CS 202 Spring 2022 - Assignment 2

Inheritance and Intro to Pointers

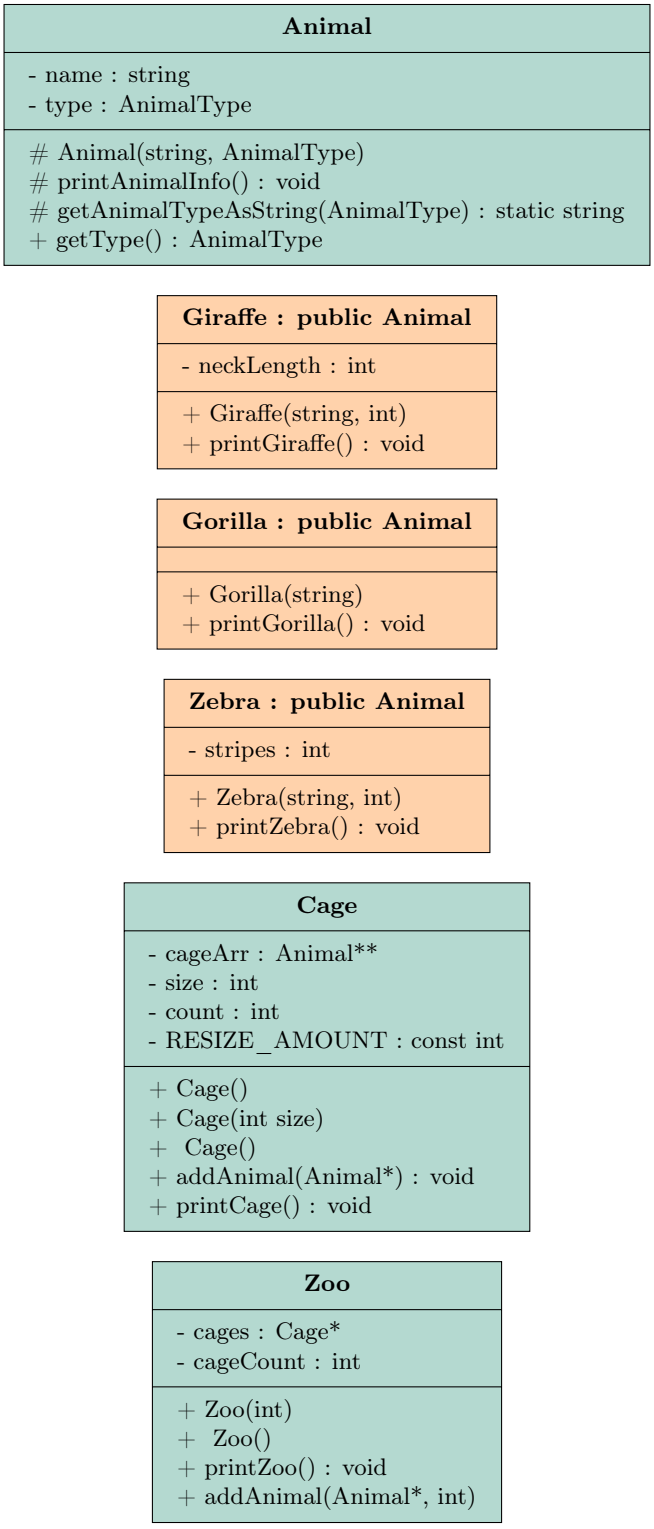
Overview

For this assignment, we'll be opening a zoo. The ethics behind zoos are controversial, so we'll just implement it virtually. Within our zoo, we will keep a number of different kinds of animals. This particular zoo will contain enclosures with some combinations of three different animals: giraffes, gorillas, and zebras.

As part of our zoo, we'll want to set up a few different classes and identify aspects that should be grouped together or have common attributes. In other words, we'll want to use **inheritance**. As part of this assignment, we will set up a Zoo class to act as a container for all of our other classes, along with a Cage to act as an enclosure for some animals, and then an Animal class to act as a base for all Animals in our Zoo. Recall that inheritance can be used to create classes with shared attributes that will copy over any members from the base to a derived class. Here the Giraffe, Gorilla, and Zebra will all have their own classes that inherit from a common Animal class. This will allow us to not only borrow members from the Animal class to have in each Animal class, but also to guarantee that they can be stored together through upcasting due to their common parent.

