Abreu, James
CS 162, Week 7
Assignment 4, Reflection

Assignment: Assignment 4 – Containers

Date Begun: November 3rd, 2016 (built off of assignment 3)

Due Date: November 20th, 2016, @ 11:59pm

Note: This design document was designed in landscape orientation to allow more space for tables and information.

Document Contents:

Click any of the links to jump to the corresponding section:

- Design Description
  - Tasks and Questions
  - Class Design
  - Other Functions
- Test Plan
- Test Results
- Comments

Design Description

Tasks and Questions: First I will begin with asking myself questions about the program. I'll organize these questions into bullet points, and refer to these questions as I design the program:

Goals

- Use linked structures to hold and manipulate data
- Modify an existing parent abstract class (for use by the derived classes)

Initial Reaction to reading assignment: Build a task list:

- ✓ Clean up previous program and prepare for its use in this assignment:
  - Clean up the battle code by breaking it into smaller chunks and using nested functions. Create a function for each logical action (special moves, roll attack, roll defense, take damage, etc..)
  - Create a bool function parameter for all functions that controls whether or not information is printed to the screen (which determines whether or not functions have side effects)
    - Adding the bool parameter for printing will allow me to get rid of the function used to run battles without displaying results (used for tests previously), making the original function more useful/flexible.

- Update menu system with new menu system that can take in any amount of menu options by passing in an array of strings as a parameter instead of passing in individual strings.
- Combine creature files into ONE file containing all of the definitions of the creature classes. Do the same for the hpp file. (This is because they are all small files, no need to be separate.)
- Update main menu look.
- Clean up any other messy code.

✓ Enter number of fighters both players will use
- Let user also select the type of monsters they would like on their team

✓ The selected fighters for each team will be placed in a linked list container

✓ Create a class called Team that will contain the linked list, as well as functions related to the linked list.
- I will use “head” and “tail” nodes to make formatting easier.
- The list will be doubly linked, for flexibility.
- An object of the Team class will be created as containers for both the active Teams as well as the Loser pile for each team (for a total of four Team objects).

✓ Before a battle, creatures will be “popped” from their Team containers when they are chosen for battle, much like how fighters in a gladiator arena are pulled from their barracks when it is time to fight.

✓ Creatures will battle each other in a lineup style format

✓ Create a function to heal a player after combat

- I will make this a function of the base Creature class since all creatures will use it.

✓ After two fighters have battle against each other, the winner will invoke the heal function and the loser will be pushed to the loser container (also a linked list).

✓ Create a system for printing Tournament results to the screen

- There should be an option to display all battle results, or simply just see the end of the Tournament result.

✓ Determine and print the result of the winner of the Tournament

- I will also display the remaining and defeated creatures for each team.

✓ Check to make sure there are no memory issues when running a Tournament several times.

- Are the dynamically allocated nodes getting deleted properly?

✓ Update design document class descriptions with new cleaned-up classes.

Class Design (shaded green): Next, before I begin coding I will consider the names of member variables and functions that I will need to use in my program. This way, I can correct any logical mistakes easily without the need to change any code:

Class Name	Overview	Member Variables	Member Functions
Die	A simple class that	• int sides	• int rollDie(int forRandom) // returns a random number between 1 and sides member variable.

Creature	Creature is an abstract base class.
	
	Uses an enum for type:
	enum CreatureType {
	TYPE_NULL = 0,
	TYPE_VAMP, // 1
	TYPE_BARB, // 2
	TYPE_BLUE, // 3
	TYPE_MEDU, // 4
	TYPE_HARR // 5
	};
		• int type
		• int numberAttackDie
		• int sizeAttackDie
		• int numberDefenseDie
		• int sizeDefenseDie
		• int armor
		• int strengthFull
		• int strengthCurrent
		• Die* attackDie(sizeAttackDie)
		• Die* defenseDie(sizeDefenseDie)
		• bool usedHogwarts
			• get and set functions for: type, numberAttackDie, sizeAttackDie, numberDefenseDie, sizeDefenseDie, armor, strengthFull, strengthCurrent, attackDie, usedHogwarts
			• int rollAttack(int random)
			• int rollDefense(int random)
			• int healCreature(int random)
			• constructor: // defaults values:
			○ type: 0 // NULL
			○ numberAttackDie: 0
			○ sizeAttackDie: 0
			○ numberDefenseDie: 0
			○ sizeDefenseDie: 0
			○ armor: 0
			○ strengthFull: 1
			○ strengthCurrent: 1
			○ AttackDie: NULL
			○ defenseDie: NULL

