Opaque Pattern

Opaque pattern is an extension of the object pattern where we make the object data structure fully private and not visible outside of the implementation.

There are multiple ways we can hide implementation and in this module we will look at how we can do it.

Defining Characteristics

Some key identifying features of the opaque pattern include:

- **Object definition is in the C file**: The struct itself is defined in the C file implementing the object instead of being defined in the header file.
- **Implementation must expose size**: we need this for allocating instances of our object. Implementation **can** also handle allocation but does not have to.
- **Implementation uses object pattern internally**: All functions can still take 'self' pointer instance as parameter - but the caller is no longer responsible for allocating the instance. If this feature is not present then we are likely dealing with the singleton pattern.
- **Uses 'new' and 'delete' idiom**: this is necessary to make a distinction between opaque object and fully defined objects and to make it clear to the caller that the caller must call delete when done with the object. We use this for dynamic allocation as I'll explain later.
- **Application (user) only deals with pointers**: since implementation is fully hidden, we only deal with opaque handles (pointer based) outside of the implementation.

Use Cases

This pattern is a useful extension to the object pattern in cases where we want to introduce a dependency barrier and ensure that we keep dependencies local to the implementation.

In such scenarios we must hide the struct definition from the user of the structure because the struct definition often contains implementation specific data structures and needs to include implementation specific headers which then also get included in code that uses the header.

- **Isolating dependencies**: We want to keep dependencies local to the implementation.
- **Prevent direct data access by user**: since the object fields are private to the implementation, they can not be accessed, changed or seen by any other code outside of the implementation.

Implementation

The biggest difference between an opaque object and the traditional object pattern with a publicly defined struct is that with the opaque pattern we must also do memory allocation.

Since the size of the opaque struct is not defined outside of the implementation C file, we need to have a way of getting this size and then we have to allocate the object either on the heap or on the stack. The third way to allocate the object is statically inside the implementation file either using code generation or as a static array from which new objects are allocated and freed.

In the case of dynamic allocation by the caller, we will need to add a method to our object to retrieve the size of the private data structure using a method call.

The public interface for the opaque object is defined in a very similar way to the object pattern, with the main difference that the structure itself is only declared in the public header (without the data fields) and then actually declared in the private C file.

File: opaque.h - public interface

```
struct opaque; // just a declaration

// init and deinit
int opaque_init(struct opaque *self);
int opaque_deinit(struct opaque *self);

// methods that operate on an opaque
void opaque_set_data(struct opaque *self, uint32_t data);
uint32_t opaque_get_data(struct opaque *self);
```

Note that any module that uses an opaque only needs to know about the forward declaration of the opaque struct - we don't need to expose the definition of the opaque struct to the rest of the application. This is the primary way in which we can hide implementation in C while still being able to use pointers to the opaque struct in the rest of our code.

File: opaque.c - private implementation

```
// actual definition of the struct in private space of the c file
struct opaque {
    uint32_t data;
};
int opaque_init(struct opaque *self){
    memset(self, 0, sizeof(*self));
    // do other initialization
    return 0;
}
int opaque_deinit(struct opaque *self){
    // free any internal resources and return to known state
    self->data = 0;
    return 0;
}
void opaque_set_data(struct opaque *self, uint32_t data){
    self->data = data;
}

uint32_t opaque_get_data(struct opaque *self) {
    return self->data;
}
```

Allocation Schemes

We can not simply do the standard stack/static allocation by the caller because we don't know the size of the struct at compile time:

```
void some_caller(...){
    struct opaque obj; // this will not work
}
```

Therefore, we need to implement an allocation scheme.

You have three different allocation schemes to choose from:

- **Stack allocation**: This scheme uses standard dynamic allocation on the stack using **`alloca`** method. **`alloca`** is like malloc, except that the allocation happens on the stack and is automatically released when the function returns. This method doesn't suffer from memory fragmentation like **`malloc`** below.
- **Dynamic allocation**: here we have to use **`malloc`** or an RTOS alternative. If you use this method on resource constrained devices then make sure that you either only allocate during initialization or that you do not use this method at all due to the risk of fragmenting the memory and reaching the point where no new instances can be allocated because of that.
- **Static allocation**: this method uses code generation or a static array. Code generation is preferable but does require that you instantiate your objects using a data representation such as using the device tree which is then parsed during build and from which instances can be created at compile time.

