Name: Carlos Carrillo-Calderon
Date: 11/23/2015
Course: CS_162

Assignment 4

Design Description:

For this analysis, I will follow the parameters defined in the book *C++ Early Objects, 2014*. Thus, an Object-oriented analysis will be performed in order to determine the requirements for this particular system, and clarify what it must be able to do, what it needs, and how the classes created for this program are related. Also, I will specify what the classes will carry out and their responsibilities.

Description of the problem

I had to design, implement, and test a program to run a tournament using the simple class hierarchy used for Assignment 4 as the basis for a fantasy combat game. The game "universe" contains Goblins, Barbarians, Reptile People, Blue Men and others. Each will have characteristics for attack, defense, armor, and strength points.

Identify classes and objects

By definition, a class, in programming, is a package that consist of data and procedures that perform operations on the data, and a set of code instruction to model real-work objects (Gaddis et al, 474). For this program, it is necessary to create a base class and a subclass for each of the characters of the game. The base class will be an abstract class, so it will never be instantiated. Each subclass will vary only in the values in the table. Since each subclass starts with the same data elements, only one constructor is needed. As part of my design, I decided to create just 1 pure virtual function, and 1 void function whose purpose is not to use the constructor for processing all the data related to the characters. In addition to this hierarchical structure, I used 2 Queue-like lists and 1 stack-like list in order to store and process data.

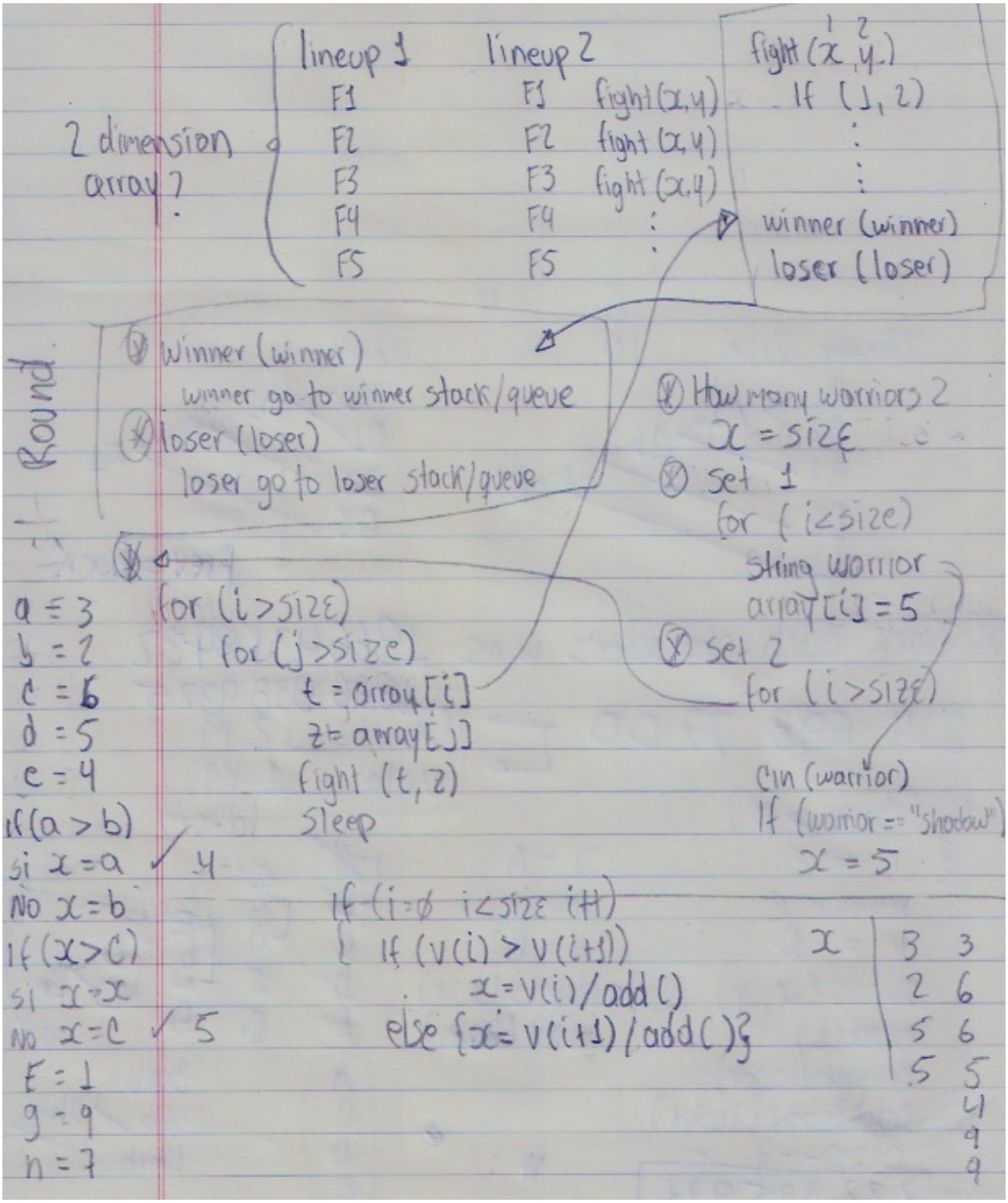
On the other hand, I decided to maintain only 2 class member variables the same as in assignment 4 program: The Strength Points variable (associated to each of the characters) and the Achilles Power variable (only associated with the Goblin character).

