Introduction

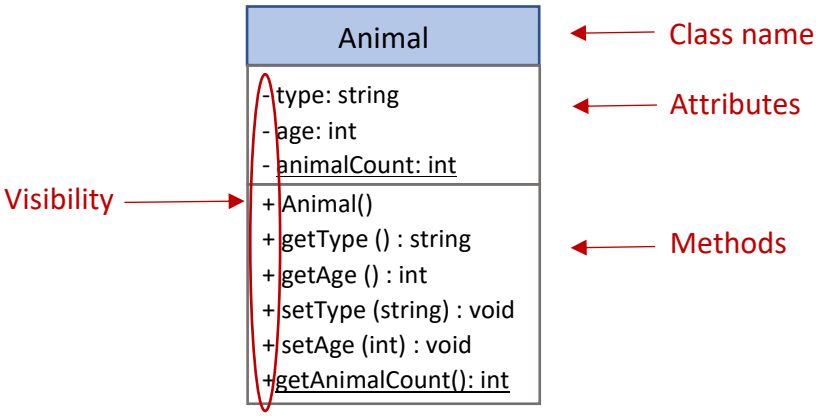
During this lab you will:

1. Create classes using a UML diagram.
2. Use classes as members of other classes (composition).
3. Practice using static class members and functions.

Part I – Understanding UML diagrams.

The Unified Modeling Language is a general-purpose modeling language in the field of software engineering that provides a standard way to visualize the design of a software system. A UML diagram is a powerful way to document the structure of a class with its data members (also called attributes, fields, or properties), methods (also called operations, or functions) and accessibilities. Though UML diagrams have many useful features, we will only look at a few at this time.

A UML diagram below shows how to document class Animal. The top portion contains the name of the class.



The second part contains member variables with their type listed after a colon. In the diagram above, the member variables are: a string *type,* an integer *age*, and a static integer *animalCount*. These three members are private, as indicated by their visibility indicator. A “minus” indicates a private visibility of a data member or a method that can only be accessed within the class. A “plus” indicates a public visibility of a member or a method that can be accessed by any class. Static variables (described in the following section) are underlined. Constants, if you have any, will be in upper case.

The third part of the diagram contains member methods, along with their incoming parameters (inside the parenthesis), followed by a colon and a function return types. We also indicate visibility (accessibility) of each member or method with a plus or a minus sign, as described above. The list usually starts with a constructor. Constructors do not

have a return type. Method *getType( )* has no incoming parameters (typical of getter methods), and has a string return type. Setters typically take a parameter, and return a void. Getters and setters are usually grouped together. Static methods are underlined (and are discussed in the next section). Constant methods are typed in upper case.

There are several rules of how elements appear in the UML diagram. Variable names usually start with a lower case letter and use a camel case, if necessary. Methods start with a lower case letter and can use camel case as well. Constant members and methods appear in the uppercase, and static members and methods are underlined, as you will see in the example below.

Part II – Composition

Classes in a software system have relationships.

The concept of composition is a design technique that is extremely useful. The idea behind composition is that we want to create sophisticated classes/models by composing them of many smaller, simpler classes. We want to combine many small pieces into a larger system.

This process of building complex objects from simpler ones is called object composition. Broadly speaking, object composition models a “has-a” relationship between two objects. A car “has-a” transmission. Your computer “has-a” CPU. The Dog “has-an” Owner. There is also a “is-a” relationship (inheritance) that we will explore in one of the future labs.

In our lab program we will have two classes: a Dog, and an Owner, and in this case an Owner is a member of the Dog class. We say that one class “has” another class as a member. The class Owner is also called immutable, once an instance is created, it can not be changed (mutated). Therefore, Owner does not have any setter methods. More on immutables later.

In the UML diagram classes that have relationships are connected with arrows. There are several types of arrows indicating the variety of relationships that classes can have, but we will not look at them today. For now the arrow indicates that an instance of class Owner will be a member of a Dog.

Part III - Static members and methods.

A class variable can be an instance variable (each instantiated object has its own copy of that variable), or it can be a class-wide variable (all objects instantiated from the same class share one copy of that variable). This is what *static* variables are. You can create as many objects of the class as you wish, and each class will have their own copy of a non-static instance variable, but they will all share the same class-wide static variable. To be able to access and manipulate that static variable (if it is private) you will also need a public static method. Static variable can also be public.

You can have more than one static variable and method in your class. Static variables are very useful, as they can be used to keep class-wide information. For example, one common use of a static variable is keeping track of how many class instances were created by incrementing the

static count variable in every constructor used to instantiate an object. Since such static variable is a class-wide variable, you can call the static function that returns that variable even when no class objects were instantiated. In this case the function will need to be preceded by the class name and the scope resolution operator. You will see it below.

