# CMSC 313 — Spring 2017
# Project 7 — Poor Man's Polymorphism

| | |
|---|---|
| **Assigned** | Monday, April 24th |
| **Program Due** | Sunday, April 30th by 11:59pm |
| **Updates:** | ▪ Fixed struct members to match main7.c<br>▪ Cleaned up punctuation in sample outputs to consistently end with '.'<br>▪ Small change to the declaration of `Lion_vtable` example. |

## Objectives

The objectives of the programming assignment are

1. to practice using malloc()
2. to practice using pointers to functions
3. to gain insight into how OOP and polymorphism are implemented

## Background

In this project, you will use plain old C structs to mock up polymorphic behavior of classes in an object-oriented programming (OOP) language. In C++, polymorphism is implemented using something called a "Vtable", or virtual function table. As discussed in class, this Vtable is an array of pointers to functions, with each method assigned to a specific position (index) in the array. So, if a class "Animal" has three virtual methods--say, eat(), speak(), and display()-- the Vtable will have three slots, and any subclasses of Animal will have Vtables with the same initial slots, plus more if additional virtual methods are defined. So, the Animal class and each of its subclasses will have their own distinct Vtables (one per class), but their Vtables will all have the same slot assignments for the initial slots, e.g., all will have their slot 0 point to some implementation of the eat() method.

Put another way, the Animal class (and therefore all Animal objects) have a Vtable that has slots for each of the class's virtual methods. Each method (eat(), speak(), etc.) will have a fixed slot index. So, invoking the appropriate method reduces to knowing the correct slot to pull out a function pointer from, and calling it. The slot numbers are assigned at compile time.

Then, each class that is derived from the Animal class will also have a Vtable with the same initial size and slot designations. So, not only all Animals, but also all Lions, will have an eat() method pointer in the same slot in each of their respective Vtables, so that it will be easy to invoke "eat()" on each of them, whether Animal or Lion.

Lastly, since the Vtable can be modified, i.e. the actual functions pointed to from the Vtable can be different in the subclasses, that allows the subclass objects to behave differently to the same method invocation. Voila: polymorphism!

Note that each and every object (instance) contains a pointer to a Vtable, which means that potentially, every object, even two objects from the same class, can potentially have distinctive Vtables, which would allow us to customize the behavior of each and every object individually. However, C++ (and Java, also) do not implement it this way: all objects of a given class share the same Vtable, so while a Lion would have a different Vtable from its parent Animal class, all Lion objects would point to the same Vtable and therefore behave similarly.

# Assignment

For this project, you will be creating the files "AnimalClasses.h" and "AnimalClasses.c" (note: *not* .cpp!). In AnimalClasses.h, you will define 3 "classes":

- a base Animal class
- a Lion class, derived from Animal
- a Fish class, again derived from Animal

When we say "class D, derived from class B", we don't really mean classes, nor do we really mean "derived", since we are programming in C, not C++. Instead, you will be defining C structs, which are very similar to C++ classes. "Derivation" in this context means the struct for the "child class" will be identical (and we do mean ***identical***) to the parent struct up to the end of the parent, with the child struct then adding more data members only ***at the end***. In C, if the first part of two structs are identical in members and their types, they will be laid out in memory in identical fashion, so a pointer to one is interchangeable with a pointer to the other, as long as you only access the data members they have in common.

Next, you will define the various methods (i.e., functions) to support these "classes", much as you would in any OOP language.

Lastly, you will create the tables of function pointers that will make our polymorphic behavior all work!

Each of these steps is outlined in detail in a section below:

## Step 1: Defining the "Class" Structs

In the file `AnimalClasses.h`, you will define each of the 3 required class's struct according to the following template:

```
/* This is the "base" class: */

typedef struct {
    /* Note the "**": this will point to an array of pointers-to-func */
    int (**vtable)(void *, char *);

    /* Following are data members for base class,
     * to be inherited by all derived classes
     */
    int foo;
```

```
    char *fum;
} Base;


/* This is a "subclass", "derived" from the class above: */

typedef struct {
    /* Any "derived class" struct: should first start with an exact copy
     * of the parent struct
     */
    int (**vtable)(void *, char *);
    int foo;
    char *fum;

    /* then any additional data members: */
    float fee;
} Derived;
```

Use the above template to define structs for an Animal, Lion, and Fish class, where Lion and Fish are child classes of Animal. Note that the "vtable" pointer is the very first field in each of our structs--this is very important!

