if (!isEmpty() && !fRight) {
        fRight = std::move(aBTree);
    }
    else {
        throw std::logic_error("Tree already has a right child or is empty");
    }
}

//Detach from left subtree
template <class T>
std::unique_ptr<BTree<T>> BTree<T>::detachLeft() {
    if (isEmpty()) {
        throw std::logic_error("Tree is empty");
    }
    return std::move(fLeft);
}

//Detach from right subtree
template <class T>
```

```
std::unique_ptr<BTree<T>> BTree<T>::detachRight() {
    if (isEmpty()) {
        throw std::logic_error("Tree is empty");
    }
    return std::move(fRight);
}
```

//Button.h

```
#pragma once
#include <SFML/Graphics.hpp>
#include <string>

class Button {
    sf::RectangleShape shape;
    sf::Text text;
    sf::Texture normalTexture; //Texture when no mouse hover
    sf::Texture hoverTexture; //Texture when mouse hover
    bool isHovering;

public:
    //Constructor
    Button(float x, float y, float width, float height, sf::Font& font, const
std::string& textStr,
        const std::string& normalTexturePath, const std::string&
hoverTexturePath) {
        shape.setPosition(sf::Vector2f(x, y));
        shape.setSize(sf::Vector2f(width, height));


        if (!normalTexture.loadFromFile(normalTexturePath)) {
            std::cerr << "Error loading normal texture" << std::endl;
        }

        if (!hoverTexture.loadFromFile(hoverTexturePath)) {
            std::cerr << "Error loading hover texture" << std::endl;
        }

        shape.setTexture(&normalTexture);
        isHovering = false;


        text.setFont(font);
        text.setString(textStr);

    }

    void update(sf::RenderWindow& window) {
        if (isMouseOver(window)) {
            isHovering = true;
            shape.setTexture(&hoverTexture);
        }
        else {
            isHovering = false;
            shape.setTexture(&normalTexture);
        }
    }

    void drawTo(sf::RenderWindow& window) {
        window.draw(shape);
        window.draw(text);
    }
```

```
    // Check if the mouse is over the button
    bool isMouseOver(sf::RenderWindow& window) {
        sf::Vector2i mousePos = sf::Mouse::getPosition(window);
        sf::Vector2f btnPos = shape.getPosition();

        if (mousePos.x > btnPos.x && mousePos.x < btnPos.x +
shape.getLocalBounds().width &&
            mousePos.y > btnPos.y && mousePos.y < btnPos.y +
shape.getLocalBounds().height) {


            return true;
        }

        return false;
    }
};
```

//Character.h

```
#pragma once
#include "STLInclude.h"
#include "Inventory.h"
#include "Action.h"

class Character
{
public:
	Character();
	Character(string name,
		int distanceTravelled,
		int gold, int level,
		int exp, int strength,
		int vitality, int dexterity,
		int intelligence, int hp,
		int stamina, int statPoints);
	virtual ~Character();

	// Functions
	void initialize(const string name);
	void printStats() const;
	void levelUp();
	void updateStats();
	void addToStat(int stat, int value);
	inline void resetHP() { this->hp = this->hpMax; }
	void addItem(const Item& item) { this->inventory.addItem(item); }
	void usePotion(int index);
	void equipWeapon(int index);
	void equipArmor(int index);
	void applyPotionEffects(const Potion& potion);
	void useActionPoints(int amount);
	int getActionPoints() const;
	void resetActionPoints();
	void increaseHP(int amount);
	void increaseStat(const string& statName, int value);


	//Accessors
	inline const int& getDistanceTravelled() const { return this-
>distanceTravelled; }
```

```
	inline const string& getName() const { return this->name; }
	inline const int& getLevel() const { return this->level; }
	inline const int& getExp() const { return this->exp; }
	inline const int& getExpNext() const { return this->expNext; }
	inline const int& getStatPoints() const { return this->statPoints; }
	inline const int& getHp() const { return this->hp; }
	inline const int& getHpMax() const { return this->hpMax; }
	inline bool isAlive() { return this->hp > 0; }
	inline const int& getStamina() const { return this->stamina; }
	inline const int& getDamageMin() const { return this->damageMin; }
	inline const int& getDamageMax() const { return this->damageMax; }
	inline const int getDamage() const { return rand() % this->damageMax +
this->damageMin; }
	inline const int& getDefence() const { return this->defence; }
	inline const int& getAccuracy() const { return this->accuracy; }
	inline const int& getGold() const { return this->gold; }
	int getStatValue(const string& statName) const;
	string getAsString() const;
	string getInventoryAsString() const;
	Item* getInventoryItem(int index) const;
	int getInventorySize() const;
	Weapon* getEquippedWeapon() const;
	Armor* getEquippedArmor() const;


	// Modifier
	inline void setDistanceTravelled(const int& distance) { this-
>distanceTravelled = distance; }
	inline void travel() { this->distanceTravelled++; }
	inline void gainExp(const int& exp) { this->exp += exp; }
	inline void gainGold(const int& gold) { this->gold += gold; }
	inline void payGold(const int gold) { this->gold -= gold; }
	void takeDamage(int damage);
	inline void retry() { this->hp = hpMax; }

	//Print to S
	void renderStats(sf::RenderWindow& window, sf::Font& font, float x, float
y) const;

private:
	int distanceTravelled;

	Inventory inventory;
	Weapon weapon;
	Armor armor;

	Weapon* equippedWeapon;
	Armor* equippedArmor;

	string name;
	int level;
	int exp;
	int gold;
	int expNext;

	int strength;
	int vitality;
	int dexterity;
	int intelligence;

	int hp;
	int hpMax;
	int stamina;
```

```
        int staminaMax;
        int damageMin;
        int damageMax;
        int defence;
        int accuracy;
        int luck;

        int statPoints;

        int actionPoints;
};
```

```
//Character.cpp
#include "Character.h"


//Default Constructor Implementation

Character::Character():equippedWeapon(nullptr), equippedArmor(nullptr)

{

        this->distanceTravelled = 0;

        this->gold = 0;


        this->name = "None";

        this->level = 1;

        this->exp = 0;

        this->expNext = 0;


        this->strength = 0;

        this->vitality = 0;

        this->dexterity = 0;

        this->intelligence = 0;


        this->hp = 0;

        this->hpMax = 0;

        this->stamina = 0;

        this->staminaMax = 0;

        this->damageMin = 0;
```

```
	this->damageMax = 0;
	this->defence = 0;
	this->accuracy = 0;
	this->luck = 0;

	this->statPoints = 0;

	this->actionPoints = this->level*2;
}

//Constructor - for loading saved characters
Character::Character(string name,
	int distanceTravelled,
	int gold, int level,
	int exp, int strength,
	int vitality, int dexterity,
	int intelligence, int hp,
	int stamina, int statPoints)
{
	this->distanceTravelled = distanceTravelled;
	this->gold = gold;

	this->name = name;
	this->level = level;
	this->exp = exp;
	this->expNext = 0;

	this->strength = strength;
	this->vitality = vitality;
	this->dexterity = dexterity;
	this->intelligence = intelligence;
```

```
	this->hp = hp;
	this->hpMax = 0;
	this->stamina = stamina;
	this->staminaMax = 0;
	this->damageMin = 0;
	this->damageMax = 0;
	this->defence = 0;
	this->accuracy = 0;
	this->luck = 0;

	this->statPoints = statPoints;

	this->actionPoints = this->level * 2;


	this->updateStats();
}


Character::~Character()
{

}

//Constructor - for creating new character
void Character::initialize(const string name)
{
	this->distanceTravelled = 0;
	this->gold = 100;
```

```
	this->name = name;
	this->level = 1;
	this->exp = 0;
	this->expNext =
		static_cast<int>((static_cast<double>(50) / 3) * ((pow(level, 3) - 6 * pow(level, 2)) +
17 * level - 12)) + 100;

	this->strength = 5;
	this->vitality = 5;
	this->dexterity = 5;
	this->intelligence = 5;

	this->hpMax = (this->vitality * 2) + (this->strength / 2) + (this->level*10);
	this->hp = this->hpMax;

	this->staminaMax = this->vitality + (this->strength / 2) + (this->dexterity / 3);
	this->stamina = this->staminaMax;

	this->damageMin = this->strength;
	this->damageMax = this->strength + 3;
	this->defence = this->dexterity + (this->intelligence / 2);
	this->accuracy = ((this->dexterity) / 2) + intelligence;
	this->luck = this->intelligence;

	this->statPoints = 0;
	this->actionPoints = level * 2;
	this->updateStats();
}

//Print all Character Attributes
void Character::printStats() const {
```

```
cout << "---- Character Sheet ----" << endl;
cout << " Name: " << this->name << endl;
cout << " Level: " << this->level << endl;
cout << " Experience: " << this->exp << endl;
cout << " Experience to Next Level: " << this->expNext << endl;
cout << " Available Stat Points: " << this->statPoints << endl;
cout << "------------------------" << endl;
cout << " Strength: " << this->strength << endl;
cout << " Vitality: " << this->vitality << endl;
cout << " Dexterity: " << this->dexterity << endl;
cout << " Intelligence: " << this->intelligence << endl;
cout << "------------------------" << endl;
cout << " HP: " << this->hp << " / " << this->hpMax << endl;
cout << " Stamina: " << this->stamina << " / " << this->staminaMax << endl;
cout << " Damage: " << this->damageMin << " - " << this->damageMax << endl;
cout << " Defence: " << this->defence << endl;
cout << " Accuracy: " << this->accuracy << endl;
cout << " Luck: " << this->luck << endl;
cout << "------------------------" << endl;
cout << " Gold: " << this->gold << endl;
cout << " Distance Travelled: " << this->distanceTravelled << endl;
cout << "------------------------" << endl;

// Show equipped weapon
Weapon* equippedWeapon = getEquippedWeapon();
cout << " Equipped Weapon:" << endl;

if (equippedWeapon) {

        cout << " Name: " << equippedWeapon->getName() << endl;
```

```
		cout << " Damage: " << equippedWeapon->getDamageMin() << " - " << equippedWeapon->getDamageMax() << " \n" << endl;

	}

	else {
		cout << " None" << endl;
	}

	// Show equipped armor
	cout << " Equipped Armor: ";
	if (equippedArmor) {
		cout << equippedArmor->getName() << endl;
		cout << " (Defence: " << equippedArmor->getDefence() << ")\n" << endl;
	}

	else {
		cout << "None\n" << endl;
	}
}

//Update Stats when level up
void Character::updateStats()
{
	this->expNext =
		static_cast<int>((static_cast<double>(50) / 3) * ((pow(level, 3) - 6 * pow(level, 2)) + 17 * level - 12)) + 100;

	this->hpMax = (this->vitality * 2) + (this->strength / 2) + (this->level * 10);
	this->hp = this->hpMax;

	this->staminaMax = this->vitality + (this->strength / 2) + (this->dexterity / 3);
```

```
	this->damageMin = this->strength;
	this->damageMax = this->strength + 3;
	this->defence = this->dexterity + (this->intelligence / 2);
	this->accuracy = (this->dexterity) * 2;
	this->luck = this->intelligence;
}

//Upgrade Stats
void Character::addToStat(int stat, int value)
{
	if (this->statPoints < value)
	{
		cout << "Error! Not Enough Statpoints! " << "\n";
	}

	else
	{
		switch (stat)
		{
		case 0:
			this->strength += value;
			cout << " Strength Increased! " << "\n";
			break;

		case 1:
			this->vitality += value;
			cout << " Vitality Increased! " << "\n";
			break;

		case 2:
```

```
                this->dexterity += value;
                cout << " Dexterity Increased! " << "\n";
                break;

        case 3:
                this->intelligence += value;
                cout << " Intelligence Increased! " << "\n";
                break;

        default:
                cout << " Invalid input. " << "\n";
                break;
        }
        this->statPoints -= value;

        this->updateStats();
    }
}

void Character::levelUp()
{
    if (this->exp >= this->expNext)
    {
        this->exp -= this->expNext;
        this->level++;
        this->expNext =
            static_cast<int>((static_cast<double>(50) / 3) * ((pow(level, 3) - 6 *
pow(level, 2)) + 17 * level - 12)) + 100;

        this->statPoints++;
```

```
		this->updateStats();



		cout << " You are now Level: " << this->level << " ! " << "\n\n";
	}


	else
	{
		cout << " NOT ENOUGH EXP! \n\n";
	}
}


string Character::getAsString() const
{
	return name + " "
		+ to_string(distanceTravelled) + " "
		+ to_string(gold) + " "
		+ to_string(level) + " "
		+ to_string(exp) + " "
		+ to_string(strength) + " "
		+ to_string(vitality) + " "
		+ to_string(dexterity) + " "
		+ to_string(intelligence) + " "
		+ to_string(hp) + " "
		+ to_string(stamina) + " "
		+ to_string(statPoints);


}
```

```
string Character::getInventoryAsString() const {
	std::stringstream ss;
	for (int i = 0; i < inventory.size(); ++i) {
		Item* item = inventory.getItem(i);
		if (item != nullptr) {
			ss << "\n Item Number: " << i + 1 << "\n" // Adding the item number
				<< item->toString() << "\n";
		}
	}
	return ss.str();
}

Item* Character::getInventoryItem(int index) const
{
	if (index >= 0 && index < this->inventory.size())
	{
		return this->inventory.getItem(index);
	}

	return nullptr;
}

int Character::getInventorySize() const
{
	return this->inventory.size();
}

void Character::takeDamage(int damage)
{
	this->hp -= damage;
```

```
	if (this->hp <= 0)
	{
		this->hp = 0;
	}


}

void Character::usePotion(int index)
{
	if (index < 0 || index >= this->inventory.size())
	{
		cout << "Invalid index. No potion at this index." << endl;
		return;
	}

	Potion* potion = dynamic_cast<Potion*>(this->inventory.getItem(index));
	if (potion)
	{
		applyPotionEffects(*potion);
		inventory.removeItem(index);
		cout << "Potion used successfully." << endl;
	}
	else
	{
		cout << "Invalid selection. No potion at this index." << endl;
	}
}


void Character::applyPotionEffects(const Potion& potion)
```

```
{

	switch (potion.getType())
	{
	case Potion::PotionType::Health:
		this->hp += potion.getPotency();
		if (this->hp > this->hpMax)
			this->hp = this->hpMax;
		break;

	case Potion::PotionType::Strength:
		this->strength += potion.getPotency();
		break;

	case Potion::PotionType::Vitality:
		this->vitality += potion.getPotency();
		break;

	case Potion::PotionType::Dexterity:
		this->dexterity += potion.getPotency();
		break;

	case Potion::PotionType::Intelligence:
		this->intelligence += potion.getPotency();
		break;
	}
}

void Character::equipWeapon(int index) {
	if (index < 0 || index >= this->inventory.size()) {
		cout << "Invalid index." << endl;
```

```
            return;
        }

        Weapon* weaponToEquip = dynamic_cast<Weapon*>(this->inventory.getItem(index));
        if (!weaponToEquip) {
            cout << "No weapon at this index." << endl;
            return;
        }

        Weapon* clonedWeapon = weaponToEquip->clone();
        if (!clonedWeapon) {
            cout << "Failed to clone weapon." << endl;
            return;
        }

        if (this->equippedWeapon) {
            this->inventory.addItem(*this->equippedWeapon);
            delete this->equippedWeapon;
        }

        this->equippedWeapon = clonedWeapon;
        this->damageMin = clonedWeapon->getDamageMin();
        this->damageMax = clonedWeapon->getDamageMax();

        this->inventory.removeItem(index);

        cout << "Weapon equipped: " << clonedWeapon->getName() << endl;
}

void Character::equipArmor(int index) {
        if (index < 0 || index >= this->inventory.size()) {
```

```
            cout << "Invalid index." << endl;
            return;
        }

        Armor* armorToEquip = dynamic_cast<Armor*>(this->inventory.getItem(index));
        if (armorToEquip) {
            // Clone the armor to equip
            Armor* clonedArmor = dynamic_cast<Armor*>(armorToEquip->clone());

            // If there's already an equipped armor, add it back to the inventory
            if (this->equippedArmor) {
                this->inventory.addItem(*this->equippedArmor);
                delete this->equippedArmor; // Delete the old equipped armor
            }

            // Equip the cloned armor and update the character's defense
            this->equippedArmor = clonedArmor;
            this->defence = clonedArmor->getDefence();

            // Remove the original armor from the inventory
            this->inventory.removeItem(index);

            cout << "Armor equipped." << endl;
        }
        else {
            cout << "No armor at this index." << endl;
        }
}

Weapon* Character::getEquippedWeapon() const
```

```
{
	return this->equippedWeapon;
}

Armor* Character::getEquippedArmor() const
{
	return this->equippedArmor;
}



void Character::useActionPoints(int amount)
{
	this->actionPoints -= amount;
	if (this->actionPoints < 0)
	{
		this->actionPoints = 0;
	}
}

int Character::getActionPoints() const
{
	return this->actionPoints;
}

void Character::resetActionPoints()
{
	this->actionPoints = level*2;
}

int Character::getStatValue(const string& statName) const {
	if (statName == "Strength") {
```

```
		return this->strength;
	}
	else if (statName == "Vitality") {
		return this->vitality;
	}
	else if (statName == "Dexterity") {
		return this->dexterity;
	}
	else if (statName == "Intelligence") {
		return this->intelligence;
	}
	else {
		cout << "Stat not found: " << statName << endl;
		return 0;
	}
}


void Character::increaseHP(int amount) {
	this->hp += amount;
	if (this->hp > this->hpMax) {
		this->hp = this->hpMax; // Ensure HP does not exceed maximum
	}
}


void Character::increaseStat(const string& statName, int value) {
	if (statName == "Strength") {
		this->strength += value;
	}
	else if (statName == "Vitality") {
```