For example, class Animal we saw above has a private *static int animalCount* that keeps track of how many Animal objects were instantiated in our program. Because this variable is private, we need an accessor function to be able to access it:

```
static int getAnimalCount() { return carCount; }
```

In our main program, when we instantiate an object, the variable animalCount will be incremented in each of the class constructors. We have to write the code to increment that variable in every constructor, because we do not know ahead of time which constructor will be used by the user to instantiate the objects. (This is at least one of the reasons that we cannot trust the compiler to provide a default constructor for us – compiler is not aware of out design considerations, and will not increment static variables for us in the provided default constructor). We can later invoke this static function, as we would any other member function to obtain the count of Animal objects. For example,

```
Animal tiger;
int count = tiger.getAnimalCount ();
```

The interesting thing is, we can even call this function without having created any Animal objects, since static variables are class-wide variables and exist even if no class objects were created. Here is how to do it:

```
int count = Animal::getAnimalCount();
```

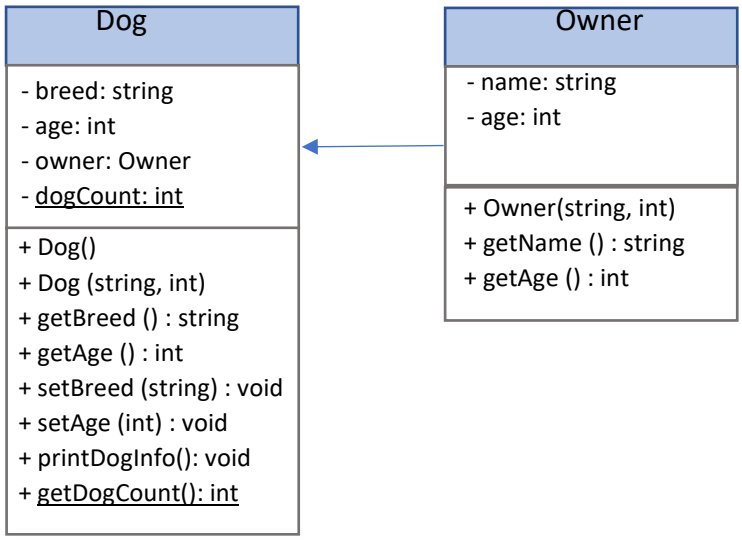
There is also one more important point. Static variables need to be initialized with their initial value (whatever it is) at the beginning of the program, but we cannot do it in the class specification (declaration) or in a constructor (but you can manupulate/increment it in the constructor). So, we need to add a line of code initializing our static variable outside of the class declaration. It is recommended to do it under the class declaration, in case all your functions are inlined and you do not have a .cpp implementation file. Otherwise, it will be done in the class implementation file that contains the definition of all the class methods. This is how you would initialize a static variable:

```
int Animal::animalCount = 0;
```

Part IV. Writing your program

Now you will write a program that consists of two classes, a Dog class and an Owner class. Their specification is shown in the UML diagram below. Notice that in our design, every Dog has an Owner class member. Class Owner is immutable, as mentioned above. An immutable

class is just a class whose members cannot be changed (mutated) after an object was instantiated. Therefore, the Owner class does not have any setter methods. The Owner's class attributes must be set at the time of creation (in the Owner's constructor). You will call Owner's constructor from inside Dog's constructor. Do not forget to do it inside *each* constructor in class Dog.



Here are the steps:

1. Create a Dog.h file that contains your Dog class declaration. Do not inline any functions there, even though they are very short. Your implementation will go into the Dog.cpp file. It is a very good practice to create separate specification and implementation files so you can practice the use of correct class layout/syntax.

2. Create an Owner.h file with Owner class declaration. Do not inline functions, as above.

3. Create Dog.cpp and Owner.cpp files. Your static variable will be initialized in the Dog.cpp implementation file.

4. Create you driver program. Before instantiating any Dog objects, print the value of the static variable *dogCount* to demonstrate that it works without creating any Dog objects. Then create several Dog objects and demonstrate that *all* your methods work. The method printDogInfo method should print all dog and owner information in the following format:

```
Before instantiating Dog objects dogCount is: 0

Dog 1:
breed:      Golden Retriever
age:    2
owner:      Kevin Smith, 34 yo
```

Dog 2:
breed: Staffordshire Terrier
age: 4
owner: Lana Strange, 23 yo

After instantiating Dog objects dogCount is: 4 // (or whatever your count is)

What/How to submit

That's it, you are done! You can now submit your zipped tarball (containing source code only) to canvas to be graded.

NOTE: If you submit the archive that cannot be open, or is corrupt, you will not be given the opportunity to redo the work and resubmit. You have one shot to get it right.