Stack Allocation

The stack allocation scheme uses standard **`alloca`** function in the C library to allocate the opaque structure on the stack:

To do this, all we need to do is add a **`size`** method to our opaque:

```
size_t opaque_size(void){
    return sizeof(struct opaque);
}
```

We can now allocate the structure on the stack and the compiler will actually throw an error (**`-Werror=return-local-addr`**) if we try to return this pointer from a function:

```
// create an opaque type on the stack
struct opaque *obj = alloca(opaque_size());
// standard init
opaque_init(obj);
// operate on the opaque
opaque_set_data(obj, 123);
// done with the object: deinit it
opaque_deinit(obj);
```

Note that this works **exactly the same** as allocating the variable on the stack. The only difference is that the size is now retrieved from a function.

Just like with normal stack variables, the compiler will not warn you if your stack space is too small and overflows during runtime. You still have to make sure your stack is sufficiently large that the variable will fit.

Dynamic allocation

The second approach is dynamic allocation. This uses **`malloc`**/**`free`**. Note that by default, on embedded systems malloc is often disabled and simply returns NULL every time. This is because if you use malloc, you must make sure that all your objects have been allocated at startup before your application init process is completed. Otherwise you may run into the situation where you suddenly run out of heap and you definitely do not want to do that at a later point.

To implement dynamic allocation, we simply implement **`new`**/**`free`**. You can also directly call malloc instead of alloca, like in the example above, but I prefer **`new`**/**`free`** just to make it clear that this is a heap allocation and that the object must be freed with the corresponding free call.

File: opaque.c - allocation (dynamic)

```
struct opaque *opaque_new(){
    // dynamically allocate an instance
    return malloc(sizeof(struct opaque));
}
void opaque_free(struct opaque **self){
    // free dynamically allocated instance
    free(*self);
    // Set the passed pointer to NULL!
    *self = NULL;
}
```

Usage:

```
// allocate new opaque on the heap
struct opaque *obj = opaque_new();
__ASSERT(obj, "Memory allocation failed!");
// call standard init
opaque_init(obj);
// operate on the opaque
opaque_set_data(obj, 456);
// deinit the object
opaque_deinit(obj);
// discard memory
opaque_free(&obj);
```

Static Allocation: Zephyr Driver Model

The Zephyr driver model uses static allocation which is linked into a list of devices. Under this model, all allocation is done at compile time using device tree as source compiled into preprocessor directives which can then be used to instantiate exactly the instances that are defined in the device tree.

The exact implementation of this scheme is outside of the scope of this module but you can explore Zephyr source code for more details on this.

The most important feature to note is that zephyr uses opaque objects for all devices which

allows things like stm32 implementation details to be kept fully private and not in conflict with any other include files - and also not accessible to the application as part of the public interface of the SDK.

You can find the usage of this approach in almost every device driver that is part of the Zephyr RTOS:

```
#define PWM_DEVICE_INIT(index)                                          \
    static struct pwm_stm32_data pwm_stm32_data_##index;                \
                                                                        \
    static const struct soc_gpio_pinctrl pwm_pins_##index[] =           \
        ST_STM32_DT_INST_PINCTRL(index, 0);                             \
                                                                        \
    static const struct pwm_stm32_config pwm_stm32_config_##index = {   \
        .timer = (TIM_TypeDef *)DT_REG_ADDR(                            \
            DT_PARENT(DT_DRV_INST(index))),                             \
        .prescaler = DT_INST_PROP(index, st_prescaler),                 \
        .pclken = DT_INST_CLK(index, timer),                            \
        .pinctrl = pwm_pins_##index,                                    \
        .pinctrl_len = ARRAY_SIZE(pwm_pins_##index),                    \
    };                                                                  \
                                                                        \
    DEVICE_DT_INST_DEFINE(index, &pwm_stm32_init, NULL,                 \
                &pwm_stm32_data_##index,                                \
                &pwm_stm32_config_##index, POST_KERNEL,                 \
                CONFIG_KERNEL_INIT_PRIORITY_DEVICE,                     \
                &pwm_stm32_driver_api);

DT_INST_FOREACH_STATUS_OKAY(PWM_DEVICE_INIT)
```

The above code will create exactly as many instances as needed **at compile time** giving us full knowledge about our memory usage while still automating the instantiation through a combination of the device tree and a build system that supports parsing the device tree and generating preprocessor directives that can then be used for instantiation.

