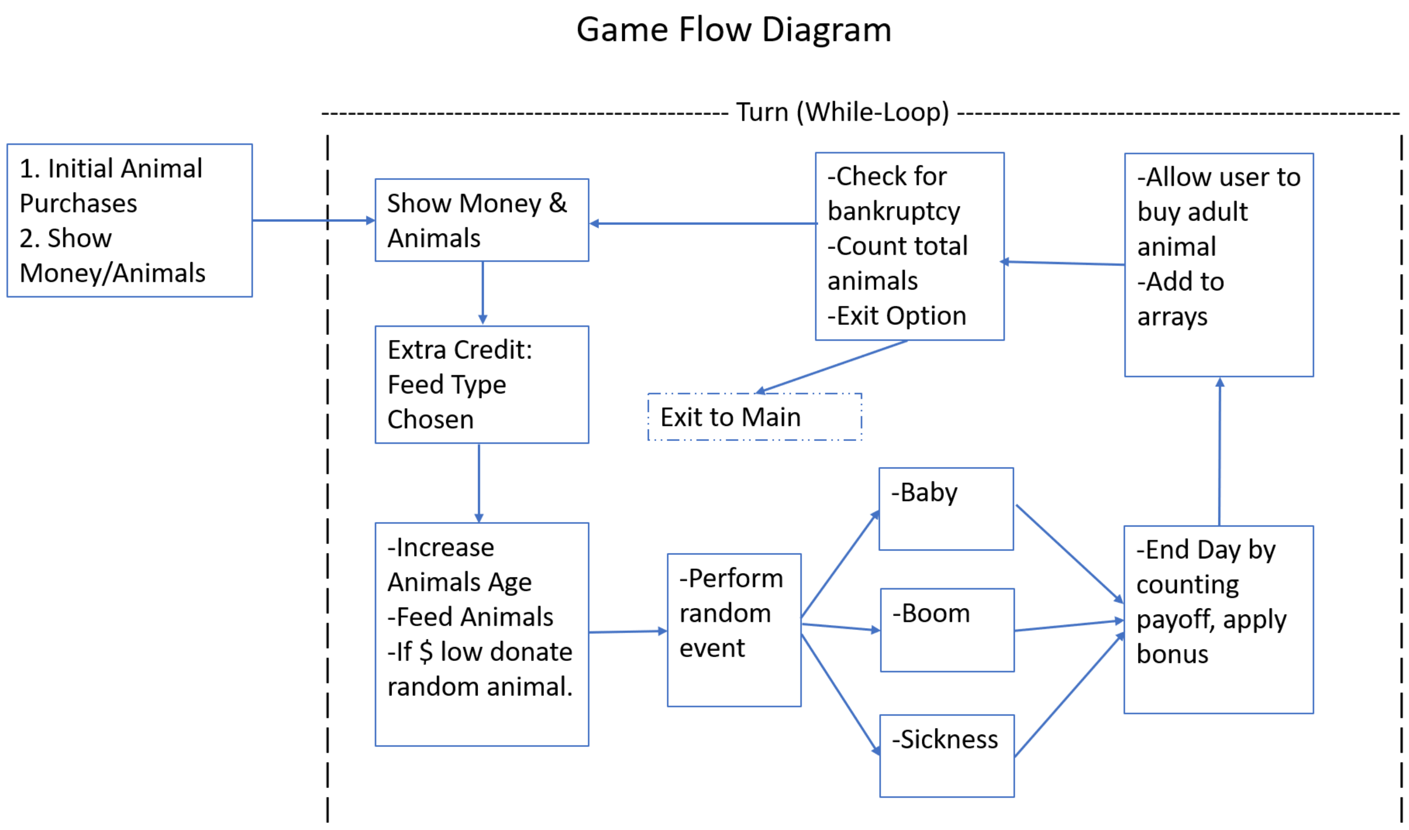
Andrew Clos
CS-162 - 400
Project 2 Design, Test Table, and Reflection of Zoo Tycoon

Initial Design:



Changes in Design:

Initially, I had planned to have the feeding of the animals take place earlier in a function called “takeTurn.” I envisioned this function being larger and handling more of the events that occurred during a “day,” but in the end, I broke the day into many more functions that were mostly called by the while loop operating from the “mainMenu” function. I also had the main menu broken up in to another part called “paramMenu” that would handle more of the user input and options that come after the initial introduction and game-start options. I ended up moving all of the “paramMenu” code in to “mainMenu” to avoid unnecessary some unnecessary data movement. This ended up making the main menu a pretty large function, but it is functional and not too complex.