Initial Design (very first sketch)

In order to learn the good habit of thinking about the application before starting to code, I crated this draft design (as requested) from which I started to build the whole concept of the game in my mind. Even though this draft is not very detailed, I may say this was the skeleton of the program. The first thing I did was to see how I could recycle pieces of code from previous assignments. The obvious reason was that this assignment gathered concepts already used in other class projects.

So I took the polymorphic abstract class implementation to produce the fights, and three linked lists to storage and process the data produced by the tournament itself. Surprisely, I used exactly these same elements until the end of the creation process.

This is the very first sketch I made to represented my original idea for the program:



My original idea for the structure of the program had this particular scheme to create a combat, to store scores, and to determine the winners of the tournament. In general, the tournament will have 2 different phases/stages. In the first stage, the warriors from one

team will fight against the warrior from the other team. In the second stage, only the winners from the fist set of fights will fight against each other. Consequently, the great winner will be determined in the second phase of the game. Here is an ordered list of processes I thought should be performed for creating the tournament:

1. User chooses number of warriors for each team.
2. User chooses the warriors for each team among 5 creature options.
3. The head of each team fights against each other.
4. The winner is stored in one linked list (winner pile) and the loser will be stored in another linked list (loser pile).
5. The winner pile is processed to make its contents/warriors fight among each other using any sort of pairing method.
6. The looser pile is empty to be reused in the finale.
7. The losers go to the loser pile.
8. The ultimate winner is the last and only value/warrior remaining in the winner pile after all the winners fight among each other.

Although I followed this plan until the end of my design, I have to redo or changed certain processes in order to make the program work. I will talk about these challenges later in this document.

Class's Attributes Definition

Now it is time to define the attributes or the data elements used to describe the objects instantiated from the classes defined above. Here are the class specifications for the Queue and Stack classes, the base class Creature, and the other 5 subclasses: Barbarian, Reptile, Blue (for Blue men), Goblin, and Shadow:

Class name: QueueLineup1 // Class to store the lineup for the first team
Attributes: QueueLineup1Node // struct to create the list nodes.
QueueLineup1Node *front; // front node
QueueLineup1Node *back; // back node

Class name: QueueLineup2 // Class to store the lineup for the second team
Attributes: QueueLineup2Node // struct to create the list nodes.
QueueLineup2Node *front; // front node
QueueLineup2Node *back; // back node

Class name: QueueWin // Class to store the winner of each combat
Attributes: QueueWinNode // struct to create the list nodes.
QueueWinNode *front; // front node
QueueWinNode *back; // back node

Class name: QueueLose // Class to store the winner of each combat
Attributes: QueueLoseNode // struct to create the list nodes.
QueueLoseNode *front; // front node
QueueLoseNode *back; // back node

Class name: Creature
Attributes: fuerza //store the Strength value for each character.
achilles //triggers the Achilles attack from Goblin.

Class name: Reptile
Attributes: //the same as Crature

Class name: Blue
Attributes: //the same as Crature

Class name: Goblin
Attributes: //the same as Crature

Class name: Shadow
Attributes: //the same as Crature

Class's Behaviors (Functions)
Since now the attributes have been defined, it is time to identify the activities or behaviors each class should perform. In software terms, these behaviors are called **functions**.

1) The functions in the **QueueLineup1** class should have these behaviors:

- Add an element to the list.
- Remove the element at front of the list.
- Return the element at front of the list.
- Display the elements from the list (front to back).

Consequently, these are the functions/methods to be created for the **QueueLineup1** class:

- **add(string):** This function add an element to the list to the back of the pile
- **remove():** This function removes the element at front of the list
- **returnName():** This function returns the element at front of the list.
- **display():** This function displays all the elements stored in the list.

2) The functions in the **QueueLineup2** class should have these behaviors:

- Add an element to the list.
- Remove the element at front of the list.
- Return the element at front of the list.
- Display the elements from the list (front to back).

Consequently, these are the functions/methods to be created for the **QueueLineup2** class:

- **add(string):** This function add an element to the list to the back of the pile
- **remove():** This function removes the element at front of the list
- **returnName():** This function returns the element at front of the list.
- **display():** This function displays all the elements stored in the list (front to back).

3) The functions in the **QueueWin** class should have these behaviors:

- Add an element to the list.
- Remove the element at front of the list.
- Return the element at front of the list.
- Display the elements from the list (front to back).

Consequently, these are the functions/methods to be created for the **QueueWin** class:

- **add(string):** This function add an element to the list to the back of the pile
- **remove():** This function removes the element at front of the list
- **returnName():** This function returns the element at front of the list.
- **display():** This function displays all the elements stored in the list (front to back).

