Virtual API Pattern

The virtual api pattern provides a way to implement abstract interfaces in a memory efficient and type safe way.

It differs from "traits" as described in "Inheritance Pattern" by being much more memory efficient without requiring multiple copies of the virtual function table to be stored in instances of an object.

Defining Characteristics

This pattern consists of following parts:

- **Constant trait objects and methods that operate on them**: these are our generic virtual functions which can be implemented in derived classes. These correspond to "Abstract Interface" in languages like C++.
- **Implementation of abstract interface**: any type that implements a particular abstract interface, always does so by implementing the virtual functions in the trait that these types inherit.
- **Traits are stored as pointers in derived class**: the trait object is always constant and is stored as a pointer. It is also passed as pointer to pointer type to all trait callbacks. This is mainly to make sure that we can scale to many instances without duplicating the function pointer table across all instances.

Use Cases

The main use case for this pattern is any situation where you want to treat a collection of implementations just the same in your application code without introducing every single type of implementation as a dependency to the code that uses it.

Essentially, you now depend on generic traits instead of on particular implementations.

- **Device Drivers**: this pattern is useful when you have many different implementations of essentially the same interface - such as for example a UART. Thus it is very often used for implementing many different device drivers that all implement the same API and to be able to treat all implementations just the same (polymorphism).
- **Plugins**: one very good example for this pattern is when you load a shared library and call some predefined function inside it which then returns an abstract interface. Plugins use this pattern a lot. This interface will continue to work even if the implementation changes since the trait definition is the only shared dependency.
- **Abstract data types**: any time we want to treat a collection of same or different objects just the same, we can use the abstract API pattern to implement an interface for interacting with these objects. Each implementation then needs to implement this interface in order to comply with the expectations of any code that would then use this object generically.

Benefits

Implementing the virtual api using a pointer to constant ops structure as we do it in this

module has the following main benefits:

- **Decoupling**: since the interface is very lightweight and consists only of function pointers, it almost completely decouples user from the implementation. If we then also use the opaque pattern, we achieve full interface decoupling.
- **Type safety**: one of the biggest benefits of implementing abstract interfaces as described in this training module is that it is completely type safe. There are no 'void*' pointers used and no casting that can potentially lead to incorrect results. Instead we use **`CONTAINER_OF`** and keep our callback data structure always constant from compile time.
- **Opaque handles**: the user deals only with opaque handles representing the abstract API and does not need to call contained functions directly. This means that the user can not accidentally modify the content of this handle because there is no visible data structure for it. The data structure is only visible to implementation of generic API and the multitude of implementations of the abstract interface.

Drawbacks

- **Complexity**: this pattern is more complex to implement than the traits pattern we looked at in the "Inheritance" module. It requires the programmer to follow a precise model of implementation without which the pattern would lack key functionality.
- **Inflexibility**: one of the key factors of this pattern is that the interface must be constant and relatively generic. If your objects are not sufficiently describable by a generic interface then it is possible that you may need multiple different interfaces in parallel.
- **Performance overhead**: although this overhead is very small, there is the overhead to always be calling the interface methods through two indirections. First one is when we call the generic wrapper method and the second indirection is when that method calls our implementation. However it is worth noting that this overhead is tiny compared to everything you are likely to do as part of the implementation method. Therefore it is usually a fair price for the improved flexibility.

Implementation

To better understand how this pattern works, let’s implement each one of the 3 components of it in sequence. Recall that one of the defining characteristics of the virtual API pattern is that it consists of three parts:

- **Generic API**: this part is exposed outside of your library. It does not include any data structures at all. All the member functions of this interface operate on pointer to pointer of the API data structure meaning that the data structure itself only needs to be declared but does not need to be defined in the public interface.
- **API interface**: this is similar to how we previously implemented traits in the chapter on inheritance. The difference is that the API data structure is only shared between the generic api and the implementations of this interface. Thus this is a "project private" api not exposed to the outside (because we want to actively prevent other code from changing it since it would break the pattern).

- **Implementation**: this is the concrete implementation. It only depends on the shared api and all other aspects of the implementation are hidden. You will have to use Opaque Pattern here or have some other way to register the abstract interface pointer so that the application can query it (for example by maintaining a list of all implementations indexed by a string or an enum - or through dynamic allocation).

Trait Definition

First we implement the api interface. This is a structure that contains only function pointers and follows the pattern of a "trait" as described in Inheritance Pattern section:

serial_ops.h

```
/** Serial API */
struct serial_ops {
    int (*write)(const struct serial_ops ** handle, const char *data, size_t size);
    int (*read)(const struct serial_ops ** handle, char *data, size_t size);
};

// You can typedef it like this if you want:
// typedef const struct serial_ops ** serial_t;
```

This header file is not part of the public interface of the library but rather is kept private and only shared with sources that implement this interface. The code that uses this interface will access it through a handle of type **`const struct <api-name> **`**.

Generic Trait API

This is a lightweight "delegator" API which connects the API to the implementation. It has access to the API structure definition and exposes a set of generic methods designed for operation on all instances of any object that implements this API:

serial.h

```
struct serial_ops;

int serial_write(const struct serial_ops ** handle, const char *data, size_t size);
int serial_read(const struct serial_ops ** handle, char *data, size_t size);
```

serial.c

```
int serial_write(const struct serial_ops ** handle, const char *data, size_t size){
    // here we DO have access to the api so we can use it directly
    return (*handle)->write(handle, data, size);
}
int serial_read(const struct serial_ops ** handle, char *data, size_t size){
    // here we DO have access to the api so we can use it directly
    return (*handle)->read(handle, data, size);
}
```

All other code in your software architecture that needs a serial device will now always be using this stable and public interface regardless of the actual implementation. This interface can be

documented and will provide a stable "contract" according to which all serial devices are accessed (or any other type of object).

