Jason Hoffman
CS162, Assignment 4
11/22/2015

Program Description

Assignment 4 is a point based fantasy game based on Assignment 3. It simulates confrontations between teams of characters, the outcome of which is recorded using a point system. Victorious characters move on to fight again, while the loser is added to a 'loser' pile of defeated characters. At the end of the game these groups are used to display the overall outcome of the game. The user is able to select the number type of characters for each team.

The result of each individual fight is decided by dice rolls and character attributes such as *armor, strength points, attack points,* and *defense points.* Each character type is unique in the way that points are derived in an attack or defense by changing the number of dice used, the number rolls, or the number of sides per die. Some characters are considered strong and likely to win often, while others will be at a disadvantage. Once a character's *strength points* have been depleted, that character is deemed to have lost the battle. Special abilities will be given to some characters which increase their likelihood of winning.

A point system will be used to create a final finishing order for the characters, as well as a winning team.

Character attributes:

Type	Attack	Defense	Armor	Strength Points
Goblin	2d6	1d6	3	8
Barbarian	2d6	2d6	0	12
Reptile People	3d6	1d6	7	18
Blue Men	2d10	3d6	3	12
The Shadow	2d10	1d6	*Special	12

*Special- The Shadow can manipulate perceptions of others. For a given attack there is a 50% chance that The Shadow is someplace else and no damage is received.

*Achilles- When fighting anything but another Goblin they might get lucky and cut an Achilles Tendon.

Note: If a Goblin rolls a 12 in attack then the target has his/her attack roll halved for the remainder of the combat.

<u>Plan for Classes</u>

The structure of each character in the game lends itself to using a base class called *Creature*, from which the five character types can inherit. Relevant items from the program description that can could be used as attributes or functions include:

Variables: armor, strength points, attack points, defense points, attack dice, defense dice, team points, individual points, group

Functions: attack, defend, create

Characters with special abilities will have their attack or defense functions overridden. Additionally, the most common attack and defense for each character will be used in the Creature class, for example, two rolls for an attack.

Creature Class

Creature will be an abstract class that contains the core parts of each creature. Common among all classes derived from creature are armor, strength points, and the attack and defend functions. Creatures will also have *attackPoints* and *defensePoints* member variables to be used to keep track of the character's status. The bool *alive* will also be added to assist with the outcome of a simulated combat, along with its associated getter function. Dice will also be a part of each object derived from creature. A class, Die, will be used for rolling attack and defense points, with each Creature object having its own die object. All creatures will also inherit a *tournamentPoints* variable from Creature which will be used to track the character's success in the game. Additionally, each creature will have a *name* and *teamName* member variables, along with the associated getters and setters.

Additional member functions:

takeDamage() - Applies defense to attack, then removes required number of strength ponts
isGoblin() - Returns false for all creatures except Goblin. Used for Goblin special attack

Creature

`private member variables:`
`  (none)`
`protected member variables:`

```
  Die *die
  int armor
  int strengthPoints
  int attackPoints
  int defensePoints
  bool alive
  int tournamentPoints
  string teamName
  string name
public member variables:
  (none)
```

```
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Creature() - Default constructor
  Creature(int armor, int strengthPoints, int attackPoints, int
defensePoints, bool alive)
  bool isGoblin()
  virtual void attack(Creature* c)
  virtual int defend()
  int getDamage()
  void takeDamage(int)
  bool isAlive()
  void setName(string n)
  string getName()
  void setTeamName(string n)
  string getTeamName()
```

Goblin Class

The Goblin character has a special attack ability, and will therefore have its *attack* function overridden. Goblin objects have an added private member variable, *halved,* which is a boolean that will be used to determine whether the opposing character will have their attack roll halved, as well as a getter function for the variable. The Creature *isGoblin()* function is overridden in Goblin and is used to prevent the Goblin special attack from being used on other Goblins.

```
                                Goblin
```

```
private member variables:
  bool halved
```

```
protected member variables:
  (none)
public member variables:
  (none)
```

```
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Goblin() - default constructor
  void setHalved(bool h)
  bool getHalved()
  bool isGoblin()
  void attack() - overridden for special ability
```