```
            this->vitality += value;
        }
        else if (statName == "Dexterity") {
            this->dexterity += value;
        }
        else if (statName == "Intelligence") {
            this->intelligence += value;
        }
        else {
            std::cout << "Unknown stat: " << statName << std::endl;
        }
        // Update other dependent attributes if necessary
        this->updateStats();
}


void Character::renderStats(sf::RenderWindow& window, sf::Font& font, float x, float y) const {
        sf::Text text;
        text.setFont(font);
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setPosition(x, y);

        std::stringstream ss;
        ss << "---- Character Sheet ----\n"
            << " Name: " << this->name << "\n"
            << " Level: " << this->level << "\n"
            << " Experience: " << this->exp << "\n"
            << " Experience to Next Level: " << this->expNext << "\n"
            << " Available Stat Points: " << this->statPoints << "\n"
            << "-------------------------" << "\n"
```

```
                << " Strength: " << this->strength << "\n"

                << " Vitality: " << this->vitality << "\n"

                << " Dexterity: " << this->dexterity << "\n"

                << " Intelligence: " << this->intelligence << "\n"

                << "-------------------------" << "\n"

                << " HP: " << this->hp << " / " << this->hpMax << "\n"

                << " Stamina: " << this->stamina << " / " << this->staminaMax << "\n"

                << " Damage: " << this->damageMin << " - " << this->damageMax << "\n"

                << " Defence: " << this->defence << "\n"

                << " Accuracy: " << this->accuracy << "\n"

                << " Luck: " << this->luck << "\n"

                << " Distance Travelled: " << this->distanceTravelled << "\n"

                << "-------------------------" << "\n"

                << " Gold: " << this->gold << "\n"

                << " Equipped Weapon: " << (this->equippedWeapon ? this->equippedWeapon-
>getName() : "None") << "\n"

                << " Equipped Armor: " << (this->equippedArmor ? this->equippedArmor-
>getName() : "None") << "\n";


        text.setString(ss.str());

        window.draw(text);

}


//Decision.h
#pragma once
#include <memory>
#include <string>
#include <iostream>
#include <SFML/Graphics.hpp>

#include "Character.h"
#include "Item.h"

//Forward Declaration
class Event;

class DecisionEffect {
public:
    virtual void applyEffect(Character& character, Item* item) = 0;
    virtual void renderEffectResult(sf::RenderWindow& window, sf::Font& font,
float x, float y) = 0;
```

```
    virtual ~DecisionEffect() {}
};

class FindItemEffect : public DecisionEffect {
    std::string itemName;

public:
    FindItemEffect() {}

    void applyEffect(Character& character, Item* item) override {
        if (item) {
            character.addItem(*item);
            itemName = item->getName();
            std::cout << "You found an item: " << itemName << "!" << std::endl;
        }
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x,
float y) override {
        if (!itemName.empty()) {
            sf::Text text;
            text.setFont(font);
            text.setString("You found an item: " + itemName + "!");
            text.setCharacterSize(24);
            text.setFillColor(sf::Color::White);
            text.setPosition(x, y);
            window.draw(text);
        }
    }
};

class DamageHealthEffect : public DecisionEffect {
    int damageAmount;

public:
    DamageHealthEffect(int damageAmount) : damageAmount(damageAmount) {}

    void applyEffect(Character& character, Item* item) override {
        character.takeDamage(damageAmount);
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x,
float y) override {
        sf::Text text;
        text.setFont(font);
        text.setString("You take " + std::to_string(damageAmount) + " damage!");
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setPosition(x, y);
        window.draw(text);
    }
};

class IncreaseStatEffect : public DecisionEffect {
    std::string stat;
    int amount;

public:
    IncreaseStatEffect(const std::string& stat, int amount) : stat(stat),
amount(amount) {}

    void applyEffect(Character& character, Item* item) override {
        character.increaseStat(stat, amount);
```

```
    }

    void renderEffectResult(sf::RenderWindow& window, sf::Font& font, float x,
float y) override {
        // Render the result of the stat increase effect.
        sf::Text text;
        text.setFont(font);
        text.setString("Your " + stat + " has increased by " +
std::to_string(amount) + "!");
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setPosition(x, y);
        window.draw(text);
    }
};



class Decision {
public:
    std::string text;
    bool requiresCheck;
    int checkThreshold;
    std::string stat;
    std::string successText;
    std::string failText;
    std::unique_ptr<DecisionEffect> successEffect;
    std::unique_ptr<DecisionEffect> failureEffect;
    std::shared_ptr<Item> droppedItem; // Smart pointers for better memory
management

    // Constructor
    Decision(const std::string& text, bool requiresCheck, int checkThreshold,
        const std::string& stat, const std::string& successText,
        const std::string& failText, DecisionEffect* successEffect = nullptr,
        DecisionEffect* failureEffect = nullptr, Item* droppedItem = nullptr)
        : text(text), requiresCheck(requiresCheck),
checkThreshold(checkThreshold),
        stat(stat), successText(successText), failText(failText),
        successEffect(successEffect), failureEffect(failureEffect),
        droppedItem(droppedItem) {}

    // Delete the copy constructor and the copy assignment operator.
    Decision(const Decision&) = delete;
    Decision& operator=(const Decision&) = delete;

    // Render function
    void render(sf::RenderWindow& window, sf::Font& font, float x, float y) const
{
        sf::Text sfmlText;
        sfmlText.setFont(font);
        sfmlText.setString(text);
        sfmlText.setCharacterSize(24);
        sfmlText.setFillColor(sf::Color::White);
        sfmlText.setPosition(x, y);
        window.draw(sfmlText);
    }

    // Move constructor
    Decision(Decision&& other) noexcept
        : text(std::move(other.text)),
        requiresCheck(other.requiresCheck),
```

```
        checkThreshold(other.checkThreshold),
        stat(std::move(other.stat)),
        successText(std::move(other.successText)),
        failText(std::move(other.failText)),
        successEffect(std::move(other.successEffect)),
        failureEffect(std::move(other.failureEffect)),
        droppedItem(std::move(other.droppedItem)) {}

    // Move assignment operator
    Decision& operator=(Decision&& other) noexcept {
        if (this != &other) {
            text = std::move(other.text);
            requiresCheck = other.requiresCheck;
            checkThreshold = other.checkThreshold;
            stat = std::move(other.stat);
            successText = std::move(other.successText);
            failText = std::move(other.failText);
            successEffect = std::move(other.successEffect);
            failureEffect = std::move(other.failureEffect);
            droppedItem = std::move(other.droppedItem);
        }
        return *this;
    }

    // Destructor
    ~Decision() = default;
};

//Decision Factory Design Pattern
class DecisionFactory {
public:
    static std::unique_ptr<Decision> createDecision(
        const std::string& text,
        bool requiresCheck,
        int checkThreshold,
        const std::string& stat,
        const std::string& successText,
        const std::string& failText,
        DecisionEffect* successEffect = nullptr,
        DecisionEffect* failureEffect = nullptr,
        Item* droppedItem = nullptr);
};
```

//Decision.cpp

```
#include "Decision.h"

std::unique_ptr<Decision> DecisionFactory::createDecision(
    const std::string& text,
    bool requiresCheck,
    int checkThreshold,
    const std::string& stat,
    const std::string& successText,
    const std::string& failText,
    DecisionEffect* successEffect,
    DecisionEffect* failureEffect,
    Item* droppedItem) {

    return std::make_unique<Decision>(
        text,
        requiresCheck,
        checkThreshold,
```

```
        stat,
        successText,
        failText,
        successEffect,
        failureEffect,
        droppedItem);
}
```

//DoublyLinkedNode.h

```
#pragma once

template<class DataType>
class DoublyLinkedNode {
public:
    typedef DoublyLinkedNode<DataType> Node;

    private:
    DataType fValue;
    Node* fNext;
    Node* fPrevious;

public:
    // Constructor for node with a value
    DoublyLinkedNode(const DataType& aValue) : fValue(aValue), fNext(nullptr),
fPrevious(nullptr) {}

    // Methods to manipulate the list
    void prepend(Node* aNode);
    void append(Node* aNode);
    void remove();

    // Accessors
    const DataType& getValue() const { return fValue; }
    Node* getNext() const { return fNext; }
    Node* getPrevious() const { return fPrevious; }

    // Check if the node is the end of the list
    bool isEnd() const { return fNext == nullptr; }
};

template<class DataType>
void DoublyLinkedNode<DataType>::prepend(Node* aNode) {
    if (!aNode) return;

    aNode->fNext = this;
    aNode->fPrevious = fPrevious;

    if (fPrevious) {
        fPrevious->fNext = aNode;
    }

    fPrevious = aNode;
};

template<class DataType>
void DoublyLinkedNode<DataType>::append(Node* aNode) {
    if (!aNode) return;

    aNode->fPrevious = this;
    aNode->fNext = fNext;

    if (fNext) {
```

```
        fNext->fPrevious = aNode;
    }

    fNext = aNode;
};

template<class DataType>
void DoublyLinkedNode<DataType>::remove() {
    if (fPrevious) {
        fPrevious->fNext = fNext;
    }

    if (fNext) {
        fNext->fPrevious = fPrevious;
    }

    fNext = nullptr;
    fPrevious = nullptr;
};
```

//DynamicArray.h

```
#pragma once

template <typename T>
class dynaArr
{
private:
	unsigned cap;
	unsigned initialCap;
	unsigned numOfEl;
	T* *arr;

	void expand();
	void initialize(unsigned from);

public:
	dynaArr(unsigned size = 5);
	dynaArr(const dynaArr& obj);
	~dynaArr();

	T& operator [] (const unsigned index);
	void operator= (const dynaArr& obj);

	unsigned max_cap();
	unsigned size();
	void push (const T element);
	void remove(const unsigned index, bool ordered = false);
	void clear();
};

template <typename T>
dynaArr<T>::dynaArr(unsigned size)
{
	this->initialCap = size;
	this->cap = size;
	this->numOfEl = 0;

	this->arr = new T * [this->cap];

	this->initialize(0);
```

```
};

template <typename T>
dynaArr<T>::dynaArr(const dynaArr& obj)
{
	this->initialCap = obj.initialCap;
	this->cap = obj.cap;
	this->numOfEl = obj.numOfEl;

	this->arr = new T * [this->cap];

	for (size_t i = 0; i < this->numOfEl; i++)
	{
		this->arr[i] = new T(*obj.arr[i]);
	}

	this->initialize(0);

};

template <typename T>
dynaArr<T>::~dynaArr()
{
	for (size_t i = 0; i < this->numOfEl; i++)
	{
		delete this->arr[i];
	}

	delete[] arr;
};

template <typename T>
T& dynaArr<T>:: operator [] (const unsigned index)
{
	if (index < 0 || index >= this->numOfEl)
		throw "Out of Indexing Operator Bounds";

	return *this->arr[index];
};

template <typename T>
void dynaArr<T>:: operator = (const dynaArr& obj)
{
	for (size_t i = 0; i < this->numOfEl; i++)
	{
		delete this->arr[i];
	}

	delete[] arr;

	this->initialCap = obj.initialCap;
	this->cap = obj.cap;
	this->numOfEl = obj.numOfEl;

	this->arr = new T * [this->cap];

	for (size_t i = 0; i < this->numOfEl; i++)
	{
		this->arr[i] = new T(*obj.arr[i]);
	}

	this->initialize(0);
```

```
};

template <typename T>
void dynaArr<T>::expand()
{
	this->cap *= 2;
	T** tempArr = new T * [this->cap];

	for (size_t i = 0; i < this->numOfEl; i++)
	{
		tempArr[i] = this->arr[i];
	}
	delete[]arr;

	this->arr = tempArr;
	this->initialize(this->numOfEl);
};


template <typename T>
void dynaArr<T>::initialize(unsigned from)
{
	for (size_t i = from; i < this->cap; i++)
	{
		this->arr[i] = nullptr;
	}
};

template <typename T>
unsigned dynaArr<T>::max_cap()
{
	return this->cap;
};


template <typename T>
unsigned dynaArr<T>::size()
{
	return this->numOfEl;
};

template <typename T>
void dynaArr<T>::push(const T element)
{
	if (this->numOfEl >= this->cap)
		this->expand();

	this->arr[this->numOfEl++] = new T(element);
};

template <typename T>
void dynaArr<T>::remove(const unsigned index, bool ordered)
{
	if (index < 0 || index >= this->numOfEl)
		throw "Out of Bounds Remove";

	if (ordered)
	{
		delete this->arr[index];
		for (size_t i = 0; i < this->numOfEl - 1; i++)
		{
			this->arr[i] = this->arr[i + 1];
		}
```

```
		this->arr[--this->numOfEl] = nullptr;
	}

	else
	{
		delete this->arr[index];

		this->arr[index] = this->arr[this->numOfEl - 1];

		this->arr[--this->numOfEl] = nullptr;
	}
}

template <typename T>
void dynaArr<T>::clear()
{
	for (size_t i = 0; i < this->numOfEl; i++)
	{
		delete this->arr[i];
	}

	this->numOfEl = 0;
};
```

//DynamicStack.h

```
#pragma once

#include "SinglyLinkedNode.h"
#include <stdexcept>
#include <iostream>

using namespace std;

template<class T>
class dynaStack
{
private:
	SinglyLinkedNode<T>* top;

public:
	dynaStack():top(nullptr){}

	~dynaStack()
	{
		clear();
	}

	//Add an element to top of the stack
	void push(const T& value)
	{
		top = new SinglyLinkedNode<T>(value, top);
	}

	//Return the top element from the stack without removing it
	T peek() const
	{
		if (isEmpty())
		{
			throw out_of_range("Stack is empty");
```

```
		}

		return top->getValue();
	}

	//Remove and return the top element from the stack
	T pop()
	{
		if (isEmpty())
		{
			throw out_of_range("Stack is empty");
		}

		T value = top->getValue();
		SinglyLinkedNode<T>* oldTop = top;
		top = top->getNext();
		delete oldTop;
		return value;
	}

	//Check if the stack is empty
	bool isEmpty() const
	{
		return top == nullptr;
	}

	//Clear the stack
	void clear()
	{
		while (!isEmpty())
		{
			pop();
		}
	}

	// Get the top node of the stack for traversal without modification
	SinglyLinkedNode<T>* getTopNode() const {
		return top;
	}



};
```

//Enemy.h

```
#pragma once

#include "STLInclude.h"

//Different Enemy types
enum class EnemyType
{
	DreadKnight = 0,
	ForestWraith,
	RockTroll,
	CrystalDragon,
	MinotaurWarlord,
	FireGolem,
	NumTypes
```

```
};


class Enemy
{
public:
	Enemy(EnemyType type, int level);
	virtual ~Enemy();

	inline bool isAlive() const { return this->hp > 0; }
	string getAsString() const;
	void takeDamage(int damage);
	void allocateActionPoints();

	inline const string& getName() const { return this->name; }
	inline int getLevel() const { return this->level; }
	inline int getDamage() const { return rand() % this->damageMax + this-
>damageMin; }
	inline int getExp() const { return this->level * 100; }
	inline int getHp() const { return this->hp; }
	inline int getHpMax() const { return this->hpMax; }
	inline int getDefence() const { return this->defence; }
	inline int getAccuracy() const { return this->accuracy; }

	void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;

private:
	string name;
	int level;
	int hp;
	int hpMax;
	int damageMin;
	int damageMax;
	int dropChance;
	int defence;
	int accuracy;
	int actionPoints;

	EnemyType type;

};


//Enemy.cpp
#include "Enemy.h"


Enemy::Enemy(EnemyType type, int  level) : type(type), level(level)
{


	switch (type)
	{
	case EnemyType::DreadKnight:
		this->name = "Dread Knight";
		this->hpMax = level * 12;
		this->damageMin = level * 2;
		this->damageMax = level * 4;
		this->defence = level * 2;
		this->accuracy = level * 2;
		break;
```

```
	case EnemyType::ForestWraith:
		this->name = "Forest Wraith";
		this->hpMax = level * 5;
		this->damageMin = level;
		this->damageMax = level * 3;
		this->defence = level;
		this->accuracy = level * 3;
		break;

	case EnemyType::RockTroll:
		this->name = "Rock Troll";
		this->hpMax = level * 15;
		this->damageMin = level * 2;
		this->damageMax = level * 3;
		this->defence = level * 5;
		this->accuracy = level;
		break;

	case EnemyType::CrystalDragon:
		this->name = "Crystal Dragon";
		this->hpMax = level * 20;
		this->damageMin = level * 4;
		this->damageMax = level * 6;
		this->defence = level * 4;
		this->accuracy = level * 2;
		break;

	case EnemyType::MinotaurWarlord:
		this->name = "Minotaur Warlord";
		this->hpMax = level * 18;
		this->damageMin = level * 3;
		this->damageMax = level * 7;
		this->defence = level * 5;
		this->accuracy = level * 2;
		break;

	case EnemyType::FireGolem:
		this->name = "Fire Golem";
		this->hpMax = level * 10;
		this->damageMin = level * 5;
		this->damageMax = level * 7;
		this->defence = level  * 2;
		this->accuracy = level;
		break;
	}


	this->hp = this->hpMax;


}

Enemy::~Enemy()
{

}

string Enemy::getAsString() const
{
	stringstream ss;

	ss << " Name: " << this->name << "\n"
```

```
		<< " Level : " << this->level << "\n"
		<< " HP: " << this->hp << " / " << this->hpMax << "\n"
		<< " Damage: " << this->damageMin << " - " << this->damageMax <<
"\n"
		<< " Defence: " << this->defence << "\n"
		<< " Accuracy: " << this->accuracy << "\n";


	return ss.str();



