Brian Palmer
Assignment 4
November 4, 2016
palmebri@oregonstate.edu

Assignment 4 Design and Reflections

Introduction

Assignment 4 culminates material from many of the labs and assignments this term. In this assignment, our goal was to create battle simulator with the creatures from last week. The user would play against the computer, each player getting 5 creatures to battle. The user gets to choose their creatures whereas the computer has five randomly assigned ones. The creatures are placed in order as they are added in a “lineup”, or a queue-like data structure, the first added creature being the first creature to battle. In battles, both creatures “popped” from their lineup to face each other. The loser goes to the players loser pile and the winner is “pushed” to the back of the winning player’s lineup. Each battle displays the matchup, descriptive events so that the user can follow exactly what occurred, the winner of the battle, and the updated lineups. These battles continue until one player loses all of their creatures. At the end, descriptive statistics display the winner of the tournament, the creatures each player lost, and point totals.

Design

The Creature base class and derived classes were copied from assignment 3. Below are the descriptions of each class from that design document. The only addition was the heal() public member function to the Creature base class indicated in bold.

Base Class:
Creature - This will be the base class of the creatures
Protected Members

- Num attack die - the number of die represents the number of times the creature can attack. The sum of each roll is the total attack.
- Num defense die - the number of times the creature can roll their defense die. The sum is the total defense the creature puts up.
- Num atk die sides - the number of sides of the attack die represents how much power their attack can have.
- Num def die sides - the number of sides of the defense die represents how much defense the creature puts up, per roll
- String type - the type is the name of the creature
- Int armor - the amount of armor the creature is wearing
- Int strength points- the amount of health that the creature has. This value can change. When it reaches zero, the creature is dead.

- **Int Max_strength_pts - max health of the creature. Initialized when creature is constructed, and is used to calculate how much creature should heal in heal() function.**

Protected helper functions

- int Armor_effect (int attack) - After the creature defends itself from an attack, the creature's armor value is subtracted from the difference. Each derived creature needs this function. It will not be virtual because as of now, no planned derived creature has a special ability that affects how their armor affect an attack.
- **Virtual** void Reduce_str_points (int attack) - this function takes the remaining attack (after the creature defends itself and the armor effect is determined) and subtracts that value from the creature's strength points value. This is a **virtual** function because **Harry Potter** can return from the dead (when strength points equal zero) and that logic will be nested within this function.

Public Functions

- **Heal() - This function adds a quarter of the creatures total health to the creatures strength points if they win their battle. For example, if a creature starts with 20 health points, they will heal 5 health points after every battle they win.**
- Get_strength_points - this returns the number of strength points. This will be useful if at later points in this project functions need to be written that will be dependent on this value.
- Get _type - this returns the creature's type. Each derived creature will have their own type, so this function will let the user know which creature they are working with.
- **Virtual** int attack() - this is the basic attack function that all derived creatures will use, unless they have a special ability. In that case, they will override this function with their own **virtual** attack function.
- **Virtual** void defend(int attack) - this is the basic defense function that all derived creatures will use. The function takes the attack and passes that value to the armor effect and reduce strength points helper functions.
- Creature Constructor (number of attack die, number of defense die, attack die sides, defense die sides, type, strength points, and armor) - The base class will set these values for all derived creatures since all derived creatures inherit the same data members. Each derived creature from this template will have default values for each data member in their own constructor that are passed to this constructor constructor. The number of attack die sides and number of defense die sides will be used to construct the creatures attack die and defense die, respectively.
- **Virtual** Destructor - Deletes the defense die and attack die that each class has.

Die class - This class will be used to create the attack die and defense die for each creature

Private data members

- Number of sides - Represents the number of sides on the die.

Public member functions

- Int roll() - Rolls the dice. A random number is generated between zero and the number of sides the die has, and is returned.
- Constructor(int number of sides) - Constructor creates a die with the passed number of sides
- Destructor - No dynamic memory is used with this class

Derived Classes:
These classes are derived from the Creature base class.

Class Barbarian:
The Barbarian has no special ability so it does not need to override any of the creatures members.
Public members

- Constructor - values passed to the base class are:
    - Num_atk_die_sides = 6
    - Num_atk_die = 2
    - Num_def_die_sides = 6
    - Num_def_die = 2
    - Armor = 0
    - Str_pts = 12
    - Type = “Barbarian”
- Destructor - No dynamic memory is allocated within the derived class, so left blank

Class Vampire
Protected members:

- **Virtual** void defend(int attack) - Vampire has a Charm ability, which can negate an attack. Vampire has a 50 percent chance of charming their opponent. If they charm their opponent, the function returns zero. The zero is passed to the other helper functions that it inherits from Creature.