I had also planned on passing the game parameters as a pointer to an array to various helper functions, but decided against and instead used private member variables with get and set methods to access them. I did end up using a 12 integer array for the establishment of new animals (more on this in the reflection section.)

Classes:

Zoo: The game will contain the menu and collect data entry from the user to set up the parameters of the game (rounds, dice status, etc). These will be passed as an array of integers to other functions as needed.

Key functions:

mainMenu: Game lives here, prompts user to play or quit, calls the other methods in proper order. Gives the users additional menu options via if-statements and presents extra credit attempts.

Andrew Clos
CS-162 - 400
Project 2 Design, Test Table, and Reflection of Zoo Tycoon

showParams: Print the legend of the game parameters (Initial Cash, Cash on Hand, number of each animal) and then print the current user entered values right below it. This is called during each turn.

newAnimals: This takes information about new animals to be created, calls the checkArray function, and then adds the new animals with the correct parameters to their arrays. Increments the animal counters.

checkArrays: Checks the arrays by comparing their size to the size of the animal counters, and doubles the array size if necessary by creating a temporary array to move the old contents in to and then deleting the memory allocated to the old array.

feed: This function deducts the feeding cost from the cash at hand. It checks the food type used to apply the correct multiplier to the food cost (part of the extra credit).

sickness: Picks a random animal to die and removes them from the array. Moves all other animals in the array one slot "to the left" and then deletes the last (now empty) element of the array's memory allocation.

boomZoo: Applies a bonus to the cash on hand based on how many tigers are currently in the zoo.

babyBorn: Checks to see if any of the animal types have a member old enough to produce offspring, and if they do picks one of them at random to have a baby. Then calls the newAnimals function to create that new baby.

endDay: Calculates the amount of money the animals have brought in and applies the tiger bonus if applicable.

randomGen: returns a random integer between an upper and lower limit that are provided by the caller.

Animal: Since this is a parent class, it will need to establish functions and data members that are common to all of the derived classes. These are mostly get and set methods for the animals age, cost, number of babies, etc.

Derived Animals (Tiger, Penguin, Turtle and New Animal): These classes contain default constructors and constructors that establish the characteristics of each species of animal. The NewAnimal function will have additional set methods for Cost, number of babies, etc. since these will be entered by the user during run time.

Input Validation: It has proven valuable to me to have created a completely separate class for input validation near the beginning of this term! I have added functions to it for different data types and for this assignment, I have added the ability to perform double validation. See more in the reflection section below.

Test Plan:

Test Category	Description of Test	Expected Results / Handling
No animals remain	With poor quality food, and initial choices of 1 animal each, the game is likely to run out of animals.	-No new babies should be born. -No feeding costs, bonuses or payoffs should be incurred. -Game should not crash.
Stepping outside an array boundary	-Step through program and watch for seg faults and garbage data after an animal is removed or added or the array size increases.	-Initially seg faults due to logical errors (see reflection!) -Should properly resize array, not step out of bounds. -May see garbage data in other variables.

Andrew Clos
CS-162 - 400
Project 2 Design, Test Table, and Reflection of Zoo Tycoon

New Animal Test	1. Data entry adhere's to limits. 2. Does it behave mathematically similar to other animals. 3. Does the presence of a new animal cause arrays to go out of bounds.	Try entering values outside acceptable -Hand calculate feeding and payoff amounts to confirm. -Perform another set of "stepping outside an array boundary" tests to cases where a new animal does exist.
Input validation: Double entry (new for me for this assignment)	1. Need to handle "." Character, but only allow one. 2. Enter symbols: '%', '&', '" '"', space, etc. both by themselves and sandwiched between valid input. 3. Will need to allow a leading zero only if it is followed by	-Should output error message that the character is not allowed, ask user to try again. -It should reject the entry, post message and ask user to try again. -In the case of a space character "y y" it may read this as a yes/true.
Input validation: is an integer	1. Entering leading zeros, any other non 1-9 char in the first position or any non 0-9 in subsequent positions. 2. '2.1' float / double handling. 3. Adhere's to size boundaries of integers. If limit is set too high or not at all.	-Explain special case of leading 0 not being allowed with cout message. -Must make sure entry isn't being truncated. If user enters '2.1,' we cannot assume they meant '2'!! -If not establishing an upper limit, warn user that their entry is above size of signed int (32,767), etc.
Random Function	1. Is it truly random – will test by running 100+ round games and looking at average values to make sure they are not skewing in a certain direction.	-Expect to see equal likelihood of sickness, bonus, baby or nothing. -Except in cases of differing food quality, where we expect to see double or half the results of illness.
Memory Leaks	1. Valgrind with -v options. 2. With arrays that have just increased in size. 3. Test case of quitting right away	-Expect to see memory leaks at first, but will look to use the delete [] function properly to remedy this -Should see it in valgrind and "vld.h" function for Visual Studio. Possible seg fault.

