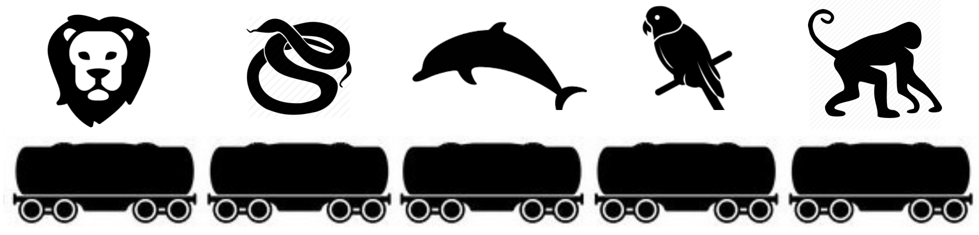
Project 4: ZooRecord “is-a” List of ? POLYMORPHISM



In Project3 we stored Animal objects into an ArrayBag. However, each entry in our data file is a different Animal-class. In Project4 we will use Polymorphism to store (via pointers) different Animal-derived objects into a List. Thus, Project4 will continue to build on the previous projects to work with **Lists** and **Polymorphism.**

We will work with the **List** class (doubly-linked List) discussed during lecture. This project will also be subdivided in **two parts**:

1. Modify the `Animal`, `Mammal`, `Bird` and `Fish` classes to support **Polymorphism**.

2. Modify ZooRecord to inherit from List and store **pointers** to `Animal`, where the pointers will actually point to either `Mammal`, `Bird` or `Fish` dynamic objects.

Implementation - 2 parts:

Part1: Modify Animal and derived classes

Modify the `Animal` class by modifying `display()` to be a **pure virtual function**. Each derived class can then **override** `display()` to display data specific to the derived object as follows:

- `Mammal::display()`
  Sample output:
  ```
  "animal_name is [not] domestic and [it is / is not] a predator,\n
  it is [not] airborne and it is [not] aquatic,\n
  it has [no] hair, [no] teeth, [no] fins, [no] tail and legs_ legs.\n\n”
  ```

Note: in the sample output above, **[text]** (text in square brackets) may or may not appear in the output, depending on the specific animal data.

- `Bird::display()`
  Sample output:
  ```
  “animal_name is [not] domestic and [it is / is not] a predator,\n
   it is [not] airborne and it is [not] aquatic.\n\n”
  ```

- `Fish::display()`
  Sample output:
  ```
  “animal_name is [not] domestic, [it is / is not] a predator\n
   and it is [not] venomous.\n\n"
  ```

Note: The formatting must match **EXACTLY**, please pay special attention to spacing, punctuation and capitalization

Part2: Modify the ZooRecord class

Modify the ZooRecord class to **inherit from List** and store **pointers to Animal** (You will find List, Node and Exception class files on Blackboard under Course Materials/ Project4 Files). ZooRecord must have at least (but is not limited to) the following methods:

```
/**parameterized constructor
 @pre the input file is expected to be in CSV (comma separated value) format
as:
"animal_name,hair,feathers,eggs,milk,airborne,aquatic,predator,toothed,backbo
ne,breathes,venomous,fins,legs,tail,domestic,catsize,class_type\n"
 @param input_file_name the name of  the input file
 @post adds Animal pointers to Animal-derived dynamic objects to record as
per the data in the input file
**/
ZooRecord(std::string input_file_name);
```

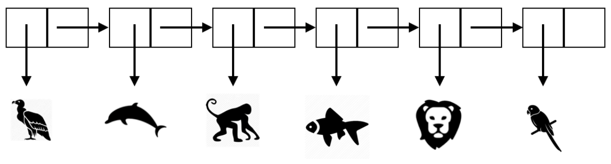
Note: our data file for this project has been modified to contain only class-types '1' (Mammal), '2' (Bird) and '4' (Fish).

```
/**@post displays all animals in record, one per line by calling appropriate
object's display method" **/
 void display();
```

