**Group Project Design + Reflection**

**Program Design**

**Problem**

In the Predator-Prey game, we will use cellular automata to create a 2D predator–prey simulation in your program. The preys are ants and the predators are doodlebugs.

**Predator-Prey Game Requirements**

Ants and doodlebugs live in a 20 \* 20 grid of cells. Only one critter may occupy a cell at a time. The grid is enclosed and no critter may move off the grid. Time is simulated in steps. Each critter performs some actions every time step.

**Input Validation:** We will recycle our InputValidation class used in Project 1. It has an overloaded function called getInt(). I chose to make InputValidation its own class so that it could be used in future programs. getInt() receives input from the user, converts it into a stringstream, which then checks to see if there is anything beyond an integer that is left over. If there is, it makes the user try to input an integer again.  
getInt(int min, int max) serves the same function as getInt() but also runs another validation checking input to see if it lies between min and max. getInt(min, max) will be useful for specifying minimum and maximum values for the following inputs:

* Menu options for continuing the game
* Menu options for buying a new adult animal
* Specifying at the beginning of the game how many of each animal type to start with (1 or 2)

**Menu: We will recycle Eric’s integer validation and menu functions.   
getValidInteger() checks input to see if it is in fact an integer using stringstream.  
displayMenu() displays choices, gets input from a user, checks to see if it is within valid range of the choice selection, then returns the choice value as an integer.**

**Critter Class:** The Critter class contains data and functions common to ants and doodlebugs.   
The Critter class will contain the following variables: row position, column position, and an enum variable representing the critter’s orientation (UP, RIGHT, DOWN, LEFT). The enum orientation variable will be useful when picking a random direction for each critter to move.   
  
This class will have a virtual function named move that is defined in the derived classes of Ant and Doodlebug. Thus, the Critter class is an abstract base class. It will also have a virtual function named breed defined in Ant and Doodlebug.

**Tiger, Penguin, and Turtle Classes:** These three classes will be implemented almost identically. Each class is a specific class that inherits the properties of the Animal class and assigns its properties to its own specific attributes based upon the specs given. Most functions/data will be used from the Animal class.

**Zoo Class:** Zoo will be the main engine class for the Zoo Tycoon game. It will hold much of the important information including: money, the creation and destruction of animal cages, each animal’s capacity, the current number of each animal, and random events.

There will also be many functions that run the game:

startZoo - startZoo asks the user how many of each animal (1 or 2) that the user would like to start the game with.  
printStats - printStats prints the stats of the zoo for each day, including: money, number of tigers, penguins, and turtles.  
liveDay - liveDay simulates a day in the zoo by calling many of the other functions that represent events of that day: printStats, age, payFoodCosts, randomEvent, getPaid, and buyAdult.  
addAnimal - addAnimal takes in three parameters: animalType, age, and qty. These parameters help in the creation of the correct animal in the zoo.  
rebuildCages - rebuildCages expands the dynamically allocated array in the case of the number of a certain animal type growing beyond what the array can hold. It expands the array by a factor of 2.  
spend - spend subtracts the cost parameter from the Zoo's money.  
birth - birth checks to see that there are adult animals (age >= 3), and if there are, creates a new animal of the type of the adult. The type of baby is decided randomly.  
age - age ages all animal in the Zoo by 1 day.  
die - die picks an animal type at random, checks to see that there is actually an animal to kill, and then proceeds to kill it :(  
getPaid - getPaid pays the zoo based on each animal's payOff amount.  
getBonus - getBonus generates a random bonus between 250 and 500 dollars for each tiger in the zoo. It adds this bonus to the zoo's money.  
payFoodCosts - payFoodCosts subtracts the food costs for each animal from the Zoo's total money.  
randomEvent - randomEvent selects a number between 0 and 3 and uses these values in the enum RandomEvents to act out a random event of: sickness, attendance boom, birth, and nothing.  
buyAdult - buyAdult at the end of every day, prompts the user to buy an adult animal of whatever type the user decides.  
  
I will also include a constructor and destructor with Zoo in order to create the animal cages dynamically and then delete them respectively.

**Main –** main will basically run a do-while loop which runs the Zoo day by day until the user ends the game or the zoo’s money gets to 0.

