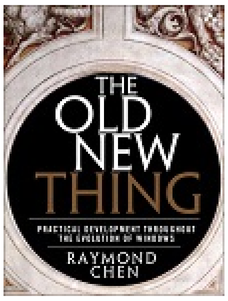
Thoughts on creating a tracking pointer class, part 3: Using a std::vector

devblogs.microsoft.com/oldnewthing/20250813-00/?p=111459

August 13, 2025



Last time, we tried to build our “tracking pointers” out of nodes in a `std::list`, but we found that the `std::list` was not expressive enough to support the operations we needed.[1]

So instead of a `std::list` of tracking pointer nodes, we can try using a vector of pointers.

```
template<typename T>
struct tracking_ptr
{
    ~tracking_ptr() {
        if (m_object) {
            trackable()->detach(this);
        }
    }

    tracking_ptr() noexcept : m_object(nullptr) {}

    ⟦ more to come ⟧

    T* get() const noexcept { return m_object; }

private:
    friend struct trackable_object<T>;

    trackable_object<T>* trackable() noexcept {
        return m_object;
    }

    tracking_ptr(T* object) : m_object(object) {
        trackable()->attach(this);
    }

    T* m_object;
};
```

I added a helper method `trackable()` to avoid having to type `m_object->trackable_object<T>::` all over the place.

This time, a `tracking_ptr` just wraps a raw pointer. We will rely on the trackable object to update the pointer as the object moves or destructs.

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

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

    // Move constructor: Transfers trackers
    trackable_object(trackable_object&& other) :
        m_trackers(std::move(other.m_trackers))) {
        update_trackers(outer());
    }

    ⟦ more to come ⟧

    tracking_ptr<T> track() {
        return { outer() };
    }

private:
    friend struct tracking_ptr<T>;

    std::vector<tracking_ptr<T>*> m_trackers;

    T* outer() noexcept
    { return static_cast<T*>(this); }

    void update_trackers(T* p) noexcept
    {
        for (auto& tracker : m_trackers) {
            tracker.m_object = p;
        }
    }

    void attach(tracking_ptr<T>* ptr) {
        m_trackers.push_back(ptr);
    }

    auto find(tracking_ptr<T>* ptr) noexcept {
        return std::find(
            m_trackers.begin(), m_trackers.end(), ptr));
    }

    void detach(tracking_ptr<T>* ptr) noexcept {
        m_trackers.erase(find(ptr));
    }
};
```

To create a tracking pointer, we attach it to the trackable object. The trackable object attaches the new tracking pointer by recording it in its vector of active tracking pointers. Note that we have to do the attach from inside the constructor because it is only then that we know what the `this` pointer is. We ran into this problem earlier when we looked at how to construct nodes of a hand-made list.

When the tracking pointer destructs, we detach it by erasing it from the vector.

When a trackable object is move-constructed, it takes over the vector of tracking pointers from the original object, and then it updates all of those pointers to point to the new object.

Okay, now to worry about assignment.

We had previously decided that on assignment, any tracking pointers to the assigned-to object expire because the contents have been obliterated.

```
trackable_object& operator=(trackable_object const& other)
{
    m_trackers.clear();
}

trackable_object& operator=(trackable_object && other)
{
    m_trackers = std::move(other.m_trackers);
    update_trackers(outer());
}
```

If we want to preserve existing tracking pointers to the assigned-to object, then things are a little different.

```
trackable_object& operator=(trackable_object const& other)
{
    // preserve destination m_trackers
}

trackable_object& operator=(trackable_object && other)
{
    m_trackers.reserve(m_trackers.size() + other.m_trackers.size());
    other.update_trackers(outer());
    m_trackers.insert(m_trackers.end(),
                      other.m_trackers.begin(),
                      other.m_trackers.end());
    other.m_trackers.clear();
}
```

For copying, we simply do nothing. All tracking pointers stay where they are. For moving, we merge the moved-from object's tracking pointers with the moved-to object's tracking pointers. The order of operations here is tricky because we need to do all potentially-throwing operations before doing any irreversible operations. So we reserve space in the `m_trackers`, which ensures that the future `insert` will not throw. Only then can we update the tracking pointers in the moved-from object to point to the moved-to object, and then transfer them to our `m_trackers`.

Okay, now we have to go back and write the move and copy constructors and assignments for the `tracking_ptr`.

Copying a tracking pointer is just registering the copy with the trackable object.

```
tracking_ptr(tracking_ptr const& other) :
    tracking_ptr(other.m_object) {}
```

We do have to teach the delegated-to constructor about the possibility of being asked to track nothing.

```
tracking_ptr(T* object) : m_object(object) {
    if (m_object) {
        trackable()->attach(this);
    }
}
```

We don't have to write a move constructor for the tracking pointer if we don't want to make any promises about the state of a moved-from tracking pointer. The move can just be a copy.

But if we want to say that moving a tracking pointer leaves an expired tracking pointer behind, then we need to clear out the old one.

```
tracking_ptr(tracking_ptr && other) :
    tracking_ptr(std::exchange(other.m_object, nullptr)) {
    if (m_object) {
        trackable()->attach(this);
        trackable()->detach(&other);
    }
}
```

The order of operations is important here. If we had detached first, and then the attach failed, we would not restore the source object to its original state, thus failing the strong exception guarantee.

But instead of attaching and detaching, we could just reuse the same entry in the tracking table. This also removes a possible exception.

```
template<typename T>
struct tracking_ptr
{
    ⟦ ... ⟧

    tracking_ptr(tracking_ptr && other) :
        tracking_ptr(std::exchange(other.m_object, nullptr)) {
        if (m_object) {
            trackable()->replace(&other, this);
        }
    }

    ⟦ ... ⟧
};

template<typename T>
struct trackable_object
{
    ⟦ ... ⟧

private:
    ⟦ ... ⟧

    void update(tracking_ptr<T>* from,
                tracking_ptr<T>* to) noexcept {
        *find(from) = to;
    }

    ⟦ ... ⟧
};
```

There are some downsides to this design. For one thing, the cost of moving or destructing a tracking pointer is linear in the number of tracking pointers that are tracking the same object. Perhaps more concerning is that the use of `std::vector` creates the possibility for low-memory exceptions, so callers have to be prepared for tracking pointer operations to fail.

We'll look at another design next time.

¹ Turns out I was wrong. Commenter LB pointed out that you can use the `splice` method to move nodes between lists. I'll add an addendum to this series that explores the splicer approach.