Andrew Clos
CS-162 - 400
Project 2 Design, Test Table, and Reflection of Zoo Tycoon

Reflection:

The overall "size" of the program seems a bit large. I feel this program bulky as I added functions, but I wasn't really sure where I could trim it down. I would appreciate feedback on this topic. I was also a bit inconsistent regarding constant global values for known limits like menu choices. I sometimes hardcoded a literal instead of adding the extra line of code to define a constant integer.

Int conversion: When passing the array of animal data to the newAnimals method, some of the data is a double and gets converted to an int value. In this case, it does not cause unexpected behavior, but it will round the user specified "cost" of a New Animal down to the nearest dollar. I chose to leave this "narrowing" in the program, but in the future will seek to avoid it since it seems a little sloppy and could produce unexpected behavior in other cases.

While loop dependent on random function to exit: In the babyBorn function, after confirming that there is at least one 'of-age' animal, I made use of a random number generator within a while loop to find an 'of-age' animal and produce a new baby. The risk here is that if my range of random numbers is off, the loop could run endlessly. Also, I could hit an unlucky sequence of random numbers and be stuck in the loop for a long time, using an unnecessarily large number of CPU cycles. In all, this like it is probably not the best coding practice, so I am hopeful for future input regarding this decision. To mitigate the chance of a problem, I created a loop counter that will exit the loop after 10,000 cycles to avoid being stuck forever. During testing, I never encountered a problem with this, so the loop counter and exit provision were removed.

Double Validation: I added double validation for this assignment and it was quite similar to my integer validation, with a couple of key differences. While my integer validation does not allow for a leading zero, I would need to allow for that with doubles, but only if the zero is immediately followed by a '.' character. (i.e. user enters '0.2') This was handled with an additional if-statement. Also, I would need to allow for a '.' Character to occur anywhere in the entry, but could only allow for one in the entire entry. This was taken care of by adding a counter to a for-loop to keep track of how many '.' characters had been found.

For feed quality, I realized that when the food quality is normal a ¼ chance of sickness is appropriate. When food quality I high, a 1/7 chance of sickness is correct (2/7 chance for each of the remaining 3 random events) and for low quality food, a 2/5s chance of sickness is needed (1/5 chance for each of the remaining 3 random events). Therefore, I took 4 * 5 * 7 = 140 and made the roll for random events between 1 and 140. Then I could easily have whole numbers (integers) for the ranges of each random event.

Avoiding calls to private members variables, instead using get and set methods. This may have made the program a bit more bulky, with many additional calls to helper functions, but it did allow me to avoid using private member variables which I believe is preferable from a style perspective.

Memory leak issues: I needed to add a for-loop to delete each element within the dynamically allocated array before delete [] array could be used. I recall needing to do this in a prior assignment where a user-sized 2x2 array was populated with values, but it took me some trial and error to get it working properly here. I can see that pointers and dynamic memory allocation are going to be a big part of C++ programming though, so I will keep practicing with this technique. I am still having valgrind errors where it thinks there is a mismatch in free / delete / delete [], but I am not sure what this means, since it does not seg fault and it shows no memory errors. Perhaps this is just an idiosyncrasy of valgrind.

Did not implement "donate animal" feature if one cannot be afforded. I had intended to add this functionality to the program, but decided not to. Instead, if the user runs out of money, the game will simply end (Game Over!) If the game were tuned a bit more closely and running low on money was a common enough occurrence, the donate or even “sell” animal feature could have been useful, but using the values provided in the program specs means the user is rarely even in jeopardy of running out of cash.