We will also work on basic pointers and memory management. A pointer can point to one of two things: a dynamic array or an object. Both of these will be used in our zoo. We want to minimize wasted space in our zoo, and so we will dynamically allocate any Cages as well as the space within those Cages. On top of that, we will only create the necessary Animals to fit into the Zoo and no extra copies, meaning that all Animals should be handled with pointers. The ***new*** keyword can be used to dynamically allocate both objects and arrays and will be needed in the later functions.

UML Diagrams



Important Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, **functions implemented for you already are in blue**, and **functions that are abstract that will be implemented later are in magenta**.

Global Functions and Constants

Animal

A simple class that acts as a base to all of the more specific Animal classes. This contains the animal's name and an enum type that keeps track of the animal's type (Gorilla, Giraffe, or Zebra). All of the specific Animals will then also have any variables or functions from this class by inheritance. Normally we would use virtuals rather than this enum for the purpose of properly using the type for functions, but we'll get to that in a later assignment.

- ***string name*** - The User's username for logging in to our site and being displayed on our pages.
- ***AnimalType type*** - An enum marking this Animal as either a GIRAFFE, GORILLA, or ZEBRA. This will be set in the constructor and is used by the implemented print functions to determine how to print. You need only set this.
- **Animal(string name, AnimalType type)** - This base class constructor just initializes the two variable members the Animal class has - being the name and the type - to the given values. When we derive classes later, remember that derived class constructors must always call one of the base class constructors (if you put none it will pick the default constructor if it exists). So when we later implement other constructors, they will need to call this one first.
- **void printAnimalInfo()** - Prints the two Animal variable members to cout. This can be called in derived classes to print these private members without the need to rewrite code or provide accessors for private members.
- **static string getAnimalTypeAsString(AnimalType animal_type)** - A static function that takes a AnimalType enum and returns a string that describes it. Used for printing in the existing code.
- **AnimalType getType()** - An accessor for the type member.

Giraffe

The Giraffe **is a** Animal, and so inherits from the Animal class. This means any members that the Animal has, the Giraffe has. That being said, private members are inaccessible in any functions we have here. The Gorilla and Zebra classes will also inherit from Animal, and this class has code written to act as an example for those two other classes. On top of the inherited Animal members, this class contains one more int variable to keep track of how long the Giraffe's neck is in feet. This allows it to expand upon the existing class it inherits from.

- ***int neckLength*** - A variable keeping track of the Giraffe's neck in feet.
- **Giraffe(string giraffe_name, int neck)** - Initializes the variable members for the Giraffe. The body of this constructor needs only to initialize the neckLength member to the passed int. Recall from earlier that a derived class must always call some base class constructor. This one will call the existing Animal constructor, passing giraffe_name as the name string parameter and the type as GIRAFFE. You can use the syntax used here as reference on how to do the other two Animal classes.
- **void printGiraffe()** - This prints the Giraffe information on top of the Animal information it inherited. In order to reuse the printAnimalInfo code we wrote, the function is called within this function first before printing the Giraffe's neckLength.

Gorilla

The Gorilla class inherits from Animal, but does not add any new variable members of its own. This class simply sets its type to be Gorilla.

- **Gorilla(string gorilla_name)** - This constructor should set the Gorilla's name to gorilla_name and the type to GORILLA. Use the Giraffe class as a reference. Since there are no Gorilla exclusive members, the body of the constructor will be empty, you need only call the Animal base class constructor.
- **void printGorilla()** - This only prints the Animal members the Gorilla has by calling the Animal's printAnimalInfo function. Again, this is similar to the Giraffe function, just with no need to print any new members.

Zebra

The Zebra inherits from Animal and has one new variable member, being the number of stripes the Zebra has.