}

void Enemy::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
	sf::Text text;
	text.setFont(font);
	text.setCharacterSize(24);
	text.setFillColor(sf::Color::White);
	text.setPosition(x, y);

	std::stringstream ss;
	ss << "Name: " << this->name << "\n"
		<< "Level: " << this->level << "\n"
		<< "HP: " << this->hp << " / " << this->hpMax << "\n"
		<< "Damage: " << this->damageMin << " - " << this->damageMax << "\n"
		<< "Defence: " << this->defence << "\n"
		<< "Accuracy: " << this->accuracy;

	text.setString(ss.str());
	window.draw(text);

	// Adjust y-position for each line of text.
	float yOffset = 0.0f;
	std::string line;
	while (std::getline(ss, line)) {
		text.setPosition(x, y + yOffset);
		text.setString(line);
		window.draw(text);
		yOffset += text.getLocalBounds().height + 5; // Adjust line spacing.
	}
}

void Enemy::takeDamage(int damage)
{
	this->hp -= damage;

	if (this->hp <= 0)
	{
		this->hp = 0;
	}


}

void Enemy::allocateActionPoints()
{
	this->actionPoints = (rand() % 3) + this->level / 2;
}
```

```
//Event,h
#include "Enemy.h"


Enemy::Enemy(EnemyType type, int  level) : type(type), level(level)
{


	switch (type)
	{
	case EnemyType::DreadKnight:
		this->name = "Dread Knight";
		this->hpMax = level * 12;
		this->damageMin = level * 2;
		this->damageMax = level * 4;
		this->defence = level * 2;
		this->accuracy = level * 2;
		break;

	case EnemyType::ForestWraith:
		this->name = "Forest Wraith";
		this->hpMax = level * 5;
		this->damageMin = level;
		this->damageMax = level * 3;
		this->defence = level;
		this->accuracy = level * 3;
		break;

	case EnemyType::RockTroll:
		this->name = "Rock Troll";
		this->hpMax = level * 15;
		this->damageMin = level * 2;
		this->damageMax = level * 3;
		this->defence = level * 5;
		this->accuracy = level;
		break;

	case EnemyType::CrystalDragon:
		this->name = "Crystal Dragon";
		this->hpMax = level * 20;
		this->damageMin = level * 4;
		this->damageMax = level * 6;
		this->defence = level * 4;
		this->accuracy = level * 2;
		break;

	case EnemyType::MinotaurWarlord:
		this->name = "Minotaur Warlord";
		this->hpMax = level * 18;
		this->damageMin = level * 3;
		this->damageMax = level * 7;
		this->defence = level * 5;
		this->accuracy = level * 2;
		break;

	case EnemyType::FireGolem:
		this->name = "Fire Golem";
		this->hpMax = level * 10;
		this->damageMin = level * 5;
		this->damageMax = level * 7;
		this->defence = level  * 2;
		this->accuracy = level;
```

```
			break;
		}


		this->hp = this->hpMax;


}

Enemy::~Enemy()
{

}

string Enemy::getAsString() const
{
		stringstream ss;

		ss << " Name: " << this->name << "\n"
			<< " Level : " << this->level << "\n"
			<< " HP: " << this->hp << " / " << this->hpMax << "\n"
			<< " Damage: " << this->damageMin << " - " << this->damageMax <<
"\n"
			<< " Defence: " << this->defence << "\n"
			<< " Accuracy: " << this->accuracy << "\n";


		return ss.str();



}

void Enemy::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
		sf::Text text;
		text.setFont(font);
		text.setCharacterSize(24);
		text.setFillColor(sf::Color::White);
		text.setPosition(x, y);

		std::stringstream ss;
		ss << "Name: " << this->name << "\n"
			<< "Level: " << this->level << "\n"
			<< "HP: " << this->hp << " / " << this->hpMax << "\n"
			<< "Damage: " << this->damageMin << " - " << this->damageMax << "\n"
			<< "Defence: " << this->defence << "\n"
			<< "Accuracy: " << this->accuracy;

		text.setString(ss.str());
		window.draw(text);

		// Adjust y-position for each line of text.
		float yOffset = 0.0f;
		std::string line;
		while (std::getline(ss, line)) {
			text.setPosition(x, y + yOffset);
			text.setString(line);
			window.draw(text);
			yOffset += text.getLocalBounds().height + 5; // Adjust line spacing.
		}
}
```

```
void Enemy::takeDamage(int damage)
{
	this->hp -= damage;

	if (this->hp <= 0)
	{
		this->hp = 0;
	}


}

void Enemy::allocateActionPoints()
{
	this->actionPoints = (rand() % 3) + this->level / 2;
}
```

```
//Event.cpp
#include "Event.h"


//Constructor and Desctructor

Event::Event() {}

Event::~Event() {}


//Display text in SFML window

void Event::displayText(sf::RenderWindow& window, sf::Font& font, const std::string& message,
bool waitForKeyPress) {

	sf::Text text;

	text.setFont(font);

	text.setString(message);

	text.setCharacterSize(30);

	text.setFillColor(sf::Color::White);

	text.setPosition(50, 200);


	window.clear();

	window.draw(text);

	window.display();


	if (waitForKeyPress) {

		sf::Event event;
```

```
            while (window.waitEvent(event)) {
                if (event.type == sf::Event::KeyPressed) {
                    break;
                }
            }
        }
}

void Event::enemyEncounter(sf::RenderWindow& window, sf::Font& font, Character& character, SinglyLinkedNode<Enemy>*& enemies)
{
        // Initialize control variables
        bool playerTurn = true;
        bool enemyTurn = false;
        bool escape = false;
        bool playerDefeated = false;
        bool enemiesDefeated = false;

        // Generate enemies
        int numOfEnemies = rand() % 4 + 1;
        EnemyType firstEnemyType = static_cast<EnemyType>(rand() % static_cast<int>(EnemyType::NumTypes));
        SinglyLinkedNode<Enemy>* enemyListHead = new SinglyLinkedNode<Enemy>(Enemy(firstEnemyType, character.getLevel() + rand() % 2), nullptr);
        SinglyLinkedNode<Enemy>* lastNode = enemyListHead;
        for (int i = 1; i < numOfEnemies; i++) {
            EnemyType enemyType = static_cast<EnemyType>(rand() % static_cast<int>(EnemyType::NumTypes));
            SinglyLinkedNode<Enemy>* newNode = new SinglyLinkedNode<Enemy>(Enemy(enemyType, character.getLevel() + rand() % 2), nullptr);
            lastNode->append(newNode);
            lastNode = newNode;
        }
```

```
	// Game loop
	while (!escape && !playerDefeated && !enemiesDefeated) {
		std::stringstream ss;

		// Handle turns
		if (playerTurn && !playerDefeated) {
			handlePlayerTurn(ss, window, font, character, enemyListHead, escape);
		}
		else if (enemyTurn && !playerDefeated) {
			handleEnemyTurn(ss, window, font, character, enemyListHead,
playerDefeated);
		}

		// Check end conditions
		playerDefeated = !character.isAlive();
		enemiesDefeated = (countNodes(enemyListHead) <= 0);

		// Clean up if the encounter is over
		if (escape || playerDefeated || enemiesDefeated) {
			while (enemyListHead != nullptr) {
				SinglyLinkedNode<Enemy>* nextNode = enemyListHead->getNext();
				delete enemyListHead;
				enemyListHead = nextNode;
			}
			enemyListHead = nullptr;
		}
	}
}
```

```
//Player Turn Handling
void Event::handlePlayerTurn(std::stringstream& ss, sf::RenderWindow& window, sf::Font& font,
Character& character, SinglyLinkedNode<Enemy>*& enemyListHead, bool& escape) {
    // Display player's turn options
    ss.str(""); // Clear the stringstream for new input for each turn

    ss << "\n ---------- CHARACTER TURN ---------- \n";
    ss << "YOU'VE ENCOUNTERED " << countNodes(enemyListHead) << " ENEMY/ENEMIES!\n";

    // Display current enemies
    displayEnemies(ss, enemyListHead);

    // Display character stats and queued actions
    displayCharacterStats(ss, character);

    // Display Queued Actions
    displayQueuedActions(ss, character);

    //Display Combat Log
    displayCombatLog(ss, character);

    // Display battle menu
    displayBattleMenu(ss);

    //Update SFML window
    updateWindow(window, font, ss);

    // Handle player input and actions
    handlePlayerInput(window, font, ss, character, enemyListHead, escape);
```

```
}

//Display Enemies
void Event::displayEnemies(std::stringstream& ss, SinglyLinkedNode<Enemy>*& enemyListHead) {



	SinglyLinkedNode<Enemy>* currentNode = enemyListHead;
	int index = 1;
	while (currentNode != nullptr) {
		Enemy& enemy = currentNode->getValue();
		if (enemy.isAlive()) {
			ss << " " << index << ": " << enemy.getName() << " Lvl: " << enemy.getLevel()
				<< " - (HP: [" << enemy.getHp() << "/" << enemy.getHpMax() << "] |
DEF: "
				<< enemy.getDefence() << " | ACC: " << enemy.getAccuracy() <<
")\n";
		}
		currentNode = currentNode->getNext();
		index++;
	}
}

void Event::displayQueuedActions(std::stringstream& ss, Character& character) {
	Queue<Action> tempQueue = this->actionQueue;


	ss << "\n ------------------------- Queued Actions ------------------------- \n";
	while (!tempQueue.isEmpty()) {
		Action currentAction = tempQueue.dequeue();
		ss << actionToString(currentAction, character) << "\n";
	}
}
```

```
void Event::displayCombatLog(std::stringstream& ss, Character& character) {

	ss << "\n ------------------------- Combat Log ------------------------- \n";
	if (combatLog.isEmpty()) {
		ss << "\nNo actions in the log.\n";

	}

	// Temporary stack to hold actions while displaying them
	dynaStack<Action> tempStack;

	// Display and transfer actions to the temporary stack
	while (!combatLog.isEmpty()) {
		Action action = combatLog.pop();
		ss << action.message << endl;  // Concatenate the custom message
		tempStack.push(action);
	}

	// Transfer actions back to the original stack
	while (!tempStack.isEmpty()) {
		combatLog.push(tempStack.pop());
	}
}

//Character stats display
void Event::displayCharacterStats(std::stringstream& ss, Character& character) {
	ss << "\n -------------------------- Quick Stats ------------------------ \n";
	ss << "Current Character HP: " << character.getHp() << " / " << character.getHpMax() << "\n";

	// Equipped weapon and armor
	Weapon* equippedWeapon = character.getEquippedWeapon();
```

```
	Armor* equippedArmor = character.getEquippedArmor();
	ss << "Equipped Weapon: " << (equippedWeapon ? equippedWeapon->getName() : "None") << "\n";
	ss << "Equipped Armor: " << (equippedArmor ? equippedArmor->getName() : "None") << "\n";

	// Action Points
	ss << "Action Points: " << character.getActionPoints() << "\n";
}

//Battle Menu Display
void Event::displayBattleMenu(std::stringstream& ss) {
	ss << "\n ------------------------ Battle Menu ------------------------ \n";
	ss << "1: Escape\n";
	ss << "2: Attack\n";
	ss << "3: Use Item\n";
	ss << "4: End Turn\n";
}

//Player Input Handling
void Event::handlePlayerInput(sf::RenderWindow& window, sf::Font& font, std::stringstream& ss, Character& character, SinglyLinkedNode<Enemy>*& enemyListHead, bool& escape) {
	int choice = 0;
	sf::Event event;
	PlayerAction nextAction = PlayerAction::None;

	//displayQueuedActions(window, font, character);
	updateWindow(window, font, ss);

	while (window.pollEvent(event)) {
		if (event.type == sf::Event::Closed) {
			window.close();
```

```
		}

		if (event.type == sf::Event::KeyPressed) {
			switch (event.key.code) {
			case sf::Keyboard::Num1:
				escape = true;
				return;
			case sf::Keyboard::Num2:
				nextAction = PlayerAction::Attack;
				break;
			case sf::Keyboard::Num3:
				nextAction = PlayerAction::UseItem;
				break;
			case sf::Keyboard::Num4:
				nextAction = PlayerAction::EndTurn;
				break;
			default:
				break; // No valid key was pressed
			}
		}
	}

	// Perform the action after polling
	switch (nextAction) {
	case PlayerAction::Attack:
		handleAttackAction(window, font, ss, character, enemyListHead);
		break;
	case PlayerAction::UseItem:
		handleUseItemAction(window, font, ss, character);
		break;
	case PlayerAction::EndTurn:
```

```
            handleEndTurnAction(window, font, ss, character, enemyListHead, escape);
            break;
        default:
            break;
        }
}

//Attack Action Handling
void Event::handleAttackAction(sf::RenderWindow& window, sf::Font& font, std::stringstream& ss, Character& character, SinglyLinkedNode<Enemy>*& enemyListHead) {
        if (character.getActionPoints() < 2) {
            ss << "Not enough action points to attack.\n";
            updateWindow(window, font, ss);
            return; // Not enough action points
        }

        // Display enemies to attack
        ss << "\n Choose an enemy (Press ENTER after selecting enemy number): ";

        updateWindow(window, font, ss);

        // Get user choice
        int choice = getPlayerChoice(window);
        cout << choice;

        // Validate choice
        if (choice < 1 || choice > countNodes(enemyListHead)) {
            ss << "Invalid enemy selection.\n";
            updateWindow(window, font, ss);
            return;
        }
```

```
        // Get targeted enemy
        Enemy* targetedEnemy = enemyListHead->get(choice - 1);

        if (!targetedEnemy || !targetedEnemy->isAlive()) {
                ss << "Targeted enemy is invalid or already defeated.\n";
                updateWindow(window, font, ss);
                return;
        }

        // Queue attack action
        queueAttackAction(character, targetedEnemy);
        ss << "Attack action queued against " << targetedEnemy->getName() << ".\n";

        updateWindow(window, font, ss);
}




void Event::queueAttackAction(Character& character, Enemy* targetedEnemy) {
        if (!targetedEnemy || !targetedEnemy->isAlive()) {
                // Handle the case where the targeted enemy is invalid or already defeated
                std::cout << "Cannot attack. Target enemy is invalid or already defeated." <<
std::endl;
                return;
        }

        // Create an attack action targeting the selected enemy
        Action attackAction(ActionType::ATTACK, &character, targetedEnemy, -1, "Attack action
queued");

        // Add the attack action to the queue
```

```
        this->actionQueue.enqueue(attackAction);

        // Deduct action points required for the attack
        character.useActionPoints(2);
}

void Event::handleUseItemAction(sf::RenderWindow& window, sf::Font& font, std::stringstream&
ss, Character& character) {
        ss.str(""); // Clear the stringstream
        ss << "Select an item to use:\n";
        ss << character.getInventoryAsString();
        updateWindow(window, font, ss);

        // Get user choice for the item
        int choice = getPlayerChoice(window);

        // Validate the choice
        if (choice < 1 || choice > character.getInventorySize()) {
                ss << "Invalid item selection. Please try again.\n";
                updateWindow(window, font, ss);
                return;
        }

        // Get the selected item from the character's inventory
        Item* selectedItem = character.getInventoryItem(choice - 1);
        if (!selectedItem) {
                ss << "Invalid item selection.\n";
                updateWindow(window, font, ss);
                return;
        }
```

```
        // Apply the item's effect
        if (auto potion = dynamic_cast<Potion*>(selectedItem)) {
                character.usePotion(choice - 1);
                ss << "Used " << potion->getName() << ".\n";
        }
        else if (auto weapon = dynamic_cast<Weapon*>(selectedItem)) {
                character.equipWeapon(choice - 1);
                ss << "Equipped " << weapon->getName() << ".\n";
        }
        else if (auto armor = dynamic_cast<Armor*>(selectedItem)) {
                character.equipArmor(choice - 1);
                ss << "Equipped " << armor->getName() << ".\n";
        }
        else {
                ss << "Item not usable.\n";
        }

        updateWindow(window, font, ss);
}

void Event::updateWindow(sf::RenderWindow& window, sf::Font& font, std::stringstream& ss) {
        sf::Text text;
        text.setFont(font);
        text.setCharacterSize(24);
        text.setFillColor(sf::Color::White);
        text.setString(ss.str());
        text.setPosition(200, 50);

        window.clear();
        window.draw(text);
        window.display();
```

```
}

int Event::getPlayerChoice(sf::RenderWindow& window) {
	std::string inputStr;
	sf::Event event;

	// Use a loop to process events
	while (window.isOpen()) {
		while (window.pollEvent(event)) {
			if (event.type == sf::Event::Closed) {
				window.close();
			}

			// Check for number key presses
			if (event.type == sf::Event::TextEntered) {
				if (event.text.unicode >= '0' && event.text.unicode <= '9') {
					// Append the character to the input string
					inputStr += static_cast<char>(event.text.unicode);
				}
			}

			// Check for Enter key press to finalize the choice
			if (event.type == sf::Event::KeyPressed && event.key.code ==
sf::Keyboard::Enter) {
				// Convert the string to an integer and return
				return !inputStr.empty() ? std::stoi(inputStr) : 0; // Return 0 if
inputStr is empty
			}
		}
	}

	return 0; // Return 0 if window is closed
```

```
}

void Event::handleEndTurnAction(sf::RenderWindow& window, sf::Font& font, std::stringstream&
ss, Character& character, SinglyLinkedNode<Enemy>*& enemyListHead, bool& escape) {
	// Process all queued actions
	processActionQueue(window, font, character);

	// Reset the character's action points for the next turn
	character.resetActionPoints();
	ss.str(""); // Clear the stringstream

	// Check if the encounter is over (either all enemies are defeated or the player has escaped)
	if (checkEndConditions(character, enemyListHead, escape)) {
		return; // Exit the encounter if it's over
	}

	// Handle the enemy turn
	bool playerDefeated = false;
	handleEnemyTurn(ss, window, font, character, enemyListHead, playerDefeated);
	if (playerDefeated) {
		ss.str("");
		ss << "You have been defeated in battle!";
		updateWindow(window, font, ss);
		// Handle player defeat
	}