Barbarian Class

The Barbarian is a standard character with no special abilities, so most of the class will be inherited from Creature. The barbarian does, however, roll twice for its defense, so *defend* will be overridden to return two rolls.

```
                         Barbarian
```

```
private member variables:
  (none)
protected member variables:
  (none)
public member variables:
  (none)
```

```
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Barbarian() - default constructor
  int defend() - overridden for two rolls
```

Reptile People

Reptile People is another class that closely resembles Creature. Its attack is three rolls, so *attack* will be overridden for three rolls. The rest of the class inherits from Creature.

```
                        Reptile People
---------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  (none)
public member variables:
  (none)
---------------------------------------------------------------
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  ReptilePeople() - default constructor
  int attack() - overridden for three rolls
```

Blue Men

Blue Men has a strong defense roll, with three rolls. Its attack is the standard two rolls. *Defend* will be overridden for three rolls, and the rest will inherit from Creature.

```
                           Blue Men
---------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  (none)
public member variables:
  (none)
---------------------------------------------------------------
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Barbarian() - default constructor
  int defend() - overridden for three rolls
```

The Shadow

The Shadow has a special defense, but will otherwise inherit from Creature. For each attack, there is a 50 percent chance that The Shadow will take zero damage. This will be built into the *defend* function.

```
                          The Shadow
---------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  (none)
public member variables:
  (none)
---------------------------------------------------------------
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  TheShadow() - default constructor
  int defend() - overridden for special ability
```

Combat

The Combat class will be used as a container for two objects in the simulated combat. Logic for each object’s attack and defend turns will be handled by the combat class, as well as checking *strengthPoints* to determine the outcome. The combat class will accomplish this with a single function, *combatBetween().* The existence of this class will allow main.cpp be contain only menu logic rather than also handling the interaction between Creature objects. The creation of characters will also be handled in the Combat class rather than in main.cpp with the function *createCharacterOfType()*, which will be responsible for creating the user’s character as well as the opposing characters. The class will also have two utility functions, *getWinner* and *getLoser,* for use in main.

```
                              Combat
-------------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  (none)
public member variables:
  (none)
-------------------------------------------------------------------
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Creature* combatBetween(Creature *p, Creature *c)
  Creature* createCharacterOfType(int n)
  Creature* getWinner(Creature *a, Creature *b)
  Creature* getLoser(Creature *a, Creature *b)
```

Object Containers

A key feature of the design of this game is using linked list containers to store the characters for each team, as well as the defeated characters as they're removed from the game. Characters who win a match are shuffled back to the end of the line, similar to a batting order, to fight again when their turn comes up. A queue would be best suited for this arrangement, as it's a first in, first out rotation. Losers are added to a pile, and displayed at the end of the game in the order that they were defeated. A stack would work well here.

Queue Class

The Queue class will be a doubly linked list designed to hold a team of Creatures. Basic functions will be:

add() - Add a creature to the front of the Queue
getFront() - Return the Creature object at the front of the Queue
removeFront() - Delete the item at the front of the Queue

Queue will contain a QueueNode struct to represent nodes in the list. Each node will have a previous and next pointer, following the philosophy of doubly linked lists. Additionally, the Queue class will have a default constructor and destructor.

```
                              Queue
---------------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  struct QueueNode
    (inside QueueNode)
     Creature* creature
     QueueNode *next
     QueueNode *prev
  QueueNode *front
  QueueNode *back
public member variables:
  (none)
---------------------------------------------------------------------
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Queue()
  ~Queue()
  Creature* getFront()
  void add(Creature*)
  void removeFront()
```

Stack Class

The Stack class will be similar to Queue, but will work as a stack. The doubly linked list will hold defeated characters to be output at the end of the game. As no *back* variable is needed for a stack, the class will simply have a *head* member variable. Functions included:

add() - Add a Creature to the stack.
removeHead() - Remove the Creature at the top of the stack.
returnHead() - Return the Creature at the top of the stack.
printStack() - To be used in main to print the contents of the stack.

