Thoughts on creating a tracking pointer class, part 12: A shared tracking pointer

devblogs.microsoft.com/oldnewthing/20250826-00/?p=111506

August 26, 2025

The tracking pointer designs we've been using so far have had *O*(*n*) complexity on move, where *n* is the number of outstanding tracking pointers. But we can reduce this to *O*(1) by the classic technique of introducing another level of indirection.

What we can do is give every trackable object a single `shared_ptr<T*>` (which we call the "tracker"), which is shared with all tracking pointers. That way, when the object is moved, we can update that single `shared_ptr<T*>`, and that updates the pointer for all the tracking pointers.

```
template<typename T> struct trackable_object;

// struct tracking_node { ... };
template<typename T>
struct tracking_ptr_base
{
    tracking_ptr_base() noexcept = default;

private:
    friend struct trackable_object<T>;
    tracking_ptr_base(std::shared_ptr<T*> const& ptr) noexcept :
        m_ptr(ptr) { }

protected:
    std::shared_ptr<T*> m_ptr;
};

template<typename T>
struct tracking_ptr : tracking_ptr_base<std::remove_cv_t<T>>
{
private:
    using base = tracking_ptr_base<std::remove_cv_t<T>>;
    using Source = std::conditional_t<std::is_const_v<T>,
        base, tracking_ptr<std::remove_cv_t<T>>>;

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

    using base::base;
    tracking_ptr(Source const& other) : base(other) {}
    tracking_ptr(Source&& other) : base(std::move(other)) {}

    tracking_ptr& operator=(Source const& other) {
        static_cast<base&>(*this) = other;
        return *this;
    }
    tracking_ptr& operator=(Source&& other) {
        static_cast<base&>(*this) = std::move(other);
        return *this;
    }
};
```

The tracking pointer (via the tracking pointer base) holds a copy of the tracker shared pointer. The small catch here is that the tracker `m_ptr` might be null if the tracking pointer was default-constructed or has been moved-from, so the `get` method needs to check for a non-null pointer before dereferencing 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 {
        return { m_tracker };
    }

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

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

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

    std::shared_ptr<T*> new_tracker()
    {
        return std::make_shared<T*>(owner());
    }

    std::shared_ptr<T*> transfer_out()
    {
        return std::exchange(m_tracker, new_tracker());
    }

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

    std::shared_ptr<T*> m_tracker = new_tracker();
};
```

The trackable object starts out with a new tracker that points to the newly-constructed object. On destruction, the trackable object nulls out the backpointer in the tracker, which causes any existing tracking pointers to expire.

As with our other trackable object implementations, copying a trackable object has no effect on the tracker, and moving it transfers the tracker to the new object, abandoning any existing tracker. When we move the tracker to the new object, we need to leave a fresh (not-yet-shared-with-anybody) tracker behind so that the moved-from object is still trackable if anybody asks.

The helper method `new_tracker()` makes a fresh tracker that tracks the current object. The helper method `transfer_out()` relinquishes the current tracker (presumably so it can be given to the moved-to object) and sets up a fresh new tracker.

Although this improves the complexity of moving a trackable object to constant time, the requirement that `m_tracker` be non-empty means that the constructors and the move-assignment operators are now throwing, because `new_tracker()` could fail.

So now we have to look at whether the inability to create a new tracker could cause us to violate our invariants.

An exception in the constructors doesn't affect our invariants because we simply decided not to exist at all.

An exception in the move assignment operator is more troublesome. If `transfer_out()` fails, we have already disconnected the trackers, so a failure to transfer out causes existing tracking pointers to the destination to expire. This violates the strong exception guarantee, which says that if an exception occurs, the object remains unchanged.

We'll fix this next time.