**Reflection**

I approached this program methodically starting with importing the easy stuff: my already written Menu and InputValidation classes. I find that starting out with these small victories gives me confidence when the project looks so daunting at first.

Afterwards, I worked on the Animal class and the other various derivatives of it. Setting up the variables and functions for these classes was straightforward. Then came the Zoo class. I heard others discussing making their dynamic Animal arrays in the Animal class itself. I seriously struggled with how to dynamically allocate memory in such a way that would store my animals in their respective cages. It was quite a pain that we could not use vectors on this assignment, but I can understand the benefit in doing it the hard way. I had seen on Piazza about using a triple pointer, and made some decent progress with it. However, I didn’t fully understand what was happening, and it made bugs difficult to fix. I decided to drop this approach for something simpler for my pea brain: double pointers. I used three double pointers to represent each type of animal’s cage. This was much more like what I had accomplished in the Langton’s Ant project, and thus easier for me to implement. I wonder if I’ll ever *really* understand triple pointers. Perhaps if I had more time to delve into the topic. I was just as satisfied getting the Animal class running with double pointers.

As I worked on the Zoo class, it appealed to my logic more to create the animal cages in the Zoo class rather than the Animal class. I alternated back and forth between the two classes and creating the dynamic memory a few times. I finally settled in on what made the most sense, having the cages in the Zoo class.

Once my struggle ended with dynamic memory (15-hour struggle…), I found the Zoo class to be lengthy, but easy to implement the functions I had plan to make. I made several small errors along the way, but all were easy to troubleshoot.

I think the only other struggle I had was with birthing a random animal if an adult animal was present. I landed on using a bool variable to determine if an adult was present. If not, the function looped through again with a different animal checking for adults.

All in all, this project was an emotional roller coaster. I was feeling low when I had spent 15 hours on dynamic allocation, wondering if I could finish the project at the snail speed. Luckily, I made it work, and the rest of the project was not too bad. In total, I believe I spent 25 hours on this project, and it has made me a better programmer to be sure.

**Test Tables**

**For InputValidation Class – This class was already tested in Project 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input Values** | **Driver Functions** | **Expected Outcomes** | **Observed Outcomes** |
| input too low or high | input < 2 input > 200 | getInt(2, 200) | You must select an integer equal to or between 2 and 200. | You must select an integer equal to or between 2 and 200. |
| input in correct range | 2 < input < 200 | getInt(2, 200) | Returns input | Returns input |
| input extreme low | input = 2 | getInt(2, 200) | Returns 2 | Returns 2 |
| input extreme high | input = 200 | getInt(2, 200) | Returns 200 | Returns 200 |
| input not an integer | input = a | getInt(2, 200) | You must enter an integer value. | You must enter an integer value. |
| input not an integer | input = 123abc | getInt(2, 200) | You must enter an integer value. | You must enter an integer value. |
| input not an integer | input = enter key | getInt(2, 200) | You must enter an integer value. | You must enter an integer value. |
| input not an integer | input = asdf 123 | getInt(2, 200) | You must enter an integer value. | You must enter an integer value. |

**For Menu Class - This class was already tested in Project 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input Values** | **Driver Functions** | **Expected Outcomes** | **Observed Outcomes** |
| input option string | “Start Langton\’s Ant simulation” | addOption(string) printOptions() main() | 1. Start Langton’s Ant simulation | 1. Start Langton’s Ant simulation |
| input option string | “Quit” | addOption(string) printOptions() main() | 1. Start Langton’s Ant simulation 2. Quit | 1. Start Langton’s Ant simulation 2. Quit |
| input too low or high | input < 1 input > 2 | getInt(1, 2) promptUserChoice() main() | You must select an integer equal to or between 1 and 2. | You must select an integer equal to or between 1 and 2. |
| input not an integer | input = a | getInt(1, 2) promptUserChoice() main() | You must enter an integer value. | You must enter an integer value. |
| input valid | input = 1 input = 2 | getInt(1, 2) promptUserChoice() getUserChoice(); | returns 1 returns 2 | returns 1 returns 2 |

