Builder

Samstag, 3. Juni 2023 18:08

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

Intro

- Builder design pattern is used for building the complex objects that are configurable, can have different set of properties (optionals) being specified: for having multiple representation of the same class.
- The creation of the class instance is separated from its representation.
- This means, the properties are set at client side with setters, rather than providing the all possible overloading versions of c-tor

The generic way to capture the common functionality of the builder could be

```
namespace utils
{

    template <typename Derived, typename...Args>
    class builder_t
    {
        public:

                /*
                 * Take all properties, by reference
                 */
                explicit builder_t(std::optional<Args>&...args) noexcept: m_properties(std::tie(args...)){}

                /**
                 * Setters - are tedious to write, especially for large number of properties
                 */
                template <std::size_t Ind>
                Derived& setter(auto&& arg)
                {
                    auto& property = std::get<Ind>(m_properties);
                    property = std::forward<decltype(arg)>(arg);

                    return impl();
                }

                /**
                 * Build the outer class
                 * @note Can't use
                 *  template <class Outer>
                 *  auto build() { return std::make_unique<Outer>(impl()); }
                 *  since the Outer c-tor should be private
                 */
                auto build() && { return impl().create(); }

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

         private:
                std::tuple<std::optional<Args>&...>m_properties;
   };
}
```

Usage

For the toy-class A, assume that we want to apply our generic builder-helper class: builder_t

```
class A
{
    // Configurable (optional) properties
    std::optional<std::string> m_name;
    std::optional<int> m_id;


    public:
        A() = delete;
        class Builder;


    private:
        /**
         * C-tor of the class
         *
         * @note private
         * @param builder Reference to the builder that will be used to configure this class
         */
        explicit A(const Builder& builder) noexcept
        {
            m_name = builder.m_name;
            m_id = builder.m_id;
        }

    public:


        /**
         *   Builder - inner class as a wrapper around the generic implementation.
         *   It's relying on yet another - CRTP pattern
         */

        class Builder final: public utils::builder_t<Builder, std::string, int>
        {
            friend class A;// class A can access the Builder's private fields and methods

            std::optional<std::string> m_name = std::nullopt;
            std::optional<int> m_id = std::nullopt;

            public:

                using base = utils::builder_t<Builder, std::string, int>;

                // Initialize the builder with current enclosing instance representation
                // It's ok, since the arguments are passed by reference (and then initialized)
                explicit Builder(const A& a) noexcept : base(m_name, m_id), m_name(a.m_name), m_id(a.m_id) {}
                Builder() noexcept: base(m_name, m_id){}

                // Setters: one can even use the helper setters directly at client side, but these wrappers
                // are for the sake of having expressive interface

                template <typename T>
                Builder&& setName(T&& name)  &&
                {
                    static_assert(details::is_string_v<T>,"<Setter> The argument is not\"string-like\"one");
                    return std::move(setter<0>(std::forward<T>(name)));
                }

                Builder&& setId(int id)  &&
                {
                    return std::move(setter<1>(id));
                }

                /**
                 * Provide the outer class factory method - that will be called
                 * inside the generic build() method.

                 * @note We can't use std::make_uinique<A>, since the outer class c-tor is private.
                */
                std::unique_ptr<A> create() const
                {
                    return std::unique_ptr<A>(new (std::nothrow) A(*this));
                }
        };// Builder
};
```

As you may observed, the all setter methods are declared as rvalue-reference qualified non-static member methods.
This is to emphasized the temporality of the inner Builder class.
It merely purpose is to configure - construct on the fly the outer enclosing class.

Example

The entire code can be found at: https://godbolt.org/z/88cffTv4K