- ***int stripes*** - The number of stripes the Zebra has
- **Gorilla(string zebra_name, int stripe_count)** - Initializes all of the Zebra's members. Similar to the previous two classes, this should call the Animal base class constructor to set the name and type, and then also set the stripes to the given int
- **void printGiraffe()** - Prints all of the Giraffe's members. This should print all of the Animal members similar to the other two classes and then finally print the stripes amount. The stripes should be printed in the format: "Stripes: x" similar to the provided sample output

Cage

The Cage class will hold a collection of different kinds of Animals within a single Cage. To do this, we will use a dynamically allocated array full of Animal*s and implement some helper functions. This class can be thought of as a variation of our dynamically allocated List example from lecture. This should allocate some amount of space, add Animals to the back of our dynamic array, and resize whenever we don't have space to put an Animal inside. In order to avoid allocating a lot of extra memory, this stores pointers to the Animals inside the Cage, rather than direct Animal objects themselves. All Animal objects will be allocated and handled by main, but just consider that once each Animal is created, it will be attempted to be added to some Cage.

- ***Animal** cageArr*** - A 1D array of pointers to all of the Animals contained within this Cage. Although this is a double pointer, only one pointer refers to an array dimension, while to other pointer comes from the actual type in that array, being Animal*. As such, when allocating for this array, we will allocate a new array of Animal*s
- ***int size*** - The allocated size of the ***cageArr***. This tells how much space the array is currently occupying (i.e. max Animals it can hold before it runs out of space)
- ***int count*** - The number of Animals actually being held in the Cage. This should always be less than or equal to the ***size***. If they are equal, we know that the Cage is full.
- ***const int RESIZE_AMOUNT*** - A constant telling how much bigger to make our ***cageArr*** whenever we run out of space. This is set to 4 in the given code. This means if we had a cage of 10 Animals and needed more space, then the new cage array should hold 14 Animals
- **Cage()** - A default constructor for Cage. This initializes the array to empty by setting the pointer to null and the size and count to 0.
- **Cage(int size)** - This constructor should allocate the ***cageArr*** to an Animal* array of the given size, then initialize the ***size*** class member to the parameter size and initialize the count of Animals to 0. Recall that to distinguish between parameters and members of the same name, we can use *this*

- **Cage()** - The Cage destructor is called whenever a Cage object is destroyed (so either when it is deleted explicitly with the keyword or when an object goes out of scope). Since the Cage class contains a pointer member to dynamically allocated memory, the destructor is necessary to write to avoid memory leaks. The cageArr will need to be deallocated, but not until after all of the Animals included within that array have all been deallocated.
- **void addAnimal(Animal* animal)** - This should add the given Animal to the cage by putting it at the end of the ***cageArr***. If the cageArr cannot hold any more Animals, it must be resized first to accommodate the new animal. So, if the cage is either empty (the array has not yet been allocated) or has no more space, allocate a new temp array of ***RESIZE_AMOUNT*** bigger than the current size, deep copy all of the old Animal*s over, and then deallocate the old array and update ***cageArr*** to point to the newly allocated array. Make sure to update the ***size*** member to the newly allocated size. Afterwards, regardless of the need to resize, we will want to put the animal at the end of the Cage. Don't overthink this; remember that a dynamic array can be used exactly like a static array after the allocation, so just use [].

Zoo

The Zoo class will hold all of the Cages that make up our Zoo. To do this, we will use one single dynamically allocated array of Cages to hold everything, with each Cage holding some number of Animals.

- ***Cage* cages*** - A pointer to the 1D array of Cages that make up our Zoo. Unlike the Cage class, we will allocate this one time dynamically and not worry about resizing.
- ***int cageCount*** - How many total Cages we have. In other words, the size of the ***cages*** array
- **Zoo(int cage_count** - Allocates an array of the given size for the ***cages*** array and sets the cageCount
- **Zoo()** - Deallocates the ***cages*** array. This should just be one line deleting that array
- **void printZoo()** - This function prints all of the Cages in the ***cages*** array
- **void addAnimal(Animal* animal, int cage_index)** - Adds the given animal to the cage at the given index in the ***cages*** array

