Object Pattern

The most basic and also perhaps the most useful design pattern in C is the object pattern. In this section I will show you how to write code that is scalable, easy to maintain, easy to understand and easy to extend as your project evolves.

In the Zephyr project this pattern is used extensively. This pattern is also often referenced as the "object oriented programming" pattern even though object oriented programming is a much bigger topic involving other patterns as well.

Definition

The object pattern groups data into a hierarchy of data structures and directs functions to operate on these data structures.

The key aspect of the object pattern is that it makes data flow along the code path instead of being referenced externally.

This has a tremendous effect on cleanliness of the whole software architecture because it gives the programmer a clear view into what data a function will modify directly.

- **Context is passed as parameter**: we use `self` parameter to access all data that belongs to the object upon which a function operates.
- **Data is never accessed globally**: any global or singleton data is never accessed directly but instead is accessed through singleton methods (see singleton pattern for more information on how this is implemented).
- **Functions do not have static data**: all data is part of the object being operated on. There is no static data in object methods. Data is either part of the object.
- **Data flows along call path**: this is an important feature that ensures we have clear boundaries between methods that use particular data. This is incredibly useful for multi-threaded environments because it makes data locking very straightforward when needed.

Use Cases

The object pattern should be your default way of implementing components in a C program. A lot of the functionality that you will ever be implementing can be classified as "functions operating on objects". Whenever you implement some functionality, try to determine "which object is this functionality going to operate on?" and then place the new function as an extension of that particular object.

The object pattern is by far the most useful of all patterns.

- **Grouping data**: you should group all variables into structs (objects) and use the object pattern as means of sorting your data hierarchically.
- **Singletons**: object pattern is the primary way to implement singletons as well because all data that was previously statically defined inside the singleton implementation can now be placed into a singleton object and all private singleton object methods can be made to operate on that object.
- **Abstract interfaces**: object pattern is key component of abstract interfaces.

- **Multi-threaded design**: object pattern is essential for multi-threaded design because thread synchronization is about "locking data - not code" and objects are essential for grouping data together so that we can have one clear lock for a group of variables we need to synchronize access to.
- **Opaque handles**: a pointer to a data structure can be exposed to the outside without exposing the data structure itself. This allows for efficient way of implementing opaque handles with custom internal allocation. Object pattern is needed because all data belonging to a handle must be separate from all other handles.

Benefits

The object is the primary tool in your programming toolbox for enabling you to untangle your source code written in C.

- **Clear Scope**: Fine grained control over data scope thus minimizing unintentional data manipulation.
- **Reentrancy**: Ensures functions are re-entrant (no globally manipulated state).
- **Easy locking**: Simplifies multi-threaded programming because you can easily locate relevant data.
- **Simplified testing**: Simplicity of testing because code can be easily compiled in isolation and fed with mock data.
- **Clear data flow**: Data flow is always through the code and not outside of it. This simplifies debugging and makes the code easier to visualize when reading it without even running or testing it.

Drawbacks

- **No ability to hide implementation**: In it's basic form (with struct declared in the header file) private fields are exposed. This adds dependencies to the code using our struct. We can only hide implementation by using an extension of this pattern such as the singleton pattern or heap object pattern.
- **Can result in high memory consumption**: sometimes it is necessary to declare static structures inside a C file - for example when you need to share some data between all instances of a particular object. This warrants occasional mix of the object pattern with singleton pattern internally where a part of the object implementation is in fact a singleton.

Implementation

The primary way in which we implement object pattern in C is by grouping all of our variables into object structures and then writing methods that operate on the object structures which are passed as a parameter to the methods (the 'self' pointer).

Let's define our object first:

File: my_object.h - definition

```
struct my_object {
    uint32_t variable;
    uint32_t flags;
};

int my_object_init(struct my_object *self);
int my_object_deinit(struct my_object *self);
```

We can now implement these methods in the C file:

File: my_object.c - implementation

```
#include "my_object.h"

int my_object_init(struct my_object *self){
    memset(self, 0, sizeof(*self));
}

int my_object_deinit(struct my_object *self){
    // cleanup
}
```

The key in object pattern is that the user is responsible for allocation of objects. User must either allocate the object on stack of the main thread or as part of another object which will be using our object:

File: application.c - usage

```
#include "my_object.h"

struct application {
    struct my_object obj;
}

int application_init(struct application *self){
    my_object_init(&self->obj);
}
```

A few implementation rules we must apply at all times:

- **Functions 'act' on objects**: Each function needs to act on an object pointed to by a "self" argument. Any additional parameters passed to the function are there to modify the behavior being done upon the 'self' object. The 'self' object is the primary object being changed and also the place where results are accumulated. Any output variables can of course also be passed as arguments and such data is considered 'exported' from the object and becomes the responsibility of the caller once the method returns.
- **Functions are prefixed with object name**: Each function that operates on an object should be prefixed with the type name of that object and also placed in a file with the same name. This is for clean organization and clarity when reading the code.

The object pattern also has another important property. It allows us to make sure that all of our functions are reentrant. To be reentrant, a function must adhere to following rules:

- **No global static data access**: It may not use global and static data - all data that it uses must come from the arguments (ie you can not access static data inside the function but the data you pass to the function can of course be static - there are only restrictions in what you get to access inside the function)
- It should not modify its own code (this one is easy to adhere to in modern software). So this is not even applicable to C programming (but is still a rule that must be adhered to in order for a function to be reentrant).
- It should not call other non-reentrant functions. The best way to ensure this is to make sure we always pass context to all methods that we call - making sure that we apply object pattern throughout the application.

