Design

1. QUEUE class
    a. Purpose: Class for the linked-list containers
        i. Data Members
            1. struct Node
                a. Character *fighterPtr
                b. Node *prev
                c. Node **next
            2. Node *aliveHead
            3. Node *deadHead
        ii. Key Functions
            1. Queue (constructor)
            2. ~Queue (destructor)
                a. go through aliveHead
                b. go through deadHead
            3. bool isAliveEmpty
            4. bool isDeadEmpty
            5. void addBack
            6. void addFront
            7. void removeFront
            8. void printAliveQueue
            9. void printDeadQueue
            10. int getNum
2. CHARACTER
    a. Abstract class
        i. Data members (protected)
            1. PlayerType
            2. Name (string)
            3. attackDie
            4. numAttackDie
            5. defenseDie
            6. numDefenseDie
            7. armor
            8. strength
            9. strengthTotal
        ii. Key functions
            1. Attack (virtual) - return int
            2. Defense (virtual) – return int
            3. getArmor
            4. getStrength
            5. setStrength
            6. randomNumberGenerator
            7. recovery – roll 1 6-sided die (e.g. 1 = 10% strength recovered)
            8. getName

9. setName

3. BARBARIAN
    a. Character subclass
        i. Data members
            1. numAttackDie = 2
            2. attackDie = 6 sided
            3. numDefenseDie = 2
            4. defenseDie = 6 sided
            5. armor = 0
            6. strength = 12
        ii. Key functions
            1. Attack (virtual) return int
            2. Defense (virtual) return int
4. VAMPIRE
    a. Character subclass
        i. Data members
            1. numAttackDie = 1
            2. attackDie = 12 sided
            3. numDefenseDie = 1
            4. defenseDie = 6 sided
            5. armor = 1
            6. strength = 18
        ii. Key functions
            1. Attack (virtual) return int
            2. Defense (virtual) return int
                a. Charm – 50% chance attack doesn't work
            3. Charm – return bool
                a. randomNumberGenerator
                    i. if 1 return true
                    ii. if 2 return false
5. BLUE MEN
    a. Character subclass
        i. Data members
            1. numAttackDie = 2
            2. attackDie = 10 sided
            3. numDefenseDie = 3
            4. defenseDie = 6 sided
            5. armor = 3
            6. strength = 12
        ii. Key functions
            1. Attack (virtual) return int
            2. Defense (virtual) return int
                a. Mob
            3. Mob

        a. Check strength
            i. 12-9 -> numDefenseDie = 3
            ii. 8-5 -> numDefenseDie = 2
            iii. 4-1 -> numDefenseDie = 1
6. MEDUSA
    a. Character subclass
        i. Data members
            1. numAttackDie = 2
            2. attackDie = 6 sided
            3. numDefenseDie = 1
            4. defenseDie = 6 sided
            5. armor = 3
            6. strength = 8
        ii. Key functions
            1. Attack (virtual) return int
                a. Glare
            2. Defense (virtual) return int
            3. Glare
7. HARRY POTTER
    a. Character subclass
        i. Data members
            1. numAttackDie = 2
            2. attackDie = 6 sided
            3. numDefenseDie = 2
            4. defenseDie = 6 sided
            5. armor = 0
            6. strength = 10
        ii. Key functions
            1. Attack (virtual) return int
            2. Defense (virtual) return int
                a. Hogwarts
            3. Hogwarts
                a. If lives = 0 && strength <= 0
                    i. Strength = 20
                    ii. Lives += 1
8. DICE
    a. To be used in the classes
        i. Data members
            1. sides
        ii. Key functions
            1. Constructor
                a. Seed random number
            2. Roll
            3. getSides

4. setSides

9. GAMEPLAY
    a. Contains the game flow
        i. Data members
            1. Rounds
            2. Queue p1
            3. Queue p2
            4. maxTeamSize
            5. p1Score
            6. p2Score
        ii. Key Functions
            1. Constructor
                a. Create matrix for each player with 5 rows and columns = maxTeamSize
            2. Destructor
                a. Deallocate memory
            3. Menu
            4. Random number generator
            5. Tournament
                a. While loop that goes through each play in the queue
                b. Calls combat function
            6. Combat (while loop that checks strength of each player)
                a. display PlayerType and name
                b. Player1Attack – for loop using player1 numAttackDie
                    i. Roll() function
                c. Player1Defense
                d. Player2Attack
                e. Player2Defense
                f. Player1Damage = player2Attack – player1Defense – player1armor
                g. Player1Strength -= Player1Damage
                h. Player2Damage = player1Attack – player1Defense – player2armor
                i. Player2Strength -= Player2Damage
                j. Check strengths and print out if someone dies
                k. Display winner of round

Test Scope	Description	Expected Outcome	Actual Outcome
Start / Lineup / Finish Menu	• Ensure only valid inputs are accepted. Non valid inputs do not crash the program.	• Only integers are accepted, anything else does not crash the program	• Only integers are accepted, anything else does not crash the program
Entire program – boundary case of 1 node	• Ensure nodes move properly from alive to dead queues	• AliveQueue is set properly. • Losers move to Front of deadQueue • Winner recover some health using recovery function • Print entire deadQueue properly	• Queues were set properly and nodes moved to correct place depending on winner/loser
Entire program – boundary case of 2 node	• Ensure nodes move properly from alive to dead queues	• AliveQueue is set properly. • Losers move to Front of deadQueue • Winner recover some health using recovery function • Print entire deadQueue properly	• Segmentation fault – Found that I was not moving the aliveHead to next, rather had it moving to prev
Entire program – boundary case of 3 node	• Ensure nodes move properly from alive to dead queues	• AliveQueue is set properly. • Losers move to Front of deadQueue • Winner recover some health using recovery function • Print entire deadQueue properly	• Segmentation fault – Found that I was not moving the aliveHead to next, rather had it moving to prev

Entire program – boundary case of 4+ node	• Ensure nodes move properly from alive to dead queues	• AliveQueue is set properly. • Losers move to Front of deadQueue • Winner recover some health using recovery function • Print entire deadQueue properly	• AliveQueue is set properly. • Losers move to Front of deadQueue • Winner recover some health using recovery function • Print entire deadQueue properly
Memory leak check	• No memory leaks	• No memory leaks	• No memory leaks

Reflection

Overall, I found myself struggling to really close out the program in the “tournament” function which had everything to do about how my QueueNodes were moving across the containers. I think a big reason for the struggles were that because we were building on top of code from the previous project, that happened to work well for me, I was not concerned about “bolting on” the solutions for project 4. However, what I ran into during the actual execution were a lot of situations, specifically boundary conditions, that I was not prepared for and had not planned for. It was a lesson for me that building on previous code requires just as much forethought and planning, if not more, to understand how the requirements can be implemented and what code needs to modified to get there.

As I mentioned above, the biggest issues I ran into was not properly thinking about the boundary conditions for the first 1,2 and 3 nodes in the aliveQueue and especially for the isutaiton of removing the node and creating a node in the loser container. It took me several iterations of looking for errors and developing some testing chunks to identify where exactly my problems were. In almost all cases, it had to do with my logic around retying the pointer relationships when removing a node. Additionally, I made my logic around the winner node move way too complicated, as I realized the head for the aliveQueue just needed to move forward by one.

I don’t think I allowed myself enough time upfront to truly understand what needed to change in project 3 code (the combat gameplay, character base and derived classes). I think my lack of appreciation of the interfaces with how to most efficiently set up and use the character pointers cost me a lot of time in the execution phase, as well as efficiency in my program.