vtable#2

Sonntag, 14. Mai 2023 20:00

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

Run-time implementation

I've tried to emulate the vtable mechanism that is embedded into compiler/linker implementation.
The first attempt was to use the very same virtual dispatching that we tried to mimic

https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/vtable.pdf

Better way - Compile-time implementation

But, there is a better way: we can implement the vtable at compile-time.
The basic idea is to use the **std::tuple as heterogenous container** - to store the vmethods of a different signatures at the point of the vtable construction - we don't need to dynamically add the vmethods, since they are known at the compile-time.
The gotcha with *std::tuple* is that it can store only distinguished types.
On the other hand, we can have the **vmethods of a same signature**.
To overcome this issue, we need to wrap our signature into **strong type**: as the way to represent the same underlying types as the unique one.

Disclaimer: Jonathan Baccara has a fantastic series on this topic
https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/
and the code is available at
https://github.com/joboccara/NamedType

Implementation

What has been changed, in compare with the run-time implementation?
First, instead of type erasure and std::any, we introduced the parameterized interface that stores the **vmethod index** and the **signature** together

```
template <typename Func>
struct IndexedMethod
{
    using method_type =std::decay_t<Func>;

    std::size_t m_index;
    method_type m_func;
};
```

Consequently, we changed the vtable implementation as well

```
template <typename...Fs>
class vtable
{
    public:
        /**
         * C-tor
         *
         * Construct the vtable, since the all vmethods are known compiler upfront
         * We don't need mechanism to dynamically add them
        */

         template <typename Func>
         using indexed_method_type = IndexedMethod<Func>;

         constexpr vtable (indexed_method_type<Fs>&&...vmethods) noexcept :
                          m_vtable(std::make_tuple(std::forward<indexed_method_type<Fs>>(vmethods)...))
         {}

         template <typename Func>
         constexpr auto get(std::size_t index) const
         {
              return getImpl<Func>(index, std::make_index_sequence<sizeof...(Fs)>{});
         }

         template <typename Func>
         constexpr void set(Func&& func, std::size_t index) noexcept
         {
             setImpl(func, index, std::make_index_sequence<sizeof...(Fs)>{});
         }

    private:
         template <typename Func, std::size_t...Is>
```

```cpp
        constexpr std::optional<Func> getImpl(std::size_t index, std::index_sequence<Is...>) const noexcept
        {
            std::optional<Func> method;
            auto find = [index, &method](const auto& vmethod)
            {
                if (vmethod.m_index ==index) method = vmethod.m_func.get(); // We store Method within the StrongType

            };

            ( find(std::get<Is>(m_vtable)),...);

            return method;
         }

         template <typename Func, std::size_t...Is>
         constexpr void setImpl(Func&& func, std::size_t index, std::index_sequence<Is...>) noexcept
         {
             auto findAndSet = [index, method = std::forward<Func>(func)](auto& vmethod)
             {
                 if (vmethod.m_index == index)
                 {
                     vmethod.m_func.emplace(method); // Override the Method in StrongType
                 }
             };

             ( findAndSet(std::get<Is>(m_vtable)),...);
         }

    private:
         std::tuple<indexed_method_type<Fs>...> m_vtable;
};
```

The **generic vmethod implementation** is also modified, to be entirely compile-time constructible

```cpp
  // vmethod generic representation

  template <typename Func>
  class Method
  {
      public:

            constexpr explicit Method(const Func& func) noexcept: m_func(func) {}
            constexpr 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>
            constexpr 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>
            constexpr Method(T* obj, method_type<T, R, Args...>method) noexcept:
               m_func([=](Args&&...args)
                      {
                          return std::invoke(method, obj, std::forward<Args>(args)...);
                      })
            {}

            // Implicit conversion operator
            constexpr operator Func() const & { return m_func; }

            // Getter
            [[nodiscard]] constexpr Func& get() const & { return m_func; }


      private:
            Func m_func;

  };
```

Demonstration

Having the class A, that introduces the vmethod A::f(), we can add into our vtable

```cpp
A() noexcept: m_vtable(
                    details::IndexedMethod <af_type>
```

```
                            {
                                .m_index = A_f_ind,
                                .m_func  = details::Method<std::function<void(int)>>(this, &A::f_default)
                            })
{}
```

Where A::f_default() is the default vmethod implementation.

@note In case of the **"pure" vmethod**, we could construct our vmethod with *nullptr*

```
.m_func  = details::Method<std::function<void(int)>>(nullptr)
```

We call the derived class implementation (if any) through the base class interface as

```
void A::f(int value) const
{
    using method_type = std::function<void(int)>;

    const auto vmethod = static_cast<std::optional<method_type>>(m_vtable.get<method_type>(A::A_f_ind));
    // "overridden" A::f() method
    // If this is a "pure" vmethod and not being overridden, and *vmethod is nullptr: std::bad_function_call exception
    // will be thrown
    if ( vmethod.has_value())
    {
        (*vmethod)(value);
    }
}
```

We "override" the base class implementation as

```
explicit B(int value) noexcept: m_value(value)
{
    // "Override" virtual method A::f()
    m_vtable.set<std::function<void(int)>>(
                details::Method<std::function<void(int)>>(this, &B::f),
                A::A_f_ind
            );
}
```

#Code:
https://godbolt.org/z/jMqPd6cvP