Project 4

For project 4, we were tasked with adding additional features to our fantasy combat game from project 3. Our revised fantasy combat game now consisted of running a team of fighters from two opposing teams. A user would designate the number of players that would play on each team. The number of fighters that could play on each team was limited to ten. Afterwards, the user would assign a character and nickname to each player on the team. After the players were assigned on each team, the head players would then fight each other in the first round. The winner of the game would go to the back of the lineup after getting a percentage of damage restored, and the loser would go to the loser stack. At the end of the game, the winner of the team would be announced on the screen and the user would be presented the option to display the losers from the loser stack. The user would then be prompted back to the menu, where he or she would be able to play again or exit.

The revised fantasy combat game design had a loser stack and line up queue. The loser stack was a top bottom structure that contained a next, previous, and creature pointer in each of its nodes. The loser class contained a displayLoser() function, which displayed the losers at the end of the game, and an addLoser() function, which added a creature to the loser stack at the end of each round. The line-up queue existed in the Queue class. The Queue class contained a moveQueue() function that moved the winner to the end of the line, a removeFront() function that removed the creature at the head of the lineup, and an isEmpty() function that returned true if there was nothing in the queue and false otherwise. The Battleground class also had a teamPlay() function and a revised menu() function. The team play function now orchestrated the combats between two teams instead of two players. The players at the head of both lineups would play until one of them died. Afterwards, the winner

would get strength restored and move to the back of the line and the loser would be extracted from the line and moved to the loser stack. The winner was determined by whichever team solely remained in either lineup. This meant that the two teams would continuously fight until all the members in either team A or team B were sent to the loser stack. The winning team was then announced on screen and the loser was given a choice to print the losers. The menu then displayed to give the option to play again or exit.

I initially encountered many problems when creating this game that forced me to make several changes. The first problem I confronted dealt with freeing memory leaks. I thought my original destructors were robust enough to be able to free all memory, but after more testing and with the help of my teacher's assistant, I realized that the memory could not be properly freed until the memory that the creature pointers pointed to within each node was freed as well. I also dealt with several segmentation faults that I had to bypass. The first was caused by a misplaced if statement within my while loop in the teamPlay() function that I resolved by placing at the end of the while loop in my function. The second was caused by the program deleting team A, team B, and lose pointers in the menu() function after the pointers were initialized in the constructors. This was problematic since each future iteration of the program would continue to delete these pointers, while the pointers themselves were created once. I resolved this by taking the contents of my constructors, which contained the dynamically allocated pointers, and placed them at the start of my menu function. Another reason why I was getting segmentation faults was because my destructor was missing an if statement. My previous destructor, checked for whether there were no nodes, a single node, or multiple nodes in my structure. This presented an issue because if there were no nodes in my destructor, yet it was checking the middle if statement condition if (front==back), my program would fault because it was checking for something that did not exist.

Thus, I resolved this issue by making two main if statements within my destructors. The first if statement executed if there was nothing in my structures, and the second if statement, which housed logic for when there more than one nodes in my structure, only executed if front was not equal to NULL. This way, my destructors no longer prematurely checked to see if there were nodes in cases where the line was empty. After I correctly placed the if statements in my teamplay() function, initialized my pointers outside of my battleground constructors, and only executed my destructor delete logic after confirming that at least one node existed in the structure, I was able to resolve all segmentation faults

Another problem that I struggled with was fixing an infinite loop conceived in my teamplay() function. This problem was evasive because my logic in the teamplay() function was correct. After more analysis, I realized later that the infinite loop existed because there was a fault in my getFront() function in my Queue class. My getFront() function did not initially account for a situation where there was nothing in the lineup. This was problematic because a large part of my teamplay() function was based on interacting with the head of both teams lineups. Thus, I resolved this problem by having the getFront() function return zero whenever there was nothing in the lineup. This enabled my if statements within my battleground functions, which relied on my getFront() function, to no longer crash the program when one of the lineups were empty.

Designing this program taught me more about memory leaks, segmentation faults, and pointers. I learned how to test for memory leaks extensively because of this program. I learning that using valgrind with leak error check is important to help narrow one's memory leaks. I also learned that it is important to test a larger program piece by piece to ensure that the smaller components of the program are working correctly and to find mistakes more

readily before the program gets larger and more complicated. For this assignment, I tested my Queue class by making a separate test main to just check the functions within the Queue class to ensure that they were behaving as expected. This was the first time I made a custom main for the purposes of testing one facet of my program. Doing so helped me isolate the problem with the getFront() function and realized that it crashed whenever there was nothing in the lineup.

