The gotcha of the C++ temporaries that don’t destruct as eagerly as you thought

devblogs.microsoft.com/oldnewthing/20221005-00

October 5, 2022

Raymond Chen

Forgetting to take a lock when updating variables is a common mistake. One way to make the mistake harder to make is to *force* the access to occur through some mechanism that proves that you possess the lock. Maybe something like this:

```
template<typename> struct LockableData;

namespace std
{
    template<typename Data>
    struct default_delete<LockableData<Data>>
    {
        void operator()(LockableData<Data>* p)
        const noexcept { p->m.unlock(); }
    };
}

template<typename Lockable>
struct [[nodiscard]] LockedData
{
    LockedData(Lockable* l = nullptr) : l(l)
    { if (l) l->m.lock(); }

    auto operator->() const noexcept
    { return std::addressof(l->data); }

private:
    std::unique_ptr<Lockable> l;
};

template<typename Data>
struct LockableData
{
    LockedData<LockableData> Lock() { return this; }

private:
    friend struct LockedData<LockableData>;
    friend struct std::default_delete<LockableData>;

    std::mutex m;
    Data data;
};
```

The idea here is that you declare some structure that holds the data you want to be protected by a common mutex. You can then wrap that data inside a `LockableData`. To access the data, you call `Lock()` to acquire the mutex and receive a `LockedData` object. You then access the structure through the `LockedData` object, and when the `LockedData` object destructs, it releases the mutex.

Using a `std::unique_ptr` with a custom deleter allows the `LockedData` object to be movable with the natural semantics. And marking the `LockedData` as `[[nodiscard]]` makes sure that you save the return value of `Lock()`; otherwise, it destructs immediately, and your lock accomplished nothing.

Here's an example usage:

```
struct WidgetInfo
{
    std::string name;
    int times_toggled = 0;
};

class Widget
{
    LockableData<WidgetInfo> info;

public:
    void SetName(std::string name)
    {
        auto lock = info.Lock();
        lock->name = name;
        lock->times_toggled = 0;
    }

    std::string GetName()
    {
        auto lock = info.Lock();
        return lock->name;
    }

    void Toggle()
    {
        { // scope the lock
            auto lock = info.Lock();
            lock->times_toggled++;
        }
        FlipSwitchesRandomly();
    }
};
```

One thing that's slightly annoying here is that in many cases, we are locking around the access to a single member. One way to avoid having to create tiny scopes is to allow the `->` operator to be used directly from the `LockableData`, so that it does a lock-access-unlock.

```
template<typename Data>
struct LockableData
{
    LockedData<LockableData> Lock() { return this; }
    auto operator->() { return Lock(); } // NEW!

private:
    friend struct LockedData<LockableData>;
    friend struct std::default_delete<LockableData>;

    std::mutex m;
    Data data;
};

class Widget
{
    LockableData<WidgetInfo> info;

public:
    void SetName(std::string name)
    {
        auto lock = info.Lock();
        lock->name = name;
        lock->times_toggled = 0;
    }

    std::string GetName()
    {
        return info->name; // lock-read-unlock
    }

    void Toggle()
    {
        info->times_toggled++; // lock-modify-unlock
        FlipSwitchesRandomly();
    }
};
```

This convenience `->` operator makes single-member updates much easier, but it also comes with a catch:

```
    void Toggle()
    {
        info->times_toggled = std::max(info->times_toggled, 10);
        FlipSwitchesRandomly();
    }
```

Do you see the problem here?

I mean, yes, there's a race condition here if two threads toggle at the same time, but that's not the problem I'm referring to. That would "merely" result in incorrect accounting.

The real problem is the double lock.

The `->` operator returns a temporary `LockedData` object, and the rules for temporary objects in C++ are that they are destructed at the end of the *full expression*.

Therefore, the evaluation of the revised code goes like this:

```
// Evaluate right hand side
LockedData<WidgetInfo> lock1 = info.operator->();
int rhs = std::max(lock1->times_toggled, 10);

// Evaluate left hand side
LockedData<WidgetInfo> lock2 = info.operator->();

// Perform the assignment
lock2->times_toggled = rhs;

// Destruct temporaries in reverse order of construction
destruct lock2;
destruct rhs;
destruct lock1;
```

Do you see the problem yet?

Since the locks are temporaries, their lifetime extends to the end of the full expression. We've seen this problem before.[1] This means that the two `info->times_toggled` operations each create a separate `LockedData` object, and each one acquires the mutex.

The result is that we acquire the mutex lock twice, which is not allowed. In practice, this results in a deadlock and forces you to issue a public apology.

The convenience `->` was a bit *too convenient*: If an object `o` is a smart pointer, people are accustomed to `o->Something` doing some work with `o` in order to produce the pointer that will be dereferenced in order to access the `Something`. What they are not accustomed to is the presence of work that occurs after the dereference to clean up.

In other words, people tend to expect that it's okay to call the `->` operator many times on the same object without consequence.

So take away that dangerous operator. People can still get the one-liner convenience, though. They just have to lock explicitly:

```
std::string GetName()
{
    return info.Lock()->name;
}
```

Having to write out the word “Lock” also makes it easier to spot that you’re locking twice within the same expression.

```
void Toggle()
{
    // suspicious double-lock - more likely to be spotted in code review
    info.Lock()->times_toggled = std::max(info.Lock()->times_toggled, 10);
    FlipSwitchesRandomly();
}
```

[1] It looks like C++23 has adopted the idea of the `out_ptr` temporary object which resets the pointer on destruction, in spite of the footguns that arise in certain usage patterns.

Raymond Chen

Follow