What we also want to have is clear data flows. The data should not flow outside of the call tree. This must be adhered to as much as possible.

If data flows outside of the call tree, the code becomes difficult to manage because there is no way to know who and when modifies that data. Object pattern does try to solve this fact by limiting the data that can be modified only to the 'self' pointer, output parameters and data modified through method calls to other objects.

It is important that data is modified through a call and never globally because a call will always need to be mocked in a unit test - giving us a clear indication that we are modifying data outside of our object. This will make your code much easier to debug.

As a general rule when implementing the object pattern we don't alter structure members directly from outside of the methods that are designed to operate on that struct (ie accept is as 'self' pointer). So anything done with variable `var` of type `struct foo` should use foo_* set of functions.

Many ancient C projects used data declared in global scope and then used "extern" to reference that data from many different places. This is the antithesis to well defined data flow. We want to have the opposite - the data flows as close to the call tree as possible. When you look at a function it should be clear where the data is coming from (one of its parameters) and where it is going (one of parameters of functions it is calling) this way it is very easy to protect this data from races and uncontrolled changes. This is not limited to a particular part of the application - it is very valuable in all parts of the application.

Best Practices

There are a few best practices to be on a lookout for when implementing this pattern:

- **Avoid static**: you should avoid static variables inside functions entirely because they break the object oriented design that the object pattern is designed to solve. They make your functions depend on more data than what is directly available through the `self` pointer.
- **Use 'self'**: unless your object pointer is an interface handle (from which you would then retrieve a 'self' pointer) do not use any other names for the variable that designates the pointer to the main context you are operating on. Use 'self' because it is compatible with C++ compiler and you remove ambiguity when you always use the same name to refer to

'self' (do not use 'me', 'dev', 'obj' or some other name - self is a standard that has become widespread even in python and rust).

- **Use consistent naming**: The main struct should have the same name as the header file it is declared in (ie for struct my_object the header should be called my_object.h and the implementation should be in my_object.c). All methods that operate on instances of the struct have the same prefix that is also the same as the struct (ie for struct my_object all methods are called my_object_something).
- **Standardize init/deinit**: There should be two standard functions: my_object_init and my_object_deinit that initialize a new instance and deinitialize it. When user instantiates an object, he should always call <object>_init() and this init function must at the very least always clear the memory of the object to zero (this is not done automatically when we are allocating a stack variable!).

Pitfalls

The most serious pitfall when implementing this pattern is that you think you can get away without using the "self" pointer.

```
// instead of doing this:
int my_object_do_something(struct my_object *self, int arg){
    self->some_var = arg;
}
// you are doing this:
static struct my_object _self;
int my_object_do_something(int arg){
    _self.some_var = arg;
}
```

Basically the pitfall is in trying to "optimize" the implementation to not include "self" as part of the argument list and instead have an instance of the object instantiated statically inside the file.

The biggest argument for this is: why require that the caller create and maintain an instance of an object when there can only be one instance afterwards and then having to pass this instance around?

The simple answer to this is that having an instance causes us to impose a structural policy on our application where my_object_do_something (which is a public method) can not just be called anywhere in the code - it requires an instance.

This requirement for an instance means that the caller needs to get that instance somewhere. If getting that instance is tricky then it serves as an indication to the programmer that he should not touch that data directly. Without the requirement of passing context along the call graph, we have no way to enforce data structure of the application.

Not passing a 'self' pointer also comes with other negative side effects:

- **Reduced clarity**: It is no longer clear whether this method operates on data aggregated under "struct my_object" or not. You lose this clarity.
- **More difficult to test**: It is bad for testability because the caller is no longer responsible

for the memory and we can not easily create multiple instances or inspect the internals of the object when testing.

Another common pitfall is the use of 'extern' variables.

This is code that looks like this:

Bad header file

```
extern uint32_t value;
```

Bad C file

```
#include <bad_header.h>
void your_method(struct your *self){
    value = 123;
}
```

This directly breaks data flow by making value of '123' flow directly into some other C file where the variable is actually defined. We have no way of observing this flow and so you should never do something like that. A better approach would be to either pass a pointer to another object that defines that variable when you are initializing your object or to restructure your source code in such a way that extern can be completely removed (you should ban the use of 'extern' variables using C scripts).

The simple guideline to follow is: **always pass all context on which your function should operate as part of arguments passed to that function.** Never access this context directly - specially if it is defined in another C file.

Follow this simple rule and you will have an architecture that enforces loose coupling by the very nature of its structure.

Alternatives

- **Opaque Pattern**: this pattern hides the implementation of the object completely by taking on the responsibility for allocating and de-allocating instances of the object. The only structure visible to the outside becomes a pointer to an instance without exposing the internals of an object instance outside of its implementation.
- **Singleton Pattern**: this is another variation of the object pattern where a subsystem may want to keep the instance of the object entirely private and only expose a global interface. This pattern is useful for services that are shared across the whole application such as logging, networking stack etc.

Conclusion

In this module we have covered the importance of the object pattern. I hope you can appreciate its simplicity and start using it to tremendously improve the organization of your source code.

Quiz

- Why is it so important to avoid static variables inside functions in C? Specially if the function is an object method?
- Why do we avoid functions without parameters? What negative property do these functions possess that make them a very bad design flaw in C source code?
- Why do we call our pointer to context 'self' and why should we avoid using other names to refer to 'self'?
- Why is it sometimes necessary to instantiate objects locally in the C file as singletons?
- Why is it sometimes necessary to only expose a pointer to the data structure outside of the implementing C file?
- Why is it a good practice to always name the header and the C file with the same name as the data object they implement?
- Why should you never use 'extern' declared variables anywhere in you C code?