Factory Pattern

The factory design pattern is a creational design pattern that provides an interface for creating objects in a super class, but allows sub-classes to alter the type of objects that will be created.

In practice (and in embedded firmware development in C) this translates into code that parameterizes the creation of objects. By now you should be familiar with the Object Pattern and factory pattern takes this to a new level.

Defining characteristics

There are several key elements to the factory design pattern:

- **Object being created**: In C, this is usually a struct which is instantiated either statically or dynamically.
- **Location of the memory**: The factory manages the memory pool and creates objects as needed. When an object is no longer needed, it is the factory that is responsible for calling the cleanup methods and placing the de-allocated object back into the pool of available objects.
- **Parameters**: factory typically creates an object based on a structure containing desired **`properties`** of the object that should be created. It does not necessarily need to return the concrete type. In the case of device drivers, all drivers are accessed through a generic **`struct device`** pointer and device drivers are "created" by a factory from device tree information.
- **Concrete implementation**: the factory implementation typically decides which object to instantiate depending on parameters given. This is the key to the flexibility of the factory pattern - it decouples implementation details from the user of the object.
- **Client or user**: the user of the factory is typically the application level code. However, creation of objects can also be completely hidden from the user and automatically instantiated at program startup (in Zephyr this is done by registering a SYS_INIT callback which is executed by the startup code).

The factory design pattern is useful for creating objects in a flexible and modular way, as it allows for the creation of different types of objects without the client having to know the specific implementation details. It also allows for easy extensibility, as new types of objects can be added without having to modify application code.

Use cases for factory pattern

In contrast with fully object oriented languages like **`C++`**, in C the use of this pattern may not seem obvious at first.

In Zephyr, you will find this pattern being used to:

- **Instantiate device drivers**: where neither our application, nor other device drivers need to know how to actually do this instantiation. We can apply the factory pattern to hide these details and make them appear automatic.
- **Memory pools and buffers**: user can define a memory pool and then create objects inside that statically allocated pool. Factory pattern is implemented using mostly macros for instantiating and accessing the pool.

- **Network device creation**: network interfaces are not defined in the device tree, but rather created and managed by the networking subsystem. A networking device is created using macros just like with memory pools and conventional device drivers - and then accessed through a `struct net_if` object.
- **File system objects**: another example of factory pattern usage in system development is the file system objects. In such a scenario, the file system implementation knows the details of what file system object should contain and so it is also responsible for creating the specific file object and returning a generic handle to it. Once again, specifics of creating the object are hidden from the user of the object.
- **Network packet creation**: a network packet is usually sent to a network interface as a buffer, but the creation of this buffer is the responsibility of multiple protocol implementations layered on top of one another. Thus the packet buffer is created in steps and then once it is ready it contains the fully constructed packet which can be sent over a physical interface.

Whenever you need to manage creation of objects and their lifetime in one place, consider applying the factory pattern to make your life easier.

Pros

- **Increased flexibility**: The factory design pattern allows for the creation of objects to be abstracted away from the application, allowing for the creation of new objects without the need to modify the user code.
- **Improved code organization**: The factory design pattern helps to separate the creation of objects from their implementation, making the code easier to read and understand.
- **Data driven architecture**: The most notable use of factory pattern is the case where an object is created based on a structured description (in JSON form, as a conventional struct or as a device tree node). This is where the factory pattern truly shines.
- **Improved testability**: When the developer needs to actively implement creation and cleanup of objects it forces the developer to think about better design. In particular it encourages creation of objects to happen in a standardized way - making it also easy to create mock-ups of the object for testing.
- **Improved code reuse**: By making creation of an object into a reusable function, this process can be repeated in multiple places in the firmware without polluting code with duplicated creation logic (while this is incredibly obvious, you would be surprised how little this principle is used in production code).
- **Improved scalability**: When creation of objects is localized and parameterized, it becomes easy to add many different kinds of objects and extend the system without having to alter existing code that creates instances of these objects. This is particularly clear in the case of device drivers and device tree.

Cons

- **Added complexity**: It may not always be necessary to have a factory. Sometimes a simple conventional `my_object_init()` method is enough. Factory pattern should be used when creation, management and cleanup of objects needs to happen in one place.

