Tommy Hollenberg
CS162 Assign3
Design Document
05/07/2017

Understanding the Problem

This program will be a text-based fantasy role-playing game. There will be four different types of creatures (incorporating a fifth creature type will earn extra credit): Humans, Elves, Balrogs, and Cyberdemons. All creatures have strength and hitpoints stats, as well as a cost to purchase and a reward for defeating them in battle. These basic features should be contained within a Creature class.

However, the types of creatures differ in some ways, most notably in how they attack. There should be a derived class for each creature type that inherits from the Creature base class. The damage inflicted during a round of combat is determined in the Creature::get_damage() function, which returns a random number between 0 and the creature's strength stat. Demons (both Balrogs and Cyberdemons) have a 5% change for a demonic attack, inflicting an additional 50 damage. Additionally, Balrogs get to attack twice each turn. Elves have a 10% for a magical attack that doubles the damage dealt. The classes associated with these creature types will contain their own get_damage() functions that call Creature::get_damage() and then handles the additional effects.

There will be a World class that contains a dynamic array of each creature type, as well as a bank that keeps track of how much money the user has accumulated. The user can add money to the World and use the in-game currency to buy additional creatures to battle.

Creatures will be pitted against each other one pair at a time. When one creature's hitpoints fall below (or equal to) zero, they die and are removed from the World object.

All classes must have constructors and appropriate use of the const qualifier for member functions. The World class must have the Big Three because it contains dynamic memory. All member variables must be private or protected, and must have designated accessor and mutator functions. Class declarations and implementations must be separated into .h and .cpp files, respectively, and there must be a driver file and a Makefile to facilitate compilation. There must be no memory leaks and all functions should be under 20 lines.

Devising a Plan

Game Design

The problem statement leaves a lot of room for personalization in terms of how the game will be played. My game will be "Gladiator Coliseum Tycoon," where you, the King, must buy creatures and pit them against each other to maximize profit from ticket sales.

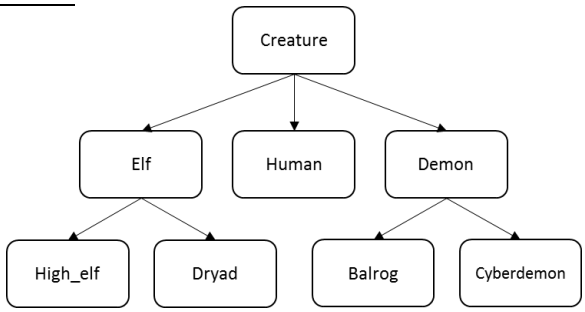
Alongside the 4 creature types listed above, there will also be a Dryad class, which inherits from the Elf class. Non-dryad elves will be associated with a High_elf class to distinguish them. Dryads will have lower attack (although they will still be able to double their damage from a magic attack), but will have a 5% chance to heal themselves to full hitpoints upon attack. Note that this effect prevents get_damage() from being const-qualified, because one of its overrides may update the hitpoints member variable.

All creatures of a certain type will be identical, and will be set up using the class' default constructor, which will call the non-default Creature constructor. An additional stat, speed, will also be included to determine which creature attacks first. The stats of each creature type are shown below:

Stat	Human	High Elf	Dryad	Balrog	Cyberdemon
Strength	50	36	24	20	60
Hitpoints	200	160	240	200	300
Speed	50	80	60	100	40
Payoff	6	8	10	16	16
Cost	10	12	15	20	20

The cost variable is how much each creature costs to recruit, and the payoff variable is how much is earned when they die. However, since the crowd is composed of a human majority and a strong elven minority, when a human or elf defeats an opponent, the payoff is multiplied by 2 or 1.5, respectively. No multiplier is applied when a demon is victorious, because they are evil. The goal of the game is to make a profit by buying gladiators and matching them up to maximize payoffs via the human and elf multipliers.

Class Inheritance Chart:



All members of the Creature, Elf, and Demon classes will be either protected or public, so that their derived classes can access them without using the accessor functions. They will only have non-default constructors, which will be protected, so no instances can be created directly without using a derived class. The most-derived classes (Human, High_elf, Dryad, Balrog, and Cyberdemon) will only have default constructors (which will be public), which will call non-default constructor of the inherited class using the predetermined stat values outlined in the table above.