Public members:

- Constructor - values passed to the base class are:
    - Num_atk_die_sides = 12
    - Num_atk_die = 1
    - Num_def_die_sides = 6
    - Num_def_die = 1
    - Armor = 1
    - Str_pts = 18
    - Type = “Vampire”
- Destructor - No dynamic memory is allocated within the derived class, so left blank

Class Blue Men
Public Members:

- **Virtual** void Defend(int attack) - For every 4 damage they receive, the blue men lose a defense die.
- Constructor - values passed to the base class are:
    - Num_atk_die_sides = 10
    - Num_atk_die = 2
    - Num_def_die_sides = 6
    - Num_def_die = 3
    - Armor = 3
    - Str_pts = 12
    - Type = “Blue Men”
- Destructor - No dynamic memory is allocated within the derived class, so left blank

Class Medusa

Public Member

- **Virtual** int attack() - Medusa’s Glare ability can turn a creature to stone. If the Medusa rolls a 12, then the medusa uses glare. Glare will KO it’s opponent (even Vampires. Vampires will not be able to Charm away a Glare).
- Constructor - values passed to the base class are:
    - Num_atk_die_sides = 6
    - Num_atk_die = 2
    - Num_def_die_sides = 6
    - Num_def_die = 1
    - Armor = 3
    - Str_pts = 8
    - Type = “Medusa”
- Destructor - No dynamic memory is allocated within the derived class, so left blank

Class Harry Potter

Private Members

- Bool second life = if true, harry potter has a second life. If he dies, then he gets 20 health points and the member is set to false

Protected Members

- **Virtual** void Reduce_str points(int attack) - If harry potter reaches zero strength points but has a second life, then he uses his second life which gains him 20 strength points. Once used, second life is set to false

Public Members

- Constructor
    - values passed to the base class are:
        - Num_atk_die_sides = 6
        - Num_atk_die = 2
        - Num_def_die_sides = 6
        - Num_def_die = 2

        - Armor = 0
        - Str_pts = 10
        - Type = “Harry Potter”
    - Private members set
        - Second life = True
- Destructor - No dynamic memory is allocated within the derived class, so left blank

Game Class

Members

- Player1 - pointer to Player. Will be User.
- Player2 - pointer to Player. Will be Computer.
- String winning - string of either winning player, or tie

Private member functions

- Void set_lineups() - sets lineups for Players. User will choose their creatures and a random selection will be made for the Player 2 (CPU)
- Creature * battle(Creature *, Creature *) - the two creatures taken as parameters will fight each other. The winner is returned.
- Void simulate_battles() - gets the front creature from both users lineups, pits them against each other, returns the winner to the lineup, pushes the loser to the losing players fainted creature stack, and lastly updates the scoreboard
- Void print_stats() - displays each players current lineup, current fainted creature stack, wins, losses, points, and lastly the winner
- add_creature_to_lineup(Creature *) - takes users Creature choice and pushes it into their lineup queue
- Creature * get_users_creature_choice() - gives user list of creatures to choose from, allocates their choice, and returns the pointer to the allocated creature.
- Setup_computer_player() - randomly chooses 5 creatures and pushes them into the players lineup
- Void update_winning() - compare both players point totals and either assigns a string of the winning player, or a string stating the game is tied
- Void press_enter_to_continue() - this function pauses the screen after every battle so that the user can either read each event that occurred during the battle, see current lineup, see who is currently winning, etc. This code was inspired by an example from http://www.cplusplus.com/forum/articles/7312/.

Public Functions

- start() - starts the game.
- Constructor and destructor

Class Player

Private members

- Queue * lineup - pointer to queue class. Queue will hold players healthy creatures.
- Stack * fainted - pointer to stack class. Stack will hold players fainted creatures.
- Int wins - player’s number of wins

- Int losses - player's number of losses
- Int points - player's number of points

Private Member functions

- Void Update_points - calculates and sets users points depending on number of wins and losses

Public Member Functions

- Queue * get_lineup - Returns pointer to players healthy creature lineup
- Stack * get_fainted - Returns pointer to players fainted creature stack
- Void Display_stats - displays number of wins, losses, and total points
- Int get_points - returns number of points that the user has
- Void won() - updates wins member and calls update points member function
- Void lost() - updates losses member and calls update points member funciton
- Constructor and Destructor

Class Stack

Private Members

- Node * top - points to the front member of the stack

Public Member functions

- Creature * pop() - Removes the top node of the stack and returns it
- ~~Creature * peek() - Returns a pointer to the first member of the stack. If there is nothing in the stack, it returns a NULL pointer. This serves as a sentinel that the stack is empty.~~ **Is_empty() can be used instead.**
- Void push (Creature *) - Takes creature pointer and assigns it to a new node, which is added to the back
- Bool is_empty() - returns true if the stack is empty
- Void Display() - displays contents in list

