UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science
COS110 - Program Design: Introduction
Assignment 2

1 Introduction

Deadline: 10th of September, 07:00

1.1 Objectives and Outcomes

The objective of this assignment is to provide a comprehensive practical exercise involving multiple classes using C++ features like polymorphism and operator overloading to produce a detailed simulation program.

1.2 Submission

All submissions are to be made to the **assignments.cs.up.ac.za** page under the COS 110 page, and for the correct assignment slot. Submit your code to Fitchfork before the closing time. Students are **strongly advised** to submit well before the deadline as **no late submissions will be accepted**.

1.3 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone elses work without consent, copying a friends work (even with consent) and copying textual material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to **http://www.ais.up.ac.za/plagiarism/index.htm** (from the main page of the University of Pretoria site, follow the *Library* quick link, and then click the *Plagiarism* link). If you have questions regarding this, please ask one of the lecturers, to avoid any misunderstanding.

1.4 Implementation Guidelines

Follow the specifications of the practical precisely. For each practical, you will be required to create your own makefile so pay attention to the names of the files you will be asked to create. If the practical requires you to submit additional files of your own, follow the file structure and format exactly. Incorrect submissions will use up your uploads and no extensions will be given. In terms of C++, unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the practical, will not be allowed. Some of the appropriate files that you submit will be overwritten during marking to ensure compliance to these requirements. If the specification makes use of text files, for providing input information, be sure to include blank text files with the specified names.

1.5 Operator Overloading

Like function overloading, operator overloading is the practice of extending the usage of the set of operators defined in C++ in ways that go beyond their normal usage. For example, while it is possible to overload a function by defining multiple versions of the same function with different argument lists, overloading an operator, like the +, would then allow for the usage of the plus in contexts outside of simple mathematical addition. It would be possible to define and use concepts like the addition of two classes in programs, extending the flexibility and ease of use possible within a given domain.

1.6 Polymorphism

Polymorphism, meaning many forms, is a useful property that is most often realised through the use of inheritance and classes. In C++, it finds common usage in the ability of a general pointer to be instantiated as a number of specific classes that all fall into the same inheritance hierarchy. In particular, it provides for greater flexibility in designing classes and enables better engineering of classes as classes in a hierarchy can share the features of their parents while still providing features and modifications of their own. All of this without changing the underlying usage of the classes in the hierarchy. Some clarification for the symbols used in the UML needs to be presented. In particular, virtual functions are shown in this UML in italics and pure virtual are in italics and also set to 0 such as *foo()=0:void*

1.7 Mark Distribution

Activity	**Mark**
(Slot 1) Gladiator Classes and Hierarchy	30
(Slot 2) Team Class	40
(Slot 3) Arena Class	30
Total	**100**

2 Assignment

In this assignment you are going to be creating a historical simulation program that will simulate gladiatorial combat as done by the gladiators of Ancient Rome.

2.1 Historical Background and General Information

Gladiators themselves are shrouded in myth due the passage of time and changing of social values. What is for certain, is that the gladiatorial games held in Rome, and other parts of their Empire, were akin to modern day massive sporting events. The gladiators themselves, many of which came from the span of Roman territory, were very much like modern professional athletes. They were well fed, so as to develop a layer of protective fat, and regularly trained and drilled as both combatants and entertainers. Casualties, while possible and not uncommon, were generally regarded as a terrible result as the event organisers would lose their investment in the fighter that had died. Of course, that did not dissuade certain event organisers from simply organising a wholesale slaughter of prisoners for example. This is not to say that being a gladiator carried the full social prestige associated with professional athletes today. Gladiators were often socially marginalised and kept under harsh discipline in order to keep them compliant. This unique combination of fascination with violence and spectacle is perhaps why gladiators still persist in modern society today.

Important to the organisation of a Roman gladiatorial combat, are the gladiator classes. Much like different positions on a football team like goalkeeper and midfield, each of the different gladiator classes fought with different weapons and armour and served in different capacities in the arena. What is perhaps unique about the gladiator classes is that each class was generally designed to fight against another specific class in single combat. In team fights, between groups of gladiators, there is a more complexity to composing a team to maximise spectacle and crowd enjoyment. The scope of this assignment will only consider 4 classes but all are briefly and generally discussed below.

