vtable

Dienstag, 2. Mai 2023 10:52

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

Intro

When we talk about runtime polymorphism, we inevitable talk about virtual methods and dynamic - late binding (linker).
In order for this mechanism to work, the compiler will generate, for each class that introduces virtual methods, the vtable - a table that stores the references on the concreate, derived class implementations (unless, the virtual methods are not the pure one - and base class provides the default implementation, which is not overridden through inheritance).
Each instance of this class, and all derived one - will be equipped with the hidden pointer to the vtable.
If derived class itself introduces additional virtual methods - well, it will have in addition its own vtable.

Imagine that you need programmatically to mimic virtual table.
How would you accomplish that?

I know - this has no pragmatic value, it's something that is embedded into compiler/linker itself.
But still.

I would go with static polymorphism instead and employ the CRTP - as already talked about
https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Invariant%2C%20covariant%20and%20contravariant.pdf

Implementation

All the examples I've found are tailored for the particular - in advance known class and their virtual methods.
To have something similar to concept of virtual dispatching, it needs to be generic enough.
Basically, we need to find the way to represent the vtable: as a map of indexes and to index attached method implementation.
The universal non-static member function signature would be (without cv-qualifiers)

```
template <typename T, typename R, typename...Args>
using method_type = R (T::*)(Args...);
```

But how to put this into homogenous associative array - map?

For that, we need some sort of type-erasure.
Ironically, we introduce the common interface for any method using the very same virtual dispatching

```
struct Supplier
{
     virtual ~Supplier() = default;

    virtual std::any get() const = 0;// type-erasure
};
```

It doesn't look promising at first glance, but it's good enough to provide the method implementation hidden behind the **std::any**, so that we can cache it as

```
std::unordered_map<std::size_t, std::shared_ptr<Supplier>>;
```

The generic model for method implementation could look like the following

```
namespace details
{
    template <typename Func>
    class Method : public Supplier
    {
        public:

              explicit Method(Func func) noexcept : m_func(func) {}
              explicit Method(Func&&func) noexcept: m_func(std::move(func)) {}

              template <typename T, typename R, typename...Args>
              using method_const_type = R (T::*)(Args...) const;
              template <typename T, typename R, typename...Args>
              using method_type = R (T::*)(Args...);

              template <typename T, typename R, typename...Args>
              Method (T* obj, method_const_type<T, R, Args...> method) noexcept:
                     m_func([=](Args&&...args)
                            {
                                return std::invoke(method, obj, std::forward<Args>(args)...);
                            })
              {}

              template <typename T, typename R, typename...Args>
              Method (T*obj, method_type<T, R, Args...> method) noexcept:
                     m_func([=](Args&&...args)
```

```
                        {
                            return std::invoke(method, obj, std::forward<Args>(args)...);
                        })
                {}


                // Supplier interface

                ~Method()override = default;
                std::any get()const override {return m_func; }


            private:
                    Func m_func;

        };


        // CTAD
        template <typename Func>
        explicit  Method(Func) ->Method<Func>;
        template <typename Func>
        explicit Method(Func&&) ->Method<Func>;
        template <typename T, typename R, typename...Args>
        Method(T*, R (T::*)(Args...) const) -> Method<std::function<R(Args...)>>;
        template <typename T, typename R, typename...Args>
        Method(T*, R (T::*)(Args...)) -> Method<std::function<R(Args...)>>;

} // details
```

It's basically the overloaded constructors set, that either store the method, or bind the method with reference to derived class specific implementation.

Now we can represent the vtable as

```
namespace details
{
    class vtable
    {
       public:
              void add_vmethod(std::size_t index, std::shared_ptr<Supplier>&& method)
              {
                  m_vtable[index] = std::move(method);
              }

              const std::shared_ptr<Supplier>& get_vmethod(std::size_t index) const
              {
                  return m_vtable.at(index);// throws if "pure virtual" method is not overridden by derived class
              }

        private:
            std::unordered_map<std::size_t, std::shared_ptr<Supplier>> m_vtable;
    };// class vtable
}// namespace details
```

We can additionally specify the universal vmethod caller as

```
namespace details
{
    template <typename Func, typename...Args>
    decltype(auto) call(const details::vtable& table, std::size_t index, Args&&...args)
    {
        try
        {
            const auto& supplier = table.get_vmethod(index);
            const auto func = std::any_cast<Func>(supplier.get());

            return func(std::forward<Args>(args)...);
        }
        catch (const std::exception& e)
        {
            // if "pure virtual" method is not overridden by derived class: rethrow
            // otherwise, provide the default implementation
            std::cout <<"Exception: "<<e.what() <<'\n';
            //throw e;
        }
    }
}// namespace details
```

Demonstration

Ok, let's assemble all this together.
Assume, we have a base class A that introduces some virtual method: A::f.
We will have a hidden (protected) pointer to the empty vtable.
This pointer will be only accessible in derived class instances, to eventually override the base class vmethod, attaching the concreate implementation

```
struct B :A
{
      explicit B(int value) noexcept : A(value)
      {

          // "override" the base class method
          // # Using the emplace like c-tor
          m_vtable->add_vmethod(f_ind, std::make_shared<details::Method<std::function<void(int)>>>(this, &B::f));
      }
 };
```

Since Method emplace c-tor binds internally the implementation with the derived class reference, the method signature can be customized with **std::function**, as universal placeholder for any callable.
We call the derived class specific implementation through the base class interface (as with CRTP).
The base class has no notion about its subtypes - yet, it knows about the method signature : the vary own, and matching index, to retrieve from the virtual table, the overridden implementation.
If the vmethod was a "pure" virtual method, and there is no overridden implementation: we can rethrow after catching the exception ("implementation not found"), or otherwise fallback to the default base class implementation

```
struct A
{
    // Quasi virtual method
    void f(int a)const
    {

        try
        {
            const auto& supplier = m_vtable->get_vmethod(A::f_ind);
            const auto func = std::any_cast<std::function<void(int)>>(supplier->get());
            func(a);
        }
         catch(const std::out_of_range& e)
        {
            // if "pure virtual" method is not overridden by derived class: rethrow
            // otherwise, provide the default implementation
            std::cout << "Exception: " << e.what() << '\n';
            // throw e;
        }
    }
};
```

Especially interesting is **destructor**, since it can't be addressed directly.
Therefore to simulate virtual destructor being overridden by the subtype, we need to introduce a private cleanup method

```
struct B : A
{
    explicit B(int value) noexcept: A(value)
    {
        // "override" the destructor

        // We can't address destructor directly - therefore we introduced private cleanup method: destroy

        m_vtable->add_vmethod(A::destructor_ind, std::make_unique<details::Method<std::function<void(void)>>>(this, &B::destroy));
     }
};
```

Code

The complete example can be found at
Compiler Explorer: https://godbolt.org/z/63GfoEWM7