Virtual functions/Function overriding

The assignment sheet requests that there be separate arrays for each creature type contained in the World class. However, it is still advantageous to use polymorphism when selecting two creatures to battle. For this reason, all Creature class functions that will be overridden in derived classes will be made virtual functions. Even without using Creature pointer polymorphism, get_species() must be a virtual function to be properly resolved when called within Creature::get_damage().

The Creature member functions that will be made virtual are get_damage() and get_species(), both of which are given in the assignment sheet, die(), which prints a unique message when the creature dies, and get_winnings(), which will be overridden in the Human and Elf classes to apply the payoff multiplier discussed above. The get_species() and die() functions will be overridden in all of the most-derived classes. The get_damage() function will be overridden in all classes except Human, which uses Creature::get_damage(), High_elf, which uses Elf::get_damage(), and Cyberdemon, which uses Demon::get_damage(). All overrides of the get_damage() function call the get_damage() function of the closest base class before handling class-specific effects.

Creature member functions that all classes will use (that will not be overridden) include accessors and mutators for the five stat variables.

The World Class

As stated in the assignment sheet, the World class will contain dynamic arrays of all five creature types, accompanied by integers that hold the number of elements. Additionally, there will be three double member variables for the current gold balance, the total amount of gold earned, and the total amount of gold spent. The last two will be used to provide a game summary when the user decides to cash out (end the game). All member variables will be private, but will be accessible outside of the class via public accessor and mutator functions. The World class will have both a default and non-default constructor, as well as user-defined destructor, copy constructor, and assignment operator overload (the Big Three), because it contains dynamic memory. Additionally, it will have a display_creatures() function that prints a numbered list of all creatures that will be used for creature selection for battling, and, of course, the battle function. The battle function logic is shown below. The take_damage() function accepts an integer, reduces the creature's hitpoints by that amount, and returns a Boolean value indicating whether the creature died or not.

```
void battle(Creature *c1, Creature *c2) {
     bool kill = false;
     // If c1 is faster than c2,
          kill = c2->take_damage(c1->get_damage()); // c1 attacks c2.
     while (kill is false) {
          // c2 attacks c1.
          if (kill is true)
               c1->die();
               add_money(c2->get_winnings(c1->get_payoff));
               // c1 dies and c2's multiplier is applied to payoff.
               remove_dead(c1); // remove c1 from the World object.
               return;
          // c1 attacks c2.
     }
     // c2 dies and c1's multiplier is applied to the payoff.
     // remove c2 from the World object.
}
```

The remove_dead() function uses another member function called find_in_array(), which searches an array for an address and returns the array index if found and -1 if not, to search the five dynamic arrays within the World class for the passed Creature pointer. Once found, it calls the appropriate function (i.e. remove_human(int index) to remove a Human object) that handles the array resizing and memory management.

The driver file

The main function will be in a file called driver.cpp. It will #include "Game.h" (discussed below), which itself #includes all the necessary header files. The main function will be very simple. It will seed the random number generator by calling srand(time(0)), print a "Welcome to Gladiator Coliseum Tycoon!" message, as shown below , and perform a short loop that declares a Game object (see below) and calls Game::play() until the user chooses to quit the program.

```
Welcome to Gladiator Coliseum Tycoon!

1. Start a new game
2. Quit
```

The Game Class

The Game class will contain all the functionality needed to play the game, and will have as members a World object, a string to hold a username, and a double that will track the amount of gold invested by the user, all private. The only public members are the default constructor, and the play() function. The play() function gets the user's username, and then calls the menu() function.