2.1.1 Trax

Descended from the Thrace region in the empire, the Trax is an mixed focused gladiator who is armed with a smaller shield and a sica, a curved sword. Although he wore a fully covering helmet, his armour was mostly padded. This light equipment makes him ideal for a balance between offence and defence and for this reason, Trax are a common gladiator class.

2.1.2 Murmillo

The heaviest, by weight and strength, the Murmillo class was only for the largest and strongest men. They would have been given the largest shields and a gladius and been largely impossible to attack from the front. Their stamina and equipment makes them great at sustaining attacks but not so much on the offence.

2.1.3 Retiarius

Armed like a fisherman, the Retiarius is given a weighted net and a trident and very little armour. With these, they can fight a wider variety of opponents, using reach and mobility to both outlast and out-strike their foe. They are extremely showy and popular for that reason, since the Retiarius would feature as a Pontarius who was a special gladiator who fought two others on a bridge.

2.1.4 Hoplomachus

The Hoplomachus is a imitation of the Greek Hoplite. Given padded armour, a shield and lance, an asta, they would fight in imitation of the Greeks whom the Romans had conquered. They are essentially an alternative to the Trax, being a bit more armoured and a bit less suited for attacking.

2.1.5 Dimachaerus

Specialised as far as possible for attack, the Dimachaerus is armed with two swords, a sica and a straight sword. They wear little armour and focus on being as quick and deadly with their blades as possible. Obviously, they opt for a high risk, high reward strategy as few opponents can survive a properly trained Dimachaerus but fewer Dimachaerus survive to become so skilled.

2.1.6 Crupellarius

The Crupellarius is a unique class, being descended from Gaelic warriors defeated by the Empire. They are unique in that they wear full metal, for the time, armour and fully enclosing helmets that bear some resemblance to the helms of later medieval knights. For this armour, they are extremely difficult to kill, being immune to most attacks against their body save for the few exposed spots. However, they lack the physicality of the Murmillo and so tire easily and cannot strike back. They are particularly unpopular in some parts of Rome due to the Gaelic revolts but they are able to soak up damage like no other class.

3 Assignment Overview

3.1 Gladiator Class Hierarchy

Presented here will be a class diagram depicting the nature of the class hierarchy formed between a parent **gladiator** class and its children, all of the specific gladiator classes mentioned earlier. The gladiators chosen here represent the more popular of the choices. The relationship is a strict one to many relationship with each specific class of gladiator being a direct descendent of the **gladiator** class. This is presented in Figure 1:

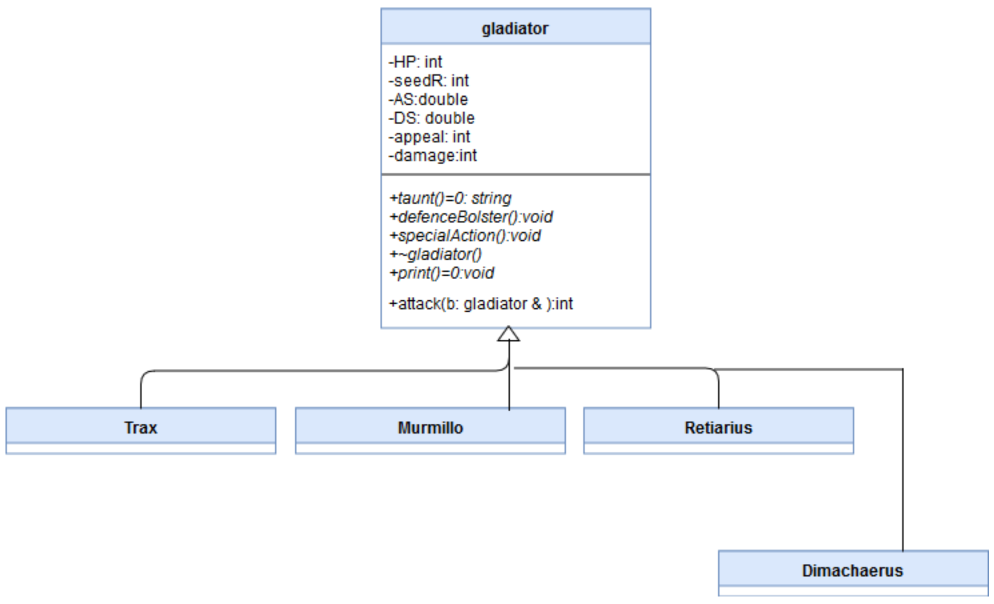

Figure 1: A UML Class diagram showing the class hierarchy

As demonstrated, the **gladiator** class is the parent class which the other 4 are descended from. In particular it is also an abstract class, having a pure virtual function and is not intended to be instantiated. Note that only salient features are captured in the above class diagram. For full specifics on each of the classes, refer below. The class specifics of the subclasses are discussed below with a UML specification and details about what specific behaviour, functions and operators each class needs to implement.

3.1.1 Gladiator Parent Class

```
gladiator
-HP:int
-AS:double
-DS:double
-appeal:int
-damage:int
-seedR:int
---------------------------
+gladiator()
+setRandomSeed():void
+getSeedR() const:int
+setSeedR(s:int):void
+getHP() const:int
+getAS() const:double
+getDS() const:double
+getAppeal() const:int
+getDamage() const:int
+setHP(s:int):void
+setAS(s:double):void
+setDS(s:double):void
```

```
+setAppeal(s:int):void
+setDamage(s:int):void
+attack(glad: gladiator &):int

+~gladiator()
+taunt()=0:string
+print()=0:void
+defenceBolster():void
+specialAction():void
```

The variables are as follows:

- HP: This is the Gladiator's health. A gladiator is able to fight with health less than 0, but doing so decreases their effectiveness dramatically. This reflects the more sportive nature of the arena where the fights are done for entertainment and casualties are by and large, very rare statistical events.
- AS: This is their Attack Skill, and represents the probability of an attack performed by them, succeeding. It will range from 0 to 1.
- DS: This is their Defence Skill, and represents the probability of an attack performed to them, failing in part to their own natural defence and armour. It represents their chance of surviving an enemy attack unscathed. It will range from 0 to 1.
- appeal: This represents their charisma and crowd appeal. The higher this number, the greater the bonus applied to renown scored when a gladiator competes.
- damage: This is the physical damage that a gladiator will cause with their weapons. Higher values represent stronger or more potent weapons and can quickly reduce opponents to 0 health.

The functions are as follows:

- ∼gladiator: This is the default destructor for the class.
- setRandomSeed: This uses the seedR variable to seed the random number generator for the gladiator class.
- taunt: This is a pure virtual function in this class. It will represent a unique and specific taunt that every gladiator class has and can do when they fight. See the specific classes for more details.
- attack: It will be used specifically to conduct a single attack from a gladiator to another. The exact functioning of this follows the given process:
  1. Generate a random number between 0 and 1. If this number is less than the attacking Gladiator's AS, the attack is successful and generates 10 Renown points. If it misses it will instead be -10 Renown points.
  2. Generate another random number between 0 and 1. If this number is less than or equal to the DS of the attacked gladiator, their armour and skill saves them and the attack is nullified. If not the attack will cause them damage.

3. Reduce the attacked gladiator's HP value by the damage value of the attacking gladiator. If this reduces a gladiator to 0 or fewer HP, it adds an additional 30 Renown points.
4. Return the number of Renown points generated plus the appeal of that gladiator as the total Renown for that attack.

This is a process that follows a certain structure. A gladiator will not need to test their DS if an attack does not hit them for example. If the attack misses, it misses and nothing further will happen. If the attack lands, then it should be necessary to check if the attack will bypass defences.