	// Prepare for the next player turn
	prepareNextPlayerTurn(ss, window, font, character, enemyListHead);
}

bool Event::checkEndConditions(Character& character, SinglyLinkedNode<Enemy>*&
enemyListHead, bool& escape) {
```

```
	// Check if all enemies are defeated
	bool allEnemiesDefeated = true;
	SinglyLinkedNode<Enemy>* currentNode = enemyListHead;
	while (currentNode != nullptr) {
		if (currentNode->getValue().isAlive()) {
			allEnemiesDefeated = false;
			break;
		}
		currentNode = currentNode->getNext();
	}

	// Check if the player has escaped or is defeated
	if (escape || !character.isAlive() || allEnemiesDefeated) {
		return true; // Encounter is over
	}

	return false; // Encounter is not over
}

void Event::displayBattleState(std::stringstream& ss, Character& character,
SinglyLinkedNode<Enemy>*& enemyListHead) {
	// Display character stats
	displayCharacterStats(ss, character);

	// Display current enemies
	ss << "\n------ Enemies ------\n";
	displayEnemies(ss, enemyListHead);
}

void Event::prepareNextPlayerTurn(std::stringstream& ss, sf::RenderWindow& window, sf::Font&
font, Character& character, SinglyLinkedNode<Enemy>*& enemyListHead) {
	// Prepare the state for the next player turn
```

```
        character.resetActionPoints(); // Reset action points
        ss.str(""); // Clear the stringstream
        displayBattleState(ss, character, enemyListHead); // Update the battle state display
        //updateWindow(window, font, ss);
}



void Event::handleEnemyTurn(std::stringstream& ss, sf::RenderWindow& window, sf::Font& font,
Character& character, SinglyLinkedNode<Enemy>*& enemyListHead, bool& playerDefeated) {
        ss.str(""); // Clear the stringstream
        ss << "\n---------- ENEMY TURN ----------\n";
        int enemyIndex = 1;

        // Iterate through each enemy in the list
        for (SinglyLinkedNode<Enemy>* currentEnemyNode = enemyListHead; currentEnemyNode
!= nullptr; currentEnemyNode = currentEnemyNode->getNext(), ++enemyIndex) {
                Enemy* enemy = &currentEnemyNode->getValue();

                if (!enemy->isAlive()) {
                        continue; // Skip if the enemy is already defeated
                }

                ss << "Enemy " << enemyIndex << ": " << enemy->getName() << "\n";
                performEnemyAttack(ss, character, *enemy);

                if (!character.isAlive()) {
                        ss << "CHARACTER DEFEATED!\n";
                        playerDefeated = true;
                        displayText(window, font, ss.str(), false); // Display the defeat message
                        return; // Exit the function if the player is defeated
                }
```

```
        }

        ss << "--------- END ENEMY TURN -----------\n";

        displayText(window, font, ss.str(), true); // Display the summary of the enemy turn

}

void Event::performEnemyAttack(std::stringstream& ss, Character& character, Enemy& enemy) {
        // Combat calculations
        int combatTotal = enemy.getAccuracy() + character.getDefence();
        int enemyTotal = enemy.getAccuracy() / static_cast<double>(combatTotal) * 100;
        int playerTotal = character.getDefence() / static_cast<double>(combatTotal) * 100;

        int combatRollPlayer = rand() % playerTotal + 1;
        int combatRollEnemy = rand() % enemyTotal + 1;

        ss << "Player Roll: " << combatRollPlayer << " vs. Enemy Roll: " << combatRollEnemy << "\n";

        // Determine if the attack hits or misses
        if (combatRollPlayer < combatRollEnemy) {
                int damage = enemy.getDamage();
                character.takeDamage(damage);
                ss << enemy.getName() << " hits for " << damage << " damage!\n\n";
                logAttack(enemy, character, damage); // Log the attack in the combat log
        }
        else {
                ss << enemy.getName() << "'s attack missed!\n\n";
                logMiss(enemy, character); // Log the miss in the combat log
        }
}
```

```
void Event::waitForKeyPress(sf::RenderWindow& window) {
	sf::Event event;
	while (window.waitEvent(event)) {
		if (event.type == sf::Event::KeyPressed) {
			break;
		}
	}
}

void Event::logAttack(Enemy& enemy, Character& character, int damage) {
	std::stringstream message;
	message << enemy.getName() << " ATTACKS character - Damage: " << damage;
	combatLog.push(Action(ActionType::ENEMY_ATTACK, &character, &enemy, damage,
message.str()));
}

void Event::logMiss(Enemy& enemy, Character& character) {
	std::stringstream message;
	message << enemy.getName() << " ATTACKS character - Missed";
	combatLog.push(Action(ActionType::ENEMY_ATTACK, &character, &enemy, 0,
message.str()));
}

void Event::executeAttackAction(sf::RenderWindow& window, sf::Font& font, Character& character,
Enemy* targetedEnemy) {
	if (!targetedEnemy || !targetedEnemy->isAlive()) {
```

```
            displayText(window, font, "Cannot attack. Target enemy is invalid or already defeated.", true);
            return;
        }


        // Calculate the base hit chance as a percentage of the character's accuracy
        int baseHitChance = character.getAccuracy() * 70 / 100; // 70% of accuracy as base hit chance


        // Subtract a portion of the enemy's defense (50% of defense)
        int defenseFactor = targetedEnemy->getDefence() * 50 / 100;


        // Calculate final hit chance
        int hitChance = max(0, baseHitChance - defenseFactor); // Ensure it doesn't go below 0
        hitChance += rand() % 100;


        // Roll for hit
        int hitRoll = rand() % 100 + 1;


        std::stringstream ss;


        ss << "Hit Roll: " << hitRoll << "\nHit Chance: " << hitChance;


        // Check if the attack hits
        if (hitRoll < hitChance) {
            int damage = character.getDamage();
            targetedEnemy->takeDamage(damage);


            ss << "\n---------- Player Turn ----------\n";
            ss << "ATTACK HIT! \n";
```

```
                ss << "\nCharacter ATTACKS " << targetedEnemy->getName() << " - Damage: " <<
damage;
                ss << "\n---------- End Player Turn ----------\n";
                displayText(window, font, ss.str(), true);

                ss.str(""); // clears ss
                ss << "\nCharacter ATTACKS " << targetedEnemy->getName() << " - Damage: " <<
damage;

                combatLog.push(Action(ActionType::ATTACK, &character, targetedEnemy, damage,
ss.str()));

                if (!targetedEnemy->isAlive()) {
                        handleEnemyDefeat(window, font, character, targetedEnemy);
                }
        }

        else {
                // Log message for a missed attack
                ss << "\n---------- Player Turn ----------\n";
                ss << "ATTACK  MISS! \n";
                ss << "Character ATTACKS " << targetedEnemy->getName() << " - Missed";
                ss << "\n---------- End Player Turn ----------\n";
                displayText(window, font, ss.str(), true);

                ss.str(""); // clears ss
                ss << "Character ATTACKS " << targetedEnemy->getName() << " - Missed";
                combatLog.push(Action(ActionType::ATTACK, &character, targetedEnemy, 0,
ss.str()));
        }
}
```

```
void Event::processActionQueue(sf::RenderWindow& window, sf::Font& font, Character& character)
{
	while (!this->actionQueue.isEmpty()) {
		Action currentAction = this->actionQueue.dequeue();
		if (currentAction.type == ActionType::ATTACK) {
			if (currentAction.targetEnemy && currentAction.targetEnemy->isAlive()) {
				executeAttackAction(window, font, character,
currentAction.targetEnemy);
			}
			else {
				displayText(window, font, "Target enemy is no longer valid.", false);
			}
		}

		// Process window events
		sf::Event event;
		while (window.pollEvent(event)) {
			if (event.type == sf::Event::Closed) {
				window.close();
				return; // Exit if the window is closed
			}
		}

		this_thread::sleep_for(chrono::seconds(1));
	}
	character.resetActionPoints();
}


void Event::handleEnemyDefeat(sf::RenderWindow& window, sf::Font& font, Character& character,
Enemy* defeatedEnemy)
{
```

```
	if (!defeatedEnemy)
	{
		//Safety check to ensure the enemy pointer is valid
		return;
	}

	std::stringstream ss;

	//Gain Experience
	int expGain = defeatedEnemy->getExp();
	character.gainExp(expGain);

	//Gain Gold
	int goldGain = rand() % (defeatedEnemy->getLevel() * 10) + 10; //Random gold dropped based on level
	character.gainGold(goldGain);

	ss << "\n ---------- ENEMY DEFEATED! ---------- \n";
	ss << "Enemy defeated! Gained " << expGain << " EXP and " << goldGain << " gold!\n";

	//Item drop chance
	int dropChance = rand() % 100;
	if (dropChance < 30) //30% chance to drop an item
	{
		//Randomly generate a weapon or armor based on enemy level
		unique_ptr<Item> droppedItem = generateRandomItem(defeatedEnemy->getLevel());

		if (droppedItem)
		{
			character.addItem(*droppedItem);
```

```
                        ss << defeatedEnemy->getName() << " dropped " << droppedItem->getName() << " !\n";
                }
        }

        // Pause for player input
        displayText(window, font, ss.str(), true);
}

unique_ptr<Item> Event::generateRandomItem(int enemyLevel) {
        int itemType = rand() % 3;
        int rarity = determineItemRarity(); // Using the function to determine rarity

        ItemFactory* factory = nullptr;
        unique_ptr<Item> item = nullptr;

        switch (itemType) {
        case 0:
                factory = new WeaponFactory();
                break;
        case 1:
                factory = new ArmorFactory();
                break;
        case 2:
                factory = new PotionFactory();
                break;
        default:
                break;
        }

        if (factory) {
```

```
            item.reset(factory->createItem(enemyLevel, rarity));
            delete factory;
        }

        return item;
}



int Event::determineItemRarity()
{
        int chance = rand() % 100;
        if (chance < 50 && chance >= 0)
        {
            return COMMON;
        }

        else if (chance < 75 && chance > 50)
        {
            return UNCOMMON;
        }

        else if (chance < 90 && chance > 75)
        {
            return RARE;
        }

        else if (chance <= 100 && chance > 90)
        {
            return LEGENDARY;
        }
}
```

```
string Event::actionToString(const Action& action, Character& character) {
	stringstream ss;
	if (action.type == ActionType::ATTACK) {
		string attackerName = (action.attacker == &character) ? "Character" : (action.targetEnemy ? action.targetEnemy->getName() : "Defeated Enemy");
		string targetName = (action.attacker == &character) ? (action.targetEnemy ? action.targetEnemy->getName() : "Defeated Enemy") : "Character";
		string damageStr = action.damage == -1 ? "Pending" : to_string(action.damage);
		ss << attackerName << " ATTACKS " << targetName << " - Damage: " << damageStr;
	}

	return ss.str();
}

void Event::puzzleEncounter(sf::RenderWindow& window, sf::Font& font, Character& character) {
	unique_ptr<BTree<Decision>> currentAdventure;

	int adventureChoice = rand() % 2;

	switch (adventureChoice) {
	case 0:
		currentAdventure = createDungeonAdventure(character);
		break;
	case 1:
		currentAdventure = createTempleAdventure(character);
		break;
	}

	processDecision(window, font, std::move(currentAdventure), character);
}
```

```
void Event::processDecision(sf::RenderWindow& window, sf::Font& font,
unique_ptr<BTree<Decision>>&& currentNode, Character& character) {
	if (!currentNode || currentNode->isEmpty()) {
		displayText(window, font, "End of the path. Exiting Puzzle Adventure.", true);
		return;
	}

	displayText(window, font, currentNode->key().text, true);

	if (currentNode->key().requiresCheck) {
		int characterStatValue = character.getStatValue(currentNode->key().stat);
		std::string checkMsg = "Checking " + currentNode->key().stat + " (" +
std::to_string(characterStatValue) + ").\n Press ENTER to continue....";
		displayText(window, font, checkMsg, true);

		if (characterStatValue >= currentNode->key().checkThreshold) {
			displayText(window, font, currentNode->key().successText, true);

			if (currentNode->key().successEffect) {
				currentNode->key().successEffect->applyEffect(character,
currentNode->key().droppedItem.get());
			}
			processDecision(window, font, std::move(currentNode->left()), character);

		}
		else {
			displayText(window, font, currentNode->key().failText, true);

			if (currentNode->key().failureEffect) {
				currentNode->key().failureEffect->applyEffect(character,
currentNode->key().droppedItem.get());
			}
```

```
                        if (!character.isAlive()) {
                                displayText(window, font, "Your character has been defeated. \n
Press ENTER to continue....", true);
                                return;
                        }
                        processDecision(window, font, std::move(currentNode->right()), character);
                }
        }
        else {
                // Handling user input through SFML
                sf::Event event;
                while (window.waitEvent(event)) {
                        if (event.type == sf::Event::KeyPressed) {
                                if (event.key.code == sf::Keyboard::Num1) {
                                        processDecision(window, font, std::move(currentNode-
>left()), character);
                                        break;
                                }
                                else if (event.key.code == sf::Keyboard::Num2) {
                                        processDecision(window, font, std::move(currentNode-
>right()), character);
                                        break;
                                }
                        }
                }
        }
}

unique_ptr<BTree<Decision>> Event::createDungeonAdventure(Character& character) {
        // Root of the dungeon adventure decision tree
        auto rootDecision = DecisionFactory::createDecision(
```

```
        "As you stand at the entrance of the ominous dungeon, you're faced with a
choice:\n\n To your left, a dimly lit corridor stretches into shadowy unknowns,\n promising silent,
creeping dangers. \n\nTo your right, the echoing sound of water \nhints at hidden depths and
possibly treacherous paths.\n Which way will you choose to venture into the mysteries that lie
ahead?.\n\nLeft: Head to the dimly lit corridor.\nRight: Head to the sound of water. \nChoose a path
: \n1.Left\n2.Right",

        false, 0, "", "", "", nullptr, nullptr, nullptr);


    auto dungeonAdventure = make_unique<BTree<Decision>>(move(*rootDecision));


    // Left subtree: Dimly lit corridor

    auto leftSubtreeDecision = DecisionFactory::createDecision(

        "At the end of the dimly lit corridor, you discover a mysterious chest, \nits ancient
wood and intricate lock whispering of long-forgotten secrets. \nBeside it stands a mystical barrier,
shimmering with an ethereal light, \nguarding secrets perhaps even older than the chest itself.
\nYou pause, weighing your decision. Do you dare open the chest and uncover its \nhidden
treasures, or will you attempt to breach the mystical barrier, \nunveiling the arcane mysteries it
conceals? The choice is yours, adventurer.\n\nLeft: Mysterious chest (Require Strength LVL
5)\nRight: Mystical barrier (Require Intelligence LVL 5)\nChoose a path : \n1.Left\n2.Right",

        false, 0, "", "", "", nullptr, nullptr, nullptr);

    auto leftSubtree = make_unique<BTree<Decision>>(move(*leftSubtreeDecision));


    // Generate an item

    unique_ptr<Item> foundItem = generateRandomItem(character.getLevel());


    // Left-Left: Mysterious chest

    auto chestDecision = DecisionFactory::createDecision(

        "A heavy chest requires strength to open. I hope you are strong enough
adventurer.",

        true, 5, "Strength",

        "You open the chest to find treasure! Check your Inventory!\n Press ENTER to
continue....",

        "The chest won't budge. You sprained an ankle. Took 5 Damage! \n Press ENTER to
continue....",

        new FindItemEffect(),

        new DamageHealthEffect(5),
```

```
            foundItem.release());
        leftSubtree->attachLeft(make_unique<BTree<Decision>>(move(*chestDecision)));

        // Left-Right: Mystical barrier
        auto barrierDecision = DecisionFactory::createDecision(
            "An arcane barrier blocks your path.", true, 5, "Intelligence",
            "You dispel the barrier! You learned a new arcane art that may prove \nuseful in the future. +1 Intelligence \n Press ENTER to continue....", "The magic is too complex. You end up with a headahce. Took 5 Damage!\n Press ENTER to continue....",
            new IncreaseStatEffect("Intelligence", 5), new DamageHealthEffect(5), nullptr);
        leftSubtree->attachRight(make_unique<BTree<Decision>>(move(*barrierDecision)));

        // Attach left subtree to root
        dungeonAdventure->attachLeft(move(leftSubtree));

        // Right subtree: Sound of water
        auto rightSubtreeDecision = DecisionFactory::createDecision(
            "Sound of water.\nLeft: Suspicious water spring.\nRight: Narrow bridge (Require Dexterity Level 5).\nChoose a path : \n1.Left\n2.Right",
            false, 0, "", "", "", nullptr, nullptr, nullptr);
        auto rightSubtree = make_unique<BTree<Decision>>(move(*rightSubtreeDecision));

        // Right-Left: Healing spring
        auto healingSpringDecision = DecisionFactory::createDecision(
            "You found a suspicious spring. It looks quite tempting to drink from it",
            true, 1, "Intelligence",
            "You feel rejuvenated! Vitality Increased. \n Press ENTER to continue....",
            "You feel rejuvenated! Vitality Increased. \n Press ENTER to continue....",
            new IncreaseStatEffect("Vitality", 1), nullptr, nullptr);
        rightSubtree->attachLeft(make_unique<BTree<Decision>>(move(*healingSpringDecision)));

        // Right-Right: Narrow bridge
```

```
	auto narrowBridgeDecision = DecisionFactory::createDecision(
		"A precarious bridge requires dexterity to cross.",
		true, 8, "Dexterity",
		"You skillfully cross the bridge! You were surprised on how you did that. +1 Dexterity \n Press ENTER to continue....",
		"You tripped and fell on your face but manage to cross. Took 5 Damage! \n Press ENTER to continue....",
		new IncreaseStatEffect("Dexterity", 1), new DamageHealthEffect(5), nullptr);
	rightSubtree->attachRight(make_unique<BTree<Decision>>(move(*narrowBridgeDecision)));