The menu() function is the main hub that will be returned to repeatedly until the user decides to end the game. The user will be given 4 options:

```
King Tommy, what is your next command?
        1. Begin tournament.
        2. View/Recruit gladiators.
        3. Invest money.
        4. Sell assets.
```

Begin tournament calls start_tournament(), but cannot be accessed until the user has recruited some gladiators. This can be done by selecting option 2, which calls the buy_creatures() function. This function calls World::display_creatures(), tells the user how much gold they have available, and gives them a list of recruitment options. Once a creature type is chosen, the user can enter how many they would like to recruit. They are denied if they do not have enough gold to cover the recruitment cost.

```
Gladiators:
        None

0 gold available.
Recruit:
        1. Humans (10 gold).
        2. High Elves (12 gold).
        3. Dryads (15 gold).
        4. Balrogs (20 gold).
        5. Cyberdemons (20 gold).
        6. Done.

1
How many? 3
Not enough money.
```

Of course, nothing can be bought until the user invests money by choosing option 3 in the menu, which calls Game::add_money() (not to be confused with World::add_money()). This function asks how much the user would like to invest, and facetiously asks for credit card information. However, anything can be entered, even just pressing enter works.

```
How much money would you like to invest? 500
Credit card number: In
Expiration Date (MM/YYYY): Your
CVV code: Dreams
500 coins added to the bank.
```

After the user has added money and recruited some creatures as gladiators, option 1 can be chosen on the menu to call start_tournament(). This function calls World::display_creatures() and gets two gladiator numbers from the user. These numbers must be different, because a gladiator cannot fight itself. These numbers are passed to the World::get_creature() function, which returns the address of the gladiators' objects, which are assigned to two Creature pointers that are passed to World::battle() to make them fight to the death. When only one gladiator remains, the function returns to the menu, where more gladiators must be recruited before another tournament can be held.

Example Battle (this is kind of a lame one, but otherwise the text would take up too many pages):

```
What a beautiful day for some carnage! The public agrees; the coliseum
is packed.

Gladiators:
1. Human (Hitpoints: 200)
2. High elf (Hitpoints: 160)
```

```
Pick a pair of combatants.
First combatant number: 1
Second combatant number: 2

High elf attacks for 31 points!
Human takes 31 damage!
Human hitpoints: 169
High elf hitpoints: 160

Human attacks for 21 points!
High elf takes 21 damage!
Human hitpoints: 169
High elf hitpoints: 139

High elf attacks for 2 points!
Human takes 2 damage!
Human hitpoints: 167
High elf hitpoints: 139

Human attacks for 11 points!
High elf takes 11 damage!
Human hitpoints: 167
High elf hitpoints: 128

High elf attacks for 35 points!
Human takes 35 damage!
Human hitpoints: 132
High elf hitpoints: 128

Human attacks for 14 points!
High elf takes 14 damage!
Human hitpoints: 132
High elf hitpoints: 114

High elf attacks for 34 points!
Human takes 34 damage!
Human hitpoints: 98
High elf hitpoints: 114

Human attacks for 47 points!
High elf takes 47 damage!
Human hitpoints: 98
High elf hitpoints: 67

High elf attacks for 18 points!
Magical attack doubles the damage!
Human takes 36 damage!
Human hitpoints: 62
High elf hitpoints: 67

Human attacks for 45 points!
High elf takes 45 damage!
```

```
Human hitpoints: 62
High elf hitpoints: 22


High elf attacks for 36 points!
Human takes 36 damage!
Human hitpoints: 26
High elf hitpoints: 22

Human attacks for 25 points!
High elf takes 25 damage!
Human hitpoints: 26
High elf hitpoints: -3
The high elf calmly accepts his fate and vanishes in a poof of dust.
16 coins added to the bank.
The crowd goes wild!
```

When the user decides the gladiator market has reached its peak, option 4 can be chosen in the menu to sell assets and get out of the game. The summarize() function is called, which gives a summary of money invested, spent on gladiators, and earned from payoffs throughout the game, as shown below.

```
King Tommy, you:
        invested 200 gold,
        spent 200 gold, and
        earned 230 gold,
        allowing you to sell the coliseum
        at a valuation of 230 gold
        for a 30 gold profit.
```

Testing Plan

As always, testing was done incrementally as I put together each piece of the program. I used valgrind to ensure that no memory leaks occurred when adding and removing creatures from the World object. I used the validation library written for the extra credit of assignment 2 to get integer and double input within a desired range.

I feel that my explanation of the program design and the sample outputs I have included demonstrate the effort and thought I have put into testing the program thoroughly.