Class Name	Overview	Member Variables	Member Functions
Vampire : Creature	charm: “Vampires can charm an opponent into not attacking. (50%) chance.”		• constructor: ○ type: 1 ○ numberAttackDie: 1 ○ sizeAttackDie: 12 ○ numberDefenseDie: 1 ○ sizeDefenseDie: 6 ○ armor: 1 ○ strengthFull: 18 ○ strengthCurrent: 18

Class Name	Overview	Member Variables	Member Functions
Barbarian : Creature			• constructor: ○ type: 2 ○ numberAttackDie: 2 ○ sizeAttackDie: 6 ○ numberDefenseDie: 2 ○ sizeDefenseDie: 6 ○ armor: 0 ○ strengthFull: 12 ○ strengthCurrent: 12

Class Name	Overview	Member Variables	Member Functions
BlueMen : Creature			• constructor: ○ type: 3 ○ numberAttackDie: 2 ○ sizeAttackDie: 10 ○ numberDefenseDie: 3 ○ sizeDefenseDie: 6 ○ armor: 3 (begins with 3) ○ strengthFull: 12 ○ strengthCurrent: 12

Class Name	Overview	Member Variables	Member Functions
Medusa : Creature			• constructor: ○ type: 4 ○ numberAttackDie: 2 ○ sizeAttackDie: 6 ○ numberDefenseDie: 1 ○ sizeDefenseDie: 6 ○ armor: 3 ○ strengthFull: 8 ○ strengthCurrent: 8

Class Name	Overview	Member Variables	Member Functions
HarryPotter : Creature			• constructor: ○ type: 5 ○ numberAttackDie: 2 ○ sizeAttackDie: 6 ○ numberDefenseDie: 2 ○ sizeDefenseDie: 6 ○ armor: 0 ○ strengthFull: 10 ○ strengthCurrent: 10

Class Name	Overview	Member Variables	Member Functions
Game	• Prompting the user for single battle or Tournament settings • Beginning the single battle or Tournament with the desired settings • Beginning each turn • Determining the outcomes of combat (see below question about how combat is resolved) • Determining game winning conditions (is the defending player's strength <= 0?)	• Creature* CreatureLeft • Creature* CreatureRight • int battleTurn • Team* teamLeft • Team* teamRight • Team* teamLeftDefeated • Team* teamRightDefeated	• Get and set functions for CreatureLeft, CreatureRight, battleTurn, and all Team pointers. • void incrementBatttleTurn() • void displayStats() • void promptGameSettings() • void singleCombat() • void beginTurn() • void resolveCombat(Creature* attacker, Creature* defender, int attackingPlayer) • string printType(int type) • Creature* allocateCreature(int type) • void Hogwarts • bool isCharmed • void useGlare • void runMob • int rollDefender • int rollAttacker • void defenderTakeDamage

			• void setTournamentSettings
			• void beginTournament
			• int tournamentRound
			
			• void testDriver()
			
			• constructor:
			  ○ battleTurn: 1
			  ○ CreatureLeft: NULL
			  ○ CreatureRight: NULL
			  ○ setTeamLeft(NULL)
			  ○ setTeamRight(NULL)
			  ○ setTeamLeftDefeated(NULL)
			  ○ setTeamRightDefeated(NULL)

Other Functions (shaded blue): These functions do not need to be a class, but it would be helpful to organize this code into these functions to break the program into smaller chunks to work with, aka encapsulation.

Function Name	Arguments	Return Value	Description
menuSelect	4 strings for the different menu options	int	Used to display menu options to user, returns users selection. Uses input validation.
inputValidation	reference to variable being checked, int min, int max	bool	Used to check if the input for an int is within a range given.
main		int	The main function will be responsible for: • Display a menu for starting the game, instructions, about author, and exit program • Creating a game object that will be used to run a game • Destroying a game object

struct name	Description	Members
CreatureNode	used by the Team class as a linked list. Holds pointers to Creature objects (dynamically allocated)	CreatureNode* next CreatureNode* previous Creature* ID

Test Plan (shaded yellow)

Test Name	Reason for Test	Test Description
Input Validation (white box testing)	There are several functions that require input from the user. I will need to test the input from the user to make sure it is valid and garbage values are not given to the program.	For input validation, I will use while loops to check the values the user has given. If not valid, the user will be told about the correct format of the input required and prompt the user to again enter the required value.
Human Testing (black box testing)	To ensure the design of the program makes sense to a human, I will invite friends or family close to me to run my program.	The friend or family member will sit at my desk and I will simply run the executable. As they are using the program, I will observe the person's reaction towards the experience, and note any confusion in the design, any bugs that occur, and the overall experience and reaction of the person using the program.
Test Driver (white box testing)	The test driver will be responsible for pitting each Creature type to battle each other creature type (including itself). This can be used for two purposes: to tweak game settings to balance gameplay, and to make sure that each class works as it should.	I will create a member function of the Game class that will be used for testing purposes. It will simulate not only each test individually, but I will also use enums to my advantage to automate the setup process.
Test Driver (Tournament)	Various amounts of teams with various types of creatures will be pitted up against each other. The test is not	I will use various kinds of test input and record the results below.

	designed to balance gameplay, rather to just make sure that no matter what options are chosen, the program does not crash or show erroneous results.	

