Locking policy

Samstag, 20. Juli 2024 16:04

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

Introduction

The famous **Andrei Alexandrescu** in his *Modern C++ Design* book published in 2000., was indeed the visionary of his time, introducing a lot of template programming technics and concepts - like Policy design and Static polymorphism that we heavily use now days.
The all these ideas are demonstrated in form of a library, **the *Loki* library** - which accompanies the book.
There was recent a public invitation, mainly for students - to modernize the library using the new C++20 standard features.

(more on that: https://cppdepend.com/blog/loki-the-best-c-library-to-learn-design-patterns-lets-modernize-it/)

The most of the concepts that are originated from this library, became a part of the new C++11 standard - like variadic templates (parameters pack), type traits, smart pointers, tuples, etc.
Therefore, it's somehow contradictable to modernize something that was inspirational for modern C++ standards at the first place.

For me, the most interesting part of the library is Policy design pattern - as the way to make the host class configurable for different kind of approaches - strategies.
(more on that: https://github.com/damirlj/modern_cpp_tutorials?tab=readme-ov-file#tut6)

One of them is a Locking policy, where Andrei introduces a different locking levels:

- **Non-locking** (single-thread) level - disable locking
- **Object-level** locking: the multiple threads will be synchronized on the same instance of the class
- **Class-level** locking: the multiple threads will be synchronized on the all instances of the same class

Implementation

Let's see, how we can re-implement these, utilizing on the features available in C++20.
This is where the *concepts* fully shine
(more on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/C%2B%2B20/Concepts.pdf)

We can specify the requirements for the locking type

```
template <typename Lock>
concept is_lockable = requires(std::remove_cvref_t<Lock>& lock)
{
     lock.lock();
     lock.unlock();
};
```

If we fallow the original design, as close as possible - we can write for Object-level locking something like

```
template <typename Host, typename Lock>
requires is_lockable<Lock>
struct ObjectLock
{
    private:// data
        mutable Lock lock_ {};

    private:// methods
        inline void lock()   const { lock_.lock(); }
        inline void unlock() const { lock_.unlock(); }
    public:
        /**
         * This is requirement on enclosing class - lock level, to have the same (name) inner type that
         * implements the actual locking strategy
        */
        struct ScopeLock final
        {
            explicit ScopeLock(const ObjectLock& obj) noexcept: obj_{obj} { obj_.lock(); }
            ~ScopeLock() { obj_.unlock(); }
            private:
                const ObjectLock& obj_;
        };
};
```

`Host` is just tagging type here - to uniquely identify the declaring type - binding it with the client class.
We don't want to use *parameterized inheritance* here - as with original approach, since client class is usually

not locker itself - but rather can be configured on different locking strategies

Similar, we can write the Class-level lock as

```
template <typename Host, typename Lock>
requires is_lockable<Lock>
struct ClassLock
{
     private: // data
          static inline Lock lock_{};
     private: //methods
          static void lock()    { lock_.lock(); }
          static void unlock() { lock_.unlock(); }
     public:
         struct ScopeLock final
          {
              explicit ScopeLock(const ClassLock& ) noexcept { lock(); }
              ~ScopeLock() { unlock(); }
          };
};
```

At the client site, we can incorporate the Locking policy as

```
template <template <class, class> class Locker, typename Lock = std::mutex>
struct Counter
{
    private:
        int count_ = 0;
        using lock_type = Locker<Counter, Lock>; // binding the host class with the Locker
        lock_type lock_ {};
    public:
        int operator()()
        { //
            typename lock_type::ScopeLock lock {lock_}; // requirement on the Locker - to have ScopeLock as inner type
            std::cout << '[' << std::this_thread::get_id() << "] counter: " << ++count_ <<'\n';
            return count_;
        }
        ...
};
```

Let's return for a moment back - to the our Lock-Level implementation.
Instead of relying on the constraint that all Lock-Level implementations have the same inner (name) type: ScopeLock, and the fact that we already have in standard library *std::quard_lock* as a scope lock in RAII sense (or even *std::scoped_lock* - for guarding more than one mutex and avoid the dead lock), we can simplify the implementation by imposing the same behavioral aspect - the same call operator signature

```
namespace locking
{
    template <typename Lock>
    using scope_lock = std::lock_guard<Lock>;

    template <typename Host, typename Lock>
    requires is_lockable<Lock>
    struct ObjectLock
    {
        private:// data
            mutable Lock lock_ {};

        public:
            /**
             * Instead of having constraint on the inner (name) type, we
             * can impose constraint on the behavioral aspect - that all Lock-Levels in our
             * Locking policy have the same call-operator signature
            */
            [[nodiscard]] inline scope_lock<Lock> operator()() const
            {
                return scope_lock<Lock> {lock_};
            }
    };
}
```

The Class-level lock becomes even simpler

```
template <typename Host, typename Lock>
requires is_lockable<Lock>
struct ClassLock
{
```

```
    private:// data
        static inline Lock lock_{}; //assuming, this is a header file

    public:
        [[nodiscard]] inline scope_lock<Lock> operator()() const
        {
            return scope_lock<Lock> {lock_};
        }
};
```

Well, for *NonLock* type, which has not been yet introduced (to disable locking mechanism), this may be a problem, having explicit return type as scope_lock<Lock>.
But we can be clever enough, and let the compiler to deduce the type - using *auto*

```
template <typename Host, typename Lock>
struct NonLock
{
    public:
        [[nodiscard]] inline auto operator()() const {}
};
```

The entire code can be found at: https://godbolt.org/z/zYo6s49G9