Class Queue

Private Members

- Node * front - points to first node in queue
- Node * rear - points to last node in queue

Public Member Functions

- Void Push(Creature *) - pushes creature pointer onto stack
- Creature * pop() - removes and returns top creature on stack
- Bool is_empty - returns true if the stack is empty
- Void display() - displays contents in list
- ~~Creature * peek() - Returns a pointer to the first member of the queue. If there is nothing in the queue, it returns a NULL pointer. This serves as a sentinel that the queue is empty.~~ **Is_empty() can be used instead.**

Tests

Tests	Plan	Expected Result	Observed Result
Basic Attack works correctly			
Barbarian rolls correct number of times	Create Barbarian Class	Barbarian rolls twice	Barbarian rolls twice
Barbarian die lands between 1 and number of sides	Use attack function	each roll is between 1 and 6	each roll between 1 - 6
Basic Defense works correctly			
Barbarian rolls correct number of times	Use defend function (ignore passed parameter)	Barbarian rolls twice	Barbarian rolls twice
Barbarian subtracts attack from defense	pass attack value 5 to defense function	should find difference; should NOT be a negative value	found correct difference
Armor is subtracted from difference	continue using defense function	should return difference; should NOT be a negative value	correct difference was returned
health points is subtracted from difference	continue using defense function	returns expected difference; should NOT be a negative value	correct difference returned
Derived Creatures special abilities properly override Creature class basic attack and defense functions			
Harry Potter Revives	Have Harry fight 100 battles against all creatures (500 total)	When he is KO'd, his health should reach zero. Then a message to the console will let the user know Harry is alive, and will display his new health points (from member)	Console displays as expected
Medusa Glares	Have Medusa fight 100 battles against all creatures (500 total)	When Medusa Glares, a message should appear on the screen letting the user know. A second message should appear letting us know that the opponent has lost all of it's strength points.	Console displays as expected
Blue Men Mob	Have Blue Men fight 100 battles against all creatures (500 total)	The conole will display how much damage was done to the Blue Men and how many defensive rolls they take. The number of rolls should diminsh by one for every 4 attack points	Console displays as expected
Vampire Charms	Have Vampire fight 100 battles against all creatures (500 total)	The console will display each time the vampire charms their opponent. The battle will end at that point and no damage will be done. The console will also display Vampire's current strength point count.	Console displays as expected
Creatures heal properly			
Winning creatures heal	After battle, check console to make sure function ran and compare against healthy queue (displays HP of each creature)	Creatures health points should be correct sum of health points remaining and health points gained.	Sum is correct
Lineups are properly created for the user and for the computer			
User can choose 5 creatures and they are added correctly	User chooses 5 creatures	Lineup should display everytime the user adds a creature, and after adding all 5 creatures	Console displays as expected
5 creatures are randomly chosen for computer	5 creatures created for computer	Computer players lineup should display at the end before any battles take place, so the user can see what creatures they are facing	Console displays as expected
Healthy creature queue and fainted creatures stack are robust			
Creatures popped from Player 1 and Player 2's lineups succesfully	Creatures are passed to battle function and they will duke it out	Creatures will attack and defend each other correctly. Each creature displays to the console.	Correct creatures battle and display on console
Losing Creature is pushed on top of stack	After the tournament, the fainted creatures will be	Record which creatures faint and the order they faint in. At the end of the simulation, check that they print out in correct order	Order is correct
Winning creature is pushed to back of healthy creature lineup	Battle creatures. The winner should be returned from the battle function and pushed back into the lineup	After the battle, the current lineups will display. The winner should appear at the back of the lineup.	Creatures are pushed onto active lineups correctly

Reflections

I thought that this assignment was fun to put together. Many components were from previous labs and past assignments, and it was neat to see everything come together. I was able to put together a majority of the project by reusing previous code, and then tweaking for it's new purpose. For example, I decided to reuse the Queue and Stack classes from a previous lab (Lab 6 I think?) because most of the functionality I needed was already there. I just needed to repurpose the code so that the value each node held was a Creature pointer instead of an integer.

I thought the hardest part of the lab was implementing the queue and stack appropriately, and implementing the interface. During my first attempt of the queue, I was getting a segmentation fault. After some testing, it turned out that I was calling the get_type() member function on a NULL pointer. I spent a fair amount of time on that issue because I was focusing on the queue too much, and not on what functions I was calling from the Creature I “popped” from the queue. The interface was also difficult just because there were a lot of requirements to meet. For example, showing the current scores, current winner, and current lineups after each battle required some research and tinkering.