Louis Adams
Project 2
2019-04-28
Reflection

Design

Animal Class

The animal class is pretty straight forward. I mostly am just following the specifications. The only tricky part here is how to design the constructors. Since I want to be able to construct animals of the NewAnimal class, which have all user-specified values, I've decided to make a constructor that takes 5 parameters, one for each Animal class member variable. Then, when I instantiate a NewAnimal object, I can call this constructor from the NewAnimal constructor. For the Tiger, Penguin, and Turtle classes, I really only need to specify the age of the animal so I made a constructor that takes a single parameter, for the age. For member functions besides the getters and setter and constructors I have a virtual destructor. I added this so that my objects of the derived NewAnimal class would be deallocated properly. I also have a virtual *getName* function which returns the name of the animal. This was really only necessary because of the derived NewAnimal class. I need to be able to get the name of the user specified animal.

Tiger/Penguin/Turtle Classes

These classes are very simple. They don't have any member variables of their own and their only member functions are a constructor, which calls the Animal constructor to initialize *age* and initializes the rest of the base class member variables within the constructor, and *getName*, which returns the name of the animal.

NewAnimal Class

This class, unlike the other derived Animal classes, has a member variable, *name,* which stores the user-specified kind of animal. A getter and setter is also needed for this variable. The constructor takes 6 parameters and calls the base class constructor to initialize base class member variables and initializes *name* in the NewAnimal constructor.

One of the challenges of implementing this class is figuring out how to construct NewAnimal objects. Calling the base class constructor from within the NewAnimal constructor works very well. The class also needs a virtual destructor, or causes memory leaks.

Zoo Class

This Zoo class is the primary class for this project and stores almost all of the logic of the game. This seems appropriate since I'm calling this the "Zoo Tycoon Game". There are variables that store the number of each kind of animal in the zoo. There are also variables for the size of each array of animals in the zoo. These are very important for allocating more memory when the array becomes full. There is a *money* variable which keeps track of the user's money in the game. Then there are 4 double pointers

to Animal objects, one for each kind of animal. The Zoo class has a constructor which takes three *int* parameters for the number of tigers, penguins, and turtles that the zoo will start with. These are specified by the user at the start of the game to be either 1 or 2. Other variables are initialized with values in the constructor such as the size of the arrays and the amount of money the user starts with. The number of new animals starts at 0. I decided to put this functionality at the end of the day when the user can then buy a new kind of animal, which is user-specified.

The *startGame* function contains the largest amount of the logic for the game. I was thinking of breaking this function up into *beginDay, takeTurn, randomEvent,* and *endDay,* but ended up only making the *randomEvent* function. Consequently, the *startGame* function may contain too much within it but it is a sequence of logic going from the creation of the zoo all the way to the closing of the zoo. I would argue that being able to see this logic in sequential order may be better than having more smaller functions but then having to jump around more. One thing I will probably change in future designs is to put the allocation and deallocation of memory in the constructor and destructor of the class, respectively. These things do seem a bit out of place being in the same function with the game logic. After the allocation of memory for the Animal objects the logic for the game begins and the cost of the starting animals is subtracted. Then the rest of the function is mostly a giant do-while loop that represents a "turn" in the game. As long as the user has a positive amount of money, the game continues to the next turn. This is all quite straightforward until I got to the purchasing of animals at the end of the day. Purchasing of tigers, penguins, and turtles was no problem, but new animals had to be completely specified by the user. I ended up making new variables from which to construct the new animals. Also, before displaying the animals available to purchase, I check to see if there are any new animals already in the zoo. If there are, then the user only has the option to buy more of that animal. If not, then the user can specify what kind of new animal she/he wants to buy.

In the middle of a turn of the game, a random event happens(or doesn't happen). Here, I call my *randomEvent* function. There are a couple tricky parts to the random events. The first is dealing with the different feed types. Depending on the feed type, there should either be a 25% chance of an animal dying, a 50% chance, or a 12.5% chance, so I get random numbers that can be divided up into these percentages. So for the premium feed type I get a number between 1 and 24 because 24 can be divided up evenly to give a 12.5% chance of an animal dying, and equal chances to the other 3 possible events. 1-3 is assigned to the event of an animal dying, 4-10, to the next event, and so on. Having my variable named *event* probably doesn't make that much sense, but I had named the variable before implementing the different feed types so there was only the numbers 1-4, and hence it made a lot more sense.