**Animal/Tiger/Penguin/Turtle Classes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input Values** | **Driver Functions** | **Expected Outcomes** | **Observed Outcomes** |
| initiates each Animal/ Tiger/ Penguin/ Turtle  object with correct values | m\_age=1+ m\_cost = 0+ m\_numBabies = 0+ m\_baseFoodCost = 10+ m\_payOff = 0+ | Animal() Tiger() Penguin() Turtle() | Correct values are stored no matter what input | Correct values are stored no matter what input |
| age each animal by 1 day | m\_age>=0 | ageUp() | m\_age increases by 1 | m\_age increases by 1 |
| isAdult: if the animal is an adult return true | m\_age < 3 m\_age >= 3 | isAdult() ageUp() | if the animal is an adult (m\_age >= 3) then return true, otherwise, return false | if the animal is an adult (m\_age >= 3) then return true, otherwise, return false |

**For Zoo Class**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Input Values** | **Driver Functions** | **Expected Outcomes** | **Observed Outcomes** |
| Printing of Zoo information | N/A | printStats() main() | Stats about the game: money and number of each animal | Stats about the game: money and number of each animal |
| Beginning – adding 1 of each type of animal to the zoo | numTigers = 1 numPenguins=1 numTurtles = 1 | startZoo() Zoo() addAnimal() main() | 1 of each animal type in the zoo | 1 of each animal type in the zoo |
| Beginning – adding 2 of each type of animal to the zoo | numTigers = 2 numPenguins=2 numTurtles = 2 | startZoo() Zoo() addAnimal()  main() | 2 of each animal type in the zoo | 2 of each animal type in the zoo |
| Beginning – adding 1 or 2 of each type of animal to the zoo | numTigers = 1 numPenguins=2 numTurtles = 1 | startZoo() Zoo() addAnimal() main() | 1 of each (Tigers/Turtles) and 2 penguins zoo | 1 of each (Tigers/Turtles) and 2 penguins zoo |
| Spending correct food costs | various 1/2 numTigers/ numPenguins/ numTurtles values | Zoo() addAnimal() payFoodCosts() main() | Correct amount is spent on food based on animal type and quantity | Correct amount is spent on food based on animal type and quantity |
| Correct random event outcomes | random events SICK ATTENDANCE BIRTH NOTHING | randomEvent() die() getBonus() birth() main() | Correct outcome happens to the Zoo based on a random value. (e.g. animal dies when sick, extra money for tigers during an attendance boom, birth of an animal, or nothing) | Correct outcome happens to the Zoo based on a random value. (e.g. animal dies when sick, extra money for tigers during an attendance boom, birth of an animal, or nothing) |
| No deaths when no animals are present | numTigers=0 numPenguins=0 numTurtles=0 | die() main() | There cannot be a death when there is not an animal to kill | There cannot be a death when there is not an animal to kill |
| Correct payouts for each animal type | various 1/2 numTigers/ numPenguins/ numTurtles values | Zoo() addAnimal() getPaid() main() | Correct payoff amounts based on each animal type and quantity | Correct payoff amounts based on each animal type and quantity |
| Buying an adult animal | 1=Tiger 2=Penguin 3=Turtle 4=No buy | addAnimal() main() | Correct animal added to the zoo with the correct expenditure | Correct animal added to the zoo with the correct expenditure |
| Game repeats until no money left | money = 100 | addAnimal() main() | After buying an animal over 100, game should end because money is <=0 | After buying an animal over 100, game should end because money is <=0 |
| Zoo doubles arrays every time max capacity is hit | tigerCapacity = 10 penguinCapacity = 10 turtleCapacity = 10 | addAnimal() x 11 addAnimal() x 20 addAnimal() x 100 main() | Add more than 10 animals of each type to each “animal cage” and see if the array doubles in size to accommodate | Add more than 10 animals of each type to each “animal cage” and see if the array doubles in size to accommodate |
| Days run events correctly in proper order | Various | liveDay() printStats() age() payFoodCosts() randomEvent() getPaid() buyAdult() | All the functions to the left execute events according to the specs and a day goes up by 1 afterwards | All the functions to the left execute events according to the specs and a day goes up by 1 afterwards |