Object Oriented Inheritance Pattern

This pattern is used to implement inheritance where one object inherits the data fields and methods of another object.

In this module we will:

- **Simple inheritance**: that uses "containment" where parent object contains the base object.
- **Pitfalls of doing this 'the usual way'**: we will look at design variations that we must avoid.
- **Practical use case**: list items where we want to organize custom items into a linked list.

Whether a programming language provides language support for inheritance or not, we still have to use the concept of inheritance to organize our software. Inheritance is not a language feature - it is an element of software architecture that spans all programming languages. Language features just add syntactical convenience when implementing these architectural features in code.

Defining Characteristics

The C programming language lacks direct language support for inheritance so we must use conventions instead when we work with inheritance in C.

A few of the conventions I will cover in this module are the following:

- **Containment of base class data**: our derived class simply contains data of its base class and uses base class methods to operate on that data. Derived object never references the base class data through a pointer.
- **Derived object tightly coupled to base data**: The derived class can directly access base class data and even though we always try to use base class methods to manipulate base class, the inheritance pattern doesn't forbid direct access to base class data.
- **Derived object must provide method wrappers**: if we want to manipulate base class data then we must always explicitly define derived object methods that would then either delegate this operation to base class method or modify the data directly. We never modify base class data directly.

Use Cases

The primary use of inheritance in C is for arranging data into more complex structures so that we can generalize algorithms that operate on our custom data structures. Examples include: lists, avl trees, etc.

- **Code reuse**: whenever we need to 'extend' the functionality of a base object, we can wrap it into a derived class and add a derived interface around the base object - but we can still reuse the code that is part of the base object implementation.
- **Generic data structures**: we use the inheritance pattern to create generic data structures such as lists in C. This allows us to reuse list manipulation functionality and even organize arbitrary structures into lists (as we have seen with the callback pattern) by simply 'inheriting' from a list node. We can do the same when we implement many

other data structures.
- **Extending other objects**: through code reuse, we can create new data structures that have extended functionality which underneath use a base data structure but expose to the user a far richer interface than the original data structure.

Benefits

Object oriented inheritance patterns are useful because they:

- **Clean design**: when we reuse the same functionality in many places it becomes easy to know what to expect when looking at the code. The opposite of this is reimplementing for example the same list algorithms throughout many different objects in the firmware - which leads to never knowing what to expect from such an implementation.
- **Clear expectations**: we can think of our code much more hierarchically and humans are far better at grasping hierarchical concepts than concepts that have many small details that need to be considered at the same time. If we structure our data hierarchically, we also create a software architecture that is very easy to grasp.
- **Simple high level architecture**: through inheritance we can compartmentalize the logic at different levels of abstraction, leading to a much simpler design at the application level because we only need to deal with high level abstractions at that level.

Drawbacks

The main drawback of this pattern is that if we really want to inherit the whole type (ie we want to inherit full api functionality of another type) then we need to create wrappers in the type that inherits the base class. These methods would then simply forward the calls to base class methods.

The reason for this is mainly that the C language doesn't support inheritance at language level meaning that we can not let the compiler and linker do this for us.

Thus, this pattern comes with the following drawbacks:

- **Unnecessary code when wrapping methods**: if we want to fully inherit a data structure and expose a full interface to the user of that structure, we must also create wrappers for the base class methods in order to be able to use the structure in a clean way.
- **Violation of encapsulation**: Inheritance by definition means that all the data of the base class automatically belongs to and can be modified by the deriving class. If the derived class directly accesses data of the base class then this can have unintended effects.
- **Added complexity**: this is particularly true for virtual inheritance. Without automatic language features it becomes harder to keep things clean and consistent across the whole project because it places the responsibility on developers to follow design patterns.

Implementation

Inheritance in C is primarily implemented through containment, meaning that one object that

inherits another object simply 'contains' the base class.

Inheriting objects

```
struct my_object {
    struct base_one base_one;
    struct base_two base_two;
};
```

Here we can say that we are making **`struct my_object`** inherit the base objects and we can then forward any operations we do on **`struct my_object`** instances to the implementation of the base objects.