I also learned how important it is to think about the sequence of your program and to anticipate how it will behave after each iteration. For example, when I allocated memory for my team and loser pointers in my battleground constructor, I knew that this would allow my program to work on the first run. However, after looping my program, I realized that allocating memory for these pointers in the constructor meant that there would be segmentation faults since they would be deleted on each iteration, but initialized only once. Thus, it was important for me to thorough imagine the flow of the program and how it would behave upon each iteration. Lastly, in this project, creating stack and queue structures helped me gained more experience with pointers. I drew out each of my functions from the loser stack and lineup queue on a piece of paper to visualize which pointers would point to the nodes in the structure. In addition, my teamplay() function heavily relied on pointers so I was able to gain more experience using creating and using different pointer variations. For example, I learned that doing teamA.getFront()->getStrength() was equivocal to doing teamA->getFront()->getStrength() after creating Queue *teamA and teamA= new Queue() in the battleground class.

Project gave me the chance to learn how to troubleshoot a larger program and how to integrated stack and queue structures in my program. Although it presented many challenges,

I learned how to test issues more granularly and how to successfully integrate new classes and structures with already existing code.

Test Plan

I tested this project by testing all instances of characters multiple times and by testing the functions used in the program to ensure they behaved correctly.

Test Case	Game	Team Size	Rounds	Characters In Team A	Characters In Team B	Game Winner	Team Winner	Winner New Strength	Driver Functions
test characters	1	5	46	1. Harry Potter A1	1. Harry Potter B1,	1. Harry Potter B1,	B	5	teamPlay()
test characters	1	5	4	2. Medusa A2	2. Bluemen B2,	2. Bluemen B2,	B	12	teamPlay()
test characters	1	5	9	3.Bluemen BMA	3. Barbarian B3,	3.Bluemen BMA	A	11	teamPlay()
test characters	1	5	12	4. Barbarian A4	4. Vampire Vam4,	4. Vampire Vam4,	B	15	teamPlay()
test characters	1	5	22	5. Vampire A5	5. Vampire Vam 5	5. Vampire Vam 5	B	12	teamPlay()
test characters	1	5	12	3.Bluemen BMA	1. Harry Potter B1,	3.Bluemen BMA	A	11	teamPlay()
test characters	1	5	6	3.Bluemen BMA	2. Bluemen B2,	2. Bluemen B2,	B	12	teamPlay()
test characters	2	4	29	1. Harry Potter 1	1.Vampire 1	1. Harry Potter 1	A	13	teamPlay()
test characters	2	4	1	2. Blue Men 2	2. Blue Men 2	2. Blue Men 2	B	6	teamPlay()
test characters	2	4	6	3. Barbarian 3	3. Blue Men 3	3. Blue Men 3	B	12	teamPlay()
test characters	2	4	12	4. Vampire 4	4. Vampire 4	4. Vampire 4	B	9	teamPlay()

test characters	2	4	8	1. Harry Potter 1	2. Blue Men 2	2. Blue Men 2	B	9	teamPlay()
test characters	3	1	5	1. Medusa 1	2.Medusa 2	1. Medusa 1	A	7	teamPlay()
test characters	4	5	55	Harry Potter AI	Harry Potter B1	Harry Potter AI	A	6	teamPlay()
test characters	4	5	9	Medusa A2	Medusa B2	Medusa A2	A	4	teamPlay()
test characters	4	5	8	Blue Men A3	Bluemen B3	Blue Men A3	B	12	teamPlay()
test characters	4	5	25	Barbarian A4	Barbarian B4	Barbarian A4	A	7	teamPlay()
test characters	4	5	26	Vampire A5	Vampire B5	Vampire B5	B	12	teamPlay()
test characters	4	5	15	Harry Potter AI	Bluemen B3	Bluemen B3	B	12	teamPlay()
test characters	4	5	14	Medusa A2	Vampire B5	Vampire B5	B	11	teamPlay()
test characters	4	5	10	Barbarian A4	Bluemen B3	Bluemen B3	B	12	teamPlay()

Test Case	Driver Functions	Observed Outcome
menu() loops correctly (test 5x)	menu(), outerMenu()	Menu loops after end of each game.
can fight up to 10 players	integerInputValidation()	Up to 10 players can fight.
round score works properly	teamPlay()	Yes; in first test fight, team B won 5 rounds, team A won 2 rounds. Team B was the official winner declared
display the contents of the loser pile, i.e. print them out in order with the loser of the first round displayed last.	displayLoser()	Loser pile displays corrects with the loser of the first round displayed last
does the new restored strengths works as expected?	restoreStrength()	Works as expected. The restored strength is accurately used when the winners combat again. BluMen strength at end of the game 11. Damage incurred: 0. New strength: 11

the head of each line up play as expected	getFront() function	Works as expected. The line up orders are aligned.
test one player on each team	teamPlay()	Works as expected.
The winner gets put at the back of her/his team's lineup.	moveQueue() function	Works as expected
display which type of creatures fought and which won on screen	teamPlay(), finalWinner()	Works as expected. Final game outcome printed after each game.
display final total points for each team	teamPlay(), finalWinner()	Works as expected. Final game outcome printed after each game.
You must display which team won the tournament.	teamPlay(), finalWinner()	Works as expected. Final game outcome printed after each game.