The Animal class should have the following fields, with given types:

- (vtable, obviously)
- char *name;
- int weight:
- char *sound;

The Lion struct should start with an exact copy of the Animal struct, and then add the following field:

- int is_alpha_male; /* This is used as a boolean */

(Hint: it would be easiest to just cut-and-paste the struct definition for Animal and add to it to create the Lion and Fish structs.)
and the Fish should add the following field to what it copied from Animal:

- char *fin_type;

Note that all data members for our "classees" will be effectively public. Also, note that there is no explicit connection between the various classes' struct definitions, other than the fact that the initial parts are identical.

## Defining the "Methods"

The functions described here should be in the file `AnimalClasses.c`: this includes both the function prototypes as well as the function definitions. They will not be called directly from code outside your AnimalClasses.c.
Just like in C++, we must define each of our methods for each of our classes. We are keeping this simple--our subclasses will not add any additional virtual methods to the Vtable beyond what the parent class defines (although it is worth thinking about how you would implement this). So, each class will have 3 methods: `XXX_eat()`, `XXX_speak()`, and `XXX_display()`, which comes out to 9 methods total across our 3 classes. Don't worry: each method implementation is a trivial 1- or 2-line implementation. Also, since this is C, not C++, the "methods" are actually just simple C functions. The exact task of each method is given in another section below.

To keep things simple, we are restricting all of our methods to take only one parameter, a string (technically, char*), and always return an int. Actually, since C won't support the implicit "this" calling object, we must pass that in, too. So I lied: every "method" function will take 2 parameters, with the first always being a pointer to the calling object. You should name your method functions in some consistent manner--we suggest _(), so for example:

```
int Animal_eat(void *this, char *arg);
...
int Fish_display(void *this, char *arg);

/* There will be 9 functions in all */
```

Next, you need to actually define the code for your class methods: the functions Lion_eat(), Lion_speak(), ..., Fish_display(). That will be 9 functions in all. These should be defined in the file AnimalClasses.c. IMPORTANT: Note that the type of the first parameter is `void *`, so inside the functions, you must cast/assign it to the correct specific type: Animal*, Lion*, or Fish*.

**The `eat` Method:**
The eat method for each class should print out a distinctive

```
Animal:{instancename} eats <arg>.
```

or:

```
Lion:{instancename} chomps <arg>.
```

or:

```
Fish:{instancename} nibbles <arg>.
```

where `<arg>` is the argument that is passed into the method (in addition to `this`). (Note that the other methods do not use their `arg` parameter for now.)

**The `speak` Method:**
The speak methods for each class are slightly different. The Animal's speak method should print out:

```
Animal:{instancename} says "{sound}".
```

where {sound} is the value of the "sound" field in that instance.
The Lion's speak method should print out:

```
Lion:{instancename} roars "{sound}".
```

and a Fish's speak method should print out:

```
Lion:{instancename} burbles "{sound}".
```

**The `display` Method:**

The display() method should print out:

```
{classname}:{instancename}:
    weight: {weight}
    sound:  {sound}
```

Additionally, for a Lion, you should print out:

```
    is alpha male: {true|false}
```

and for a Fish, add:

```
    fin type: {finType}
```

**Implementing the Methods:**

You should start by writing the three methods for the Animal class. It should then be very easy to just cut-and-paste this to create the 3 methods for the Lion class, and the 3 methods for the Fish class. They are very similar.
For now, have all your methods return 0.

## Defining the Vtable

We must now define the Vtables for our polymorphism to work. We must define one Vtable per class--3 in all. First, we need to assign the slots in our Vtable. ***IMPORTANT:***This should be in the file AnimalClasses.h:

```
#define EAT_METHOD     0
#define SPEAK_METHOD   1
#define DISPLAY_METHOD 2
```

Once you have these offsets defined, you should statically define the Vtable for each class, following this example. (***IMPORTANT:***This should be in the file AnimalClasses.c, *after* the function prototypes for the 9 method functions.)

```
static int (*Lion_vtable[3])(void *, char *) = {
    &Lion_eat,
    &Lion_speak,
    &Lion_display
};
```

You will need a separate Vtable for each of your 3 classes.
Then, when constructing objects of a given class (described below), you will set the `vtable` field of the object to point to the appropriate Vtable (one of the 3 you just defined).