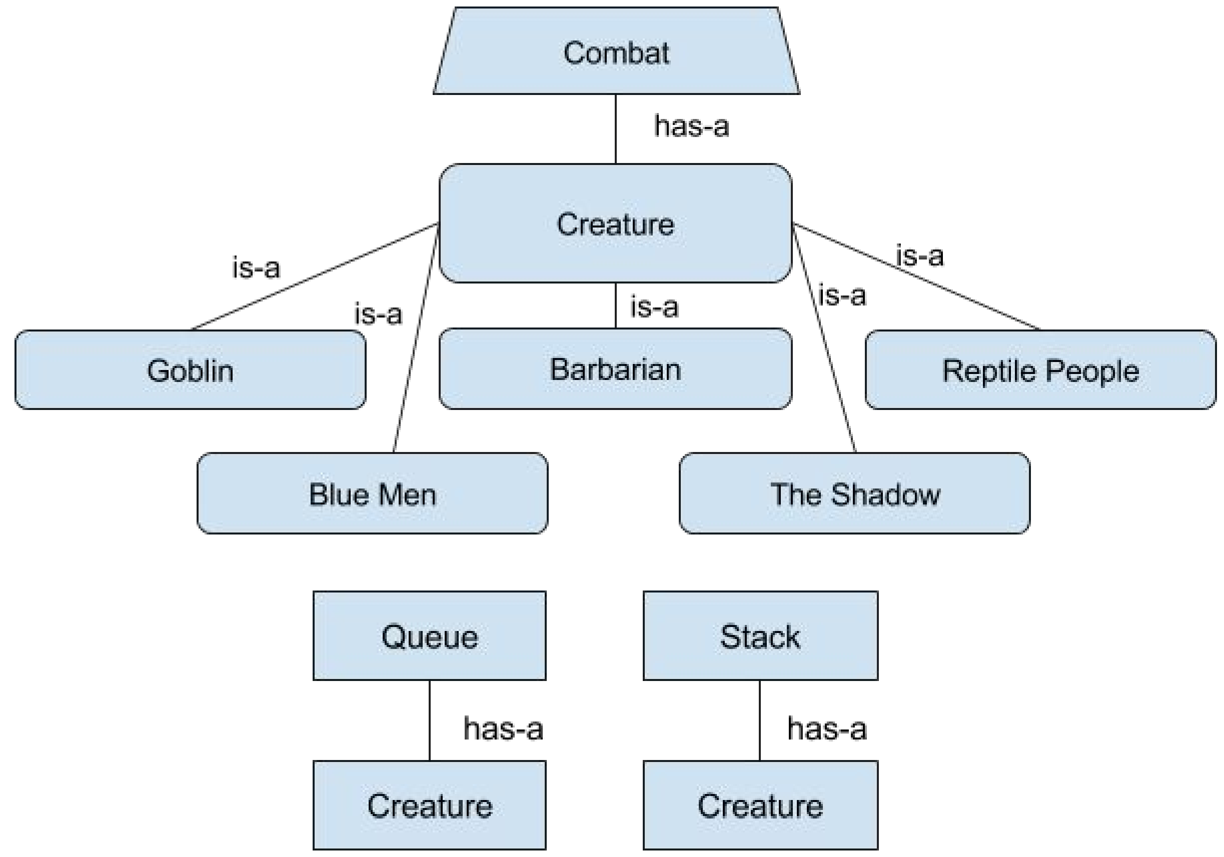
```
                              Stack
---------------------------------------------------------------------
private member variables:
  (none)
protected member variables:
  struct StackNode
    (inside StackNode)
     Creature* creature
     StackNode *next
     StackNode *prev
  StackNode *head
public member variables:
  (none)
```

```
private member functions:
  (none)
protected member functions:
  (none)
public member functions:
  Stack()
  ~Stack()
  Creature* returnHead()
  void add(Creature*)
  void removeHead()
```

Scoring

This game is designed to have a tournament format. Each team will have an overall score, which will be the sum of the scores of each member of the team. Team members earn points by inflicting damage during combat. Whether the player wins or loses, each point of damage inflicted will count as a point.