- **Lack of flexibility**: The factory pattern should not use hard-coded parameters. It is the most flexible when it is fully data driven - i.e. you have a piece of code that is able to produce a variety of objects based on a data representation.
- **Testing difficulties**: Sometimes this pattern can make the code more difficult to test. For example, Zephyr device drivers that instantiate device driver objects through a separate section in the executable file need to have the device creation macros redefined to do nothing in the test to avoid the device driver being instantiated in the test before the test code is able to create all the mocks.

Most of the time these cons are non-issues because the factory pattern, when used in the right context, is the most logical solution to the problem of object creation.

Implementation

There are several ways in which we can implement the factory pattern. We will cover the most relevant styles of implementation that are suitable particularly for embedded firmware development in C without access to native object oriented language constructs that are for instance available in `C++`.

- **Simple factory**: In this implementation, a factory class is created that has a static method for creating objects of different classes. The factory class determines which class to instantiate based on the input provided by the user.
- **Embedded factory**: object creation is done through lists of constructors placed in special sections of the executable itself.
- **Abstract factory**: In this implementation, a factory class is created that has methods for creating groups of related objects. The factory class has sub-classes that override these methods to create specific groups of objects.
- **Prototype factory**: In this implementation, the factory class has a prototype object that it clones to create new objects. The user can then request new objects by specifying the prototype object to clone.
- **Pooled factory**: In this implementation, the factory class has a static method for creating objects of a specific class, and uses a pre-allocated static pool for allocating objects. The user can then request an instance by specifying the key for the desired object (this is used in Zephyr for example for managing bt_conn objects inside the bluetooth stack).

Simple factory

To implement the simple factory we will use the abstract object pattern and parameterize our factory using a simple enum. We will completely hide the object types we are creating and do the whole creation process entirely using the parameters passed to our factory method.

shape.h

```
// this is our public header
enum shape_type {
    SHAPE_CIRCLE,
    SHAPE_SQUARE
};
```

```
struct shape_api {
    void (*draw)(shape_t shape);
};

// define a generic shape handle
typedef struct shape_api ** shape_t;

struct shape {
    const struct shape_api * const api;
};

shape_t shape_create(enum shape_type type);
void shape_draw(shape_t shape);
```

shape.c: data structures

```
#include <shape.h>

struct shape_circle {
    struct shape shape;
    float radius;
};

struct shape_rectangle {
    struct shape shape;
    float width, height;
};
```

shape.c: circle api

```
static void _shape_circle_draw(shape_t shape){
    struct shape_circle *self = CONTAINER_OF(shape, struct shape_circle, shape.api);
    // now we can access private methods
    _do_draw_circle(self->radius);
}

const struct shape_api _circle_api = {
    .draw = _shape_circle_draw
};

static void shape_circle_init(struct shape_circle *self){
    memset(self, 0, sizeof(*self));
    self->api = &_circle_api;
}
```

shape.c: rectangle api

```
static void _shape_rectangle_draw(shape_t shape){
    struct shape_rectangle *self = CONTAINER_OF(shape, struct shape_rectangle,
shape.api);
    // now we can access private methods
    _do_draw_rectangle(self->width, self->height);
}

const struct shape_api _rectangle_api = {
    .draw = _shape_rectangle_draw
};

static void shape_rectangle_init(struct shape_rectangle *self){
    memset(self, 0, sizeof(*self));
    self->api = &_rectangle_api;
}
```

shape.c: factory methods

```
void shape_draw(shape_t shape){
    // call the virtual method
    (*shape)->draw(shape);
}

shape_t shape_create(enum shape_type type){
    switch(type){
        case SHAPE_CIRCLE: {
            struct shape_circle *shape = (struct shape_circle*)your_alloc();
            shape_circle_init(shape);
            return &shape->api;
        } break;
        case SHAPE_RECTANGLE: {
            struct shape_rectangle *shape = (struct shape_rectangle*)your_alloc();
            shape_rectangle_init(shape);
            return &shape->api;
        } break;
    }
    return NULL;
}
```

To use this factory we can now simply do the following:

application.c: usage

```
int main(void){
    shape_t rect = shape_create(SHAPE_RECTANGLE);
    shape_t circle = shape_create(SHAPE_CIRCLE);

    shape_draw(rectangle);
    shape_draw(circle);
}
```

The result is that we don’t need to worry about the creation of each object. The factory class handles this for us.