List item type

You have already seen the basic concept of inheritance in the module covering the callback pattern. For convenience, we will recap it here. The basic concept we would like to implement is to reuse the functionality responsible for arranging items into a list by inheriting from the list item.

A data structure that we would like to arrange into a list can inherit from a list item like this:

List type

```
struct _snode {
    struct _snode *next;
};

typedef struct _snode sys_snode_t;

struct _slist {
    sys_snode_t *head;
    sys_snode_t *tail;
};

typedef struct _slist sys_slist_t;
```

Inheriting list item

Without going into details of the implementation, we can now inherit from the list item and by doing so make it possible to arrange arbitrary structures into a list:

Usage example

```
struct my_object {
    sys_snode_t node;
};

sys_slist_t list;
struct my_object obj1;
struct my_object obj2;

sys_slist_init(&list);
```

```
sys_slist_append(&list, &obj1.list);
sys_slist_append(&list, &obj2.list);
```

Note that it is very important that we take care of only having items of one particular type in our list. Otherwise we don't have a way of telling which item we have when we iterate the list.

Iterating the list

We can now iterate the list by going through the linked list of the items and using CONTAINER_OF to get pointer to our enclosing object from the pointer to the list item:

CONTAINER_OF

```
sys_snode_t *node;
SYS_SLIST_FOR_EACH_NODE(&list, node){
    struct my_object *self = CONTAINER_OF(node, struct my_object, node);
    ...
}
```

The above code presupposes that all list items in the list are of type "struct my_object".

Multiple Inheritance

There is nothing stopping us from embedding multiple different types into our structure and this has the effect of simple multiple inheritance in C.

```
struct base_one {
    uint32_t x;
};

struct base_two {
    uint32_t x;
};

void base_one_init(struct base_one *self){
    memset(self, 0, sizeof(*self));
}

void base_one_do_something(struct base_one *self){
    printk("%s\n", __func__);
}

void base_two_init(struct base_two *self){
    memset(self, 0, sizeof(*self));
}
```

```
struct derived {
    struct base_one base_one;
    struct base_two base_two;
};

void derived_init(struct derived *self){
    base_one_init(&self->base_one);
    base_two_init(&self->base_two);
```

```
}

void derived_do_something(struct derived *self){
    printk("%s\n", __func__);
    base_one_do_something(&self->base_one);
}

void main(void)
{
    struct derived derived;

    derived_init(&derived);
    derived_do_something(&derived);
}
```

Note that there is a lot of similarity with this approach in C compared to the same approach in for example C. In fact, C often just adds syntax features that make this approach simpler to implement. At the core, it is the philosophy that matters and not language features.

Code in C++

```
Base::do_something();
```

Simply becomes the following code in C:

```
base_do_something(&self->base);
```

Traits And Behaviors

In C++ we usually refer to overridden functions as "virtual functions" and to classes that only outline virtual functions with NULL implementations as "abstract interfaces".

In practice, what we are actually trying to implement is "traits".

Traits is a newer concept and is used extensively in more modern languages like Rust. It is not the syntax that matters - what matters is the intent and purpose of why we event want traits in the first place.

We will look closer into abstract interfaces in the "Virtual API Pattern". For now let's just do a gentle introduction to traits.

A trait is defined as: a collection of behaviors that an object must implement in order to be treated as "an object that has trait X" where X is typically a class that outlines the definitions of these behaviors - as function pointers.

```
struct trait_x {
    void (*do_x)(struct trait_x *self);
};

struct trait_y {
    void (*do_y)(struct trait_y *self);
};

static inline void trait_x_do(struct trait_x *self){
```

```
    self->do_x(self);
}

static inline void trait_y_do(struct trait_y *self){
    self->do_y(self);
}
```