TO-DO / HINTS

You will need to finish the implementation of various class member functions. Our declarations are provided within the header .h files. Think of declarations as similar to prototypes from CS 135; they give the name and parameter list for a function that we will implement later within our cpp files. You will need to finish these implementations, the definitions of the functions declared. Some functions have already been defined for you. You will only need to modify the unfinished functions within the **cage.cpp** and **animal.cpp** and submit just these 2 files, everything else has been written for you. The header files will contain the relevant variables and members you have access to for each class.
You can implement the classes in any order, but in terms of difficulty, the recommended order is: Gorilla, Zebra, Zoo, and end with Cage.
When implementing the Gorilla and Zebra classes, do not be afraid to use the existing Giraffe code for reference, everything will be very similar. Use the existing header files to see what members belong to what class and what can be accessed where (i.e. what is private, protected, public?). It is recommended to start with the Zoo class to get a handle on dynamic allocation before moving over to the Cage class. Once you feel comfortable allocating an array of Cages, try to think about how to similarly allocate an array of Animal*s. The examples from Canvas should act as a good reference; the code will be nearly identical aside from any types.

Compiling

This assignment has multiple files to better organize our code. When compiling multiple files, recall that we want to compile any existing cpp implementation files, but not header files, since those get added to our cpp files when we do an #include. Here, we have 3 cpp files: main.cpp, cage.cpp, and animal.cpp. To compile, we might do something like:

```
g++ -o zoo main.cpp cage.cpp animal.cpp
```

This will compile and link the three cpp files after including any other files, and then generate an output executable named zoo that can be run with ./zoo. Alternatively for this assignment, a makefile is provided. Recall that a makefile is a special type of file containing commands similar to what we just wrote with the intention of making the process easier as well as avoiding recompiling already compiled files that have not been updated. If you look in the provided makefile, you should notice a command very similar to the above one inside. To execute a makefile, simply use the Bash command **make** (Note that Bash is the terminal language on Linux and Mac).

Sample Run

```
blacks1@sally:~$./zoo
How many cages would you like to create?
3
How many animals should cage 0 hold?
4
Animal 0
What kind of animal should this be ('g', 'G', 'z')?
g
What should the animal be named?
Necky
How long is its neck?
5
Animal 1
What kind of animal should this be ('g', 'G', 'z')?
G
What should the animal be named?
Turk
Animal 2
What kind of animal should this be ('g', 'G', 'z')?
z
What should the animal be named?
Stripey
How many stripes does it have?
100
Animal 3
What kind of animal should this be ('g', 'G', 'z')?
G
What should the animal be named?
DK
How many animals should cage 1 hold?
1
Animal 0
What kind of animal should this be ('g', 'G', 'z')?
z
What should the animal be named?
```

```
Pokey
How many stripes does it have?
50
How many animals should cage 2 hold?
5
Animal 0
What kind of animal should this be ('g', 'G', 'z')?
g
What should the animal be named?
Giraffy
How long is its neck?
10
Animal 1
What kind of animal should this be ('g', 'G', 'z')?
G
What should the animal be named?
Diddy
Animal 2
What kind of animal should this be ('g', 'G', 'z')?
G
What should the animal be named?
Funky
Animal 3
What kind of animal should this be ('g', 'G', 'z')?
G
What should the animal be named?
Dixie
Animal 4
What kind of animal should this be ('g', 'G', 'z')?
z
What should the animal be named?
Horse
How many stripes does it have?
0

Cage# 0
There are 4 animals in this cage
0.
Necky the Giraffe
Neck Length: 5

1.
Turk the Gorilla

2.
Stripey the Zebra
Stripes: 100

3.
DK the Gorilla


Cage# 1
There are 1 animals in this cage
```

```
0.
Pokey the Zebra
Stripes: 50


Cage# 2
There are 5 animals in this cage
0.
Giraffy the Giraffe
Neck Length: 10

1.
Diddy the Gorilla

2.
Funky the Gorilla

3.
Dixie the Gorilla

4.
Horse the Zebra
Stripes: 0
```