Project 1: Welcome to 23500

This is a trivial project whose objective is to refresh your knowledge of classes and basic OOP such as inheritance. This will require you to implement the `Animal` abstract class along with the derived classes `Bird`, `Fish`, and `Mammal` (both the header and implementation files).

You will also be dealing with polymorphism and virtual functions which will be a new topic you will learn this semester during lecture.

In order to successfully complete this project, we strongly recommend that you look back to your CSCI 135 coursework as a reference and recall the difference between header and implementation files.

Background:

These classes deal with animals and their characteristics such as if they have fins or tails for example. Most of the code should be self-explanatory as to what it does. If you see a "predator" which is a boolean then you can assume true means the animal is a predator and false means it isn't.

Implementation:

You must separate interface from implementation (.hpp and .cpp files) Work through the tasks sequentially (implement and test). Only move on to a task when you are positive that the previous one has been completed correctly. Remember that the names of function prototypes and member variables must exactly match those declared in the respective header file when implementing a class. Implement the class based on the following specification (FUNCTION PROTOTYPES AND MEMBER VARIABLE NAMES MUST MATCH EXACTLY). Recall that accessor functions (e.g. getName()) are used to access the private data members (e.g. all getName() will do is return name_), while mutator functions give a value to the data members. You will also need to override classes (function that have the name but behave differently based on the class that calls them.) **Remember, you must thoroughly document your code!!!**

Remember, you must thoroughly document your code!!!

Task: Making the `Animal` class!

Implement the following constructors, members and methods:

public methods and functions:

```
Animal(); //default constructor
Animal(std::string name, bool domestic = false, bool predator = false); //parameterized
constructor

/** Gives you the animal's name
        @return the name_
*/
std::string getName() const;

/** Checks if animal is domestic
        @return true if domestic_ is true
*/
bool isDomestic() const;

/** Checks if animal is a predator
        @return true if predator_ is true
*/
bool isPredator() const;

/** Sets the animal's name to string name
        @param name of the animal you want to set it to
*/
void setName(std::string name);

/**
        Mutates the animal's domestic_ variable to true
        @post domestic_ is set to true
*/
void setDomestic();

/**
        Mutates the animal's predator_ variable to true
        @post predator_ is set to true
*/
void setPredator();

/**
        This is what makes the class abstract
        You will override this for each inherited class
*/
virtual void display() = 0;
```

protected members:

```
std::string name_;
bool domestic_;
bool predator_;
```

Notice the default argument in the parameterized constructor, do not include default arguments in the implementation files (cpp) but rather only in the hpp files.

Task: Making the `Bird` class!

Implement the following constructors, members, and methods:

public methods and functions:

```
Bird(); //default constructor
Bird(std::string name, bool domestic = false, bool predator = false); //parameterized constructor

/** Checks to see if Bird airborne_ is true
        @return airborne_
*/
bool isAirborne() const;

/** Checks to see if Bird aquatic_ is true
        @return aquatic_
*/
bool isAquatic() const;

/** Mutates airborne_ and sets it to true
        @post airborne_ is set to true
*/
void setAirborne();

/** Mutates aquatic_ and sets it to true
        @post aquatic_ is set to true
*/
void setAquatic();

/** @post displays bird data in the form:
        “animal_name is [not] domestic and [it is / is not] a predator,\n
        it is [not] airborne and it is [not] aquatic.\n”
*/
void display() override;
```

protected members:

```
bool airborne_;
bool aquatic_;
```

Notice you will have to call the base class constructor (Animal in this case) inside the `Bird` constructors in order to assign values such as name, domestic, and predator. Since `Bird` class inherits from `Animal` class, it inherits `Animal`'s public and protected members.

Task: Making the `Fish` class!

Implement the following constructors, members and methods:

public methods and functions:

```
Fish(); //default constructor
Fish(std::string name, bool domestic = false, bool predator = false); //parameterized constructor

/** Checks to see if venomous_ is true
        @return venomous_
*/
bool isVenomous() const;

/** Mutates venomous_ and sets it to true
        @post venomous_ is set to true
*/
void setVenomous();

/** @post displays fish data in the form:
        "animal_name is [not] domestic, [it is / is not] a predator\n
        and it is [not] venomous.\n"
*/
void display() override;
```

protected members:

```
bool venomous_;
```

Notice you will have to call the base class constructor (Animal in this case) inside the `Fish` constructors in order to assign values such as name, domestic, and predator.

Task: Making the `Mammal` class!

Implement the following constructors, members and methods:

public methods and functions:

```
//Alright, I'm getting tired of all the comments, you should understand what these do by now!
```

```
Mammal();
Mammal(std::string name, bool domestic = false, bool predator = false);
bool hasHair() const;
bool isAirborne() const;
bool isAquatic() const;
bool isToothed() const;
bool hasFins() const;
bool hasTail() const;
int legs() const;
void setHair();
void setAirborne();
void setAquatic();
void setToothed();
void setFins();
void setTail();
void setLegs(int legs);

/** @post displays bird data in the form:
        "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”
*/
void display() override;
```

protected members:

```
bool hair_;
bool airborne_;
bool aquatic_;
bool toothed_;
bool fins_;
bool tail_;
int legs_;
```

Notice you will have to call the base class constructor (Animal in this case) inside the `Mammal` constructors in order to assign values such as name, domestic, and predator.

Testing

How to compile:

```
g++ <test main file> -std=c++17
```

You must always implement and test you programs **INCREMENTALLY!!!** What does this mean? Implement and test one method at a time.

- Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable).
- Implement the next function/method and test in the same fashion. **How do you do this?** Write your own `main()` function to test your class. In this course you will never submit your test program, but you must always write one to test your classes. Choose the order in which you implement your methods so that you can test incrementally: i.e. implement mutator functions before accessor functions. Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use stubs: a dummy implementation that always returns a single value for testing Don't forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Grading Rubric

Correctness 80% (distributed across unit testing of your submission) **Documentation 10% Style and Design 10%** (proper naming, modularity, and organization)

Important: You must start working on the projects as soon as they are assigned to detect any problems with submitting your code and to address them with us **well before** the deadline so that we have time to get back to you **before** the deadline. This means that you must submit and resubmit your project code **early** and **often** in order to resolve any issues that might come up **before** the project deadline.

There will be no negotiation about project grades after the submission deadline.

Submission:

You will submit **the following files**:

`Animal.hpp` `Animal.cpp` `Bird.hpp` `Bird.cpp` `Fish.hpp` `Fish.cpp` `Mammal.hpp` `Mammal.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.