National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS213: Advanced Programming

Class: BESE – 4B

Lab 8: Polymorphism

Date: 9th May, 2016

Time: 10:00 AM to 12:50 PM

Instructor: Mr. Numair Khan

Lab 8: Polymorphism

Introduction

In our last two classes we have studied how polymorphism, or run-time method binding, works in C++ and Java, and how it is implemented using vtables. In this lab we will be simulating polymorphism in C, which is not an object-oriented language. In practical scenarios, you will rarely be using C if you need polymorphic behavior in your program. The purpose of this lab, then, is to familiarize you with how polymorphism works under the hood and, hence, to understand the tradeoffs involved in utilizing polymorphism in any object-oriented language like Java, C++ or C# (bear in mind that originally, C++ compilers used to translate C++ code into C code first, and then call a C compiler to generate object files and for linking).

Moreover, although rare, there may be scenarios where you would like to leverage the high-performance of a low-level language like C, while also benefiting from the organizational benefits of object oriented programming. Having the knowledge to implement *partial* object-oriented behavior in such cases can be beneficial (Chipmunk2D, a 2D physics engine for computer games, achieves this to great effect).

Objectives

After performing this lab students will be able to:

- Simulate object-oriented behavior in C
- Write low-level code to implement polymorphism for any high-level language

Tools/Software Requirement

- C

Deadline

- The deadline for submitting Lab 8 is 23:55PM on Tuesday, 10th April 2016.

You will be implementing the vector and matrix classes you implemented in an earlier lab, but this time around you will be doing it using only C code. The matrix class has the following data members:

1. Number of rows
2. Number of columns

And the following methods:

1. Right multiply the matrix with another matrix
2. Add the matrix to another matrix
3. Find the L1 norm (what the hell is that?!) of the matrix

The vector class inherits from the matrix class with the added restriction that the number of columns is always equal to 1. The vector class adds the following new data members:

1. The maximum element of the vector
2. The minimum element of the vector

The 3rd method of the matrix class - calculate the L1 norm – is a virtual method. It is overridden in the vector class (since the L1 norms is calculate differently for vectors). The vector class also overrides the addition method – but this method is not a virtual method (hence, it is bound statically).

Lab Task 1 – Methods

Now, for simulating the behavior of the above described class in C. Classes in C++ can be replaced with structs in C. How will inheritance work with structs? Also, structures will not allow methods to be defined in them.

This is a good point to discuss a new calling convention. Thus far, you have learnt about the `_cdecl` and `__stdcall` calling conventions. Another calling convention, used for class member functions in C++, is `__thiscall.` You can read about the finer details of this calling convention online, but one of its important features is that a pointer to the object on which the function is being called is passed automatically to the function. You have probably already used the `this` keyword in Java, which is nothing more than a pointer to the current object.

Consider an example.

```
class myClass {

   public:

      void myFunction (int arg1);

};

myClass myClassObject;

myClassObject.myFunction(20);
```

The above code will be translated to the following by the compiler. Notice how the compiler adds an extra argument to the method.

```
class myClass {

   public:

      void myFunction (myClass* this, int arg1);

};

myClass myClassObject;

myClassObject.myFunction((myClass*)&myClassObject, 20);
```

This gives us a clue about how to go about simulating methods in C. Create global functions which accept as their first argument the structure on which they are to operate:

```
typedef struct MyStruct_ {

   int a;

   float b;

} MyStruct;

void myStructMethod( MyStruct* this, int arg1) { … }

…

MyStruct A;

myStructMethod(&A, 20);
```

What if we would like to use syntax similar to C++ to call the methods: `myClassObject.myFunction(20)`? You have already learnt how to do this in C (that is, to treat functions like data-types).

Implement the methods and data members of both the vector and matrix classes using structs in C. Overridden methods should be implemented for each class separately. You should be able to call the methods using syntax similar to C++:

```
MyStruct A;

init_myStruct();  //acts as constructor

A.myStructMethod( &myStructObject, 20);
```

For each "class" create a function which acts as a constructor (you may call this function `init_*()`) . Whereas, the C++ constructor is called automatically by the compiler, you will have to explicitly call the `init()` function whenever an "object" of your "class" is created. Among other things, you may want to use the `init()` function to set up the C++ like method calling behavior just discussed.

Lab Task 2 – Vtables

Now for the virtual method in our class. For each class create an array of function pointers (like you did in your very first lab). This will be the vtable for the class:

```
typedef void (*vmethod) ();
```

…

```
vmethod *vtable_vector_class = malloc(sizeof(vmethod));

//create another array as the vtable of the matrix class
```

Now add a pointer to this array to the definition of the class you created above using structs. Call this pointer `vtable_ptr.` Any call to a virtual method should then translate as follows:

```
myObject.l1norm();      //in C++

myObject.vtable_ptr[0]; //in our C implementation
```

As you can see the actual method called can be changed at runtime by making `vtable_ptr` point to a different vtable.

Your C code should allow the following usage (read the code carefully):

```
Matrix *m = (Matrix*)malloc(sizeof(Vector));
init_matrix(m);    //initialize the static fields of m AS A MATRIX
m->vtable_ptr = ?; //Which vtable should this point to?
m.add(m, m2);     //Will this call be bound statically or
                  //dynamically?
                  //If statically, which add() function is called? The
                  //one defined for Matrix or for Vector. Remember,
                  //this is a decision you make in the init function
m->vtable_ptr[0]();
```

Looking at the above code you can also understand why C++ allows an object of a child class to be assigned to an object of a parent class, but not vice versa. Vector inherits from Matrix so we know all the fields in Matrix are also present in Vector. Hence, we can call `init_matrix()` above by passing it a Vector object, and we know it will not try to initialize a field that does not exist. If, instead, we had tried to call `init_vector()` by passing it a Matrix object, we would have run into an error when the function tried to set the data member representing the minimum element of the vector – this data member does not exist for Matrix objects.

Deliverables

A text file containing the address of the GIT repository to which you will upload your source code.

If you have any questions, the Piazza forum is the way to go.