Outline



Test Plan

Test	Input	Outcome
1	Creature derived object set to call inherited attack() function	Integer representing attack is returned
2	Creature derived object set to call inherited defend function	Integer representing defense is returned and strength points are reduced
3	Goblin object is set to call overridden attack function with a roll of 12	Goblin *halved* member variable is set to true, reducing further attacks to object by half
4	Barbarian object is set to call overridden defend function	Integer is returned containing the sum of two die rolls
5	Reptile People object is set to call overridden attack function	Integer is returned containing the sum of three die rolls
6	Blue Men object is set to call overridden defend function	Integer is returned containing the sum of three defense rolls

7	The Shadow object is set to call overridden defend function	An integer is returned 50% of the time that causes the object to take zero damage
8	A Creature derived object’s isAlive() function is called	A boolean is returned
9	A Creature derived object’s *strengthPoints* go below zero after an attack	Creature’s *isAlive* variable is set to false, *strengthPoints* are set to zero in defend() function
10	Creature derived object pointer is instantiated and assigned to uninstantiated Creature variable	Object retains its properties
11	Combat class is instantiated and two Creature objects are provided to its combatBetween function	The objects’ attack and defend functions are called until a Creature object is returned as the winner
12	Inherited getDamage function is called from Creature derived class	Object’s strengthPoints member variable is returned
13	Combat class’s create character function is called for each character type	A character of the the designated type is created
14	A Creature object is added to an empty Queue	Creature is added to Queue with front and back pointing to same creature. ‘Prev’ and ‘next’ both point to NULL
15	A Creature object is added to a queue containing one other creature	Creature becomes new ‘back’ of queue with ‘prev’ pointer pointing to front and ‘next’ pointer pointing to NULL
16	removeFront is called on empty Queue	Function exits
17	getFront is called on empty Queue	Function exits
18	removeFront is called on Queue containing one Creature	Creature is deleted. Front and back both point to NULL
19	A Creature object is added to an empty Stack	Creature is added to stack. Head points to Creature. ‘prev’ and ‘next’ both point to NULL
20	A Creature object is added to a stack containing one Creature	Creature is added to stack. Head moves to new creature
21	removeHead is called on empty stack	Function exits

22	returnHead is called on empty stack	Function exits
23	Combat’s getWinner is called	Two creatures are compared. Winning creature is returned with points added to tournamentPoints variable
24	Combat’s getLoser is called	Losing creature is returned

<u>Reflection</u>

This project turned out to be much more complex than I had originally planned. Before even getting to the particulars of the game, I struggled with creating the Queue and Stack classes. Although I lost quite a bit of time on this, the game ended up coming together.

The Queue and Stack were created using Lab 6 as a shell, with changes to make them doubly linked. A separate program was used to test the structures. My primary method of testing was using breakpoints to make sure all pointers were property created and pointing to the correct objects. Later, I output the contents of the containers to see them working more quickly and make some final checks for proper operation.

Scoring also turned out to be somewhat difficult. I had originally planned to keep score based on damage inflicted, with the sum of these points adding up to a team score. This seemed to require too much modification to the Queue class, so I instead chose to use a point system based on the difficulty of defeating each character. These scores add up to create the team score.

Another problem I ran into toward the end of the project is returning a “top 3” when fewer than three characters were used per team, or if the winning team ended with fewer than three. I was trying to avoid using a vector, however, ended up doing so in the end. Remaining characters are added to a vector and sorted, then, if the vector contains less than three Creatures, the creature from the loser pile with the most points is added, as this character, despite losing, is technically the third place finisher.

Additionally, I left out any mention of restoring a character’s strength after a win. In this program, strength is fully restored when a Creature object moves on to the next round.

A number of functions were added to the project that were not described in my plan:

Queue

frontToBack - Cycles the front Creature to the back of the Queue after a win
calculatePoints - For use in main.cpp. Calculates the team's total points for output
isEmpty - A function was needed to check whether the Queue is empty

Stack

returnContents - Returns the entire stack in the form of a vector.

Creature

incrementPoints - Increases points for Creature after a win
getTournamentPoints - getter for tournamentPoints member variable
getDefeatPoints - For use in scoring. Getter for defeat points member variable

main.cpp

creatureSort - Sorts vectors of creatures for output of final result