3) The functions in the **StackLose** class should have these behaviors:

- Add an element to the list.
- Remove the element at the top of the pile.
- Return the element at the top of the pile.
- Display the elements from the list from top to bottom.

Consequently, these are the functions/methods to be created for the **StackLose** class:

- **add(string):** This function add an element to the list to the back of the pile
- **remove():** This function removes the element at front of the list
- **returnName():** This function returns the element at front of the list.
- **display():** This function displays all the elements in the list from top to bottom.

5) The functions in the **Creature** class should have these behaviors:

- Mutate, get, and return the double variable Strength (fuerza).
- Mutate the int variable achilles.
- Calculate the Strength value of each warrior/character after each attack.
- Roll the dice corresponding to each warrior/character.

Consequently, these are the functions/methods to be created for the **Creature** class:

- **setFuerza(double):** This function mutates the member variable fuerza(Strength).
- **double getFuerza():** This function returns the member variable fuerza(Strength).
- **setAchilles(int):** This function mutates the member variable achilles.
- **virtual double attacks (double, double) = 0:** This is a pure function that is overloaded or redefined according to fight features from each character.
- **Void diePlayer(int, int):** This function performs the actual die rolling according to number of dice and side corresponding to each character.

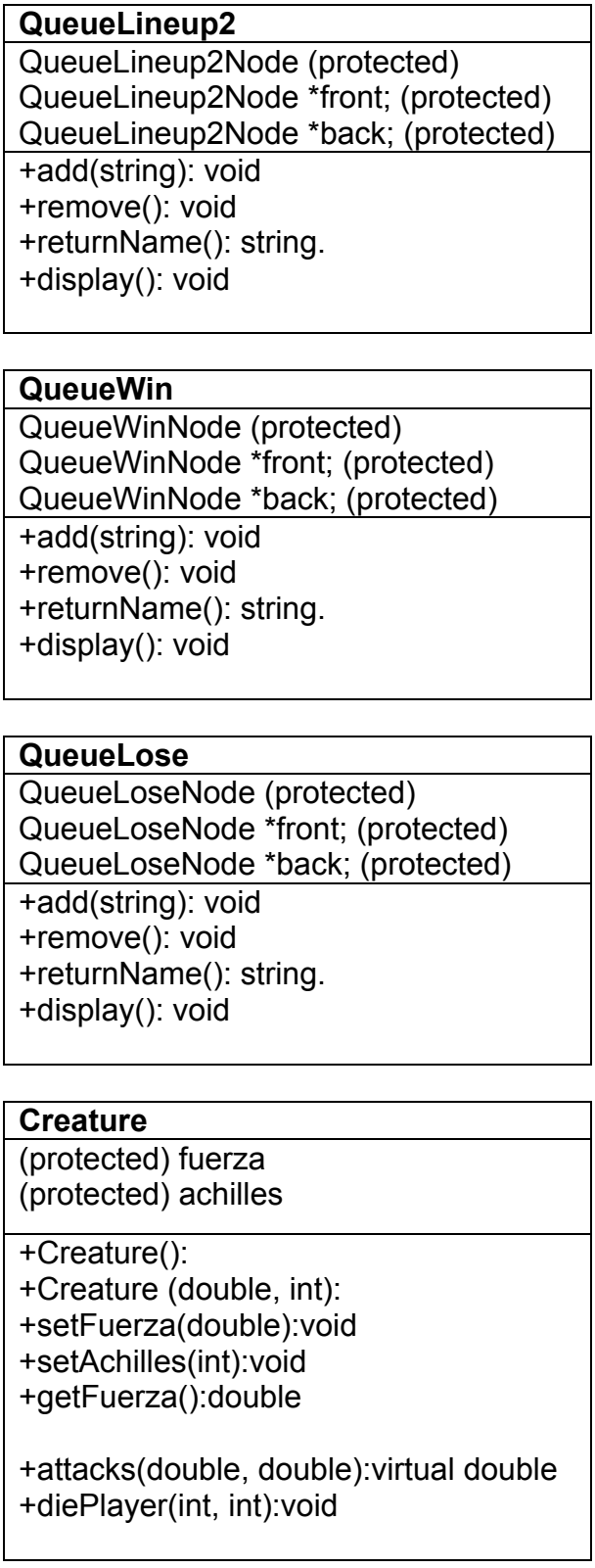
2) The only function in the **Barbarian, Reptile, Blue Men, Goblin,** and **Shadow** classes is **double attacks (double, double)** and it will obviously have very similar behaviors in each subclass, but in general this function will:

- Calculate the Strength value of each warrior/character after each attack.

UML Class Diagrams
Now that it has been possible to identify the class members and the behaviors that they should perform, it is also possible to draw a UML class diagram in order to visualize the **class members** and **functions** that each class should have.