	// Attach right subtree to root
	dungeonAdventure->attachRight(move(rightSubtree));

	return dungeonAdventure;
}
unique_ptr<BTree<Decision>> Event::createTempleAdventure(Character& character) {
	// Root of the temple adventure decision tree
	auto rootDecision = DecisionFactory::createDecision(
		"As you approach the entrance of the ancient temple, \nshrouded in the mists of time, you are presented with a pivotal choice. \nTo your left, an overgrown passage, its path concealed by the relentless \ngrasp of nature, \nsuggests secrets hidden and reclaimed by the earth. \n\nTo your right, an echoing hall, its vastness reverberating with the \nwhispers of ages past, \nbeckons you towards the unknown grandeur and potential perils within.\n\n You can only choose one path. \nWill it be the tangled mysteries of the overgrown passage \nor the daunting expanse of the echoing hall?.\n\nLeft: Head to the overgrown passage.\nRight: Head to echoing hall. \n\nChoose a path : \n1.Left\n2.Right",
		false, 0, "", "", "", nullptr, nullptr, nullptr);
	auto templeAdventure = make_unique<BTree<Decision>>(move(*rootDecision));

	// Left subtree: Overgrown passage
	auto leftPathDecision = DecisionFactory::createDecision(
		"You go through the overgrown passage and you found \na hidden alcove and a suspicious shiny door in your way.\n\nLeft: Hidden alcove.\nRight: Suspicious Shiny Door \nChoose a path : \n1.Left\n2.Right",
```

```
        false, 0, "", "", "", nullptr, nullptr, nullptr);
    auto leftPath = make_unique<BTree<Decision>>(move(*leftPathDecision));

    // Left-Left: Hidden alcove
    auto hiddenAlcoveDecision = DecisionFactory::createDecision(
        "You found a hidden alcove filled with ancient texts. \nThese would require some knowledge of the old arts to decipher (Intelligence LVL 12)", true, 12, "Intelligence",
        "You decipher the texts and gain knowledge! +1 Intelligence", "The writings are too cryptic. You get a headache. Took 5 Damage!",
        new IncreaseStatEffect("Intelligence", 1), new DamageHealthEffect(5), nullptr
    );
    leftPath->attachLeft(make_unique<BTree<Decision>>(move(*hiddenAlcoveDecision)));

    // Left-Right: Trapped floor
    auto trappedFloorDecision = DecisionFactory::createDecision(
        "Oh no, the door closed behind you and you find yourself in a room with hidden floor traps. (Dexterity LVL 10)", true, 10, "Dexterity",
        "You navigate the traps safely! +1 Dexterity", "You trigger a trap but escape narrowly. You sprained an ankle. Took 5 Damage!",
        new IncreaseStatEffect("Dexterity", 1), new DamageHealthEffect(5), nullptr
    );
    leftPath->attachRight(make_unique<BTree<Decision>>(move(*trappedFloorDecision)));

    // Attach left subtree to root
    templeAdventure->attachLeft(move(leftPath));

    // Right subtree: Echoing hall
    auto rightPathDecision = DecisionFactory::createDecision(
        "You arrive at the Echoing hall and found a Guarded Relic \non your left and a Mysterious Statue on your right.\nLeft: Guarded relic \nRight: Mysterious statue \nChoose a path : \n1.Left\n2.Right",
        false, 0, "", "", "", nullptr, nullptr, nullptr
    );
```

```
    auto rightPath = make_unique<BTree<Decision>>(move(*rightPathDecision));

    // Right-Left: Guarded relic
    auto guardedRelicDecision = DecisionFactory::createDecision(
        "A relic is suddenly guarded by spectral forces. (Require Strength Lvl 15)", true, 15, "Strength",
        "You fend off the guardians and claim the relic! +1 Strength", "The spectral forces overpower you, but you escape. Barely. Took 5 Damage!",
        new IncreaseStatEffect("Strength", 1), new DamageHealthEffect(5), nullptr
    );
    rightPath->attachLeft(make_unique<BTree<Decision>>(move(*guardedRelicDecision)));

    // Right-Right: Mysterious statue
    auto mysteriousStatueDecision = DecisionFactory::createDecision(
        "A statue with a riddle. (Require Intelligence Lvl 14)", true, 14, "Intelligence",
        "You solve the riddle and find a hidden treasure! +1 Intelligence", "The riddle baffles you, until you get a migraine. Took 5 Damage!",
        new IncreaseStatEffect("Intelligence", 1), new DamageHealthEffect(5), nullptr
    );
    rightPath->attachRight(make_unique<BTree<Decision>>(move(*mysteriousStatueDecision)));

    // Attach right subtree to root
    templeAdventure->attachRight(move(rightPath));

    return templeAdventure;
}
```

//Game.h

```cpp
#include "DynamicArray.h"
#include "Event.h"
#include "Shop.h"
#include "STLInclude.h"
#include "Button.h"

class Game {
public:
    Game(sf::RenderWindow& window, sf::Font* font);
    virtual ~Game();

    void initializeShop();
    void showTypingEffect(const std::string& text, unsigned int maxWidth);
    void initGame();
    bool mainMenu();
    void createNewCharacter();
    void levelUpCharacter();
    void saveCharacters();
    void loadCharacters();
    void selectCharacter();
    void travel();
    void rest();
    void inventoryMenu();
    void addEnemy(const Enemy& enemy);
    void printEnemies(sf::RenderWindow& window) const;

    inline bool getPlaying() const { return this->playing; }


    string loadStoryFromFile(const std::string& filename);

    void openShop();
    void handleCharacterDeath();
    void characterSheet();
    void levelUpMenu();
    void loadExistingCharacters();
private:
    int choice;
    bool playing;
    int activeCharacter;
    dynaArr<Character*> characters;
    std::string fileName;
    SinglyLinkedNode<Enemy>* enemies;

    void drawText( const std::string& text, float x, float y);
    void updateGame();
    bool isWeapon(const std::string& itemName);
    bool isArmor(const std::string& itemName);

    sf::Music gameMusic;

    Shop* shop;

    sf::RenderWindow& window; // Reference
    sf::Font* font;
    std::vector<Button> buttons;

    sf::Texture backgroundTexture;
    sf::Sprite backgroundSprite;
```

```
    sf::Texture RestTexture;
    sf::Sprite RestSprite;

    sf::Texture CharSheetTexture;
    sf::Sprite CharSheetSprite;


};

//Game.cpp
#include "Game.h"


Game::Game(sf::RenderWindow& window, sf::Font* font)

        : window(window), font(font), enemies(nullptr), characters(5), shop(nullptr)

{

        playing = true;

        activeCharacter = 0;

        fileName = "characters.txt";

}


Game::~Game()

{

  while (enemies != nullptr) {

    auto toDelete = enemies;

    enemies = enemies->getNext();

    delete toDelete;

  }


  for (size_t i = 0; i < characters.size(); ++i) {

    delete characters[i];

  }


        delete shop;

}
```

```
//Functions
void Game::initializeShop() {
	if (characters.size() > activeCharacter && characters[activeCharacter] != nullptr) {
		delete shop; // Clean up the old shop if it exists
		shop = new Shop(characters[activeCharacter]->getLevel());
	}
	else {
		std::cerr << "Active character not set. Shop cannot be initialized." << std::endl;
	}
}

string Game::loadStoryFromFile(const std::string& filename) {
	std::ifstream inFile(filename);
	std::string story((std::istreambuf_iterator<char>(inFile)), std::istreambuf_iterator<char>());
	return story;
}

void Game::showTypingEffect(const std::string& text, unsigned int maxWidth) {
	sf::Text storyText;
	storyText.setFont(*this->font);
	storyText.setCharacterSize(30);
	storyText.setFillColor(sf::Color::White);
	storyText.setPosition(50.f, 50.f);

	std::string displayedText = "";
	std::string line = "";
	std::string word = "";
	float yPos = 50.f;

	for (char c : text) {
		if (c == ' ' || c == '\n') {
```

```
                std::string tempLine = line + word + c;
                storyText.setString(tempLine);

                if (storyText.getLocalBounds().width > maxWidth || c == '\n') {
                        displayedText += line + "\n";
                        line = word + c;
                        yPos += storyText.getLocalBounds().height;
                }
                else {
                        line = tempLine;
                }

                word = "";
        }
        else {
                word += c;
        }

        storyText.setString(displayedText + line + word);

        this->window.clear();
        this->window.draw(storyText);
        this->window.display();

        sf::sleep(sf::milliseconds(10)); // Typing effect delay
}

displayedText += line;
storyText.setString(displayedText);

displayedText += "\nPress Enter to continue...";
```

```
        storyText.setString(displayedText);

        this->window.clear();
        this->window.draw(storyText);
        this->window.display();

        // Wait for Enter key press
        bool waitingForEnter = true;
        while (this->window.isOpen() && waitingForEnter) {
            sf::Event event;
            while (this->window.pollEvent(event)) {
                if (event.type == sf::Event::Closed) {
                    this->window.close();
                }
                if (event.type == sf::Event::KeyPressed) {
                    if (event.key.code == sf::Keyboard::Enter) {
                        waitingForEnter = false;  // Exit the loop when Enter is
pressed
                    }
                }
            }
        }
}

void Game::initGame() {
        if (!backgroundTexture.loadFromFile("Assets/IMG/main_menu.png")) {
            std::cerr << "Failed to load background texture" << std::endl;
        }
        backgroundSprite.setTexture(backgroundTexture);

        if (!gameMusic.openFromFile("Assets/AUDIO/main3.ogg")) {
```

```
                std::cerr << "Failed to load music" << std::endl;
        }
    ifstream in("characters.txt");

    Weapon::initNames();
    Armor::initNames();
    Potion::initNames();

        // After loading characters
        if (in.is_open()) {
                this->loadCharacters();
                this->initializeShop();
        }

        else {
                // If no characters are loaded, create a default character
                this->createNewCharacter();
                this->initializeShop();
        }
}

void Game::addEnemy(const Enemy& enemy) {
    auto newNode = new SinglyLinkedNode<Enemy>(enemy, enemies);
    enemies = newNode;
}

void Game::printEnemies(sf::RenderWindow& window) const {
        auto current = enemies;
        float yPos = 20.0f;

        while (current != nullptr) {
```

```
            sf::Text enemyText;
            enemyText.setFont(*this->font); // Dereference the font pointer here
            enemyText.setString(current->getValue().getName());
            enemyText.setCharacterSize(24);
            enemyText.setFillColor(sf::Color::White);
            enemyText.setPosition(10.f, yPos);

            window.draw(enemyText);

            yPos += 30.0f;
            current = current->getNext();
        }
}

bool Game::mainMenu() {

        // Load and set the main menu background
        if (!backgroundTexture.loadFromFile("Assets/IMG/prologue2.png")) {
            std::cerr << "Failed to load main menu background texture" << std::endl;
        }

        backgroundSprite.setTexture(backgroundTexture);

        //Banner
        sf::Texture rectangleTexture;
        if (!rectangleTexture.loadFromFile("Assets/IMG/Buttons/banner3.png")) {
            std::cerr << "Failed to load rectangle texture" << std::endl;
        }

        // Black Rectangle
        sf::Sprite bannerSprite;
```

```
bannerSprite.setTexture(rectangleTexture);
bannerSprite.setPosition(50, 500);

// Welcome text
sf::Text welcomeText;
welcomeText.setFont(*this->font);
welcomeText.setString("Main Menu");
welcomeText.setCharacterSize(30);
welcomeText.setFillColor(sf::Color::White);
welcomeText.setPosition(165, 525);



// Main menu Buttons
//Travel Button
buttons.push_back(Button(100, 600, 89, 27, *this->font, "",
        "Assets/IMG/Buttons/Travel.png", "Assets/IMG/Buttons/Travel2.png"));

//Shop Button
buttons.push_back(Button(100, 625, 67, 29, *this->font, "",
        "Assets/IMG/Buttons/Shop.png", "Assets/IMG/Buttons/Shop2.png"));

//Level Up Button
buttons.push_back(Button(100, 650, 122, 29, *this->font, "",
        "Assets/IMG/Buttons/Lvlup.png", "Assets/IMG/Buttons/Lvlup2.png"));

//Rest Button
buttons.push_back(Button(100, 675, 65, 27, *this->font, "",
        "Assets/IMG/Buttons/Rest.png", "Assets/IMG/Buttons/Rest2.png"));

//Show Character Sheet Button
```

```
buttons.push_back(Button(100, 700, 197, 27, *this->font, "",
        "Assets/IMG/Buttons/CharInfo.png", "Assets/IMG/Buttons/CharInfo2.png"));

//Show Inventory Button
buttons.push_back(Button(100, 725, 202, 29, *this->font, "",
        "Assets/IMG/Buttons/inventory.png", "Assets/IMG/Buttons/inventory2.png"));

//Create New Character Button
buttons.push_back(Button(100, 750, 187, 27, *this->font, "",
        "Assets/IMG/Buttons/newChar.png", "Assets/IMG/Buttons/newChar2.png"));

//Save Current Character Button
buttons.push_back(Button(100, 775, 205, 27, *this->font, "",
        "Assets/IMG/Buttons/save.png", "Assets/IMG/Buttons/save2.png"));

//Load Saved Character BUtton
buttons.push_back(Button(100, 800, 205, 27, *this->font, "",
        "Assets/IMG/Buttons/select.png", "Assets/IMG/Buttons/select2.png"));

//Quit Button
buttons.push_back(Button(100, 825, 60, 27, *this->font, "",
        "Assets/IMG/Buttons/quit.png", "Assets/IMG/Buttons/quit2.png"));



while (this->window.isOpen()) {
        sf::Event event;

        // Update character info text based on the current character's status
        if (characters.size() > 0 && characters[activeCharacter]) {
                std::string characterInfo = "";
```

```
                if (characters[activeCharacter]->getExp() >= characters[activeCharacter]->getExpNext()) {
                        characterInfo += " - Level Up Available!";
                }

        }

        while (this->window.pollEvent(event)) {
                if (event.type == sf::Event::Closed) {
                        this->window.close();
                }

                for (auto& button : buttons) {
                        button.update(this->window);
                }

                if (event.type == sf::Event::MouseButtonPressed) {
                        if (event.mouseButton.button == sf::Mouse::Left) {

                                if (buttons[0].isMouseOver(this->window)) {
                                        // Travel button action
                                        travel();
                                        break;
                                }

                                else if (buttons[1].isMouseOver(this->window)) {
                                        //Shop

                                        openShop();
                                        break;
                                }
```

```
else if (buttons[2].isMouseOver(this->window)) {
        //Level up
        levelUpMenu();
        break;
}

else if (buttons[3].isMouseOver(this->window)) {
        //Rest
        rest();
        break;
}

else if (buttons[4].isMouseOver(this->window)) {
        //Character Information
        characterSheet();
        break;
}

else if (buttons[5].isMouseOver(this->window)) {
        //Character Inventory
        inventoryMenu();
        break;
}

else if (buttons[6].isMouseOver(this->window)) {
        //Create New Character
        createNewCharacter();
        break;
}
```

```
                                else if (buttons[7].isMouseOver(this->window)) {
                                        //Save Current Character
                                        saveCharacters();
                                        break;
                                }

                                else if (buttons[8].isMouseOver(this->window)) {
                                        //Select Saved Character
                                        selectCharacter();
                                        break;
                                }

                                else if (buttons[9].isMouseOver(this->window)) {
                                        this->window.close();
                                        return false; // Return false to indicate game should
exit
                                }
                        }
                }
        }

        this->window.clear();
        this->window.draw(backgroundSprite);
        this->window.draw(bannerSprite);

        // Draw the welcome text and character info
        this->window.draw(welcomeText);


        for (auto& button : buttons) {
                button.drawTo(this->window);
```

```
                }

                this->window.display();
        }
        return true;
}

void Game::openShop() {

        // Check if shop is initialized
        if (shop) {
                shop->openShop(*characters[activeCharacter], this->window, *this->font);
        }

        else {
                // Display error message or initialize shop if needed
                sf::Text errorMessage("Shop is not available.", *this->font, 24);
                errorMessage.setFillColor(sf::Color::White);
                errorMessage.setPosition(50.f, 50.f);

                this->window.clear();
                this->window.draw(errorMessage);
                this->window.display();
                sf::sleep(sf::seconds(2)); // Display message for 2 seconds

                // Return to main menu
                mainMenu();
        }
}

void Game::characterSheet() {
```

```
if (!CharSheetTexture.loadFromFile("Assets/IMG/main_menu6.png")) {
	std::cerr << "Failed to load main menu background texture" << std::endl;
}
CharSheetSprite.setTexture(CharSheetTexture);
CharSheetSprite.setPosition(0.f, 0.f);

sf::RectangleShape rectangle;
rectangle.setSize(sf::Vector2f(325, 600));
rectangle.setFillColor(sf::Color::Black);
rectangle.setOutlineColor(sf::Color::Black);
rectangle.setOutlineThickness(5);
rectangle.setPosition(40, 290);

if (characters.size() > 0 && characters[activeCharacter]) {
	// Clear the window
	this->window.clear();

	this->window.draw(CharSheetSprite);
	this->window.draw(rectangle);

	// Call the renderStats method from the Character class
	characters[activeCharacter]->renderStats(this->window, *this->font, 50.f, 300.f);

	// Display the character sheet

	this->window.display();

	// Wait for the user to press Enter to return to the main menu
	bool waitingForEnter = true;
	while (this->window.isOpen() && waitingForEnter) {
		sf::Event event;
```

```
                        while (this->window.pollEvent(event)) {
                                if (event.type == sf::Event::Closed) {
                                        this->window.close();
                                }
                                else if (event.type == sf::Event::KeyPressed && event.key.code ==
sf::Keyboard::Enter) {
                                        waitingForEnter = false;
                                }
                        }
                }
        }
}