Test Results (shaded red)

Input	Output

• Team Size: 2
• Creatures Player 1: BlueMen, Medusa
• Creatures Player 2: Vampire, Barbarian
 	
Test Description: small Team size, randomly chosen creature types for each Team.
	• Number of Rounds: 2
	• Which team won: Team 1
	• Round Results:
	  ○ Round 1: Team 1 wins
	  ○ Round 2: Team 1 wins
	• Survived Creatures Player 1: BlueMen, Medusa
	• Survived Creatures Player 2: none

• Team Size: 12
• Creatures Player 1: Barbarian, Barbarian, Medusa, Vampire, Barbarian, Medusa, BlueMen, Vampire, BlueMen, Medusa, Medusa, Barbarian
• Creatures Player 2: BlueMen, Vampire, BlueMen, Harry Potter, BlueMen, Harry Potter, Harry Potter, Barbarian, Medusa, Barbarian, Harry Potter, BlueMen
 	
Test Description: large Team size, randomly chosen creature types for each Team.
	• Number of Rounds: 23
	• Which team won: Team 1
	• Round Results:
	  ○ Round 1: Team 2 wins
	  ○ Round 2: Team 1 wins
	  ○ Round 3: Team 2 wins
	  ○ Round 4: Team 2 wins
	  ○ Round 5: Team 2 wins
	  ○ Round 6: Team 1 wins
	  ○ Round 7: Team 1 wins
	  ○ Round 8: Team 2 wins
	  ○ Round 9: Team 1 wins
	  ○ Round 10: Team 2 wins

	○ Round 11: Team 2 wins ○ Round 12: Team 2 wins ○ Round 13: Team 2 wins ○ Round 14: Team 1 wins ○ Round 15: Team 1 wins ○ Round 16: Team 1 wins ○ Round 17: Team 1 wins ○ Round 18: Team 2 wins ○ Round 19: Team 2 wins ○ Round 20: Team 1 wins ○ Round 21: Team 1 wins ○ Round 22: Team 1 wins ○ Round 23: Team 1 wins • Survived Creatures Player 1: BlueMen • Survived Creatures Player 2: none
• Team Size: 6 • Creatures Player 1: Vampire, Vampire, Vampire, Vampire, Vampire, Vampire • Creatures Player 2: Vampire, Vampire, Vampire, Vampire, Vampire, Vampire Test Description: medium Team size, one creature type chosen for all Teams.	• Number of Rounds: 9 • Which team won: Team 1 • Round Results: ○ Round 1: Team 1 wins ○ Round 2: Team 2 wins ○ Round 3: Team 1 wins ○ Round 4: Team 1 wins ○ Round 5: Team 1 wins

	○ Round 6: Team 2 wins ○ Round 7: Team 1 wins ○ Round 8: Team 2 wins ○ Round 9: Team 1 wins • Survived Creatures Player 1: Vampire, Vampire, Vampire, • Survived Creatures Player 2: none

Comments (shaded grey)

Problems I Came Across	How I solved it:
• Needed a way to allocate any class type depending on user input.	• created a new function called Creature* allocateCreature(int type) that dynamically allocates the correct type of creature depending on the int passed in.
• When printing the nodes in the Team queue, if there was no nodes or one node, grammatical errors occurred.	• I added a counter int that increments each time the while loop is ran. That way I can keep track of how many times the iterator has passed through the while loop. If it only passed through once, there is no need for the word "and". And if it didn't pass through at all then the list must be empty and the word "(none)" is printed instead.
• The amount of files in the project folder was getting overwealming.	• I've reduced the amount of files for the program by consolidating all of the class header and definitions into two primary files: creatureTypes.hpp and creatureTypes.cpp. This saved a lot of wasted space.
• Working with the logic of creatures fighting each other is getting confusing.	• I've abstracted out logical chunks of the battle system to make it easier to read. Now there are separate functions for special moves and rolling for attack and defense for example.
• The only difference between two battle functions is that one prints out things and the other doesn't	• I've added a parameter for the battle function (a bool), to print out action descriptions only when the bool argument is true.