Embedded factory

This pattern is especially prevalent in embedded firmware development and it is built around defining C macros to instantiate variables inside specific sections of the executable file itself. When the executable is loaded into memory or executed from flash, these variables appear as a simple list of pointers inside that section and each pointer then points to a data structure which defines an object to be created.

We can then have init code that simply loops through these pointers and creates all of the objects from the data that has been automatically placed there by the compiler.

This method is most often used in systems development where we have full control over the code that is executed before "main". In fact, this method is always used in all C programs - just that it is implemented by compiler libraries. In such scenario, the C startup code executes constructors for static and global variables. From user perspective, this looks like pure magic..

uart_drier.c: factory pattern in action

```
DEVICE_DT_INST_DEFINE(index,                                \
            &uart_stm32_init,                       \
            NULL,                           \
            &uart_stm32_data_##index, &uart_stm32_cfg_##index,  \
            PRE_KERNEL_1, CONFIG_SERIAL_INIT_PRIORITY,      \
            &uart_stm32_driver_api);                \
                                    \
STM32_UART_IRQ_HANDLER(index)

DT_INST_FOREACH_STATUS_OKAY(STM32_UART_INIT)
```

Abstract factory

We can also virtualize the factory itself. This is done by having a generic shape factory which in turn has multiple implementations (where each implementation will be responsible for creating a certain class of shapes). Our resulting objects are all shapes - but we can now have multiple factories for creating these shapes.

shape_factory.h: abstract api

```
struct shape_factory {
    const struct shape_factory_api * const api;
};

shape_t shape_factory_create_shape(shape_factory_t factory, enum shape_type type);
```

shape_factory_yellow.h

```
struct shape_factory_yellow {
    struct shape_factory factory;
};

static inline shape_factory_t shape_factory_yellow_cast_to_factory(struct
shape_factory_yellow *self){
    return &self->factory.api;
}
void shape_factory_yellow_init(struct shape_factory_yellow *self);
```

shape_factory_yellow.c

```
shape_t _yellow_create(shape_factory_t factory, enum shape_type type){
    struct shape_factory_yellow *self = CONTAINER_OF(factory, struct
shape_factory_yellow, factory.api);
    switch(type){
        case SHAPE_CIRCLE: {
            return _create_yellow_circle();
        } break;
        case SHAPE_RECTANGLE: {
            return _create_yellow_rectangle();
        } break;
    }
    return NULL;
}

const struct shape_factory_api _yellow_api = {
    .create = _yellow_create
};

void shape_factory_yellow_init(struct shape_factory_yellow *self){
    memset(self, 0, sizeof(*self));
    self->api = &_yellow_api;
}
```

We can then create a multitude of other factories that create other kinds of shapes. The usage of the factory still remains the same:

```
void main(void){
    struct shape_factory_yellow yellow_fac;
    shape_factory_yellow_init(&yellow_fac);
    shape_factory_t fac = shape_factory_yellow_cast_to_factory(&yellow_fac);
    shape_t circle = shape_factory_create_shape(fac, SHAPE_CIRCLE);
    shape_draw(circle);
}
```

The implementation details are once again hidden from the user and the code can remain very clean. It doesn’t matter how many factories we have, the code that is designed to deal with shapes without needing to know what shape it is dealing with will still work exactly the same.

Prototype factory

This kind of factory is suitable for cases where we want to avoid constructing a complex object from disk each time. It is an optimization. One example is when we have cad software and it has a number of standard shapes that it is able to create in the viewport. We can load the models of these shapes ahead of time and then simply make new copies of them in the scene.

The prototype factory pattern works the same way.

However, it is important to note that to make it work we should have a virtual clone method as part of our shape API:

```
struct shape_api {
    void (*draw)(shape_t shape);
    void (*clone)(shape_t shape);
};
...
static shape_t _circle_prototype;
static shape_t _rectangle_prototype;
```

```
shape_t shape_create(enum shape_type type){
    switch(type){
        case SHAPE_CIRCLE: {
            return shape_clone(_circle_prototype);
        } break;
        case SHAPE_RECTANGLE: {
            return shape_clone(_rectangle_prototype);
        } break;
    }
    return NULL;
}
void shape_factory_init(void){
    _circle_prototype = _load_circle_from_disk();
    _rectangle_prototype = _load_rectangle_from_disk();
```

The implementation of virtual methods works according to the same principles as in simple factory. Here we are just copying an object instead of initializing it from scratch.