//Create new Character window
void Game::createNewCharacter() {
        std::string inputName;

        // Initialize text objects for display
        sf::RectangleShape rectangle;
        rectangle.setSize(sf::Vector2f(250, 100));
        rectangle.setFillColor(sf::Color::Black);
        rectangle.setOutlineColor(sf::Color::Black);
        rectangle.setOutlineThickness(5);
        rectangle.setPosition(450, 500);

        sf::Text promptText;
        promptText.setFont(*this->font); // Set the font (dereference the pointer)
        promptText.setString("Enter Character Name:"); // Set the text
        promptText.setCharacterSize(24); // Set the character size
        promptText.setFillColor(sf::Color::White);
        promptText.setPosition(460, 500);
```

```
sf::Text inputText;
inputText.setFont(*this->font); // Set the font (dereference the pointer)
inputText.setString(""); // Initially empty string
inputText.setCharacterSize(24); // Set the character size
inputText.setFillColor(sf::Color::White);
inputText.setPosition(460, 520);

bool nameEntered = false;

while (this->window.isOpen() && !nameEntered) {
	sf::Event event;
	while (this->window.pollEvent(event)) {
		if (event.type == sf::Event::Closed) {
			this->window.close();
		}
		// Handle text input event
		if (event.type == sf::Event::TextEntered) {
			if (event.text.unicode < 128) {
				inputName += static_cast<char>(event.text.unicode);
				inputText.setString(inputName);
			}
		}
		// Handle Enter key for finalizing name
		if (event.type == sf::Event::KeyPressed) {
			if (event.key.code == sf::Keyboard::Enter) {
				nameEntered = true;
			}
			// Handle backspace for editing name
			else if (event.key.code == sf::Keyboard::BackSpace &&
!inputName.empty()) {
```

```
                                inputName.pop_back();
                                inputText.setString(inputName);
                        }
                }
        }

        this->window.clear();
        this->window.draw(backgroundSprite);
        this->window.draw(rectangle);
        this->window.draw(promptText);
        this->window.draw(inputText);
        this->window.display();

        if (nameEntered) {
                // Check if name already exists
                for (size_t i = 0; i < this->characters.size(); i++) {
                        if (inputName == this->characters[i]->getName()) {
                                inputName = ""; // Reset name if it already exists
                                nameEntered = false; // Restart name entry
                                inputText.setString(inputName);
                                break;
                        }
                }
        }
}

if (!inputName.empty()) {
        Character* newCharacter = new Character(inputName, 0, 100, 1, 0, 5, 5, 5, 5, 22, 8,
0);
        this->characters.push(newCharacter);
        this->activeCharacter = this->characters.size() - 1;
```

```
                this->initializeShop(); // Initialize shop for the new character
                mainMenu();  // Start the game with the new character
        }
}

void Game::levelUpCharacter() {
        if (this->characters[this->activeCharacter]->getExp() >= this->characters[this->activeCharacter]->getExpNext()) {

                // Level up character
                this->characters[this->activeCharacter]->levelUp();

                sf::Text levelUpText("Choose a stat to upgrade:", *this->font, 24);
                levelUpText.setFillColor(sf::Color::White);
                levelUpText.setPosition(10.f, 10.f);

                //Options of stats to upgrade
                std::vector<std::string> optionsText = { "Strength", "Vitality", "Dexterity", "Intelligence" };
                std::vector<sf::Text> options;

                for (size_t i = 0; i < optionsText.size(); ++i) {
                        sf::Text option(optionsText[i], *this->font, 24);
                        option.setFillColor(sf::Color::White);
                        option.setPosition(10.f, 50.f + i * 40.f);
                        options.push_back(option);
                }

                int selectedOption = -1;

                while (this->window.isOpen() && selectedOption == -1) {
                        sf::Event event;
```

```
                while (this->window.pollEvent(event)) {
                        if (event.type == sf::Event::KeyPressed) {
                                if (event.key.code >= sf::Keyboard::Num1 &&
event.key.code <= sf::Keyboard::Num4) {
                                        selectedOption = event.key.code -
sf::Keyboard::Num1;
                                        this->characters[this->activeCharacter]-
>addToStat(selectedOption, 1);

                                        // Display a message indicating the stat has been
upgraded
                                        sf::Text upgradeText("Stat upgraded successfully!",
*this->font, 24);
                                        upgradeText.setFillColor(sf::Color::White);
                                        upgradeText.setPosition(10.f, 200.f);
                                        this->window.draw(upgradeText);
                                        this->window.display();
                                        sf::sleep(sf::seconds(3)); // Display the message for
3 seconds
                                        break;
                                }
                        }
                }

                this->window.clear();
                this->window.draw(levelUpText);
                for (auto& option : options) {
                        this->window.draw(option);
                }
                this->window.display();
        }
    }
    else {
```

```
                // Handle the case when the character does not have enough experience to level up
                sf::Text noLevelUpText("Not enough experience to level up.", *this->font, 24);
                noLevelUpText.setFillColor(sf::Color::White);
                noLevelUpText.setPosition(10.f, 10.f);
                this->window.clear();
                this->window.draw(noLevelUpText);
                this->window.display();
                sf::sleep(sf::seconds(2)); // Display the message for 2 seconds
        }

}

void Game::saveCharacters() {
        ofstream outFile(fileName);

        if (outFile.is_open()) {
                for (size_t i = 0; i < characters.size(); i++) {
                        Character* character = characters[i]; // Get the pointer to the character

                        outFile << "BEGIN_CHARACTER\n";
                        outFile << character->getAsString() << "\n"; // Use pointer to call getAsString
                        outFile << "BEGIN_INVENTORY\n";

                        for (int j = 0; j < character->getInventorySize(); ++j) {
                                Item* item = character->getInventoryItem(j); // Use pointer to call
getInventoryItem
                                if (item != nullptr) {
                                        // Check item type and add prefix
                                        if (dynamic_cast<Weapon*>(item)) {
                                                outFile << "Weapon ";
                                        }
```

```
                                        else if (dynamic_cast<Armor*>(item)) {
                                                outFile << "Armor ";
                                        }
                                        else if (dynamic_cast<Potion*>(item)) {
                                                outFile << "Potion ";
                                        }

                                        // Write item details
                                        outFile << item->toString() << "\n";
                                }
                        }

                        outFile << "END_INVENTORY\n";
                        outFile << "END_CHARACTER\n";
                }
        }

        else
        {
                std::cerr << "Unable to open file for saving characters." << std::endl;
        }

        outFile.close();
}


bool Game::isWeapon(const string& itemName) {
        static const set<string> weaponNames = {
                "Combat Knife", "Sword", "Axe", "PickAxe", "WarHammer", "Katana"
        };
```

```
	return weaponNames.find(itemName) != weaponNames.end();
}

bool Game::isArmor(const string& itemName) {
	static const set<string> armorNames = {
		"Leather Armor", "Iron Armor", "Gold Armor", "ChainMail Armor", "Diamond Armor", "Nether Armor"
	};

	return armorNames.find(itemName) != armorNames.end();
}

void Game::loadExistingCharacters() {
	ifstream inFile("characters.txt");

	if (inFile.is_open()) {
		this->loadCharacters(); // Use your existing loadCharacters logic
		mainMenu();  // Return to main menu after loading
	}
	else {
		sf::Text errorText("No saved characters found.", *this->font, 24);
		errorText.setFillColor(sf::Color::White);
		errorText.setPosition(50.f, 50.f);
		this->window.draw(errorText);
		this->window.display();
		sf::sleep(sf::seconds(2));
		mainMenu();
	}
}

void Game::loadCharacters() {
```

```
	ifstream inFile(fileName);
	string line;

	if (inFile.is_open()) {
		Character* temp = nullptr;
		bool loadingCharacter = false;
		bool loadingInventory = false;


		while (getline(inFile, line)) {
			//Starting point to detect Character Object
			if (line == "BEGIN_CHARACTER") {
				temp = new Character();
				loadingCharacter = true;
				loadingInventory = false;
			}

			//Ending point to detect Character Object
			else if (line == "END_CHARACTER") {
				if (temp != nullptr) {
					this->characters.push(temp);
					cout << "Character " << temp->getName() << " loaded!" <<
endl;
				}
				loadingCharacter = false;
				temp = nullptr; // Reset temp to nullptr after adding to characters
			}

			//Starting point to detect Inventory Object under Detected Character Object
			else if (line == "BEGIN_INVENTORY") {
				loadingInventory = true;
```

```
                }

                //Ending point to detect Inventory Object under Detected Character Object
                else if (line == "END_INVENTORY") {
                        loadingInventory = false;
                }

                //Start loading Character Details
                else if (loadingCharacter && !loadingInventory && !line.empty()) {
                        stringstream str(line);
                        string name;
                        int distanceTravelled, gold, level, exp, strength, vitality, dexterity,
intelligence, hp, stamina, statPoints;
                        str >> name >> distanceTravelled >> gold >> level >> exp >> strength
>> vitality >> dexterity >> intelligence >> hp >> stamina >> statPoints;
                        temp = new Character(name, distanceTravelled, gold, level, exp,
strength, vitality, dexterity, intelligence, hp, stamina, statPoints);
                }

                //Start Loading Inventory Item Details
                else if (loadingInventory && !line.empty()) {
                        stringstream itemStream(line);
                        string itemType;
                        itemStream >> itemType;

                        if (itemType == "Weapon") {
                                // Load weapon data
                                string name;
                                int damageMin, damageMax, level, buyValue, sellValue,
rarity;
                                itemStream >> name >> level >> rarity >> damageMin >>
damageMax >> buyValue >> sellValue;
```

```
                Weapon* weapon = new Weapon(damageMin,
damageMax, name, level, buyValue, sellValue, rarity);
                temp->addItem(*weapon);
                delete weapon;
            }
            else if (itemType == "Armor") {
                // Load armor data
                string name;
                int defence, level, buyValue, sellValue, rarity;
                itemStream >> name >> level >> rarity >> defence >>
buyValue >> sellValue;
                Armor* armor = new Armor(defence, name, level, buyValue,
sellValue, rarity);
                temp->addItem(*armor);
                delete armor;
            }

            else if (itemType == "Potion") {
                // Load potion data
                string name, description, typeStr;
                int level, rarity, potency;

                itemStream >> name >> description >> level >> rarity >>
potency;

                Potion::PotionType type = Potion::PotionType::Health; //
Default type
                if (!typeStr.empty()) {
                    if (typeStr == "Health") {
                        type = Potion::PotionType::Health;
                    }
                    else if (typeStr == "Strength") {
                        type = Potion::PotionType::Strength;
```

```
                                        }
                                        else if (typeStr == "Vitality") {
                                                type = Potion::PotionType::Vitality;
                                        }
                                        else if (typeStr == "Dexterity") {
                                                type = Potion::PotionType::Dexterity;
                                        }
                                        else if (typeStr == "Intelligence") {
                                                type = Potion::PotionType::Intelligence;
                                        }
                                        else {
                                                cout << "Unknown potion type: " << typeStr
<< endl;
                                        }
                                }
                                Potion* potion = new Potion(name, level, rarity, type,
potency);
                                temp->addItem(*potion);
                                delete potion;
                        }
                }
        }
        inFile.close();

        if (this->characters.size() == 0) {
                throw "No Characters Available.";
        }
}
```

```
void Game::selectCharacter() {


        // Check if there are characters to select
        if (characters.size() > 0) {
                sf::Text title("Select Character:", *this->font, 24);
                title.setFillColor(sf::Color::White);
                title.setPosition(10.f, 10.f);

                std::vector<sf::Text> characterOptions;

                // Create a menu option for each character
                for (size_t i = 0; i < characters.size(); i++) {
                        std::string optionText = std::to_string(i + 1) + ". " + characters[i]->getName()
+ " (Level: " + std::to_string(characters[i]->getLevel()) + ")";
                        sf::Text option(optionText, *this->font, 20);
                        option.setFillColor(sf::Color::White);
                        option.setPosition(10.f, 50.f + i * 30.f);
                        characterOptions.push_back(option);
                }

                int selectedCharacter = -1;

                // Event loop for character selection
                while (this->window.isOpen() && selectedCharacter == -1) {
```

```
                sf::Event event;
                while (this->window.pollEvent(event)) {
                        if (event.type == sf::Event::Closed) {
                                this->window.close();
                        }

                        if (event.type == sf::Event::KeyPressed) {
                                if (event.key.code >= sf::Keyboard::Num1 &&
event.key.code <= sf::Keyboard::Num9) {
                                        int choice = event.key.code - sf::Keyboard::Num1;
                                        if (choice >= 0 && choice <
static_cast<int>(characters.size())) {
                                                selectedCharacter = choice;
                                        }
                                }
                        }
                }

                this->window.clear();
                this->window.draw(title);
                for (auto& option : characterOptions) {
                        this->window.draw(option);
                }
                this->window.display();
        }

        if (selectedCharacter >= 0) {
                activeCharacter = selectedCharacter;
                this->initializeShop(); // Initialize shop for the selected character
        }
    }
    else {
```

```
                sf::Text noCharacterText("No characters to select.", *this->font, 24);
                noCharacterText.setFillColor(sf::Color::White);
                noCharacterText.setPosition(50.f, 50.f);
                this->window.clear();
                this->window.draw(noCharacterText);
                this->window.display();
                sf::sleep(sf::seconds(2));
        }

        // Return to the main menu after selection or if there are no characters
        mainMenu();
}




void Game::travel() {
        // Increment the travel distance of the active character
        characters[activeCharacter]->travel();




        // Generate a random event
        Event event;
        int eventType = rand() % 2;  // Randomly decide the type of event

        // Check the type of event and handle it accordingly
        if (eventType == 0) {
                // Handle enemy encounter event
                event.enemyEncounter(this->window, *this->font, *characters[activeCharacter],
enemies);
```

```
        }
        else if (eventType == 1) {
                // Handle puzzle encounter event
                event.puzzleEncounter(this->window, *this->font, *characters[activeCharacter]);
        }

        // Update the this->window display and handle event aftermath
        this->window.clear();
        // ... Draw relevant game elements if needed
        this->window.display();

        // Update the game state after the event
        // For example, check if the character leveled up or if they are still alive
        if (!characters[activeCharacter]->isAlive()) {
                handleCharacterDeath();
        }
}

void Game::handleCharacterDeath() {
        // Display death message
        sf::Text deathText("Your character has died. \nWould you like to start a new character? (Y/N)", *this->font, 24);
        deathText.setFillColor(sf::Color::White);
        deathText.setPosition(100, 100); // Adjust as needed

        this->window.clear();
        this->window.draw(deathText);
        this->window.display();

        // Wait for user input
        sf::Event event;
```

```
        while (this->window.waitEvent(event)) {
                if (event.type == sf::Event::Closed) {
                        this->window.close();
                        exit(0); // Exit the game
                }
                else if (event.type == sf::Event::KeyPressed) {
                        if (event.key.code == sf::Keyboard::Y) {
                                createNewCharacter(); // Create a new character
                                break;
                        }
                        else if (event.key.code == sf::Keyboard::N) {
                                this->window.close();
                                exit(0); // Exit the game
                        }
                }
        }
}



void Game::rest() {
        // Load and set the shop background
        if (!RestTexture.loadFromFile("Assets/IMG/rest1.png")) {
                std::cerr << "Failed to load main menu background texture" << std::endl;
        }
        RestSprite.setTexture(RestTexture);
        RestSprite.setPosition(0.f, 0.f);

        if (characters.size() > activeCharacter && characters[activeCharacter]) {
                Character* activeChar = characters[activeCharacter];
                int cost = (activeChar->getHpMax() - activeChar->getHp()) * 2;
```

```
		// Display a welcome message
		sf::Text welcomeText("Ah, welcome, traveler! Weary from your journey, are you? \nStep right in and make yourself at home.", *this->font, 24);
		welcomeText.setFillColor(sf::Color::White);
		welcomeText.setPosition(50.f, 750.f);

		// Check if the character has enough gold
		if (activeChar->getGold() >= cost) {
			activeChar->resetHP(); // Restore HP to maximum
			activeChar->payGold(cost); // Deduct the gold cost

			// Display a success message
			sf::Text restText("You rented a room and took a rest for a day. Healing Cost: " + to_string(cost) + " Gold.", *this->font, 24);
			restText.setFillColor(sf::Color::White);
			restText.setPosition(50.f, 800.f);

			this->window.clear(); // Clear the this->window
			window.draw(RestSprite);
			this->window.draw(welcomeText); // Draw the welcomeText
			this->window.draw(restText); // Draw the restText
			this->window.display(); // Display the updated window

			// Wait for a key press before returning to the menu
			sf::Event event;
			while (this->window.waitEvent(event)) {
				if (event.type == sf::Event::KeyPressed) {
					break;
				}
			}
```

```
                // Re-draw the main menu
                mainMenu();
            }
            else {
                // Display a failure message
                sf::Text restText("Not enough gold to rest.", *this->font, 24);
                restText.setFillColor(sf::Color::White);
                restText.setPosition(50.f, 815.f);

                this->window.clear(); // Clear the this->window
                this->window.draw(restText); // Draw the restText
                this->window.display(); // Display the updated this->window

                // Wait for a key press before returning to the menu
                sf::Event event;
                while (this->window.waitEvent(event)) {
                    if (event.type == sf::Event::KeyPressed) {
                        break;
                    }
                }