It is important to highlight that the minus sing to the left of each attribute indicates that it is a private member. Similarly, the plus sign to the left of each function indicates that it is a public member. Thus, this could be the UML (Unified Modeling Language) diagram for the Creature class:

QueueLineup1
QueueLineup1Node (protected) QueueLineup1Node *front; (protected) QueueLineup1Node *back; (protected)
+add(string): void +remove(): void +returnName(): string. +display(): void


QueueLineup2
QueueLineup2Node (protected)
QueueLineup2Node *front; (protected)
QueueLineup2Node *back; (protected)
+add(string): void
+remove(): void
+returnName(): string.
+display(): void
QueueWin
QueueWinNode (protected)
QueueWinNode *front; (protected)
QueueWinNode *back; (protected)
+add(string): void
+remove(): void
+returnName(): string.
+display(): void
QueueLose
QueueLoseNode (protected)
QueueLoseNode *front; (protected)
QueueLoseNode *back; (protected)
+add(string): void
+remove(): void
+returnName(): string.
+display(): void
Creature
(protected) fuerza
(protected) achilles
+Creature():
+Creature (double, int):
+setFuerza(double):void
+setAchilles(int):void
+getFuerza():double
+attacks(double, double):virtual double
+diePlayer(int, int):void

Barbarian
- fuerza (inherited from Creature) - achilles (inherited from Creature)
+attacks(double, double):virtual double

Reptile
- fuerza (inherited from Creature) - achilles (inherited from Creature)
+attacks(double, double):virtual double

Blue Men
- fuerza (inherited from Creature) - achilles (inherited from Creature)
+attacks(double, double):virtual double

Goblin
- fuerza (inherited from Creature) - achilles (inherited from Creature)
+attacks(double, double):virtual double

Shadow
- fuerza (inherited from Creature) - achilles (inherited from Creature)
+attacks(double, double):virtual double