- print: This is a pure virtual function used to print out the gladiator's statistics.
- defenceBolster: This represents a defensive bolster. A gladiator will use this to increase their stats and recover during the fight and depending on the type, different effects will result.
- specialAction: This represents a special action that tries to bolster the gladiator's popularity with the crowd.
- getter and setter: Each of the variables of the class has a getter and a setter assigned to it that simply returns or overwrites the variable's value.

3.1.2 Trax

```
trax
-sprints:int
--------------------------
+getSprints():int
+setSprints(s:int):void
+trax()
+~trax()
+taunt():string
+defenceBolster():void
+specialAction():void
+print():void
```

The variables specific to the Trax are:

- sprints: This is a special stat used to represent the mobility of the Trax Gladiator. They will use their sprints as part of their special action but they only have a finite number of them available in any given fight.

The methods of the class are:

- getter and setter: A specific getter and setter for the sprints variable.
- ∼trax(): This is the destructor for the class. It will print "Trax removed" without the quotation marks and with a new line at the end.

- trax: This constructor has a default profile to use when creating a Trax gladiator. The statistics that are the default values for the class are below:

  1. HP: 16
  2. AS: 0.65
  3. DS: 0.55
  4. appeal: 5
  5. damage: 1
  6. sprints: 5
  7. seedR: 100

- print: This method will print out the statistics of the gladiator in a row by row fashion. For example:

  ```
  HP: 16
  AS: 0.65
  DS: 0.55
  Appeal: 1
  Damage: 1
  Sprints: 5
  SeedR: 100
  ```

- taunt: This method will return a unique message. The message is ”THRACE REMEMBERS!”, without quotations.
- defenceBolster: This bolster will decrease the Trax’s AS by 0.05 and increase his DS by 0.15, representing him focusing more on his greater mobility to the detriment of his short ranged weapon.
- specialAction: When activated, this must check how many sprints the Trax has left. If the Trax has sprints left, he will use one and increase his appeal by 2 and his AS by 0.1.

3.1.3 Murmillo

```
murmillo
---------------------------
+murmillo()
+~murmillo()
+taunt():string
+defenceBolster():void
+specialAction():void
+print():void
```

The methods of the class are:

- murmillo: This constructor has a default profile to use when creating a Murmillo gladiator. The statistics that are the default values for the class are below:

1. HP: 30
2. AS: 0.45
3. DS: 0.75
4. appeal: 1
5. damage: 1
6. seedR: 100

- ∼murmillo: This is the destructor for the class. It will print "Murmillo removed" without the quotation marks and with a new line at the end.
- taunt: This method will return a unique message. The message is "I AM INVINCIBLE!", without quotations
- print: This method will print out the statistics of the gladiator in a row by row fashion. For example:

```
HP: 16
AS: 0.65
DS: 0.55
Appeal: 1
Damage: 1
SeedR: 100
```

- defenceBolster: This bolster will increase his DS by 0.01 and increase his damage by 1, representing him focusing entirely on guarding any attacks coming from his opponent and being more dangerous on a counter-attack.
- specialAction: When activated, the Murmillo will change his fighting style depending on his current condition. Adjust his stats based on the following conditions:
  1. If his health is less than or equal to 5, drop his DS to 0.5 and increase his AS by 0.2. Additionally, increase his damage to 3.
  2. If his health is greater than 5 but less than or equal to 10, drop his AS to 0.35 and increase his appeal by 2.
  3. If his health is greater than 10, increase his appeal by 3.

3.1.4 Retiarius

```
retiarius
-control:int
---------------------------
+getControl():int
+setControl(s:int):void
+retiarius()
+~retiarius()
+taunt():string
+defenceBolster():void
+specialAction():void
+print():void
```

The variables specific to the Retiarius are:

- control: This variable represents the capacity of the Retiarius to use his net to control the enemy fighter. The longer a fight drags on, the more the Retiarius is likely to critically expose an enemy to an attack. This variable has a getter and a setter.