Deriving traits

We can now create an object that derives these traits:

```
struct derived_with_traits {
    struct trait_x trait_x;
    struct trait_y trait_y;
};

static void derived_with_traits_do_x(struct trait_x *trait){
    struct derived_with_traits *self = CONTAINER_OF(trait, struct
derived_with_traits, trait_x);
    printk("%p: %s\n", self, __func__);
}

static void derived_with_traits_do_y(struct trait_y *trait){
    struct derived_with_traits *self = CONTAINER_OF(trait, struct
derived_with_traits, trait_y);
    printk("%p: %s\n", self, __func__);
}

void derived_with_traits_init(struct derived_with_traits *self){
    self->trait_x = (struct trait_x) {
        .do_x = derived_with_traits_do_x
    };
    self->trait_y = (struct trait_y) {
        .do_y = derived_with_traits_do_y
    };
}
```

Using trait objects

We can then instantiate the derived objects as follows:

```
void main(void)
{
    struct derived_with_traits dwt;
    derived_with_traits_init(&dwt);

    // from now on we can treat them generically as either one of the traits!
    struct trait_x *dwx = &dwt.trait_x;
    struct trait_y *dwy = &dwt.trait_y;

    trait_x_do(dwx);
    trait_y_do(dwy);
}
```

Traits allow us to have multiple derived classes implementing different behaviors for generic operations. We can have as many "do_x" implementations for trait X as there are classes that derive that trait - yet we can treat all objects that implement "trait X" just the same.

We are going to cover a more generic implementation of traits as part of the 'Virtual API Pattern'.

Best Practices

- **Always use Object Pattern**: this is a major prerequisite for being able to further clean up your architecture using inheritance. If your application is not currently using object pattern for organizing all variables then you will have a hard time implementing clean inheritance as well.
- **Understand the role of inheritance**: the primary role of inheritance is code reuse. The secondary role is to keep your functionality hierarchical so that it is easier to maintain.
- **Use delegator methods**: when you need to access some data or functionality in the base class and this property can be considered a property of the derived class as well, create a derived class method and then simply delegate to the base class.
- **Trait callbacks always static**: always implement trait callbacks as static functions inside the implementation C file. This ensures that you will not forget to update them when you make changes to your implementation.

Common Pitfalls

- **Fragile base class**: if changes to the base class alter the behavior of the derived class then it becomes specially important that you thoroughly verify the expectations you place on the derived class.
- **Code duplication**: if you have many base class methods and you then implement lots of wrappers in the derived class, you end up with many functions that just forward the call to another method of the derived class. This is essentially dead weight.
- **Trying to implement inheritance before Object Pattern**: this is worth repeating multiple times: use the Object Pattern as much as possible and it is likely that the proper inheritance relationships will come into view naturally.

Alternatives

- **Improving the base class**: sometimes it is better to simply add additional functionality to the base class rather than try to create a derived class. Inheritance in C should be used primarily in cases where you would like to combine several functionalitities in a higher level object.
- **Virtual API Pattern**: this pattern deals with fully abstract interfaces. The difference is that an abstract interface is a shared data structure that is always constant and shared between all instances of an object while a trait is a structure contained in each instance where the function pointers are not necessarily always constant. Virtual API is a more robust and memory efficient way to implement pure virtual functions in C.
- **Facade Pattern**: Since C language doesn't naturally provide a syntax for separating

inheritance from composition, many patterns look similar in their implementation. A pattern where an object inherits functions and then delegates the calls to a base class may look very similar to a facade. The difference is in the intent (the intent of a Facade is to simplify the interface which is not the same as the intent of direct delegation).

Conclusion

Hopefully this module has given you a better understanding of inheritance in C. I have not covered all nuances of this pattern here and it seems to me that there is a lot more to be said about it. I may update this module in the future as I actively seek out more example to include here.

This module paves the way for understanding more complex patterns that you can use in your C programming to create a clean and maintainable software architecture.

Quiz

- What are the main reasons for using inheritance in your software architecture?
- How is multiple inheritance pattern implemented in C and what naming convention is it good to use when naming your methods to make this easier to maintain?
- What are the benefits of using traits and how do they relate to multiple inheritance?
- What main drawback do traits have when they are embedded into the derived struct compared to pure virtual interfaces which are constant and only referenced from the derived struct through a pointer to the interface?