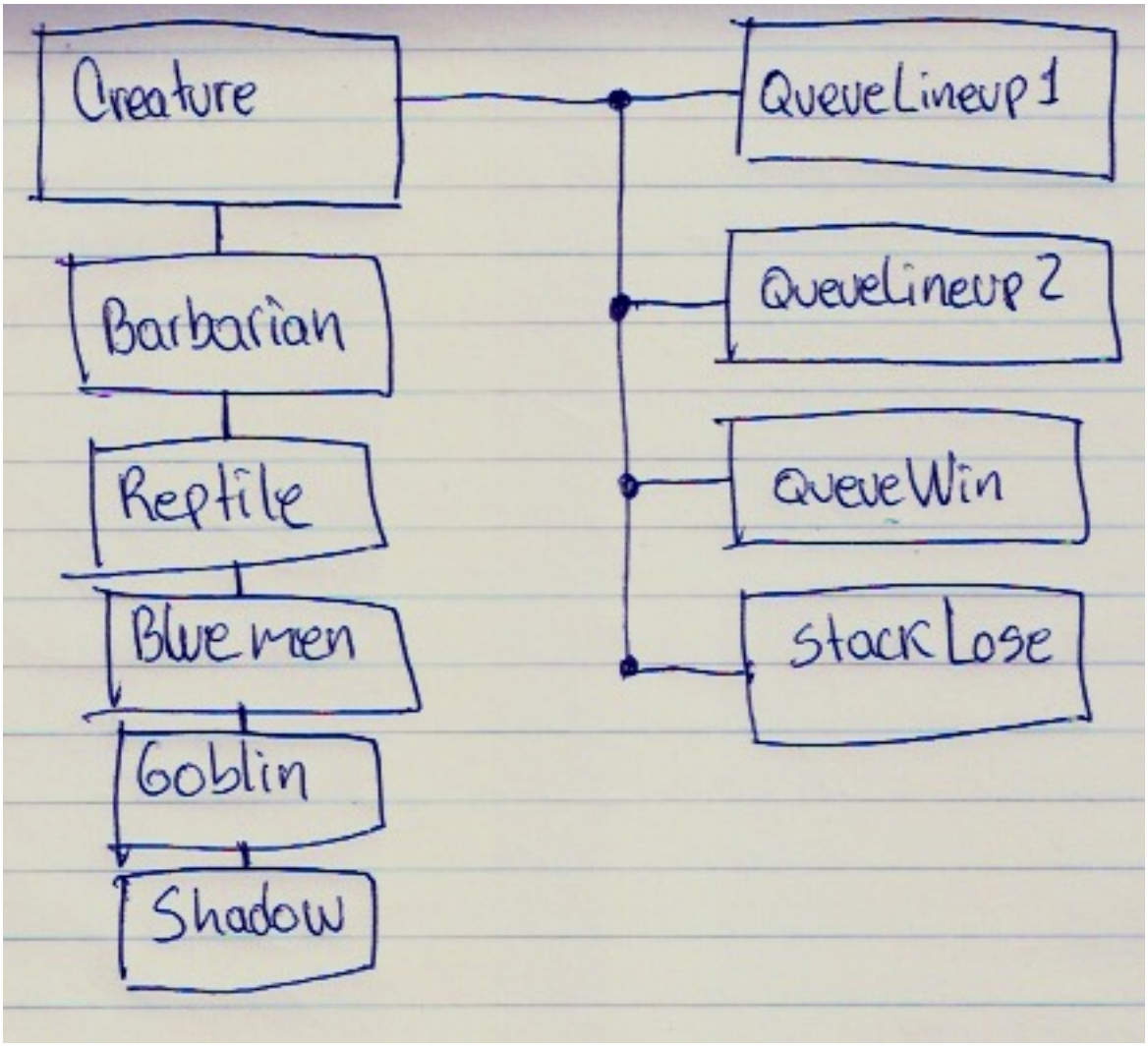
Relationships between classes

Now is time to determine the possible relationships that exist between the 6 classes. As established by Gaddis (2014), there could be three types of formal relationships among classes Access (Uses-a), Ownership (Has-a) and Inheritance (Is-a). In terms of access, the UML diagrams above clearly shows the private and public attributes that are related between the 6 classes. Thus, it could be said that there is an **Inheritance (Is-a)** relationship between the 6 classes. As shown in the UML diagrams above, Creature is the mother/base class of all the other classes. Actually the fact that the base class is an

abstract class means that it can't work by itself since it cannot be instantiated. Reciprocally, the children classes can only work through the mother-base class. Additionally, all the class members from the base class are inherited by the subclasses, so it is possible to say that this is a perfect example of an **Inheritance (Is-a)** relationship.

On the other hand, we may have an Ownership (Has-a) relationship between the abstract base class and the Queue and Stack linked list classes. The can be established by observing the way how the Creature class and subclasses share data with the functions of the Queue and Stack classes.

This type of relationships can give us a clue of how a class ***hierarchy diagram*** may look like for this program:



Main Function
In order to articulate the whole design, I created this overview of the main function or driver program to create character objects, which make attack and defense rolls

required to show my classes work correctly:

1. Create a main menu: In this menu I prompt the user to choose both the attacker and the defender. In order to do this, I assigned a number to each warrior in the following order:

   PLEASE SELECT THE ATTACKER:
   "Enter 1 to attack with BARBARIAN.
   "Enter 2 to attack with REPTILE."
   "Enter 3 to attack with BLUE MEN."
   "Enter 4 to attack with GOBLIN."
   "Enter 5 to attack with THE SHADOW."

   PLEASE SELECT THE RIVAL CREATURE:
   "Enter 1 to fight against BARBARIAN."
   "Enter 2 to fight against REPTILE."
   "Enter 3 to fight against BLUE MEN."
   "Enter 4 to fight against GOBLIN."
   "Enter 5 to fight against THE SHADOW."

2. Create a conditional for each possible fight: Although it was very time consuming, I believed this was the most effective way to simulate possible fight scenarios. Additionally, I decided to set the initial Strength values of the fighters inside this conditional in order to make things easier (in my opinion). For instance, if the user wants Barbarian to attack Reptile, the corresponding conditional would be:

```
if(choice1 == "BARBARIAN" && choice2 == "REPTILE")
 {
      strengthAttack = 12;
      strengthDefense = 18;
```

   Where **choice1** corresponds to the attacker and **choice2** corresponds to the defender.

3. Create class objects to make a single attack: For this task, I created objects in terms of the creature that was receiving the attack. The reason why I did this is because the Strength value changes only in terms of the defender but not the attacker. So this was the code I used to make one of the creatures (Reptile) attacks to another creature (Barbarian):

```
Barbarian myBarbarian(strengthDefense, 1);
myBarbarian.diePlayer(2, 1);
```

   It is important to clarify that the parameter value located next to the Strength parameter is the Achilles value/parameter, which only changes to 2 when Goblin scores 12 during an attack.

4. Create class objects to make a fight: Since a fight is only a fight when the 2 contenders attack to each other, I had to come up with an idea to make that possible. And the solution was simple but effective: I switch the character numbers in the class call and the strength values in the constructor. So this is the code I used to simulate a “round” between the Reptile and Barbarian:

```
Reptile attacks/Barbarian self-defends:
strengthDefense = fuerzaDefense;
Barbarian myBarbarian(strengthDefense, 1);
myBarbarian.diePlayer(2, 1);
fuerzaDefense = myBarbarian.getFuerza();

Barbarian Attacks/Reptile self-defends:
strengthAttack = fuerzaAttack;
Reptile myReptile(strengthAttack, 1);
myReptile.diePlayer(1, 2);
fuerzaAttack = myReptile.getFuerza();
```

5. Create a loop to perform the fight: I created a loop to process the input provided by the user in order to make the team fight possible. This loop takes the front elements from each pile and send them to be processes by the objects() function. Here is the piece of code that executes this process:

```
// Loop to perform the actual tournament
for (int i=0; i<TEAMSIZE; i++){
// Take fighter1 from the pile/list
std::string warrior1 = myQueueLineup1.returnName();
// Take fighter2 from the pile/list
std::string warrior2 = myQueueLineup2.returnName();
objects(warrior1, warrior2); // Make fighters subclass objects
myQueueWin.add(winnerWarrior); // Send winners to the winner pile
myStackLose.add(loserWarrior); // Send losers to the loser pile
// Remove node from the pile to prepare next combat
myQueueLineup1.remove();
// Remove node from the pile to prepare next combat
myQueueLineup2.remove();
system("sleep 2");}
```

