Factory method

Montag, 27. November 2023 14:48

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Introduction

I've encountered quite oft that people reuse one and the same Factory Method implementation - a textbook example introduced in *Effective Modern C++* by **Scott Meyers**.
And I want to say it right at the beginning - this is not how the Factory Method in reality supposed to be implemented, for several reasons.
The most obvious one - Factory Method shouldn't be aware of the types it fabricates.

The Factory Method is about:
- **Dependency injection**: at client call-spot will be decided which one from family of related types - or better to say, types that are not necessarily polymorphic by inheritance (*dynamic polymorphism*), but also non-related types with polymorphic behavior (*static polymorphism*) will be created.
- **Creational mechanism**: it's about having the centralized method - a gateway that implements the allocation policy that will be consistently applied for generating each and every type.
It's not necessarily allocation on the heap - the other allocation strategies can be used as well, depends on the use-case.

Implementation

Dynamic polymorphism

The widely used Scott Meyers textbook example is "switch/case" factory method that based on the given enum (or even string key) generates the type.
Not only that the method shouldn't be aware of the types it produces, it's also against the *Open-Closed Principle*, since adding a new type (through inheritance) result in modifying existing method.
And it's also error-prone.

Let's start with the Universal Factory Method, that uses default allocation policy - on heap

```
namespace details
{
    // Universal Factory Method
    // It will create any type of T on the heap: returning the std::unique_ptr so that
    // client code takes the ownership
    template <typename T, typename...Args>
    auto create(Args&&...args)
    {
        if constexpr(std::constructible_from<T, Args...>) {
          return std::make_unique<T>(std::forward<Args>(args)...);
        } else {
          return nullptr;
        }
    }
}
```

The simplified implementation could look like

```
namespace factory::dynamic
{
    using dynamic_types = enum class DynamicTypes
    {
        Derived1,
        Derived2,
        Derived3
    };

    struct Derived1 {};
    struct Derived2 {};
    struct Derived3 {};

    struct Base
    {
        // Factory method: has to be aware of all concreate - derived types
        template <typename T, typename...Args>
        requires std::is_base_of_v<Base, T>
        static auto create(dynamic_types type, Args&&...args)
        {
            using enum dynamic_types;
            // This violates the Open-Closed principle
            switch(type)
            {
```

```
                case Derived1 : return details::create<Derived1>(std::forward<Args>(args)...);
                case Derived2 : return details::create<Derived2>(std::forward<Args>(args)...);
                // It's also error-prone
                //case Derived3 : return details::create<Derived2>(std::forward<Args>(args)...);
                case Derived3 :return details::create<Derived3>(std::forward<Args>(args)...);
                default: break;// what if there is a new switch/type is required - and introduced in enum?
            }
            return nullptr;
        }
    };
}
```

We can improve this by using the **IOC (Inversion Of Control)** containers.
Briefly, it relies on the RTTI - *std::type_index* as a key within predefined *std::unordered_map*
that holds the factory methods as the matching values, that either return already created - shared instance, or
the creation strategy could be to generate a new instance of a type on each invocation.

More on that: https://github.com/damirlj/modern_cpp_tutorials#tut5

Static polymorphism

It's fair to assume that the family of types for which we want to specify the Factory Method is known upfront - at compile time.
Therefore, we can use one of the compile-time technics to implement the method itself.
One possibility would be to use CRTP pattern

```
namespace factory::crtp
{
    template <typename Derived>
    class Base
    {
        Base() = default;
        friend Derived;

        public:
               // Factory method: as a single gateway to produce any derived implementation of the
               // Base class interface.
               template <typename...Args>
               static auto create(Args&&...args)
               {
                   return details::create<Derived>(std::forward<Args>(args)...);// Derived class c-tor
                   // return Derived::create(std::forward<Args>(args)...);// Derived class own Factory Method

               }

               void doSomething()const
               {
                  impl().doSomethingImpl();
               }

         private:
              Derived& impl() {return static_cast<Derived&>(*this);}
             const Derived& impl() const {return static_cast<const Derived&>(*this);}
    };
}
```

Additionally, we have two options for the Factory Method - either to implement allocation policy internally, that will be uniformly applied
(as we should do), or to delegate this to Derived class dedicated Factory Method - forcing that each Derived class implements its own creational
mechanism (which is error-prone, since there is no guarantee on the same deallocation type attached to the returned type:
that the same allocation policy will be consistently used).

Concepts

Starting with C++20 we have a Concepts - as template parameter compile-time predicates: the constraints on the template
parameter substitution, that real types have to satisfy to be a valid candidates.
These (compound) constraints can actually specify the **interface** - behavioral aspect that each type has to match against.
This way, we can simplify our CRTP approach using the concepts instead

```
using mode_type = enum class Mode
{
    drive_forward,
    drive_backward,
    parked,
    neutral
};

// Vehicle interface
template <typename Vehicle>
concept is_vehicle = requires(Vehicle&v, const mode_type mode )
{
```

```
    {v.startEngine()} ->std::same_as<void>;
    {v.drive(mode)} ->std::same_as<void>;
    {v.breaking()} ->std::same_as<bool>;
    {v.stopEngine()} ->std::same_as<void>;
};
```

Now, we can specialize our universal factory method (on the **heap**)

```
// For vehicles tailored Factory Method as free function.
// It imposes restriction to the all vehicles that implement the very same interface: that are having the same
// behavior without being polymorphic by inheritance
template<is_vehicle V, typename...Args>
auto createVehicle(Args&&...args)
{
    return details::create<V>(std::forward<Args>(args)...);
}
```

Or, we can use the one that provides the allocation on the **stack**

```
namespace stack_alloc
{
    struct StackDeleter
    {
        // Call operator
        void operator()(auto* ptr) const {if(ptr)[[likely]]std::destroy_at(ptr);}
    };

    template <typename T>
    using unique_ptr = std::unique_ptr<T, StackDeleter>;

    template <typename T, typename Buffer, typename...Args>
    [[nodiscard]] auto create(Buffer& buffer, Args&&...args)
    {
        auto* ptr = std::construct_at(reinterpret_cast<T*>(buffer.data()), std::forward<Args>(args)...);
        return unique_ptr<T>(ptr, StackDeleter{});
    }
}

template<is_vehicle V, typename Buffer, typename...Args>
[[nodiscard]] auto createVehicleStack(Buffer& buffer, Args&&...args)
{
    return stack_alloc::create<V>(buffer, std::forward<Args>(args)...);
}
```

We can write even more generic Factory Method with arbitrary allocation policy - so that we can choose between dynamic (heap) and allocation on the stack.
More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/desing%20patterns/Observer.pdf

The entire code: https://godbolt.org/z/x99Y61P73