The methods of the class are:

- ∼retiarius: This is the destructor for the class. It will print "Retiarius removed" without the quotation marks and with a new line at the end.
- retiarius: This constructor has a default profile to use when creating a Retiarius gladiator. The statistics that are the default values for the class are below:
  1. HP: 17
  2. AS: 0.70
  3. DS: 0.25
  4. appeal: 6
  5. damage: 3
  6. control: 0
  7. seedR: 100
- print: This method will print out the statistics of the gladiator in a row by row fashion. For example:

```
HP: 16
AS: 0.65
DS: 0.55
Appeal: 1
Damage: 3
Control: 5
SeedR: 100
```

- taunt: This method will return a unique message. The message is "NEPTUNE TAKE YOU!", without quotations.
- defenceBolster: This bolster will increase his DS by 0.1 and his AS by 0.01. However it will decrease his appeal by 1.
- specialAction: When activated, the Retiarius increases his control by 1. If the amount of control he has is a multiple of 2, then he will again an additional 0.05 DS. If it is a multiple of 5, he will gain 2 appeal.

3.1.5 Dimachaerus

```
dimachaerus
---------------------------
+dimachaerus()
+~dimachaerus()
+taunt():string
+defenceBolster():void
+specialAction():void
+print():void
```

The methods of the class are:

- dimachaerus: This constructor has a default profile to use when creating a Dimachaerus gladiator. The statistics that are the default values for the class are below:

  1. HP: 16
  2. AS: 0.75
  3. DS: 0.15
  4. appeal: 10
  5. damage: 4
  6. seedR: 100

- print: This method will print out the statistics of the gladiator in a row by row fashion. For example:

  ```
  HP: 16
  AS: 0.65
  DS: 0.55
  Appeal: 1
  Damage: 4
  SeedR: 100
  ```

- ∼dimachaerus: This is the destructor for the class. It will print "Dimachaerus removed" without the quotation marks and with a new line at the end.

- taunt: This method will return a unique message. The message is "MARS WILL HAVE YOUR HEAD!", without quotations.

- defenceBolster: This bolster will increase his DS by 0.01 and his AS by 0.01. However it will decrease his appeal by 1.

- specialAction: When activated, the Dimachaerus will drop his appeal by 1 to gain more 2 HP.

3.2 Team

The **team** class encapsulates and holds the individual gladiator objects as a collective fighting team. Here is where a number of the operator overloads will be implemented so that creating and managing gladiator teams is an easy process.

```
team
-fightingTeam: gladiator **
-teamSize:int
--------------------------------
+team()
+~team()
+getSize():int
+setSize(s:int):void
+operator[](i:int):gladiator &
+operator+(add:int *):team &
+operator-(sub:int *):team &
+operator--():team
+friend operator<<(output:ostream &,t:const team &):ostream &
```

The variables of the class are as follows:

- fightingTeam: A 1D array of gladiator pointers. This can accept and store and type of gladiator mentioned in the hierarchy.
- teamSize: The size of the teams in terms of the number of gladiators.

The methods have the following behaviour:

- team: This is simply a default constructor for the class. It does not need to do more than instantiate the class.
- ∼team: This is destructor of the class. It must deallocate the memory assigned by the fightingTeam variable.
- getSize: This returns the size of the team;
- setSize: This sets the size of the team.
- operator[]: This is an overload of the subscript operator. It will receive an index, like an array would, and then returns a gladiator reference to the gladiator who is stored at the index passed into this operator. Essentially, it receives an index and then returns the gladiator stored in the team at the index. No need to check if the indices provided are within bounds.
- operator+: This operator adds a team of gladiators by receiving an array of integers. The integer array will contain values 1 to 4, representing Trax, Murmillo, Retiarius and Dimachaerus gladiators respectively. You must insert the type of gladiator specified at the array index into the fightingTeam variable. Remember that the memory will have to be allocated freshly. You can assume the passed in team will match the size of the teams already in use.

- operator-: This is an opposite process of the + operator. It will receive an array of integers which again specifies gladiator types stored in the team. If that gladiator type is found at the index in the passed in array, that gladiator pointer must be deallocated.

  For example, if the fightingTeam has [1,2,1] and then the operator is passed [3,1,1] then the final result will be [1,2,NULL] as only the third index has the same type as the argument passed in.

