Observer

Sonntag, 3. September 2023 14:54

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

Introduction

Observer belongs to the architectural patterns - describing how the bigger modules of a system can interact between each other, just by sending the messages.
It's **event-driven** architecture, where event will be sent to the all receivers expecting the very same *message*: to all *actors* being *subscribed* to that particular event.
I've deliberately used in the same sentence the *synonyms* - other terms that are commonly used to describe this pattern: as **asynchronous massage-based communications** between software components.
But unlike actors, that have a dual-nature: can act as a Sender and Receiver at the same time, the Observer pattern distinguish two entities: *Publisher* - the one who emits the events (or rather messages - items), and one or more *Subscribers* - the one who listen on these messages: event notifications.

My favorite example of this pattern is Reactive library, for which I've dedicated the entire article
https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/android/RxAndroid.pdf
It has 3O notation: it has clearly decoupled **O**bservable (Publisher) from **O**bserver(Subscriber), providing the rich set of **O**perators - chained operations that usually do some kind of transformations on the downstream, until the items reach the subscribers.
This functional approach, that hides the implementation details, is known as declarative - in opposite to imperative approach, which leads to more concise, more expressive code which is easier to read and therefore understand, easier to maintain.
Not only that, but it allows us to specify different thread contexts in which Observable will emit the items, from the one in which Observer will observe these items.
And on top of it - it comes with laziness in mind: in terms of creation and/or emission of (cold) observables.

Implementation details

More about that and other patterns and SOLID principles you can find in "*C++ Software Design*"[1] by **Klaus Iglberger**, who's book served me in a great deal as inspiration for implementing the Observer pattern in a way I did, along with Reactive library.
As Klaus mentioned in his book - we design our code to be flexible enough to react on any change (in requirements), either by adding a new types, or by adding a new operations.
The key is to determine the variation points - and keep them in isolation.
Here, the Observer (Subscriber) is our variation point - different Observers will provide a different callable at the subscription point: a different way of handling the same event.

Taking the Reactive library as a reference, the Observer interface will look like

```cpp
// Observer callbacks interface
template <typename T, typename Error = std::exception_ptr>
struct IObserver {
    using value_type = T;
    using error_type = Error;


    virtual void onNext(const value_type& ) = 0;
    virtual void onNext(value_type&& ) = 0;

    virtual void onError(error_type ) = 0;
    virtual void onCompletion() = 0;

    virtual ~IObserver() = default;
};
```

The first choice would be to implement it in OO fashion - providing derived class specific implementation.
Since we talking here about designing by adding a new types, I've decided to use another design pattern: **Type Erasure**.
https://github.com/damirlj/modern_cpp_tutorials#tut7

It's fair enough to say, that Observer (Subscriber) implementation is rather about Type Erasure and some performance optimization using the stack as a storage, rather than constantly reach for the heap.
Idea with Type Erasure is that you have:

- An internal - hidden interface: the very same IObserver interface
- For which you provide the *private* (pimpl idiom) implementation of this interface, as a wrapper around the real - *external* implementation which is injected via templatized constructor (external polymorphism) of an enclosing class: Observer

```cpp
//Templated C-tor: customization point
template <typename Func>
explicit Observer(Func&& func) noexcept:
observer_(make_unique<ObserverImpl<Func>>(std::forward<Func>(func)))
{}
```

Our template parameter *Func* - argument *func* is the callable that will be actually invoked
at the point when Observable emits the new item.
ObserverImpl<Func> is internal implementation, as a wrapper around the provided callback - callbacks.
We holds the pointer to the base IObserver class and relies on the return type covariant, when we call the unqualified *make_unique*() utility method.
It's another customization point - the storage type, which will be provided as a storage policy (static polymorphism)

```
namespace details
{

     template <class Allocator>
     struct Deleter
     {

          using allocator_type = std::remove_cvref_t<Allocator>;

         explicit Deleter(allocator_type& allocator) noexcept : allocator_(allocator) {}

         void operator()(auto* ptr) {
             allocator_.deallocate(ptr);
         }
         private:
             Allocator& allocator_;
     };

     template <typename T, class Allocator>
     using unique_ptr = std::unique_ptr<T, Deleter<Allocator>>;

     template <typename T, class Allocator, typename...Args>
     [[nodiscard]]auto make_unique(Allocator& allocator, Args&&...args) {
           return details::unique_ptr<T, Allocator>(allocator.template allocate<T, Args...>
     (std::forward<Args>(args)...), Deleter<Allocator>{allocator});
     }
}// namespace details
```