Zephyr also has the advantage of having a lot of custom linker scripts that enable much of the initialization to happen completely automatically by the system code before even calling the application **`main`** function.

Benefits

- **Hides implementation**: removes the need of the code that uses the object to include dependencies upon which the module depends.
- **Limits dependencies**: the components and libraries upon the object depends no longer 'leak' into the code that uses the object - not even through header files.

Drawbacks

- **Requires allocation scheme**: this pattern must either use **`alloca()`**, **`malloc()`** or implement a custom static allocation scheme for creating new objects - which increases complexity.

- **Prevents data structuring**: with object pattern you would normally instantiate each object as part of a clean data hierarchy with 'application' struct being the top level enclosing data structure. Since opaque pattern actively prevents instantiation of object on the stack, it also prevents objects from being instantiated inside that data hierarchy.

The fact that opaque pattern requires an allocation scheme is the main disadvantage of the opaque pattern. Since we do not know the size of the object, we can not allocate it and initialize it in the same way that we usually do it with the object pattern. Instead, the object either needs to be allocated on the heap, or it needs to be allocated internally as part of static allocation scheme within the implementation of the opaque object.

Best Practices

- **Use stack allocation whenever possible**: this works exactly the same as working with object pattern and is the most lightweight approach.
- **Use the `new/delete` idiom**: if you do use heap, it's a good idea to make this clear, by adding the new/delete methods. Always check for failed allocations.

Common Pitfalls

- **Running out of stack space**: if your objects are too big and you allocate them using the stack allocation scheme, then you can easily smash the stack. However, this is not specifically a problem with this pattern, but rather a general danger that is always present when you use stack allocated variables in general.
- **Memory fragmentation**: if you are using the dynamic memory allocation scheme and you frequently allocate/deallocate large numbers of objects of different sizes then you can end up running out of contiguous blocks of memory and your memory allocation will fail. To avoid this, if you have limited memory avoid **`malloc`** completely at runtime. If you really need to use it, use it only during initialization stage and do not reallocate anything at runtime.

Alternatives

- **Object Pattern**: The object pattern is a clear alternative, with slightly more simplicity - if you are ok with exposing all dependencies to any other code that includes your header.
- **Singleton Pattern**: The singleton pattern is essentially the opaque pattern - but with the main difference that not even the pointer to the context is passed around. Singleton pattern should be mainly limited to implementation of software wide subsystems and services where instantiation of multiple instances does not make sense at all. Note however that even a singleton object should internally use the object pattern to increase quality of code organization and ease of testing.
- **Abstract API Pattern**: An abstract interface by definition acts as an opaque object. It is a structure that holds only function pointers and each function is then able to retrieve pointer to the implementation specific data using **`CONTAINER_OF`** macro. Abstract interfaces are useful when we want to generalize over a class of objects - and we then by definition make all of our objects opaque. This pattern is more heavyweight than simple

opaque object.

Conclusion

In this module we have looked at the Opaque Pattern which is primarily used for hiding internals of a data structure from the code that uses that data structure.

The opaque pattern is an extension of the object pattern but keeps data definition inside the C file.

To make this work, the opaque pattern must provide an allocation scheme for creating new objects. This allocation may either use stack allocation like the object pattern, dynamic heap memory or static allocation if the number of objects to be created is known at compile time.

Quiz

- What are some of the reasons why we would want to hide the data structure of an object outside of its implementation?
- Why does the opaque pattern require a custom allocation scheme?
- How does using the opaque pattern affect the structure of the data of the application? How does it change the way we structure application data?
- Why is it a good idea to automate the allocation of objects at compile time?