Bridge design pattern

Freitag, 19. Mai 2023 15:17

Intro

In short, Bridge design pattern is

- Structural design pattern that
- Separates the Abstraction from the concreate Implementation, so that
  these two can vary independently, in two separate hierarchies.
  It can be seen as **pimpl** idiom, since it hides the implementation details at client side.

Usage example

We can define the universal factory method as

```
namespace details
{
    template <typename T>
    struct Factory final
    {
        /**
            Universal factory method
            Creates the std::unique_ptr<T> for an arbitrary arguments list
            Advantage: the c-tor of type T after specialization, can be changed - but
            the factory method remains the same
        */
        template <typename...Args>
        static std::unique_ptr<T>create(Args&&...args)
        {
            if constexpr(std::is_constructible_v<T, Args...>)
            {
                return std::make_unique<T>(std::forward<Args>(args)...);
            }

            return nullptr;

        }
     };
}
```

For the sake of arguments, let's define the Implementation common interface - using run-time virtual dispatching

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

     // Implementation common interface

    virtual void f() = 0;
};
```

We introduce the Abstraction as a class template - with an Implementation policy, as the way to inject the concreate implementation at compile-time: at client customization point side.
For the static polymorphism - more on that subject
https://github.com/damirlj/modern_cpp_tutorials/tree/main#tut6

This is our Abstraction, with its own hierarchy, that can vary independently from Implementation hierarchy.

```
template <typename Implementation>
struct Abstraction
{
    virtual ~Abstraction() = default;

     // Abstraction common interface

    virtual void g() = 0;
    virtual void h() = 0;

    explicit Abstraction(std::unique_ptr<Implementation> pimpl) noexcept:
         m_pimpl(std::move(pimpl))
    {}

    template <typename...Args>
    explicit Abstraction(Args&&...args) noexcept:
         m_pimpl(details::Factory<Implementation>::create(std::forward<Args>(args)...))
    {}

    protected:
        std::unique_ptr<Implementation> m_pimpl;
};
```

Well, that's it - more than less everything what we need.
In Implementation branch, we can have then something like

```
// Concreate implementations

struct A1 : IImplementation
{
    void f() override { puts("A1::f()"); }
};


struct A2 : IImplementation
{
    explicit A2(int id) noexcept: m_id(id) {}

    void f() override { puts("A2::f()"); }

    int get() const { return m_id; }

    private:
        int m_id;
};
```

On the client side, we can finally bridge these two as

```
struct Client1: public Abstraction<A1>
{
    using base = Abstraction<A1>;
    using base::base;

    virtual ~Client1() override = default;

    void g() override
```

```
    {
        printFunc();

        m_pimpl->f();
        puts("Client1::g()");
    }

    void h() override
    {
        printFunc();

        m_pimpl->f();
        puts("Client1::h()");
    }
};


struct Client2: public Abstraction<A2>
{
    using base = Abstraction<A2>;
    using base::base;// base class c-tors

    virtual ~Client2() override = default;

    void g() override
    {
        printFunc();

        m_pimpl->f();// from implementation "borrowed" functionality

        puts("Client2::g()");
        std::cout << "id=" << m_pimpl->get() <<'\n';
    }

    void h() override
    {
        printFunc();

        m_pimpl->f();

        puts("Client2::h()");
        std::cout << "id=" << m_pimpl->get() <<'\n';
    }
};
```

Code

The entire code is also available at: https://godbolt.org/z/doxh136PM