Function pointers in C can be used to perform object-oriented programming in C.

For example, the following lines is written in C:

```
String s1 = newString();
s1->set(s1, "hello");
```

Yes, the `->` and the lack of a `new` operator is a dead give away, but it sure seems to imply that we're setting the text of some `String` class to be `"hello"`.

By using function pointers, **it is possible to emulate methods in C**.

How is this accomplished?

The `String` class is actually a `struct` with a bunch of function pointers which act as a way to simulate methods. The following is a partial declaration of the `String` class:

```
typedef struct String_Struct* String;

struct String_Struct
{
    char* (*get)(const void* self);
    void (*set)(const void* self, char* value);
    int (*length)(const void* self);
};

char* getString(const void* self);
void setString(const void* self, char* value);
int lengthString(const void* self);

String newString();
```

As can be seen, the methods of the `String` class are actually function pointers to the declared function. In preparing the instance of the `String`, the `newString` function is called in order to set up the function pointers to their respective functions:

```
String newString()
{
    String self = (String)malloc(sizeof(struct String_Struct));

    self->get = &getString;
    self->set = &setString;
    self->length = &lengthString;

    self->set(self, "");

    return self;
}
```

For example, the `getString` function that is called by invoking the `get` method is defined as the following:

```
char* getString(const void* self_obj)
{
    return ((String)self_obj)->internal->value;
}
```

One thing that can be noticed is that there is no concept of an instance of an object and having methods that are actually a part of an object, so a "self object" must be passed in on each invocation. (And the `internal` is just a hidden `struct` which was omitted from the code listing earlier -- it is a way of performing information hiding, but that is not relevant to function pointers.)

So, rather than being able to do `s1->set("hello");`, one must pass in the object to perform the action on `s1->set(s1, "hello")`.

With that minor explanation having to pass in a reference to yourself out of the way, we'll move to the next part, which is **inheritance in C**.

Let's say we want to make a subclass of `String`, say an `ImmutableString`. In order to make the string immutable, the `set` method will not be accessible, while maintaining access to `get` and `length`, and force the "constructor" to accept a `char*`:

```
typedef struct ImmutableString_Struct* ImmutableString;

struct ImmutableString_Struct
{
    String base;

    char* (*get)(const void* self);
    int (*length)(const void* self);
};

ImmutableString newImmutableString(const char* value);
```

Basically, for all subclasses, the available methods are once again function pointers. This time, the declaration for the `set` method is not present, therefore, it cannot be called in a `ImmutableString`.

As for the implementation of the `ImmutableString`, the only relevant code is the "constructor" function, the `newImmutableString`:

```
ImmutableString newImmutableString(const char* value)
{
    ImmutableString self = (ImmutableString)malloc(sizeof(struct ImmutableString_Struct))

    self->base = newString();

    self->get = self->base->get;
    self->length = self->base->length;

    self->base->set(self->base, (char*)value);

    return self;
}
```

In instantiating the `ImmutableString`, the function pointers to the `get` and `length` methods actually refer to the `String.get` and `String.length` method, by going through the `base` variable which is an internally stored `String` object.

The use of a function pointer can achieve inheritance of a method from a superclass.

We can further continue to **polymorphism in C**.

If for example we wanted to change the behavior of the `length` method to return `0` all the time in the `ImmutableString` class for some reason, all that would have to be done is to:

1. Add a function that is going to serve as the overriding `length` method.
2. Go to the "constructor" and set the function pointer to the overriding `length` method.

Adding an overriding `length` method in `ImmutableString` may be performed by adding an `lengthOverrideMethod`:

```
int lengthOverrideMethod(const void* self)
{
    return 0;
}
```

Then, the function pointer for the `length` method in the constructor is hooked up to the `lengthOverrideMethod`:

```
ImmutableString newImmutableString(const char* value)
{
    ImmutableString self = (ImmutableString)malloc(sizeof(struct ImmutableString_Struct))

    self->base = newString();

    self->get = self->base->get;
    self->length = &lengthOverrideMethod;

    self->base->set(self->base, (char*)value);

    return self;
}
```

Now, rather than having an identical behavior for the `length` method in `ImmutableString` class as the `String` class, now the `length` method will refer to the behavior defined in the `lengthOverrideMethod` function.

I must add a disclaimer that I am still learning how to write with an object-oriented programming style in C, so there probably are points that I didn't explain well, or may just be off mark in terms of how best to implement OOP in C. But my purpose was to try to illustrate one of many uses of function pointers.

For more information on how to perform object-oriented programming in C, please refer to the following questions:

- Object-Orientation in C?
- Can you write object oriented code in C?