Thoughts on creating a tracking pointer class, part 14: Nonthrowing moves with the shared tracking pointer

devblogs.microsoft.com/oldnewthing/20250828-00/?p=111524

August 28, 2025

So far, we've been working on an alternate design for tracking pointers, but we found that it had the unfortunate property of having potentially-throwing constructors and move assignment operations.

We can make these operations non-throwing by removing the need for a trackable object always to have a ready-made tracker. Instead, we can create a tracker on demand the first time somebody asks to track it. The exception doesn't go away, but it defers it to the time a tracking pointer is created. This is arguably a good thing because it makes tracking pointers "pay for play": You don't allocate a tracker until somebody actually needs it.

```
template<typename T>
struct trackable_object
{
    trackable_object() noexcept = default;
    ~trackable_object()
    {
        set_target(nullptr);
    }

    // Copy constructor: Separate trackable object
    trackable_object(const trackable_object&) noexcept :
        trackable_object()
    { }

    // Move constructor: Transfers tracker
    trackable_object(trackable_object&& other) noexcept :
        m_tracker(other.transfer_out()) {
        set_target(owner());
    }

    // Copying has no effect on tracking pointers
    trackable_object&
        operator=(trackable_object const&) noexcept
    {
        return *this;
    }

    // Moving abandons current tracking pointers and
    // transfers tracking pointers from the source
    trackable_object&
        operator=(trackable_object&& other) noexcept {
        set_target(nullptr);
        m_tracker = other.transfer_out();
        set_target(owner());
        return *this;
    }

    tracking_ptr<T> track() /* noexcept */ {
        ensure_tracker();
        return { m_tracker };
    }

    tracking_ptr<const T> track() const /* noexcept */ {
        ensure_tracker();
        return { m_tracker };
    }

    tracking_ptr<const T> ctrack() const /* noexcept */ {
        ensure_tracker();
        return { m_tracker };
    }

private:
    std::shared_ptr<T*> mutable m_tracker;

    T* owner() const noexcept {
        return const_cast<T*>(static_cast<const T*>(this));
    }

    void ensure_tracker() const
    {
        if (!m_tracker)
        {
            m_tracker = std::make_shared<T*>(owner());
        }
    }

    std::shared_ptr<T*> transfer_out()
    {
        return std::move(m_tracker);
    }

    void set_target(T* p)
    {
        if (m_tracker)
        {
            *m_tracker = p;
        }
    }
};
```

We make the `m_tracker` mutable because `ensure_tracker()` might be asked to create a tracker on demand from a const reference.

Creating the tracker on demand removes the exception from the default constructor, the move and copy constructors, and the move and copy assignments. The potentially-throwing behavior moves to the `track()` and `ctrack()` methods, but that can be sort of justified on the principle of “pay for play”.

Now, if you look more closely at what we have, you may notice that the `shared_ptr` is overkill. We don’t use weak pointers, and all of our operations are single-threaded, so the atomic memory barriers inside the `shared_ptr` operations are not necessary. We’ll create a “limited-use single-threaded” version of the `shared_ptr` next time.