The idea behind is the ability to optimize the code by providing the more efficient storage than the heap, especially in embedded environment where this is limited or even prohibited.
We can therefore introduce, behind the default - dynamic storage, the storage on the stack (similar to std::pmr allocators)

```
template <std::size_t Capacity, std::size_t Allignment>
struct StackStorage
{
    template <typename T, typename...Args>
    T* allocate(Args&&...args)  {
        static_assert(sizeof(T) <= Capacity, "Storage capacity exceeded");
        return std::construct_at(reinterpret_cast<T*>(buffer_.data()), std::forward<Args>(args)...);
    }


    void deallocate(auto* ptr) {
        if (ptr) std::destroy_at(ptr);
    }

    private:
        alignas(Allignment) std::array<std::byte, Capacity> buffer_;
};
```

Now we can write the enclosing Observer class as

```
template <typename T, typename Storage, typename Error = std::exception_ptr>
struct Observer {

        using value_type = T;
        using error_type = Error;


        template <typename Func>
        using onNextCallback = Consumer<value_type, Func>;
        using onErrorCallback = std::function<void(const error_type&)>;
        using onCompletionCallback = std::function<void()>;

    private:
        // Type erasure
        struct IObserver {
              virtual void onNext(const value_type& ) = 0;
              virtual void onNext(value_type&& ) = 0;

              virtual void onError(error_type ) = 0;
              virtual void onCompletion() = 0;
              virtual ~IObserver() = default;
```

```cpp
            // prototype design pattern
            using unique_ptr = details::unique_ptr<IObserver, Storage>;
            virtual unique_ptr clone(Storage& storage) const = 0;
      };

      template <typename Func>
      struct ObserverImpl final :IObserver
      {
          …
          IObserver::unique_ptr clone(Storage& storage) const override {
              return details::make_unique<ObserverImpl>(storage, *this);
      }

      };// ObserverImpl
      …
};// Observer
```

where the storage is default-constructable member variable, so that for all internal allocation/deallocation we can efficiently pass it by reference (@see details::Deleter class)

```cpp
Storage storage_{};
IObserver::unique_ptr observer_;//pimpl idiom

template <typename O, typename...Args>
requires std::is_base_of_v<IObserver, O>
auto make_unique(Args&&...args) {
   return details::make_unique<O, Storage>(storage_, std::forward<Args>(args)...);
}
```

So that at client side we can specify the desired storage as

```cpp
// Alias as customization point for storage
// using Storage = details::DynamicStorage;
using Storage = details::StackStorage<128u, alignof(void*)>;
using PersonObserver = Observer<Person, Storage>;
```

Our *onNext*() method - that will be called by the Observable whenever there is a new item to be emitted, has two overloaded versions, to support move semantic as well.
We can use a small utility class for that (similar to Java Consumer<T> class)

```cpp
//For declaring the callable that can be invoked with both,
//lvalue and rvalue reference argument, something like supporting the
//both signatures at the same time
// - std::function<void(const auto& )>
// - std::function<void(auto&&)>
template <typename T, typename Func>
requires std::invocable<Func, T>
struct Consumer
{

    explicit Consumer(Func&&func) noexcept(
             std::is_nothrow_copy_constructible_v<Func> ||
             std::is_nothrow_move_constructible_v<Func>) :func_(std::forward<Func>(func))
    {}

    void apply(const T& arg) {
        applyImpl(arg);
    }

    void apply(T&& arg) {
        applyImpl(std::move(arg));
    }

private:
    template <typename U>
    requires std::convertible_to<U, T>
    void applyImpl(U&& u) {
        std::invoke(func_, std::forward<U>(u));
    }

    std::remove_cvref_t<Func> func_;
};
```