6. Create a loop for the finale: After having the results from the team combat, the next step was to create a loop to make sure all the winners fight against each other. In order to do this, it was necessary to use the winner queue structure and the loser stack structure. The idea was to remove the loser from the winner pile until only one element/object remains in the winner pile. Here is the piece of code that executes this process:

```
// only combats -1 are needed to determine the Great WINNER!!
TEAMSIZE = TEAMSIZE - 1;
```

```
// Loop for the Great Finale: Removes the 2 warriors on the front of
the pile and re-insert the winner to the tail/back of the pile
    do{
         // Take fighter1 from the pile/list
        std::string fighter1 = myQueueWin.returnName();
        cout<<"Fighter1: "<<fighter1<<endl;
        // Remove node from the pile to prepare the combat
        myQueueWin.remove();
        // Take fighter2 from the pile/list
        std::string fighter2 = myQueueWin.returnName();
        cout<<"Fighter2: "<<fighter2<<endl;
        // Remove node from the pile to prepare next combat
        myQueueWin.remove();
        myQueueWin.display();
        cout<<endl;

        objects(fighter1, fighter2); // Make fighters subclass objects
        // Send winner to the back of the winner pile
        myQueueWin.add(winnerWarrior);
        // Send winners to the finishers pile
        myStackLose.add(loserWarrior);
        cout<<endl;

        TEAMSIZE--;
        system("sleep 2");

    }while(TEAMSIZE > 0);
```

7. Display the final results: Finally, it was necessary to use the display functions from the list classes in order to show the final results to the user.

Tournament Structure
The tournament score system was designed as follow:

1) The head of each team fight against each other until one of the warriors loses all of its strength points. These updates are made after this combat to death:

   - The winner recovers some strength points (up to 4 points).
   - The winner is added to the winners group (winners will fight again in the next phase of the tournament).
   - The loser is added to the loser group (the losers are out of the tournament and will never fight again).
   - The winner's team gets 1 point.
   - The loser's team gets no points.

2) This is the way in which the finale is developed in the program. I chose this system in order to properly used the tool we could use to built the program. Thus, these are the action that happen in the finale:
   - The winners will fight according to their order of arrival to the pile. So the first warrior arrived to the pile fights against the 2nd warrior arrived. Then the 4th with the 5th and so on.

   - The winner stays and goes to the back of the winner pile.

   - The loser is sent to the loser pile and out of the tournament, so it will never fight again.

   - At the end, only one warrior will remain in the winner pile, therefore, that warrior will be the Great Winner/Champion of the tournament.

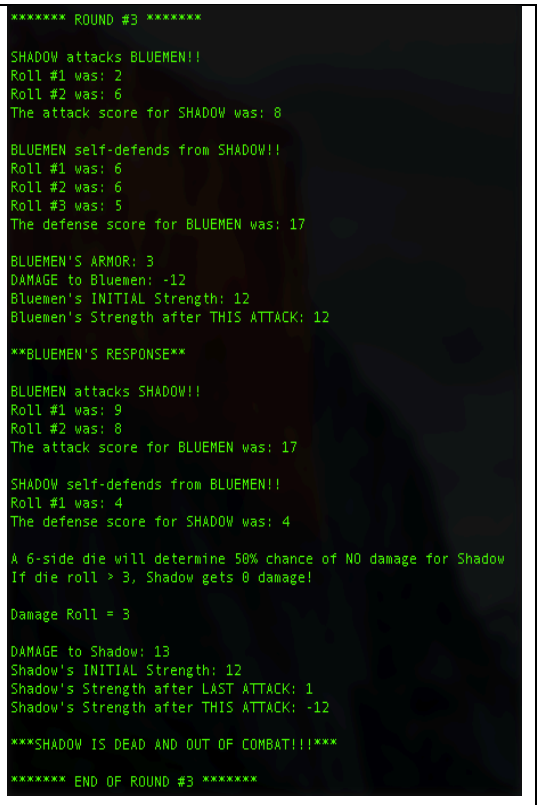
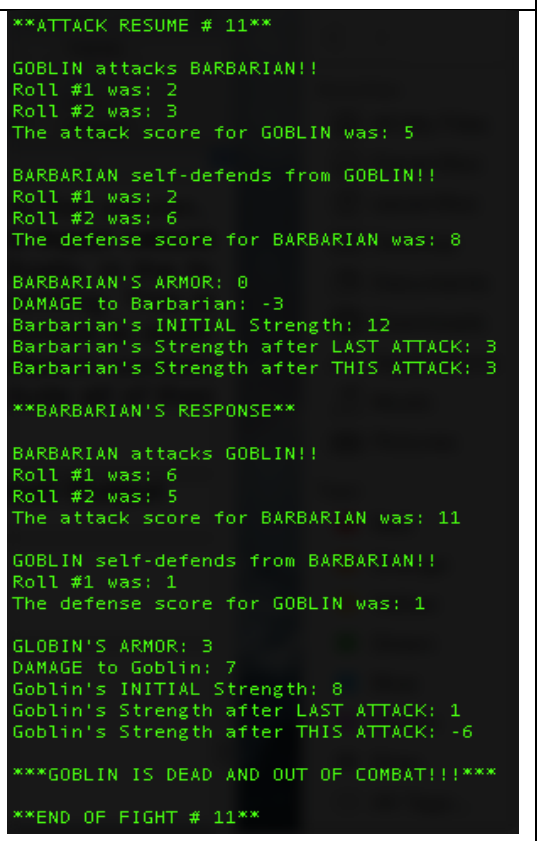
   - The last to arrivals to the loser pile will be the 2nd and 3rd place respectively.