In C, this pattern is cumbersome and a little confusing. It simply has too much indirection to keep track of. Personally I rarely use this approach, but it is still good to know that this possibility exists and how to implement it.

Pooled factory

Memory allocation is an important question to consider. When we are working with Zephyr or microcontroller firmware in general, we don't want to use dynamic allocation. We can do just fine without it as well.

One approach is to use existing memory pools of objects. This can be as simple as having a fixed size number of instances while keeping track of slots that are in use.

To implement this, we can define a new structure that is a union of all available shapes:

```
union shape_slot {
    struct shape_circle circle;
    struct shape_rectangle rectangle;
};
struct shape_slot {
    bool empty;
    union shape_slot data;
};

static struct shape_slot _slots[CONFIG_MAX_SLOTS];
```

We can then proceed to allocate our slots as needed. The union is a way to make sure that we can keep all slots inside a single array. We simply define our **`struct shape_slot`** to be the size of the shape that has the largest structure. While this does waste some memory, it considerably simplifies our implementation. Provided that shapes are roughly the same size, the memory wasted inside the slots will be minimal.

Best practices

- **Clear interfaces**: Clearly define the interface for creating objects, as well as the specific types of objects that can be created. You should be naming your objects properly and structuring your code according to principles of good object oriented design.
- **Singleton factory**: Use a singleton factory class to ensure that only one instance of the factory exists at any given time. In C this is almost the default mode of operation for the simple factory. For abstract factories you may want to initialize the factories at bootup before entering `main`. You can accomplish this by using boot time initialization callback in Zephyr.
- **Parameterize your creation**: Think of the factory as a way of creating memory objects from an XML or JSON description. Your factory takes in parameters in a simple form and creates specific objects according to these parameters without exposing the object implementation to the user.
- **Utilize the abstract api pattern**: Use CONTAINER_OF and use the abstract api method. This keeps your code very type safe and robust at runtime. It is difficult to go wrong when you apply the api pattern.
- **Optimize when needed**: Use the prototype factory when a complex object takes longer time to create from scratch than to copy in memory.

Pitfalls of Factory Pattern

- Insufficient parametrization
- Too complex api for creating objects

The main pitfall of the factory pattern is if you have low degree of parametrization and end up simply calling the object `init` method from the factory method. This kind of scenario is simply making your code more complex without adding any real value to it.

Alternatives

- **Object Pattern**: sometimes a simple `my_object_init` constructor is sufficient. Most of the time you would be creating the objects directly - either in static memory space or inside a memory pool. You would use the standard object `init` pattern for constructor and `deinit` for cleanup. For most use cases this is sufficient and no factory is needed.
- **Singleton Pattern**: This design pattern uses an api of functions that do not take an object as parameter. This effectively helps one to hide the implementation inside the C file but unfortunately has the drawback of there only ever being one instance of the object to operate on (the static instance created in the C file).
- **Bridge Pattern**: This design pattern involves creating an abstraction layer between two different components, allowing them to interact without being tightly coupled. This method means that we 'extend' one object with additional methods defined in a separate C file exposing a separate API where the public headers of that api do not depend on the implementation of our specific objects being created. This means that the bridge api depends on accessing private data - but not our user code.

Conclusion

We have learned:

In this post we have covered the purpose and use cases of the factory design pattern. This pattern is useful when creation of objects based on a 'recipe' (parameters) is desired and where creation is best done in one place because the correctness of the creation process depends on the intricate combination of parameters.

We have also covered key elements of the factory design pattern and a number of practical implementation strategies in C that are suitable specifically for embedded software development.

Quiz

- What is the main purpose of the factory design pattern?
- What are the key elements of the factory design pattern?
- What are some benefits of using the factory design pattern?
- What are some examples of when the factory pattern is used in Zephyr? How are objects and devices initialized before main application is started?
- In what scenarios is the factory pattern useful for managing the creation and lifetime of objects?
- How can we easily have a pool of slots from which to allocate our objects and manage it in an object oriented (and type safe) manner?
- What are some of the pros and cons of the factory pattern?
- What are some of the alternative patterns that you can use when factory pattern may not be a good fit?

Why would you want to use an abstract factory?

What is the primary defining characteristic of the factory?

Why is the factory pattern a little harder to test?

What is the primary way in which the factory pattern helps you simplify your application code?