Observable (Publisher) implementation is much more straightforward.
It stores (as an textbook example) the weak reference (std::weak_ptr) to the Observers (1:n relationship), that will be subscribed to receive the newly emitted items (optionally to capture exceptions/completion event)

```cpp
template <typename Observer>
class Observable {
  public:
```

```
        using observer_type = std::decay_t<Observer>;
      /**
      Subscribe the observer, by meaning of the callable
      that will be invoked each time when there is something to be
      emitted.
      @note In order to utilize on RVO, the return value must not be discarded
   */
      [[nodiscard]] std::shared_ptr<observer_type> subscribe(const observer_type& observer) {
              auto subscription = std::make_shared<observer_type>(observer);
              observers_.push_back(subscription);
              return subscription;
        }
  private:
        //template <typename Item>
        //using notify_f = void (observer_type::*)(Item&&);

        template <typename T, typename Func>
        void notifyImpl(T&& item, Func&& func) {// fun is the propriate IObserver callback
                using std::cbegin;
                using std::cend;

                auto notify = [item = std::forward<T>(item),func =std::forward<Func>(func)]
                              (const auto& observer) {
                              if (auto ptr = observer.lock()){
                                std::invoke(func, *ptr, item);
                              }
                };

                std::for_each(cbegin(observers_), cend(observers_), notify);
        }
  public:
        template <typename T>
        requires std::is_base_of_v<typename observer_type::value_type, T> ||
                 std::convertible_to<T, typename observer_type::value_type>

        void notify(T&& value) {
                auto dispatch = [](observer_type& observer, auto&& val)
                {
                        using U =std::decay_t<decltype(val)>;

                        if constexpr(std::is_rvalue_reference_v<U>){
                                observer.onNext(std::forward<U>(val));
                                return;
                        }
                        observer.onNext(val); // lvalue reference overloading call
                };

                notifyImpl(std::forward<T>(value), dispatch);
        }

        void notifyError(typename observer_type::error_type error) {
             notifyImpl(error, &observer_type::onError);
        }

        void notifyCompletion() {
            notifyImpl(&observer_type::onCompletion);
        }


  private:
      std::vector<std::weak_ptr<observer_type>> observers_;
};
```

To at least try to mimic the Reactive library implementation with the Operator set, here is some clumsy attempt to implement the *map* operator: it applies transformation function to the each emitted item down the stream. It's actually implemented more like *flatMap*,
where the result will be a new Observable: O<f(T)> , as a monad: map(O<T1>, f: T1->T2): O<T2>

```
//Operators: map example

template <typename T, typename Observer, typename Func>
requires std::invocable<Func, T>
auto map(Func&& func) {
        // New - transformed observable (Decorator pattern - kind of)
        struct MappedObservable : Observable<Observer> {
              explicit MappedObservable(Func&& func) noexcept:
                      func_(std::forward<Func>(func)){}

              using value_type = std::remove_cvref_t<T>;

              void notify(const value_type& arg) {// "decorate" base-class implementation by applying
transformation function
                    try{
                          auto&& value = std::invoke(func_ ,arg);
```

```
                Observable<Observer>::notify(std::forward<decltype(value)>(value));
            }catch(...) {
                const auto error = std::current_exception();
                Observable<Observer>::notifyError(error);//you can't have uniform, overloaded
notify(error) method, since predominant function template in overloading set
            }

        }
        private:

            Func func_;

    };

    return MappedObservable{std::forward<Func>(func)};
}
```

Code example

Compiler Explorer: https://godbolt.org/z/1WPj6MTYf

Links

1. https://meetingcpp.com/mcpp/books/book.php?hash=60bbda6997fb45126b54f7e933ba6d8febb91fa6