                // Re-draw the main menu
                mainMenu();
            }
        }
}



void Game::inventoryMenu() {
```

```
float xPos = 50.f; // Starting x-position for inventory items
float yPos = 50.f; // Starting y-position for inventory items
float columnWidth = 300.f; // Width of each inventory column
const float windowMargin = 50.f; // Margin at the right edge of the window


sf::Text inventoryTitle("Inventory. What do you wish to equip? (Press ESC to return to main menu). ", *this->font, 24);
inventoryTitle.setFillColor(sf::Color::White);
inventoryTitle.setPosition(50.f, 10.f);




std::vector<sf::Text> inventoryTexts; // Store inventory texts for re-drawing

// Display inventory items
for (int i = 0; i < characters[activeCharacter]->getInventorySize(); ++i) {
    Item* item = characters[activeCharacter]->getInventoryItem(i);
    sf::Text itemText;

    if (item) {
        itemText.setFont(*this->font);
        itemText.setString(std::to_string(i + 1) + ": " + item->toString());
        itemText.setCharacterSize(20);
        itemText.setFillColor(sf::Color::White);
        itemText.setPosition(xPos, yPos);

        inventoryTexts.push_back(itemText); // Store the text for re-drawing
```

```
                        yPos += 200.f; // Increment y-position for the next item

                        // Check if yPos exceeds window height and there is space for a new column
                        if (yPos + 40.f > this->window.getSize().y && xPos + columnWidth +
windowMargin < this->window.getSize().x) {
                            xPos += columnWidth; // Move to a new column
                            yPos = 50.f; // Reset y-position for the new column
                        }
                    }
                }
                this->window.clear();
                this->window.draw(inventoryTitle);
                // Re-display inventory items using inventoryTexts
                for (auto& text : inventoryTexts) {
                    this->window.draw(text);
                }

                this->window.display();



                while (this->window.isOpen()) {
                    sf::Event event;
                    while (this->window.pollEvent(event)) {
                        if (event.type == sf::Event::Closed) {
                            this->window.close();
                        }

                        // Check for number keys to select inventory items
                        if (event.type == sf::Event::KeyPressed && event.key.code >=
sf::Keyboard::Num1 && event.key.code <= sf::Keyboard::Num9) {
                            int choice = event.key.code - sf::Keyboard::Num1;
```

```
                    if (choice >= 0 && choice < characters[activeCharacter]->getInventorySize())
                    {
                        // Equip or use the selected item
                        Item* selectedItem = characters[activeCharacter]->getInventoryItem(choice);
                        if (dynamic_cast<Weapon*>(selectedItem)) {
                            characters[activeCharacter]->equipWeapon(choice);

                        }
                        else if (dynamic_cast<Armor*>(selectedItem)) {
                            characters[activeCharacter]->equipArmor(choice);

                        }
                        else if (dynamic_cast<Potion*>(selectedItem)) {
                            characters[activeCharacter]->usePotion(choice);

                        }

                        // After equipping an item, return to the main menu
                        mainMenu();
                        return; // Exit the function

                    }
                }

                else if (event.key.code == sf::Keyboard::Escape)
                {
                    mainMenu();
                    return;
                }
            }
```

```
            }
}


void Game::drawText(const std::string& text, float x, float y) {
    sf::Text sfmlText;
    sfmlText.setFont(*this->font);
    sfmlText.setString(text);
    sfmlText.setCharacterSize(24);
    sfmlText.setFillColor(sf::Color::White);
    sfmlText.setPosition(x, y);
    this->window.draw(sfmlText);
}




void Game::updateGame() {
            this->window.clear(); // Clear the screen
            this->window.draw(backgroundSprite); // Draw the background
            mainMenu(); // Example call
            this->window.display(); // Update the screen
}

void Game::levelUpMenu() {
    Character* currentChar = characters[activeCharacter];

    if (currentChar->getExp() >= currentChar->getExpNext()) {
        // Level up the character
        currentChar->levelUp();
```

```
// Display message for leveling up
sf::Text levelUpMessage("You have leveled up! Choose a stat to increase: \n1. Strength \n2. Vitality \n3. Dexterity \n4. Intelligence \n\nWhich stat do you wish to upgrade?", *this->font, 24);
levelUpMessage.setFillColor(sf::Color::White);
levelUpMessage.setPosition(50.f, 50.f);

this->window.clear();
this->window.draw(levelUpMessage);
this->window.display();

// Allow the player to allocate stat points
while (currentChar->getStatPoints() > 0 && this->window.isOpen()) {
   sf::Event event;
   while (this->window.pollEvent(event)) {
      if (event.type == sf::Event::KeyPressed) {
                                        switch (event.key.code) {
                                        case sf::Keyboard::Num1:
                                                currentChar->addToStat(0, 1);
                                                break;
                                        case sf::Keyboard::Num2:
                                                currentChar->addToStat(1, 1);
                                                break;
                                        case sf::Keyboard::Num3:
                                                currentChar->addToStat(2, 1);
                                                break;
                                        case sf::Keyboard::Num4:
                                                currentChar->addToStat(3, 1);
                                                break;
                                        default:
                                                break; // No valid key was pressed
```

```
                                        }
            }
          }
        }

        // Redraw the main menu after allocating stat points
        mainMenu();

    } else {
        // Character does not have enough experience to level up
        sf::Text noLevelUpText("Not enough experience to level up.", *this->font, 24);
        noLevelUpText.setFillColor(sf::Color::White);
        noLevelUpText.setPosition(50.f, 50.f);

        this->window.clear();
        this->window.draw(noLevelUpText);
        this->window.display();
        sf::sleep(sf::seconds(2)); // Display message for 2 seconds

        // Redraw the main menu
        mainMenu();
    }
}
```

//Inventory.h

```
#pragma once

#include "STLInclude.h"
#include "Weapon.h"
#include "Armor.h"
#include "Item.h"
#include "Potion.h"


class Inventory
{
public:
        Inventory();
        ~Inventory();
```

```
        //copy constructor
        Inventory(const Inventory& obj);

        void addItem(const Item& item);
        void removeItem(int index);

        // Accessors
        Item* getItem(int index) const;
        int size() const;
        void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;

private:
        //Using Doubly Linked List
        DoublyLinkedNode<Item*>* head;
        DoublyLinkedNode<Item*>* tail;
        int itemCount;
};
```

//Inventory.cpp

```
#include "Inventory.h"

Inventory::Inventory() : head(nullptr), tail(nullptr), itemCount(0){}

Inventory::~Inventory()
{
    while (head != nullptr)
    {
        DoublyLinkedNode<Item*>* temp = head;
        head = head->getNext();
        delete temp->getValue();
        delete temp;
    }
}

Inventory::Inventory(const Inventory& obj) : Inventory() {
    if (obj.head == nullptr) {
        return; // Nothing to copy
    }

    DoublyLinkedNode<Item*>* current = obj.head;
    while (current != nullptr) {
        Item* currentItem = current->getValue();
        if (currentItem != nullptr) {
            addItem(*currentItem);
        }
        else {
            // Handle the error: currentItem is null
        }
        current = current->getNext();
    }
}


void Inventory::addItem(const Item& item)
{
    DoublyLinkedNode<Item*>* newNode = new DoublyLinkedNode<Item*>(item.clone());
    if (tail != nullptr)
    {
```

```
        tail->append(newNode);

    }
    else
    {
        head = newNode;
    }

    tail = newNode;
    itemCount++;
}


void Inventory::removeItem(int index) {
    if (index < 0 || index >= itemCount)
    {
        throw out_of_range("Index out of bounds");
    }

    DoublyLinkedNode<Item*>* current = head;
    for (int i = 0; i < index; ++i)
    {
        current = current->getNext();
    }

    if (current == head)
    {
        head = current->getNext();
    }

    if (current == tail)
    {
        tail = current->getPrevious();
    }

    current->remove();
    delete current->getValue();
    delete current;
    itemCount--;
}


Item* Inventory::getItem(int index) const {
    if (index < 0 || index >= itemCount)
    {
        throw out_of_range("Index out of bounds");
    }

    DoublyLinkedNode<Item*>* current = head;
    for (int i = 0; i < index; ++i)
    {
        current = current->getNext();
    }

    return current->getValue();
}

int Inventory::size() const
{
    return itemCount;
}
```

```
void Inventory::render(sf::RenderWindow& window, sf::Font& font, float x, float
y) const {
    sf::Text text;
    text.setFont(font);
    text.setCharacterSize(24);
    text.setFillColor(sf::Color::White);

    float yOffset = 0.0f;
    DoublyLinkedNode<Item*>* current = head;
    int index = 0;

    while (current != nullptr) {
        Item* currentItem = current->getValue();
        if (currentItem != nullptr) {
            text.setPosition(x, y + yOffset);
            text.setString(std::to_string(++index) + ": " + currentItem-
>getName());
            window.draw(text);

            yOffset += text.getLocalBounds().height + 5; // Adjust line spacing.
        }

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

//Item.h

```
#pragma once

#include "STLInclude.h"
enum rarity
{
	COMMON,
	UNCOMMON,
	RARE,
	EPIC,
	LEGENDARY
};

class Item
{
public:
	Item();
	Item(int level, int rarity);
	Item(string name,
		int level,
		int buyValue,
		int sellValue,
		int rarity);

	virtual ~Item();

	inline string debugPrint() const { return this->name; }
	virtual Item* clone() const = 0;
	virtual string toString() const = 0;

	//Accessors
	inline const string& getName() const { return this->name; }
	inline const int& getLevel() const { return this->level; }
	inline const int& getBuyValue() const { return this->buyValue; }
	inline const int& getSellValue() const { return this->sellValue; }
```

```
	//Rarity of Items converted to string
	inline string getRarity() const {
		switch (this->rarity) {
		case COMMON: return "Common";
		case UNCOMMON: return "Uncommon";
		case RARE: return "Rare";
		case EPIC: return "Epic";
		case LEGENDARY: return "Legendary";
		default: return "Unknown";
		}
	}

	//Modifiers
	inline void setName(string name) { this->name = name; }

	//Static
	inline static int calculateBuyValue(int level, int rarity){ return 100 *
level * (rarity + 1); }
	inline static int calculateSellValue(int level, int rarity) { return
calculateBuyValue(level, rarity) / 2; };

	void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;
private:
	string name;
	int level;
	int buyValue;
	int sellValue;
	int rarity;
};
```

//Item.cpp

```
#include "Item.h"

Item::Item()
{
	this->name = "NONE";
	this->level = 0;
	this->buyValue = 0;
	this->sellValue = 0;
	this->rarity = 0;
}

Item::Item(int level, int rarity)
{
	this->name = "NONE";
	this->level = rand() % level+3;

	this->buyValue = level * rand()%rarity+1;
	this->sellValue = this->buyValue / 2;
}

Item::Item(string name, int level, int buyValue, int sellValue, int rarity)
{
	this->name = name;
	this->level = level;
	this->buyValue = buyValue;
	this->sellValue = sellValue;
	this->rarity = rarity;
```

```
}

Item::~Item()
{

}

void Item::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
	sf::Text text;
	text.setFont(font);
	text.setCharacterSize(24);
	text.setFillColor(sf::Color::White);
	text.setPosition(x, y);

	std::stringstream ss;
	ss << "Name: " << this->name << "\n"
		<< "Level: " << this->level << "\n"
		<< "Rarity: " << getRarity() << "\n"
		<< "Buy Value: " << this->buyValue << "\n"
		<< "Sell Value: " << this->sellValue << "\n";

	text.setString(ss.str());
	window.draw(text);
}
```

//main.cpp

```
#include "Game.h"

using namespace std;

int main()
{
    // Create the main window
    sf::RenderWindow window(sf::VideoMode(1024, 1024), "Adventure Quest",
sf::Style::Close);

    // Load background image
    sf::Texture backgroundTexture;
    if (!backgroundTexture.loadFromFile("Assets/IMG/menu3.png")) {
        std::cerr << "Error loading background.png" << std::endl;
        return EXIT_FAILURE;
    }

    // Create background sprite
    sf::Sprite background;
    background.setTexture(backgroundTexture);

    // Load font
    sf::Font font;
    if (!font.loadFromFile("Assets/FONT/font8.ttf")) {
        std::cerr << "Error loading font" << std::endl;
        return EXIT_FAILURE;
    }

    // Load and play music
    sf::Music menuMusic;
    if (!menuMusic.openFromFile("Assets/AUDIO/main3.ogg")) {
        std::cerr << "Error loading music" << std::endl;
        return EXIT_FAILURE;
```

```
    }

    menuMusic.setLoop(true); // Loop the music
    menuMusic.play();
    menuMusic.setVolume(10.f); // Set volume to 10%

    bool gameStarted = false;
    bool showBackstory = false;

    // Start the game loop
    try {
        while (window.isOpen()) {
            // Process events
            sf::Event event;
            while (window.pollEvent(event)) {
                // Close window: exit
                if (event.type == sf::Event::Closed) {
                    window.close();
                }

                // Check for Enter key press to start the game
                if (event.type == sf::Event::KeyPressed) {
                    if (event.key.code == sf::Keyboard::Enter && !gameStarted) {
                        gameStarted = true;
                        showBackstory = true;
                    }
                }
            }

            window.clear();
            window.draw(background);
            Game game(window, &font);

            if (!gameStarted) {
                //do nothing
            }

            else if (showBackstory) {
                // Main game logic

                std::string rpgStory =
game.loadStoryFromFile("Assets/TXT/Introduction.txt");
                game.showTypingEffect(rpgStory, 850);

                showBackstory = false;  // Reset after showing
            }

            else {
                game.initGame();
                while (!game.mainMenu()) {
                    break;
                }

            }
            window.display();

        }
        return EXIT_SUCCESS;
    }

    catch (const std::runtime_error& e) {
        std::cout << "Exiting game: " << e.what() << std::endl;
        return EXIT_SUCCESS;
```

```
    }

}
```

//ItemFactory.h

```
#pragma once
#include "Weapon.h"
#include "Armor.h"
#include "Potion.h"

//Factory Design Pattern
class ItemFactory {
public:
    virtual ~ItemFactory() {}
    virtual Item* createItem(int level, int rarity) const = 0;
};

class WeaponFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        return new Weapon(level, rarity);
    }
};

class ArmorFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        return new Armor(level, rarity);
    }
};

class PotionFactory : public ItemFactory {
public:
    Item* createItem(int level, int rarity) const override {
        Potion::PotionType type = static_cast<Potion::PotionType>(rand() % 6);
        int potency = level * 5;
        return new Potion(Potion::namesMap[type], level, rarity, type, potency);
    }
};
```

//Iterator.h

```
#pragma once

#include "SinglyLinkedNode.h"

template <class DataType>
class Iterator
{
private:
      SinglyLinkedNode<DataType>* currentNode;

public:
      //Constructor
      Iterator(SinglyLinkedNode<DataType>* node = nullptr): currentNode(node){}

      //Dereference operator
      DataType& operator*()
      {
            return currentNode->getValue();
      }
```

```
        //Prefix incremenet operator
        Iterator& operator++()
        {
                if (currentNode)
                {
                        currentNode = currentNode->getNext();
                }
                return *this;
        }

        //Postfix increment operator
        Iterator operator++ (int)
        {
                Iterator temp = *this;
                ++(*this);
                return this;
        }

        //Equality check operator
        bool operator==(const Iterator& other) const
        {
                return currentNode == other.currentNode;
        }

        bool operator!=(const Iterator& other) const
        {
                return !(*this == other);
        }
};
```

//Potion.h

```
#pragma once
#include "Item.h"

class Potion : public Item
{
public:
        enum class PotionType
        {
                Health = 0,
                Strength,
                Vitality,
                Dexterity,
                Intelligence
        };

        Potion();
        Potion(string name, int level, int rarity, PotionType type, int potency);

        virtual ~Potion();

        virtual Potion* clone() const override;
        virtual string toString() const override;

        //Accessors
        PotionType getType() const;
        int getPotency() const;

        static void initNames();

        static map<PotionType, string> namesMap;
```

```
	void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;
private:
	PotionType type;
	int potency;
};
```

//Potion.cpp

```
#include "Potion.h"
//Initialize static namesMap container
map<Potion::PotionType, string> Potion::namesMap;

void Potion::initNames()
{
	namesMap[PotionType::Health] = "Potion of Healing";
	namesMap[PotionType::Strength] = "Elixir of Strength";
	namesMap[PotionType::Vitality] = "Essence of Vitality";
	namesMap[PotionType::Dexterity] = "Brew of Dexterity";
	namesMap[PotionType::Intelligence] = "Tonic of Intelligence";
}

Potion::Potion()
	: Item(), type(PotionType::Health),potency(0){}

Potion::Potion(string name, int level, int rarity, PotionType type, int potency)
	:Item(namesMap[type], level, calculateBuyValue(level,
rarity),calculateSellValue(level, rarity), rarity),
	type(type), potency(potency){}

Potion::~Potion(){}

Potion* Potion::clone() const
{
	return new Potion(*this);
}

string Potion::toString() const
{
	string typeStr;
	switch (type)
	{
	case PotionType::Health: typeStr = "Health"; break;
	case PotionType::Strength: typeStr = "Strength"; break;
	case PotionType::Vitality: typeStr = "Vitality"; break;
	case PotionType::Dexterity: typeStr = "Dexterity"; break;
	case PotionType::Intelligence: typeStr = "Intelligence"; break;
	}
	string str =
		" -------------------- \n Name: " + this->getName()
		+ " \n Description: +1 " + typeStr
		+ " \n Potency: " + to_string(this->getPotency())
		+ " \n Buy Value: " + to_string(this->getBuyValue()) + " Gold"
		+ " \n -------------------- ";


	return str;
}

Potion::PotionType Potion::getType() const
```

```
{
	return type;
}

int Potion::getPotency() const
{
	return potency;
}

void Potion::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
	sf::Text text;
	text.setFont(font);
	text.setCharacterSize(24);
	text.setFillColor(sf::Color::White);
	text.setPosition(x, y);

	std::string typeStr;
	switch (type) {
	case PotionType::Health: typeStr = "Health"; break;
	case PotionType::Strength: typeStr = "Strength"; break;
	case PotionType::Vitality: typeStr = "Vitality"; break;
	case PotionType::Dexterity: typeStr = "Dexterity"; break;
	case PotionType::Intelligence: typeStr = "Intelligence"; break;
	}

	std::stringstream ss;
	ss << "Name: " << getName() << "\n"
		<< "Type: " << typeStr << "\n"
		<< "Potency: " << potency << "\n"
		<< "Level: " << getLevel() << "\n"
		<< "Rarity: " << getRarity() << "\n";

	text.setString(ss.str());
	window.draw(text);
}
```