Implementation header

```
struct serial_impl {
    const struct serial_ops *serial;
};

void serial_impl_init(struct serial_impl *self);
const struct serial_ops ** serial_impl_to_serial(struct serial_impl *self);
```

API Implementation

The implementation depends on the serial API and implements the functions defined in it. It can also provide a way to cast from a specific type to a serial interface handle (of type **`const struct serial_ops **`** in this case) but this is not necessary if your whole implementation is opaque because you will be creating this object internally and then only exporting the pointer to the interface.

```
#include "serial_ops.h"
#include "serial_impl.h"

static int serial_impl_write(const struct serial_ops ** handle, const char *data,
size_t size){
    // here we can resolve concrete type in the usual type safe way
    struct serial *self = CONTAINER_OF(handle, struct serial_impl, serial);
    // just print a simple message
    printk("write: %s", data);
    return size;
}

static int serial_impl_read(const struct serial_ops ** handle, char *data, size_t
size){
    // Similar read implementation here
    return -1;
}

static const struct serial_ops _ops = {
    .write = serial_impl_write
    .read = serial_impl_read
};

void serial_impl_init(struct serial_impl *self){
    memset(self, 0, sizeof(*self));
    self->serial = &_ops;
}

const struct serial_ops ** serial_impl_to_serial(struct serial_impl *self){
    return &self->serial;
}
```

Usage

It is now very easy to use this interface. If you are creating the implementation locally then you will of course need to initialize it first, but after that you will only be dealing with the handle:

```
// initialization of implementation (usually would be done separately from usage
below)
struct serial_impl serial;
serial_impl_init(&serial);

// get the handle that can be passed around and used with the generic api
const struct serial_ops ** handle = serial_impl_to_serial(&serial);

const char *data = "Hello World!\n";
serial_write(handle, data, strlen(data));
```

All delegation is handled internally by the lightweight public API we have defined.

Best Practices

- **Trait ops must be thoroughly defined**: every implementation must follow clear implementation guidelines and the behavior of each implementation should be the same.
- **Typedef the handle type for simplicity**: since the handle type actually declares a new type and is used repetitively in many places it is good to typedef it. You can make a typedef exception for this type and add it to your typedefs file (so checkpatch does not flag it).
- **Keep your traits constant**: this pattern relies on the interfaces rarely changing so you should try to think ahead when designing the interface and then keep it constant. Any changes to the expected behavior of the interface itself will likely require substantial changes to all implementations.

Common Pitfalls

- **Overusing abstract interfaces**: if everything in your software is done through an abstract interface, you will have interfaces that need to be updated often and updating the interface requires reworking parts of the implementations. So you should reserve use of abstract interfaces only for places where polymorphism makes sense (ie treating many items using the same set of actions).
- **Insufficient testing**: the full set of interface requirements must always be verified using unit tests for each implementation of that interface. Not doing this will lead to missed bugs that occur when you update the interface but fail to update all implementations.
- **Overly specific interfaces**: the interface should capture generic functions of a large category of objects. You should design your interface in such a way that you rarely experience situations where you need to write dummy methods for an interface because some operation of the interface does not make sense for your implementation.
- **Inconsistent implementation of virtual api pattern**: it is important that you apply this pattern consistently and always follow the same implementation approach.

This pattern in particular must always be implemented consistently because it often defines widely used APIs which are then implemented by many modules and if you later have to change the way you do abstract interfaces, you will be breaking a lot of code.

One example of this pitfall is the Zephyr project that has implemented devices in a way where there is no direct connection between handle and api (there is no way of knowing if a generic device handle implements a certain particular api). Changing this approach is not easy now because all of the Zephyr code base uses the old approach. So unfortunately we have to work with a type-unsafe **`struct device`** handles instead. Don't make this happen in your project.

Alternatives

- **Callback Pattern**: the callback pattern is a simpler version of the abstract interface where instead of defining a generic API we work with lists of callbacks.
- **Event Bus Pattern**: an event bus can implement a data driven abstract interface where both the function name and data can be encoded into an event and published on the event bus. This pattern is much more complex than the abstract interface pattern and often more suitable for many-to-many publish subscribe scenarios.

Conclusion

In this module I have shown you how to implement generic virtual APIs in C. This is a powerful pattern that allows you to separate your implementation from usage completely.

It keeps the code that calls the generic API very clean, consistent and straightforward and ensures that all access to specifics must be channeled through the generic API without ever accessing any private implementation specific methods.

Quiz

- How does the Virtual API pattern differ from the trait objects described in the Inheritance Pattern?
    1. It doesn't differ at all.
    2. Trait objects are i) not constant and ii) trait objects waste memory if stored for each instance.
    3. Virtual api is not the same as traits at all.
- How does defining our API handles as pointer-to-pointer help us stay memory efficient, particularly with large number of instances?
    1. Since virtual api does not change after compile time, it is sufficient to store only a pointer to our API in our implementation object.
    2. It makes it easier for us to retrieve pointer to the implementation object.
    3. It allows us to have individual api objects for each instance making it simple to keep the two connected in a type safe manner.
- Why is it so important that the virtual api does not contain any implementation specific details at all?
    1. Because it would duplicate these details between virtual api and implementation.

    2. Because it would require every single implementation to implement these details and if this does not make sense we would end up with a lot of useless code.
    3. Because we can not pass implementation specific details through a generic api.
- Why do we need to have a generic api along side of our trait objects?
    1. Because the generic api provides a clear set of actions that can be performed on pointers to the api objects. We can document this api and use it just like conventional functions instead of having to dereference pointers each time.
    2. Because we can not implement virtual functions without an additional api.
    3. Because the api does preliminary calculations before passing data to the implementation functions.