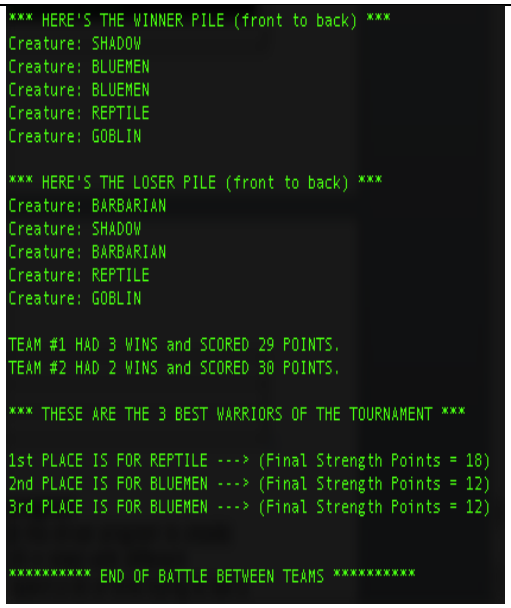
Test Plan

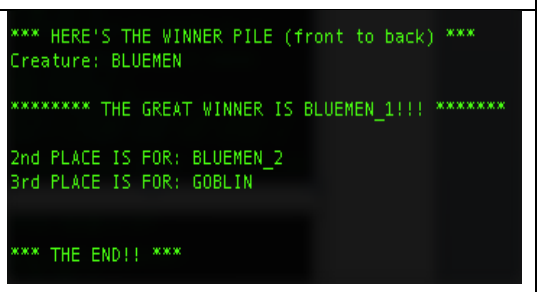
In order to test this program, I created a 4-column table. In the first column, I described **what I was going to test**. In the second column, the possible **Input values** to perform the test. In the third column, the **expected outcomes**. And finally, in the forth column, the **actual values or output** that the program shows after running. This is the table I used. In my real testing process, I make each warrior had a combat to death with every other warrior. Since there are 5 warriors (5^2), I tested 25 different combats (including each character against itself). For logistic reasons, I wont include all of these 25 combats. I'll just show 1 random combat per character (but I invite the evaluator to play the game a little bit). It is also relevant to say that the strongest fighters were Blue Men and Reptile respectively, followed by Shadow and Barbarian. And the weakest warrior (without Achilles power) was Goblin.

For the finale, additional input was no necessary since it also depends on the initial input provided by the user. In other words, the random element will decide the elements to be inputted for the finale. So the next table will show a good sample of the most important testing elements I used to see if my program was properly working:

Section to be tested	Input provided	Expected Output	Actual Output
attacks(dob, dob) diePlayer(int, int)	Attacker: Barbarian Defender: Goblin	Correct attack score calculation for Barbarian. Correct attack score calculation for Goblin. Correct defense score calculation for Barbarian. Correct defense score calculation for Goblin. Correct Strength calculation for Barbarian. Correct Strength calculation for Goblin. One of the warriors should die at some point.	**GOBLIN'S RESPONSE** GOBLIN attacks BARBARIAN!! Roll #1 was: 5 Roll #2 was: 2 The attack score for GOBLIN was: 7 BARBARIAN self-defends from GOBLIN!! Roll #1 was: 5 Roll #2 was: 1 The defense score for BARBARIAN was: 6 BARBARIAN'S ARMOR: 0 DAMAGE to Barbarian: 1 Barbarian's INITIAL Strength: 12 Barbarian's Strength after LAST ATTACK: 7 Barbarian's Strength after THIS ATTACK: 6 **END OF FIGHT # 9** **ATTACK RESUME # 10** BARBARIAN attacks GOBLIN!! Roll #1 was: 5 Roll #2 was: 3 The attack score for BARBARIAN was: 8 GOBLIN self-defends from BARBARIAN!! Roll #1 was: 3 The defense score for GOBLIN was: 3 GLOBIN'S ARMOR: 3 DAMAGE to Goblin: 2 Goblin's INITIAL Strength: 8 Goblin's Strength after LAST ATTACK: 1 Goblin's Strength after THIS ATTACK: -1 ***GOBLIN IS DEAD AND OUT OF COMBAT!!!***
attacks(dob, dob) diePlayer(int, int)	Attacker: Reptile Defender: Blue Men	Correct attack score calculation for Reptile. Correct attack score calculation for Blue Men. Correct defense score calculation for Reptile. Correct defense score calculation for Blue Men. Correct Strength calculation for Reptile. Correct Strength calculation for Blue Men. One of the warriors should die at some point.	**ATTACK RESUME # 18** REPTILE attacks BLUE_MEN!! Roll #1 was: 4 Roll #2 was: 6 Roll #3 was: 4 The attack score for REPTILE was: 14 BLUE_MEN self-defends from REPTILE!! Roll #1 was: 2 Roll #2 was: 3 Roll #3 was: 6 The defense score for BLUE_MEN was: 11 BLUE'S ARMOR: 3 DAMAGE to Blue: 0 Blue's INITIAL Strength: 12 Blue's Strength after LAST ATTACK: 1 Blue's Strength after THIS ATTACK: 1 **BLUE_MEN'S RESPONSE** BLUE_MEN attacks REPTILE!! Roll #1 was: 10 Roll #2 was: 8 The attack score for BLUE_MEN was: 18 REPTILE self-defends from BLUE_MEN!! Roll #1 was: 4 The defense score for REPTILE was: 4 REPTILE'S ARMOR: 7 DAMAGE to Reptile: 7 Reptile's INITIAL Strength: 18 Reptile's Strength after LAST ATTACK: 2 Reptile's Strength after THIS ATTACK: -5 ***REPTILE IS DEAD AND OUT OF COMBAT!!!*** **END OF FIGHT # 18**

attacks(dob, dob) diePlayer(int, int)	Attacker: Blue Men Defender: Shadow	Correct attack score calculation for Blue Men. Correct attack score calculation for Shadow. Correct defense score calculation for Blue Men. Correct defense score calculation for Shadow. Correct Strength calculation for Blue Men. Correct Strength calculation for Shadow. One of the warriors should die at some point.	
attacks(dob, dob) diePlayer(int, int)	Attacker: Goblin Defender: Barbarian	Correct attack score calculation for Goblin. Correct attack score calculation for Barbarian. Correct defense score calculation for Goblin. Correct defense score calculation for Barbarian. Correct Strength calculation for Goblin. Correct Strength calculation for Barbarian. One of the warriors should die at some point.	

attacks(dob, dob)
diePlayer(int, int)
 	
myQueueLineup1.
returnName()
 	
myQueueLineup2.
returnName()
 	
objects(stg, stg);
 	
myQueueWin.
add();
 	
myStackLose.
add(string);
 	
myQueueLineup1.
remove();
 	
myQueueLineup2.
add(string);
	All the elements inputted by the user,
	
	(In this case, all 5 warriors were entered for each team)
		A winner list.
		
		A loser list.
		
		Winner team according to wins and Strength points score.
		
		Winner team according to wins and Strength points score.
		
		The 3 best warriors of the Team1 Vs. Team2 tournament according to Strength points score
			

attacks(dob, dob)
diePlayer(int, int)
 	
myQueueWin.
returnName()
 	
myQueueWin
remove();
 	
objects(stg, stg);
 	
myQueueWin.
add();
 	
myStackLose.
add(string);
	All the elements stored in the winner pile/queue	Only one element in the winner pile.
		
		The last 2 losers (which are eventually the 2nd and 3rd place in the competition.)
			

Problems while designing and implementing your program

In general, the main problem I faced was to properly set the driver program to create character objects. Since there was several scenarios (25 in total) with different parameters, I inserted a lot bugs during this process. I spent a lot of time trying to set a combat so the program could actually be playable and graphically appreciated to see status of the warriors after each round accurately. This is actually the first time in my role as a programmer that I had to face a very tedious and complex debugging process. I may say that attention to little details was the real problem.

Additionally, I struggled a little bit deciding what the functions really should do. I was not sure if the attack class should also handle the die roll. Also, I wasn't sure if I should control all the data through the constructor or if I should use a function to make things

less messy. I finally decide to use both and I don't regret to do it. Personally, I believe it was a good I idea.

On the other hand, I have to recognized that it was difficult to learn how to manipulate the linked list to make them fit into my design. Pointers are not one of my strengths so I found it difficult to figure out hoe to use these structures.

Finally, I also spent a good chunk of time thinking about the initial design. I had to do a lot of reading and research. This was also my first time working with games like this. Although it was fun, it also was frustrating at some moments.

Conclusion:
In conclusion, this assignment was a great exercise to practice software design based on an OOD approach. Also, this program finally made me understand the concept of heritage, which has a lot of benefits but also certain level of complexity. In theory, if I follow these same steps when working on this type of projects, the design can be more accurate. Also, this reinforced on me the good habit of thinking about design before starting to code. In general, this was a great learning experience.