- operator−−: This operator will deallocate all of the memory assigned to the fightingTeam variable when activated. Essentially this will free up all of the memory that has been assigned. The fightingTeam variable will be NULL after the operation has been completed.

- operator<<: This is an overload of the output operator. This will print out the types of the gladiators contained within the team. For example:

  ```
  Gladiator 1: Trax
  Gladiator 2: Retiarius
  Gladiator 3: Trax
  ```

You will be allowed to use the following libraries: **string, cstdlib, iostream**.

3.3 Arena Class

The **arena** class is where the actual simulation will happen. As specified in the UML diagram below, it will contain a number of functions for constructing gladiator teams and then pitting them against each other in combat.

```
arena
-redRenown:int
-blueRenown:int
-turnLimit:int
-teamSize:int
-randomSeed:int
-redTeam: team*
-blueTeam: team*
-----------------------------
+arena(seed:int, tSize:int,numTurns:int)
+~arena()
+createRedTeam(types:int*):void
+createBlueTeam(types:int*):void
+purgeBlueTeam(types:int*):void
+purgeRedTeam(types:int*):void
+clearBlueTeam():void
+clearRedTeam():void
+runSimulation():void
+displayResults():void
+rollDice():double
```

The variables of the class are described as follows:

- redRenown,blueRenown: Renown governs the success of the gladiatorial team in terms of their appeal to the audience and the spectacle of the fight they put on. Both teams, the red and blue, will have their own separate score that tracks their renown. Actions that they take will increase and decrease their renown and the winner between the two teams is the team with the highest score. Both teams will start with 0 renown and gain it over the course of the battle.
- turnLimit: This variable provides a strict limitation on the number of turns that can be fought in the simulation. Each turn allows all gladiators of both teams to act, so the limit is required to prevent fights from dragging out too long.
- teamSize: This simply specifies how large both teams will be. The teams are of the same size.
- randomSeed: This is a numerical value fed to the random number generator. It will be used to control the randomness of the simulation.
- redTeam,blueTeam: These are pointers to team objects which are used to contain all of the gladiator classes.

The functions of the class are described, with associated behaviour, as:

- purgeRedTeam: This will remove all of the gladiators in the red team that are found in the array that is passed in. Remember that a gladiator is only removed if, at a specified index x, the team and passed in array have the same type of gladiator assigned to that index.
- purgeBlueTeam: This will remove all of the gladiators in the blue team that are found in the array that is passed in. Remember that a gladiator is only removed if, at a specified index x, the team and passed in array have the same type of gladiator assigned to that index.
- clearRedTeam: This will remove all of the gladiators in the red team and free up the memory for usage.
- clearBlueTeam: This will remove all of the gladiators in the blue team and free up the memory for usage.
- arena: This is the constructor for the arena class. When created, the passed in arguments are used to allocate memory for the two teams as well as for seeding the random number generator. The renown of both teams is set to 0 as well as initialising the team variables.
- ∼arena: This is the destructor for the arena. The gladiator classes themselves can make do with a default destructor but the destructer here must deallocate and set to null, both of the teams.

- createRedTeam: This function will receive an integer array with a list of values. The passed in value will be the size of the gladiator team limit created earlier. In this array, 1 stands for Trax,2 stands for Murmillo, 3 stands for Retiarius and 4 stands for Dimachaerus. Using these values, instantiate instances of each gladiator in the position, based on the indices of the passed in array, in the redTeam. For example, the array [1,4,4,2,3] refers to a 5 man team with a Trax in index 0 and a Retiarius in index 4 and so on. This information is passed into the generation of a team object. Ideally this is accomplished through the methods provided by the **team** class. After the team is created, display which of the gladiators were chosen as according to the **team** class.