The second tricky part of the random events was the event where animals have babies. This took a lot of Boolean logic make work correctly. First, I had to check whether it's possible for the animal to have babies, and then if so choose an animal type randomly only among those types where it's possible to have babies. I found that the best way to do this was to first be sure that some animal is capable of having a baby and then keep looping until an animal capable of having babies is chosen.

***menu* Function**

The menu function has just one variable, *choice*, to hold the user's choice. The function returns this value. The menu displays the menu to the user, prompts the user for a choice, and then calls the *getInteger* function to validate that an integer was entered. A simple loop will be used to validate the range of the integer entered. This is a simple menu that just shows the default number of starting animals that the zoo will contain and gives the user the option to start the game immediately, quit, or change the number of starting animals.

***getInteger* Function**

I was able to reuse this function from previous programs. It uses *getline* to get a line of input from the user and stores it in a string. It then checks the string to see if every character is a digit. If not, it is rejected and the user is prompted again. If every character is a digit, then the string is converted to an *int* using stringstream. If the extraction of an *int* fails (this should only happen at this point if too large of a number was entered), then the number is rejected and the user is prompted again. The stringstream is also cleared and extraneous input is ignored. If an *int* is successfully extracted, it is returned. Right now, this function works only with positive integers.

***getString* Function**

This function is used to validate string input. For this program, it's used to reject anything that is not an alphabetic character (a-z/A-Z) or a space. These are the only characters that I thought would be appropriate for naming an animal. I used this function to get the name of a NewAnimal. Since the function uses *getline* and allows spaces, inputs such as "woolly mammoth" are accepted.

***outputMessage* Function**

This function takes a string as a parameter, opens the file called "file.txt" and outputs the string to it. This was needed to implement the status messages for extra credit.

***inputMessage* Function**

I wrote this function so that it takes a line number int as a parameter, and then takes that line from the file, stores it in a string, and returns it. I then realized it would be easier to just always output to the first line of the file and then always input the first line of the file.

***main* Function**

main sets the default number of starting tigers, penguins, and turtles to 1 and then gets the user's choice by calling the *menu* function. A do-while loop is used so that the user can choose to play the game as many times as she/he wishes without quitting. The range of the *int* variables is validated using simple while loops. If the user chooses to play the game, a Zoo object is instantiated and it's *startGame* function is called. If the user chooses to play again, his starting numbers of animals would remain the same for the second game unless she/he decided to change them.

Test Plan

Test Scope	Description	Expected Output	Actual Output
menu function – getting a choice from the user	User enters a negative number	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a non-number character	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a float	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a number out of the range 1-3	Error from *main* telling user to enter a number in range	Error from *main* telling user to enter a number in range
main – selecting the number of tigers, penguins, and turtles before starting the game	User enters a negative number	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a non-number character	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a float	Error message from *getInteger* and re-prompt the user	Error message from *getInteger* and re-prompt the user
	User enters a number out of the range 1-2	Error from *main* telling user to enter a number in range	Error from *main* telling user to enter a number in range
startGame – selecting different feed types	User enters a 1 for cheap feed	animals should be more likely to die	animals do die twice as often on average
	User enters a 2 for generic feed	all random events should be equally likely	all random events do seem to occur with the same frequency
	User enters a 3 for premium feed	animals should be less likely to die during random events	animals do die less frequently
startGame – check daily profit	profit equals profit from all animals plus bonus	all profits are accumulated correctly	all profits are accumulated correctly
startGame – buying animals	buy a tiger	money is subtracted, payoff from next day should be noticeably increased	money is subtracted, payoff from next day should be noticeably increased
	buy a penguin	money is subtracted, payoff from next day should be noticeably increased	money is subtracted, payoff from next day should be noticeably increased
	buy a turtle	money is subtracted, numTurtles is increased	money is subtracted, numTurtles is increased