## Defining "Constructors"

You will define a combination allocator/constructor for each class, named `new_()` (the exact name is important!). The function must be declared in `AnimalClasses.h`, and defined in `AnimalClasses.c` For example:

```
Lion *new_Lion(void) {
   ...
}
```

When called, it should malloc() the appropriate amount of space for the object, and most importantly, initialize the "vtable" member to point to the appropriately defined Vtable for that class (you made this in the previous section). You do not need to initialize any of the other fields. The caller is responsible for that.
The function should return a pointer to the newly-allocated object.

## Invoking Our "Virtual Methods"

Lastly, you must implement the polymorphically-behaving method invocation, by defining the function "invoke()", which takes a pointer to the invoking object, the method index (e.g., EAT_METHOD), and the single argument that the method all take, a char*. It should then use the object argument to look up the Vtable, and call the function in the correct slot, passing in two parameters: the pointer to "this", as well as the char* argument. invoke(void *this, METHOD, char *arg);
Here is the prototype, which should go into AnimalClasses.h:

```
int invoke(Animal *this, int method_slot, char *arg);
```

The definition should go in AnimalClasses.c. You must fill in the body. It should use `this` to access the vtable field, index into the vtable array using `method_slot` to get a pointer to the appropriate function, and call that function passing in both `this` and `arg` as the two arguments, and return the value returned by the pointed-to function. (For example, the pointed-to function might be `Lion_eat()`.)
Hint: the entire body of `invoke()` can be a single line of code!

## Testing Your Code

After you are done writing the code, your file `AnimalClasses.h` should contain:

- Definitions for the three typedef'ed structs `Animal`, `Lion`, and `Fish`.
- #define's for the 3 methods' indices (e.g.: "`#define EAT_METHOD 0`")
- Function prototypes for the three "constructors" `new_{Animal,Lion,Fish}()`
- Function prototype for the function `invoke()`

Your file `AnimalClasses.c` should `#include` <stdlib.h> (for malloc()), <stdio.h> (for printf()), and obviously AnimalClasses.h, and should contain:

- Definitions for the three Vtables `{Animal,Lion,Fish}_vtable`
- Definitions for the three "constructors" `new_{Animal,Lion,Fish}()`
- Definitions for the 9 "methods" `{Animal,Lion,Fish}_{eat,speak,display}()`
- Definition for the function `invoke()`

We have provided a test program "main7.c", which should compile cleanly, and produce the following output:

```
$ ./main7.out
Invoking methods for object 0 (class Lion):
Lion:Nala roars "GRRRR".
Lion:Nala chomps impala.
Lion:Nala:
weight: 1000
sound: GRRRR
is alpha male: false
------------------------

Invoking methods for object 1 (class Fish):
Fish:Nemo burbles "blub".
Fish:Nemo nibbles worms.
Fish:Nemo:
weight: 1
sound: blub
fin type: round
------------------------

Invoking methods for object 2 (class Animal):
Animal:JaneDoe says "mmph".
Animal:JaneDoe eats paste.
Animal:JaneDoe:
weight: 10
sound: mmph
------------------------

Invoking methods for object 3 (class Fish):
Fish:Dory burbles "blub".
Fish:Dory nibbles squid.
Fish:Dory:
weight: 5
sound: blub
fin type: pointed
------------------------

Invoking methods for object 4 (class Lion):
Lion:Simba roars "GROWL".
Lion:Simba chomps mice.
Lion:Simba:
weight: 1001
```

```
sound: GROWL
is alpha male: true
------------------------
```

We have also provided a Makefile as well. These various files are in the directory:
"`/afs/umbc.edu/users/p/a/park/pub/cmsc313/spring17/proj7/`"
You should add your two files `AnimalClasses.h` and `AnimalClasses.c` to the directory and run "make", then run main7.out to see if your code works. You should not see ***any*** warnings during the compile!
We strongly encourage you to also tinker with the main7.c provided to test out your code more thoroughly. However, this time, we are not requiring that you submit any additional test code.

## What to Submit

Before you submit your program, record a sample run of your program using the UNIX script command.

Use the UNIX `submit` command on the GL system to turn in your project. You should submit *three* files: (1) the C header file defining your class structs; (2) the C source code that implements the methods, vtables, etc. and (3) the typescript file of your sample run. The UNIX command to do this should look something like:

```
submit cs313_park proj7 AnimalClasses.h AnimalClasses.c typescript
```