Sean Mulholland ID No: 932652783 mulholls@onid.oregonstate.edu

Assignment 4 Design Document-

Design:

Fortunately my Assignment 3 was already designed in a way I could easily add on to it in an incremental approach. My design choice was to overhaul the single character creation into a function that creates a queue of Creature*'s. By using a queue, it should readily build a "lineup" of creatures that can be manipulated easily. Because they're pointers, and I took advantage of this in Assignment 3, I can use polymorphism to very easily set up a simple loop of a set length call my creature creation function from Assignment 3 and simply "push" it into the team queue. Once the queue is created, I can just return it to the calling function.

The next challenge was adding a replenish health function. I decided to take a fairly simple route. I plan on adding a variable for max health that each creature's constructor will initiate to the original strength value (max HP). From there, I can write a simple member function replenishHealth() that can create a Dice object with a number of sides equivalent to the maxHealth value. A simple roll of the dice between turns can generate a health value to replenish, and a comparison will prevent the creature from over healing between turns.

For combat, I already had a good 1 v 1 combat function. I plan to use this function again in a while loop that repeats while the queues holding the teams both have a size > 0. At the end of combat, I will utilize a stack container to receive the loser. That way, when a creature is defeated they go into the "loser pile" in the order that they lose. I can then compare the creature in the stack to the creatures at the front of the queues and the team with the loser on it will simply have it popped. The winning team will have the creature moved to the back of the queue calling the replenish health function to fight again. This is great because by using pointers at the end of each battle there will only be one unique pointer for each creature in their respective containers.

Once a team has won, they may have more than 1 creature left in their team. I decided to determine inter team rank (doesn't REALLY matter that much), they would just battle one another. The other route I thought about was storing a points variable in the creature instance. I decided against that simple because I didn't want to write another member data point. The inter-team battles will not affect any points between teams.

This leads to the point system. I decided to go on a team based point system. Each fight the winning team gets 1 point for the first 'n' fights, where n is the max number of creatures on the team. If there are battles beyond this amount (creatures fighting again) they will receive 2 points for their victory. For instance, if there are 5 creatures on each team the first 5 battles are worth 1 point. If one team doesn't dominate the first round, there can be up to a total of 9 battles which means that the next 4 battles are worth 2 points. The winner is determined by the sum of these points. The rank should readily reflect the point system too. If it comes down to the last creatures, the points could be very close if the battles trade off (team1 = 1 + 1 + 1 + 2 + 2 = 7, team2 = 1 + 1 + 2 + 2 = 6) or if one team loses the first four battles, but wins the rest it could be more spread (team1 = 1 + 1 + 1 +1 + 1 = 5, team2 = 2 + 2 + 2 + 2 = 8). If one team wins all the battles, they should come up dominant (team1 = 5, team2 = 0) but the team battle will determine the winning character. For instance, a blue man is particularly strong. If he fought first, and wiped the enemies and then fought a bunch of weaklings on his own team he would be a clear victor and this would be reflected in the points and rankings.

All these designs should readily fit into 2-3 new functions that should work with the existing assignment 3 code I'm basing this off.

Test Plan:

The test plan will be pretty basic. Since I'm using the same combat function, I already know it's tested. I will however need to test a variety of team sizes to make sure that the ranks are appropriate. I don't think that this will be an issue because I plan on using a counter variable in a loop and the queue should handle the memory management.

For the replenish health function, I just need to check to see if it is working appropriately. It's based on the dice that I already tested so I have to ensure that the creatures will only have their max health and not heal too much. I can test this pretty easily by outputting the creatures health before and after the healing process. It should work without a hitch because the dice works already, and a simple comparison between the maxHealth variable and the current strength variable can be assigned to maxHealth's value if it is too high.

The point system is also pretty straight forward. I'm using a 2x1 integer array and just adding the current points variable to the array. I don't expect any issues with the points process. To check the ranking I plan on just explicitly outputting the rank in the stack to ensure it matches to the battle outcomes. It's a pretty simple process, so it should work pretty straightforwardly.

My biggest concern is memory management. I've just finally wrapped my head around the actual issue of using pointers and dynamic allocation and realized how important it is to properly delete the instances when they're done and not earlier. Because of this, I plan to use a tool that I just learned about called valgrind that can check to ensure that the allocations match with the memory releases. I've never used it before, but being able to just check that the allocations match the releases is good enough.

Reflection:

I had no issues with this assignments basic functionality right off the bat. All the code I wrote last week was easily adaptable and already was tested and worked. I just added each new chunk of code and then would test the new function until I'm sure it worked. Incremental development was my friend in this process as there were a number of variables I needed to recreate or pass as I added the functions so a simple initialization or assignment fixed most errors I had because I had a compile-time error.

My biggest fault, and biggest learning experience, was with the memory management involved with the creature instances. I ran into the problem of having 2 pointers pointing to the same instance. I didn't fully understand when I started what the delete function did and ran into some accidently deleted references. Fortunately the locations where these problems were occurring were easily identified using the incremental process. Segmentation faults were pretty upsetting for a bit, but I got through them and learned so much from them during this process. It helped during the final testing to use valgrind and count the allocations and deallocations by the final program.

Everything else worked really easily, although after reviewing last week's code I would have totally rehashed a number of functions if I had more time. Some of the development was good code, but it didn't match to the existing code as well. Instead of basing the program off the old code with it's old design structure I should have created a new design structure and used the old code to meet the new requirements. Fortunately it all worked in the end because each function served a big purpose and was fairly easily adaptable.