Thoughts on creating a tracking pointer class, part 15: A custom shared pointer

devblogs.microsoft.com/oldnewthing/20250829-00/?p=111526

August 29, 2025

Last time, we made our trackable object implementation's constructors and assignment operations non-throwing, although it came at a cost of making the act of creating a tracking pointer potentially-throwing. At the end, I noted that we didn't use all the features of C++ shared pointers. We never used weak pointers or thread safety, so we can replace shared pointers with a custom version that supports only single-threaded shared references.

The single-threaded simplification is significant because it removes the need for atomic operations and allows the compiler to do reordering and coalescing of reference count manipulations.

```
template<typename T>
struct tracking_ptr_base
{
    tracking_ptr_base() noexcept = default;
    tracking_ptr_base(tracking_ptr_base const& other) noexcept
        : m_ptr(other.copy_ptr()) {}
    tracking_ptr_base(tracking_ptr_base&& other) noexcept = default;
    ~tracking_ptr_base() = default;

    tracking_ptr_base&
        operator=(tracking_ptr_base const& other) noexcept
    {
        m_ptr = other.copy_ptr();
        return *this;
    }

    tracking_ptr_base&
        operator=(tracking_ptr_base&& other) noexcept = default;

    operator bool() const noexcept
    {
        return get() != nullptr;
    }

protected:
    friend struct trackable_object<T>;

    struct data
    {
        data(T* tracked) noexcept : m_tracked(tracked) {}
        unsigned int m_refs = 1;
        T* m_tracked;
    };

    struct deleter
    {
        void operator()(data* p)
        {
            if (--p->m_refs == 0)
            {
                delete p;
            }
        }
    };

    tracking_ptr_base(T* p) noexcept : m_ptr(new data(p))
    {
    }

    T* get() const noexcept
    {
        return m_ptr ? m_ptr->m_tracked : nullptr;
    }

    void set(T* ptr) noexcept
    {
```

```
        m_ptr->m_tracked = ptr;
    }

    std::unique_ptr<data, deleter> copy_ptr() const noexcept
    {
        if (m_ptr) ++m_ptr->m_refs;
        return std::unique_ptr<data, deleter>(m_ptr.get());
    }

    std::unique_ptr<data, deleter> m_ptr;
};
```

This looks like a lot of code, but it's really accomplishing very little.

The basic operations on the pointer are incref and decref. The incref operation increments the `m_refs` and the decref operation decrements the `m_refs`, destroying the `data` if the reference count goes to zero.

Copying the `tracking_ptr_base` copies the pointer and performs an incref. Destructing the `tracking_ptr_base` performs a decref, and moving the `tracking_ptr_base` moves the pointer from the source to the destination, decref'ing any existing pointer in the destination. (The responsibility to decref the pointer moves from the source to the destination.)

By building on top of `std::unique_ptr` with a custom deleter, we get cleanup and move implementations for free.

Okay, I think this mostly wraps up the development of a tracking pointer implementation. I have no idea if it is any use, but it was a nice exercise trying to figure out how to implement it.

(Okay, there's a follow-up I had promised to write after the main series is over. So there's at least one more part to go.)