- createBlueTeam: This function will receive an integer array with a list of values. The passed in value will be the size of the gladiator team limit created earlier. In this array, 1 stands for Trax,2 stands for Murmillo, 3 stands for Retiarius and 4 stands for Dimachaerus. Using these values, instantiate instances of each gladiator in the position, based on the indices of the passed in array, in the blueTeam. For example, the array [1,4,4,2,3] refers to a 5 man team with a Trax in index 0 and a Retiarius in index 4 and so on. This information is passed into the generation of a team object. Ideally this is accomplished through the methods provided by the **team** class. After the team is created, display which of the gladiators were chosen as according to the **team** class.

- rollDice: This function returns a random number generated between 0 and 1.

- displayResults: This will display the results of the simulation. Follow the formatting specifics very clearly. If the two teams have the same renown, then they are tied and the winner is instead "Tied". The format of this is given in an example below:

  ```
  Fight Finished
  Winner: Red Team
  Red Renown: 50
  Blue Renown: 40
  Turns: 15
  ```

- runSimulation: This function does most of the heavy lifting and processing in terms of the arena class. The structure of the simulation consists of a number of turns determined by the turn limit. In each turn, both teams will have actions. Each gladiator of every team will take their actions which will consist of an attack against the corresponding member of the enemy team. For example, a Trax at index 1 in the red team will fight against the Murmillo at index 1 of the blue team.

The runSimulation function itself is quite complicated so a more complete presentation of its workings is needed. The function when called, should conduct a number of turns equal to the turn limit. After this, the combat is at an end and must end. A turn has the following parts:

1. For each gladiator who is below 0 HP, reduce their AS and DS by 0.05. Increase their appeal by 1. This reflects that while becoming a lot less physically capable as fighters, the crowd would love their mounting wounds and persistence to fight.

2. The red team will always act first and then the blue team will take their actions. Each gladiator of the team for the turn now does the following actions:

    (a) Generate a random double number between 0 and 1. If the number is greater than 0.5, the gladiator will perform an attack against their opposite in the other team. Otherwise, they will do a defensive bolster action.

    (b) Generate a random double number between 0 and 1. If the number is greater than 0.85, the gladiator will perform their special action.

3. After all gladiators of one team have performed this sequence, the other team will then do the same sequence, but with them as attackers, and the turn will end.

Teams will gain renown for the following actions:

1. Landing a successful attack provides 10 Renown Points
2. Reducing an enemy to 0 or fewer HP provides 30 Renown Points
3. Missing an attack reduces Renown by 10 points
4. Doing a special action gives 5 Renown Points

The actions are stackable. If an attack lands, it should stack the Renown. For example, if an attack lands and then reduces the enemy to 0 or fewer HP, they accrue 40 Renown before the appeal is added. The appeal, which is added based on the individual gladiator, should be added to the total Renown tally but this is handled in the attack function itself.

3.4 Submission

You will have to submit your code to 3 separate upload slots. You will have a maximum of 5 uploads per slot for this assignment. The slots are very limited so do not rely on Fitchfork to debug your code.

The code that you are specifically required to provide, that is the code that you must implement for each of the slots, is given below:

- Slot 1: **trax.h,trax.cpp, gladiator.cpp, gladiator.h, murmillo.h, murmillo.cpp, retiarius.h, retiarius.cpp, dimachaerus.cpp, dimachaerus.h**
- Slot 2: **team.h, team.cpp,trax.h,trax.cpp, gladiator.cpp, gladiator.h, murmillo.h, murmillo.cpp, retiarius.h, retiarius.cpp, dimachaerus.cpp, dimachaerus.h**
- Slot 3: **arena.cpp, arena.h,team.h, team.cpp,trax.h,trax.cpp, gladiator.cpp, gladiator.h, murmillo.h, murmillo.cpp, retiarius.h, retiarius.cpp, dimachaerus.cpp, dimachaerus.h**

Also remember that for each slot you must provide a **main** as well as a **makefile** for that slot.

You will notice that the slots follow a progression. Part of the challenge of this assignment, and part of the point, is to have experience constructing a relatively large scale system made of many parts that have to operate with each other. The consequence of this, is that later parts of the assignment require working code for the prior parts. Be sure to work incrementally and test as you go to ensure that your simpler classes are working properly before moving onto the later parts of the assignment.