Name: Christopher Matian
OID: 933308644
Date: 11/18/2017

Initial Design Strategy and File Structure

The initial design of my FCS project built on top of what I had already created back in project 3. I made plans to expand the combat function I had built, and I also planned to create several lists (2 queues and 1 stack) for storing the various objects that I would need for the game.

I began by mapping out the files I would likely need shown below:

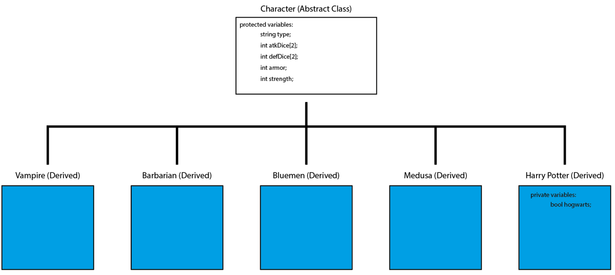
Header File	Function File
	main.cpp
validation.hpp	validation.cpp
menu.hpp	menu.cpp
combat.hpp	combat.cpp
character.hpp	character.cpp
vampire.hpp	vampire.cpp
barbarian.hpp	barbarian.cpp
bluemen.hpp	bluemen.cpp
medusa.hpp	medusa.cpp
harrypotter.hpp	harrypotter.cpp
queue.hpp	queue.cpp
stack.hpp	stack.hpp
selection.hpp	selection.hpp

One of the new additions I made to the file group was the queue, stack, and selection files. Queue and stack are both relatively self-explanatory and I ended up using the same code I had used in lab 6 and 7. The minor difference was I had to make the pop() function return a value rather than just function as a void function. I needed the pop function to return the character so that I could then use the character and display information about it.

The selection files were created in order to store the function prototype and the function itself for creating a new character. I designed it so that it would return a Character pointer based on what the user selected. I also made it so that the character menu was displayed each time the function was called. This helped clear up the overhead because I didn't need to clutter up my main function with showMenu() function calls.

Simple Class Hierarchy Diagram

The only thing that really changed in the class hierarchy was that I added two additional protected variables to the base class. The variables I added were name and max. The name was set by calling a function called setName(), and the max was a variable that kept track of the max strength of the character. This was used for the recovery function.



Design Changes, Problems, and Final Strategy

So lots of problems and changes this time around. I'll break them up into sections because it's easier to format that way:

- Recursive function changed in combat function
Initially, I had a recursive function that I was using to simulate combat between two opponents. The problem I ran into with this methodology in project 4 was that I couldn't figure out how to make the recursive function and keep track of multiple combatants. I ultimately decided on scrapping it and moving to a more primitive yet functional combat simulator that checked the status of each character, and also determined when one of the lists had run out of fighters.

-Memory Leaks
The second issue I started bumping into was memory leaks caused by overwriting the Character pointer I had created and losing the memory address in the heap. I resolved this issue by creating two vectors (one for each team) and pushing the characters into that each time the tournament ran.

```
for (int i = 0; i < charPerTeam; i++) {
                    cout << "Selecting Character " << i + 1 << " for team two." << endl;

                    c2 = charSelect("Team Two");

                    c2->setTeam("Team Two");

                    v2.push_back(c2);


                    team2.push(v2[i]);

                }
```

This is a block of code which illustrates what I was attempting to do. I then issued the following line of code:

```
for(int i = 0; i < charPerTeam; i++) {

                    // Delete each vector index
                    delete v1[i];
                    delete v2[i];

                }
                // Clear the Vectors
                v1.clear();
                v2.clear();
```

at the bottom of the game loop. This refreshed the vectors and deleted the stored memory addresses. For good measure, I had to issue .clear() on the vectors afterwards in order to clear up some strange, pedantic errors when running valgrind. I understand that my solution isn't particularly ideal, but it solved the problem and managed to get the game leak free.

-Remove Function Creation
The other addition I had to make was a removal function for my queue lists. For some reason, when a team completely dominated (won 3/3 rounds of combat for example) the pop function was storing the value and not removing it. I created the remove function (i.e. copy/pasted from my older labs) and adapted it so that when the tournament had no more participants, it would clean out the lists until there were only null pointers. This had the added advantage of cleaning up the game so that the older group wasn't "floating" into the new team upon creation.

```
// Clear nodes from the team 2 list
          while (team1.head != nullptr) {
              team1.remove();
          }
```

- Stack Overflow Issues
When I was testing the program in valgrind I ran into this stack overflow issue which would crop up the moment I tried to issue the recovery function. No matter what I did, the first time the recovery function was called program would prematurely run out of memory. Somehow I was hitting the 8mb cap that valgrind has for testing applications which didn't really make sense.

I ended up tracking it down to the recovery function because I noticed that the logs would immediately crash the moment that function was called. Additionally, it was reference the random_device I was using for generating a random number in the function (mersenne twister). To resolve the issue, I scrapped the mersenne twister engine and went back to rand() in order to get the function to work and no longer cause a stack overflow.

Test Cases:

Test validateBounds function for bad inputs such as negative values and letters. (Test 1)	-10	validateBounds(intent);	Should catch the bad submission and query the user again.	**Please enter a value that corresponds with the menu item above.**
Test validateBounds function for bad inputs such as negative values and letters. (Test 2)	e	validateBounds(intent);	Should catch the bad submission and query the user again.	**Please enter a value that corresponds with the menu item above.**
Test validateBounds function for bad inputs such as negative values and letters. (Test 3)	eeee	validateBounds(intent);	Should catch the bad submission and query the user again.	**Please enter a value that corresponds with the menu item above. (x4)**
Simulate battle between 3 vampires and 3 barbarians		combat()	Vampires should win the combat most of the time.	Vampires win combat.
Simulate battle between 4 vampires and 4 bluemen		combat()	Bluemen should win combat.	Bluemen win combat - noted that blue men weren't recovering dice when recovering health
Test to make sure that bluesmen are restoring the proper amount of dice.		combat()	Bluemen defense dice are incremented appropriately.	Bluesmen defense dice are properly re-added when they recover health.
Simulate battle between 3 vampires and 3 medusas		combat()	Vampires should win the combat	Medusa wins the combat
Stress test and repeat the above test case again.		combat()	same as above	Vampires win combat
Stress test and repeat the above test case again		combat()	same as above	Vampires win combat
Simulate battle between 3 vampires and 3 harry potters		combat()	Vampires should win	Vampires win the tournament 3-1.
Lists are being cleared each time the tournament ends		print() and remove()	List should be empty when printed after calling the remove function on the while loop	List is empty and contains only nullptrs.
Recovery function is running properly and restoring health no greater than the maximum health of the character.		recovery()	Recovery function should restore health to the winner of each combat.	Health is restored to each character but the value is exceeding the maximum health threshold.
Check for memory leaks after running the tournament several times		Run Game	No memory leaks	Several blocks are being lost during transition to the next game.