//Puzzle.h

```
#pragma once

#include "STLInclude.h"

class Puzzle
{
public:
	Puzzle(string fileName);
	virtual ~Puzzle();
	string getAsString();

	inline const int& getCorrectAns()const { return this->correctAnswer; }

	void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;

private:
	string question;
	vector<string> answers;
	int correctAnswer;
};
```

//Puzzle.cpp

```
#include "Puzzle.h"
```

```
Puzzle::Puzzle(string fileName)
{
	this->correctAnswer = 0;

	ifstream inFile(fileName);
	int numOfAns = 0;
	string answer = "";
	int correctAns = 0;

	if (inFile.is_open())
	{
		getline(inFile, this->question);
		inFile >> numOfAns;
		inFile.ignore();

		for (size_t i = 0; i < numOfAns; i++)
		{
			getline(inFile, answer);
			this->answers.push_back(answer);
		}


		inFile >> correctAns;
		this->correctAnswer = correctAns;
		inFile.ignore();

	}

	else
		throw("Could no open puzzle!");

	inFile.close();
}

Puzzle::~Puzzle()
{

}

string Puzzle::getAsString()
{
	string answers = "";

	for (size_t i = 0; i < this->answers.size(); i++)
	{
		answers += to_string(i) + ": " + this->answers[i] + "\n";
	}

	/*return this->question + "\n" + "\n"
		+ answers + "\n"
		+ to_string(this->correctAnswer) + "\n";*/

	return this->question + "\n" + "\n"
		+ answers + "\n";

}

void Puzzle::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
	sf::Text text;
	text.setFont(font);
	text.setCharacterSize(24);
```

```
        text.setFillColor(sf::Color::White);

        // Render the question
        text.setString(question);
        text.setPosition(x, y);
        window.draw(text);

        // Calculate the height of the text for spacing
        float textHeight = text.getLocalBounds().height + 5;

        // Render each answer
        for (size_t i = 0; i < answers.size(); ++i) {
                text.setString(std::to_string(i) + ": " + answers[i]);
                text.setPosition(x, y + (i + 1) * textHeight);  // Position each
answer below the previous
                window.draw(text);
        }
}
```

//Queue.h

```
#pragma once

#include "SinglyLinkedNode.h"
#include <stdexcept>


template <class T>
class Queue
{
private:
        SinglyLinkedNode<T>* front;
        SinglyLinkedNode<T>* rear;
        int size;

public:
        //Constructor
        Queue(): front(nullptr), rear(nullptr), size(0){}

        //Destructor
        ~Queue()
        {
                while (!isEmpty())
                {
                        dequeue();
                }
        }

        // Copy constructor
        Queue(const Queue<T>& other) : front(nullptr), rear(nullptr), size(0) {
                SinglyLinkedNode<T>* current = other.front;
                while (current != nullptr) {
                        this->enqueue(current->getValue());
                        current = current->getNext();
                }
        }

        // Copy assignment operator
        Queue<T>& operator=(const Queue<T>& other) {
                if (this != &other) { // Protect against self-assignment
                        // Clear the current queue
                        while (!this->isEmpty()) {
```

```
                        this->dequeue();
                }

                // Copy elements from the other queue
                SinglyLinkedNode<T>* current = other.front;
                while (current != nullptr) {
                        this->enqueue(current->getValue());
                        current = current->getNext();
                }
        }
        return *this;
}

//Enqueue
void enqueue(const T& value)
{
        SinglyLinkedNode<T>* newNode = new SinglyLinkedNode<T>(value,
nullptr);
        if (isEmpty())
        {
                front = rear = newNode;
        }

        else
        {
                rear->append(newNode);
                rear = rear->getNext();
        }
        size++;
}

//Dequeue
T dequeue()
{
        if (isEmpty())
        {
                throw out_of_range("Queue is empty");
        }

        T value = front->getValue();
        SinglyLinkedNode<T>* toRemove = front;
        front = front->getNext();
        delete toRemove;
        size--;

        if (isEmpty())
        {
                rear = nullptr;
        }

        return value;
}


//Check if the queue is empty
bool isEmpty() const
{
        return front == nullptr;
}

//Get the size of the the queue
int getSize() const
{
```

```
            return size;
        }

        //Peek at the front of the queue
        T& peek() const
        {
            if (isEmpty())
            {
                throw out_of_range("Queue is empty");
            }

            return front->getValue();
        }
};

//Shop.h
#pragma once

#include "Character.h"
#include "STlInclude.h"
#include "Weapon.h"
#include "Armor.h"
#include "Potion.h"

class Shop {
public:
        Shop(int playerLevel);
        ~Shop();

        void openShop(Character& character, sf::RenderWindow& window, sf::Font&
font);
        void displayItemsForSale(sf::RenderWindow& window, sf::Font& font) const;
        void purchaseItem(Character& character, int index, sf::RenderWindow&
window, sf::Font& font);
        int getItemsCount() const;

private:
        SinglyLinkedNode<Item*>* itemsForSaleHead;
        int playerLevel;

        void generateItemsForSale();
        void clearShop();
        int calculateItemPrice(const Item* item) const;

        sf::Texture ShopTexture;
        sf::Sprite ShopSprite;
};



//Shop.cpp
#include "Shop.h"

Shop::Shop(int level) : playerLevel(level), itemsForSaleHead(nullptr)
{
    srand(static_cast<unsigned>(time(0)));
    generateItemsForSale();
}

Shop::~Shop()
{
    clearShop();
```

```
}

void Shop::displayItemsForSale(sf::RenderWindow& window, sf::Font& font) const {
    int index = 1;
    float xPos = 50.0f; // Starting y-position for displaying items

    for (SinglyLinkedNode<Item*>* current = itemsForSaleHead; current != nullptr;
current = current->getNext()) {
        int price = calculateItemPrice(current->getValue());
        sf::Text itemText;
        itemText.setFont(font);
        itemText.setString(std::to_string(index) + ": " + current->getValue()-
>toString());
        itemText.setCharacterSize(24);
        itemText.setFillColor(sf::Color::White);
        itemText.setPosition(xPos, 650.f); // Adjusted position for better
visibility

        window.draw(itemText);
        xPos += 320.0f; // Increased space between lines for clarity
        index++;
    }
}


int Shop::getItemsCount() const {
    int count = 0;
    for (SinglyLinkedNode<Item*>* current = itemsForSaleHead; current != nullptr;
current = current->getNext()) {
        count++;
    }
    return count;
}


void Shop::openShop(Character& character, sf::RenderWindow& window, sf::Font&
font) {
    bool shopOpen = true;

    // Load and set the shop background
    if (!ShopTexture.loadFromFile("Assets/IMG/shop2.png")) {
        std::cerr << "Failed to load main menu background texture" << std::endl;
    }
    ShopSprite.setTexture(ShopTexture);
    ShopSprite.setPosition(0.f, 0.f);


    // Display a welcome message
    sf::Text welcomeTextShop("Good day, brave adventurer! You've come to the
right place for all your battle needs. \nHere at my shop, we have a fine
selection of armor, weapons, \nand potions to aid you on your quests", font, 24);
    welcomeTextShop.setFillColor(sf::Color::White);
    welcomeTextShop.setPosition(50.f, 525.f);


    // Set up input prompt and user input text
    sf::Text inputPrompt("Enter item number to purchase or '0' to exit:", font,
24);
    inputPrompt.setFillColor(sf::Color::White);
    inputPrompt.setPosition(50.f, 850.f);

    sf::Text userInput("", font, 24);
    userInput.setFillColor(sf::Color::White);
```

```
    userInput.setPosition(750.f, 850.f);

    std::string userEnteredText;

    while (shopOpen) {
        // Event handling
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
                shopOpen = false;
            }

            if (event.type == sf::Event::TextEntered) {
                if (event.text.unicode < 128) {
                    userEnteredText += static_cast<char>(event.text.unicode);
                    userInput.setString(userEnteredText);
                }
            }

            if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Enter) {
                    int itemIndex = std::stoi(userEnteredText) - 1;
                    if (itemIndex >= 0 && itemIndex < getItemsCount()) {
                        purchaseItem(character, itemIndex, window, font);
                    }
                    else if (itemIndex == -1) {
                        shopOpen = false; // Exit shop if 0 is entered
                    }
                    userEnteredText = ""; // Clear the entered text
                }
                else if (event.key.code == sf::Keyboard::BackSpace &&
!userEnteredText.empty()) {
                    userEnteredText.pop_back();
                    userInput.setString(userEnteredText);
                }
            }
        }

        // Rendering
        window.clear();
        window.draw(ShopSprite);
        window.draw(welcomeTextShop); // Draw the welcomeTextShop
        displayItemsForSale(window, font);  // Display the items for sale
        window.draw(inputPrompt);
        window.draw(userInput);
        window.display();
    }
}

void Shop::purchaseItem(Character& character, int index, sf::RenderWindow&
window, sf::Font& font) {
    Iterator<Item*> it = Iterator<Item*>(itemsForSaleHead);

    for (int i = 0; i < index; ++i) {
        ++it;
    }

    sf::Text feedbackText;
    feedbackText.setFont(font);
    feedbackText.setCharacterSize(24);
    feedbackText.setPosition(100.f, 950.f);
```

```
    if (*it) {
        int price = calculateItemPrice(*it);
        if (character.getGold() >= price) {
            character.payGold(price);
            character.addItem(**it);
            feedbackText.setString("You purchased: " + (*it)->getName() + ". It
has been added to your inventory.");
        }
        else {
            feedbackText.setString("Not enough gold!");
        }
    }
    else {
        feedbackText.setString("Invalid item selection.");
    }

    // Display the feedback for a short duration
    window.clear();
    displayItemsForSale(window, font); // Redraw the items for sale
    window.draw(ShopSprite); // Draw the welcomeTextShop
    window.draw(feedbackText);
    window.display();
    sf::sleep(sf::seconds(2)); // Hold the screen for 2 seconds to allow the user
to read the message
}


void Shop::generateItemsForSale() {
    clearShop();

    // Random Potion
    Potion::PotionType randomPotionType = static_cast<Potion::PotionType>(rand()
% static_cast<int>(Potion::PotionType::Intelligence) + 1);
    string potionName = Potion::namesMap[randomPotionType];
    itemsForSaleHead = new SinglyLinkedNode<Item*>(new Potion(potionName,
playerLevel, rand() % 5, randomPotionType, rand() % playerLevel * 10 + 10),
nullptr);

    // Random Armor
    itemsForSaleHead->setNext(new SinglyLinkedNode<Item*>(new Armor(playerLevel,
rand() % 5), nullptr));

    // Random Weapon
    itemsForSaleHead->getNext()->setNext(new SinglyLinkedNode<Item*>(new
Weapon(playerLevel, rand() % 5), nullptr));
}


void Shop::clearShop()
{
    while (itemsForSaleHead)
    {
        SinglyLinkedNode<Item*>* temp = itemsForSaleHead;
        itemsForSaleHead = itemsForSaleHead->getNext();
        delete temp->getValue(); //Delete item
        delete temp; //Delete node
    }
}

int Shop::calculateItemPrice(const Item* item) const {
    if (item)
    {
        int rarityValue = 0;
```

```
        string rarityStr = item->getRarity();

        if (rarityStr == "Common") {
            rarityValue = COMMON;
        }
        else if (rarityStr == "Uncommon") {
            rarityValue = UNCOMMON;
        }
        else if (rarityStr == "Rare") {
            rarityValue = RARE;
        }
        else if (rarityStr == "Epic") {
            rarityValue = EPIC;
        }
        else if (rarityStr == "Legendary") {
            rarityValue = LEGENDARY;
        }

        return item->getBuyValue();
    }

    return 0;
}
```

//SinglyLinkedNode.h

```
#pragma once
#include <stdexcept>

template<class DataType>
class SinglyLinkedNode {
public:
    typedef SinglyLinkedNode<DataType> Node;

private:
    DataType fValue;
    Node* fNext;

public:
    // Corrected constructor
    SinglyLinkedNode(const DataType& aValue, SinglyLinkedNode* aNext)
        : fValue(aValue), fNext(aNext) {}

    // Methods to manipulate the list
    void append(Node* aNode);
    void remove();
    int size() const;
    void setNext(Node* nextNode);
    DataType* get(int index);

    // Accessors
    DataType& getValue() { return fValue; }
    Node* getNext() const { return fNext; }

    // Check if the node is the end of the list
    bool isEnd() const { return fNext == nullptr; }
};

template<class DataType>
void SinglyLinkedNode<DataType>::append(Node* aNode) {
    if (!aNode) return;
```

```
    Node* current = this;
    while (current->fNext != nullptr) {
        current = current->fNext;
    }
    current->fNext = aNode;
};

template<class DataType>
void SinglyLinkedNode<DataType>::remove() {
    if (!fNext) return;

    Node* temp = fNext;
    fNext = fNext->fNext;
    delete temp;
};

template<class DataType>
int SinglyLinkedNode<DataType>::size() const {
    int count = 0;
    const Node* current = this;
    while (current != nullptr) {
        count++;
        current = current->fNext;
    }
    return count;
};

template<class DataType>
DataType* SinglyLinkedNode<DataType>::get(int index) {
    Node* current = this;
    int currentIndex = 0;

    while (current != nullptr && currentIndex < index) {
        current = current->fNext;
        currentIndex++;
    }

    if (current == nullptr) {
        return nullptr; // Index out of range
    }

    return &(current->fValue);
};

template<class DataType>
void SinglyLinkedNode<DataType>::setNext(Node* nextNode) {
    fNext = nextNode;
};

template<class DataType>
int countNodes(const SinglyLinkedNode<DataType>* head) {
    int count = 0;
    const SinglyLinkedNode<DataType>* currentNode = head;
    while (currentNode != nullptr) {
        count++;
        currentNode = currentNode->getNext();
    }
    return count;
};
```

//STLInclude.h

```cpp
#pragma once
#include<ctime>
#include <vector>
#include <sstream>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <set>
#include <thread>
#include <chrono>
#include <map>
#include <memory>
#include "SinglyLinkedNode.h"
#include "DoublyLinkedNode.h"
#include "DynamicArray.h"
#include "DynamicStack.h"
#include "Iterator.h"
#include "Queue.h"
#include "BTree.h"
#include <stdlib.h>
#include <string>
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
using namespace std;
```

//Weapon.h

```cpp
#pragma once

#include "STLInclude.h"
#include "Item.h"


class Weapon:public Item
{
public:
	Weapon();
	Weapon(int level, int rarity);
	Weapon(int damageMin,
		int damageMax,
		string name,
		int level,
		int buyValue,
		int sellValue,
		int rarity);

	virtual ~Weapon();

	//Pure virtual
	virtual Weapon* clone() const override;


	//Functions
	string toString() const;

	//Accessors
	inline int getDamageMin() const { return this->damageMin; }
	inline int getDamageMax() const { return this->damageMax; }

	//Modifiers
```

```
	static dynaArr<string> names;
	static void initNames();

	void render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const;

private:
	int damageMin;
	int damageMax;
};
```

//Weapon.cpp

```
#include "Weapon.h"

dynaArr<string> Weapon::names;

void Weapon::initNames()
{
	Weapon::names.push("Combat Knife");
	Weapon::names.push("Great Sword");
	Weapon::names.push("Axe");
	Weapon::names.push("PickAxe");
	Weapon::names.push("WarHammer");
	Weapon::names.push("Katana");
}

Weapon::Weapon()
	:Item("NONE", 0, 0, 0, COMMON) { // Default constructor
	this->damageMax = 0;
	this->damageMin = 0;
}

Weapon::Weapon(int level, int rarity)
	:Item(Weapon::names[rand() % Weapon::names.size()],
		level,
		calculateBuyValue(level, rarity),
		calculateSellValue(level, rarity),
		rarity)
{
	this->damageMax = rand() % level * (rarity + 1);
	this->damageMax += (rarity + 1) * 5;
	if (rarity == 3)
		this->damageMax += level * 5;
	else if (rarity == 4)
		this->damageMax += level * 10;
	this->damageMin = this->damageMax / 2;
}

Weapon::Weapon(int damageMin, int damageMax, string name, int level, int
buyValue, int sellValue, int rarity)
	: Item(name, level, buyValue, sellValue, rarity) {
	this->damageMin = damageMin;
	this->damageMax = damageMax;
}

Weapon::~Weapon()
{

}
```

```cpp
Weapon* Weapon::clone() const
{
	return new Weapon(*this);
}

string Weapon::toString() const
{
	string str =
		" -------------------- \n Name: " + this->getName()
		+ " \n Level: " + to_string(this->getLevel())
		+ " \n Rarity: " + this->getRarity()
		+ " \n Damage: " + to_string(this->damageMin) + " - " +
to_string(this->damageMax)
		+ " \n Buy Value: " + to_string(this->getBuyValue()) + " Gold"
		+ " \n -------------------- ";

	return str;
}

void Weapon::render(sf::RenderWindow& window, sf::Font& font, float x, float y)
const {
	sf::Text text;
	text.setFont(font);
	text.setCharacterSize(24);
	text.setFillColor(sf::Color::White);
	text.setPosition(x, y);

	// Create the string representation of the weapon
	std::string weaponInfo = toString();

	// Set the string to the text object
	text.setString(weaponInfo);

	// Draw the text to the window
	window.draw(text);
}
```