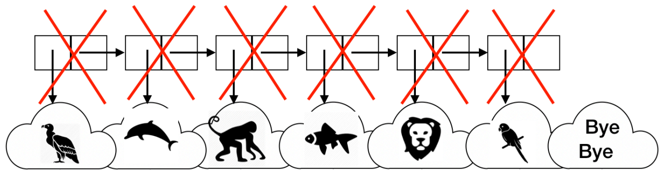
You must also **provide or allow for a default constructor** (e.g. explicitly tell compiler to provide one with keyword `default` in interface)

Extra Credit (10 points):

If you've been paying attention, you should be very bothered by the way we implemented ZooRecord thus far. ZooRecord stores pointers to dynamically allocated Animal-derived objects, but the base class (List) does not know about this.

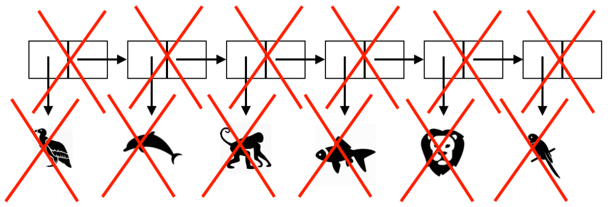


This means that if the ZooRecord is cleared or when a node is deleted or the ZooRecord object goes out of scope (the destructor is invoked), the List-derived remove() and clear() methods will only delete the dynamic nodes leaking the dynamic Animal-derived objects pointed to at each node.



To take care of this (for extra credit) do the following:

- Provide a destructor for ZooRecord that will take care of deleting the dynamic Animal-derived objects pointed to by the pointers stored in the ZooRecord nodes (you don't need to delete the nodes because ~List() will be invoked anyways. If you wish to delete the nodes as well for efficiency, when ~List() is invoked it will already be empty). You can do so by overriding clear() to delete all dynamic Animal-derived objects and then explicitly calling List::clear() to delete the nodes. You can call clear() from ~ZooRecord().
- Override `remove()` to delete the dynamic Animal-derived objects as well as the nodes. Don't forget to set the pointers stored in the nodes to `nullptr`.



Note: analogously, attention should be paid to copies as well <u>but to contain the size of the project we will not worry about that here.</u> Just keep in mind that copy constructors and assignment operators should both be taken care of to be used in ZooRecord in order to make "deep copies" of the Animal-derived dynamic objects.

Testing:

You must always test your implementation **INCREMENTALLY**!!!

In `main()` (not for submission) first test your modifications from Part 1. Instantiate objects of type Mammal, Bird and Fish and make calls to each object's display() to check that data is displayed correctly.

When Part 1 is implemented and tested then move on to Part 2, instantiate in main() a ZooRecord object, and call dIsplay() on it. Make sure that Polymorphism is correctly implemented (each different object type is correctly calling its own version of display()).

Grading Rubric:

- **Correctness 80%** (distributed across unit testing of your submission)
- **Documentation 10%**
- **Style and Design 10%** (proper naming, modularity and organization)
- A submission that implements all required classes and/or functions but does not compile will receive 40 points total (including documentation and design).

Submission:

You will submit all files for the 5 classes (**10 files**):

- The Animal class (`Animal.hpp` and `Animal.cpp`)
- The Mammal class (`Mammal.hpp` and `Mammal.cpp`)
- The Bird class (`Bird.hpp` and `Bird.cpp`)
- The Fish class (`Fish.hpp` and `Fish.cpp`)
- The ZooRecord class (`ZooRecord.hpp` and `ZooRecord.cpp`)

Your project must be submitted on Gradescope.

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You MUST test and debug your program locally.

Before submitting to Gradescope you MUST ensure that your program compiles (with g++) and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the "Programming Rules" document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you don't have through Gradescope.
"But it ran on my machine!" is not a valid argument for a submission that does not compile.
Once you have done all the above you submit it to Gradescope.

The due date is Friday October 11 by 5pm. No late submissions will be accepted.

Have Fun!!!!!

Image credits: https://www.iconfinder.com