	buy a new animal	user should be able to specify values	user can specify values
	buy a new animal the next day	name of previously specified new animal should display, should not have option to specify values	name of new animal does display with no option to specify values
randomEvent – test death of an animal random event	there is 1 or more tigers, penguins, or turtles	only one animal should die	1 animal dies
	there are 0 tigers, penguins, or turtles	no animal should die	no animal dies
	there is no new animal	no animal should die	no animal dies
	there is a new animal	new animal should die	new animal dies
randomEvent – test boom in zoo attendance	bonus is added depending on the number of tigers	bonus is added to the profit for the day	bonus is added to the profit for the day
randomEvent – baby animal random event	test whether baby tigers can be born	a tiger is born	a tiger is born
	test whether baby penguins can be born	5 penguins are born	5 penguins are born
	test whether baby turtles can be born	10 turtles are born	10 turtles are born
	test whether baby new animals can be born	baby new animals are born, even user-specified numbers of babies such as > 10	baby new animals are born, even user-specified numbers of babies such as > 10
	test this event when there are no adults capable of having babies	message should display saying there are no animals capable of having babies	message displays
randomEvent – 4th random event	test event where nothing happens	message displays saying nothing random has happened	message displays
getString function	user enters a name for a new animal with characters other than a-z/A-Z/” “	input is rejected and user is reprompted	input is rejected and user is reprompted
	user enters a name of new animal using characters a-z/A-Z and spaces	input is accepted	input is accepted
outputMessage function	check contents of file.txt after a random event occurs	random event message should be in the file	message is in the file
inputMessage function	check input of random event message from file	random event message gets read and displays	random event message gets read and displays

	to console output	as console output	as console output
Memory Leaks	Run game, making sure to buy 10 or more tigers, testing the *addTiger* and *allocTigers* functions	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game, making sure to buy 10 or more penguins, testing the *addPenguin* and *allocPenguins* functions	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game, making sure to buy 10 or more turtles, testing the *addTurtle* and *allocTurtles* functions	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game, making sure to buy 10 or more new animals, testing the *addNewAnimal* and *allocNewAnimals* functions	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game, quit, and run again	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game until tigers are killed in a random event, testing *deleteTiger*	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game until penguins are killed in a random event, testing *deletePenguin*	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game until turtles are killed in a random event, testing *deleteTurtle*	All heap blocks freed and no errors	All heap blocks freed and no errors
	Run game until new animals are killed in a random event, testing *deleteNewAnimal*	All heap blocks freed and no errors	All heap blocks freed and no errors

Reflection

The Zoo Tycoon project was definitely the most challenging and stress inducing project I've done so far. I had thought was getting the hang of pointers and using *new* and *delete* until this project, and then I realized I had a lot to learn still.

The primary challenge I had with this project was figuring out how to create arrays of animals, expand them dynamically, and instantiate the animals dynamically. I didn't have a fundamental understanding of how the pointers were needed in order to point to a location in memory where the objects would be instantiated. I actually had to reach out on Slack to other classmates for the first time in this program to try and get a better understanding of this concept. I didn't understand that the *new* operator returns an address and that the *delete* operator also operates on an address. My first attempts were trying to "delete" dereferenced objects, rather than the pointer. I don't think I understood what I was doing for quite a while with my *allocTigers* etc. functions. I would create a temporary array of pointers to hold the value of the *tigers* array but I didn't understand that I was only copying the addresses of the Tiger objects into the temp array, and not the Tiger objects themselves. Therefore, I would be trying to "delete" objects from the temp array after copying them back to my *tigers* array even though the objects themselves shouldn't even be affected by the *allocTigers* function, it's only allocating more memory for more pointers to Animal objects. This was the sort of thing that I needed to mull over every day and then sleep on and then mull over again the next day until I finally started to get it. Even once I had the basic functionality implemented I still had memory leaks for the longest time. I finally realized that although I had *deleteTiger* etc. to deallocate memory for dead animals, I had forgot to deallocate memory for any animals left living at the end of the game, as well as for the arrays of pointers. Once this was done, most of the memory leaks were gone except one block which I couldn't seem to free no matter what I did. Finally, after a lot of reading online about using *new* and *delete*, I figured out that I may need a virtual destructor for my derived class and base class. I knew it was my NewAnimal class that was causing the memory leak because whenever I didn't purchase any new animals, I had no leaks at all. After adding the virtual destructors, even with nothing in the body of the destructor, the leaks were all gone!

