**Effectiveness & Approach**

The design and intent of Program 3 was to create a monster battle program that utilized object oriented programming as well as operator overloading. This goal was achieved utilizing single inheritance with a Base Class named Monster and four Derived Classes, Dragon, Vampire, Werewolf, and Zombie. The Base Class would house the majority of the monster’s attributes while the derived would focus on what made that type of monster unique among its class. The execution of this program was largely successful, and I feel that the importance of operator overloading was captured within the program.

My initial design was to have a Base Class, four derived classes that represented the different Monsters, as well as 2 different node classes for the two different data structures and a list class to manage them and utilize dynamic binding for the Monsters. Throughout my implementation, it became apparent that the focus of the assignment was becoming on the data structures and not on the operator overloading or monsters themselves. Keeping operator overloading in mind as my main goal, I shifted the paradigm of this program away from dynamic binding and into the client program as my focal point to enact operator overloading.

This shift took the form of not housing my actual monster objects inside of the Binary Search Tree but using the BST to represent available monsters as a copy of the real objects housed inside of the client program. This enabled me to separate the data structure implementation and the operator overloading implementation and increase the effectiveness of both. This was massively beneficial for my program’s operator overloading functionality and expedited its ability to facilitate a meaningful battle simulation for the user to engage with. In terms of cohesion, this shift enabled the client program to create and manage the monster object’s effectively, while simultaneously using the list class to create and manage the data structures as well. By separating the two major components of the assignment, my BST and the array of DLL were implemented utilizing potions of the same data from the monsters themselves.

**Operator Overloading**

My approach with operator overloading was to implement it with the monster’s in mind as the primary focus. The implementation focused largely on the base class data within the monster and coordinating that data with the monster battle simulation. The arithmetic operators (+,+=,=) were utilized the to alter the attributes of the monsters and simulate upgrading their health, defense, and strength. The relational and equality operators were utilized to compare the different monsters within the battle simulation to determine which monster was the winner and utilize the arithmetic operators to alter their stats accordingly.

Additionally, another operator that was overloaded was the output operator (<<). Overloading this operator enabled the program to display the data from any point within the client program and enhanced the monster battle simulation by displaying the monsters attributed before and after with battles without the need of a display function. Due to most of the monster’s attributes existing in the base class, some operators were overloaded in both the derived and base class, while others were overloaded in the derived class and would call a base class function.

**Object Oriented Programming**

In terms of object oriented programming, I believe that my program was successful in most portions of the assignment. The object oriented portion of my program was different from previous assignments, as this time it came from solely the monster base class and the derived classes and there was no implementation within the data structures. The OOP that was implemented with my monsters was effective and enabled my derived monsters to share and manipulate a lot of the commonalities between the monsters.

Given that operator overloading was the focus of the assignment, being able to enact it on not only the derived class, but also the base class was pivotal. Utilizing single inheritance enabled me to manipulate the derived class information as needed, but also opened the door for the base class data to be manipulated as well. This resulted in all my operators having the dual functionality of changing both the base and derived classes.

I believe that an approach that incorporated the monsters into the data structures with dynamic binding would have resulted in a better result than my design in terms of OOP. This would have enabled the program to not have differentiate between what type of monster it was and use them as needed. I do believe this would have been at the sacrifice of operator overloading and given that this was the focus of the assignment, I am confident in my design to separate the two.

**Data Structures**

The two data structures within this assignment were a Binary Search Tree as well as an Array of Doubly Linked List. The Binary Search Tree housed duplicate information for each of the monsters and upon traversal, the nodes acted as a quick reference for the user to see all available monsters within the program and their pertinent information. The Array of Doubly Linked Lists acted as the storage for the monster’s abilities. Each index of the array represented a different derived monster class. Each of the nodes within each given DLL housed a specific monster’s abilities. This data structure cataloged and separated each of the monster’s ability information and provided the information about the monsters to the user as needed. Both data structures were implemented in their entirety, with display one type, display all, remove, and remove all as primary features.

**Efficiency**

The efficiency of my program was largely positive. Categorizing the data structures as well as having individual monster arrays enabled quick access to the data housed within the program in most cases. The array of DLL was beneficial by being able to only need to traverse one of the desired indices as needed, but also detrimental as a concurrent search was needed in some functionalities to find the correct data. I believe my data structures and monster battle simulation work fluidly. They create, manage, and destroy the content of monsters efficiently using recursion and loops. I was able to complete the project with no memory leaks and without cutting any corners for the programs design.

**Conclusion**

I believe that my approach and implementation of the programming assignment was successful. With the main goal of operator overloading, my program made use of the hierarchy and specific class roles to implement it as well as engage with the user of the program. I would have liked to have a stronger amount of cohesion within my data structure and my class objects but implementing the monsters within the client program resulted in a beneficial result for both operator overloading as well as some portions of object oriented programming.