The New Animal class in general was quite a challenge and added a lot of complexity to the project. Besides the virtual destructor that was needed to prevent memory leaks, I also needed a way to store the name of the animal that the user would give it and return that name. Thus, I created another virtual function *getName*, that would return the name of the animal. I also created my *getString* function solely to validate the input from the user regarding the name of the new animal. This function probably could have been more strict, such as limiting the length of the input entered by the user, but it does eliminate any input that's not an alphabetic character or a space. I do like that it allows for spaces. The NewAnimal constructor also took a while to make since I needed 6 parameters, and it had to pass 5 of those parameters to the base class constructor. I also chose to introduce the new animal functionality at the end of a turn in the game, giving the user the chance to buy a new animal then. In hindsight, it may have been easier to give the user the option to create a new animal in the menu before the game starts. If I had done this, then allocating memory for all 4 kinds of animals would have been nearly identical. As I did it, the *newAnimals* array started with 0 animals in it while the other arrays all

had at least 1 animal already. When the user has the option to purchase an animal at the end of a turn I also had to check whether a new animal already exists in the zoo or not, and if one did then I had to display the name of that animal to the user rather than giving him the option to create the animal. If the new animal did not exist, the user should have the option to create one and specify all of its attributes.

The next major challenge I had was implementing the different feed types. I have a separate function for random events so I decided that my *randomEvent* function could take the feed type (as an *int*) as a parameter. I then needed different sets of random numbers depending on the feed type. I then had to make sure that if the feed type was cheap, there was a 50% chance of an animal dying, if the feed type was generic, 25%, and if the feed type was premium, then 12.5% of the time an animal would die. This added a lot of tedious details to my conditional statements. I needed to check for the feed type together with the range of the random value in order to execute one of the random events.

The next challenge was how to make sure that an animal had a baby only if they were capable of doing so, and choosing that animal randomly. It was easy enough to check to see if any type of animal was capable of having babies or if none were. It was harder to choose a random animal only among those that were capable of having babies. I finally got the idea to create a bool called *babyEventDone* as well as bools called *tigerParentPossible, etc.* and then search randomly among the animal types and add a baby only if the *tigerParentPossible, etc.* was true, and then set *babyEventDone* to true. If the selected animal couldn't be a parent, the loop would continue and another animal type would be chosen randomly. I hadn't worked with bools a lot in my past programs so this way of searching didn't pop into my head very quickly. It didn't help that I mixed up my assignment operators and equality operators among all my bool variables.

The last challenge I had working on this project was outputting the random event status messages to a file, and then reading them in again, and displaying them as console output. Outputting the message was very straight forward. I just made a function that took a string as a parameter, created an ofstream object, opened a file, and output the message to the file. I designed the input function to take an *int* as a parameter which would be a line number. The function would then read the line in the file corresponding to this line number. I was thinking I would be able to specify the line that I want and input that line. However, this didn't really make sense due to the fact that the random events are random and there's no way of knowing what will be on any line. So, I simplified how got a line of input. I simply output a message to the first line and then input a message from the first line.

Overall, I learned a great deal while working on the Zoo Tycoon project. The topics I learned the most about were dynamic allocation of memory, the *new* and *delete* operators, dynamically creating objects, inheritance, polymorphism, string input validation, file input and output, constructors calling base class constructors, member initialization lists, and virtual destructors. The sheer size of this project was quite overwhelming and I actually put in more time into this class in the last two weeks than I did in the first two weeks which really surprised me, but it was a satisfying experience and it did a lot to boost my confidence.