Thoughts on creating a tracking pointer class, part 4: Using a circular doubly linked list

devblogs.microsoft.com/oldnewthing/20250814-00/?p=111482

August 14, 2025

Our previous attempt to create a tracking pointer class had the downside of incurring a potential memory allocation when a tracking pointer is copied or moved. We can get rid of this allocation by making the tracking pointer itself provide the necessary memory. To do this, we want to create a doubly-linked list (doubly linked to keep operations at *O*(1)) where the tracking pointers are themselves nodes in the list. We saw earlier that the C++ standard library `std::list` doesn't give access to the nodes. So I guess we'll just do it ourselves?

We start with a `tracking_node` which is just a node in a circularly linked list.

```
struct tracking_node
{
    tracking_node* next;
    tracking_node* prev;

    struct as_solo{};
    struct as_join{};
    struct as_displace{};

    tracking_node(as_solo) noexcept :
        next(this), prev(this) {}

    tracking_node(as_join, tracking_node& other) noexcept :
        next(std::exchange(other.next, this)),
        prev(&other) {
        relink();
    }

    tracking_node(as_displace, tracking_node& other) noexcept :
        next(std::exchange(other.next, &other)),
        prev(std::exchange(other.prev, &other)) {
        relink();
    }

    ~tracking_node() { unlink(); }

    tracking_node(tracking_node const&) = delete;
    void operator=(tracking_node const&) = delete;

    void unlink() noexcept {
        next->prev = prev;
        prev->next = next;
    }

    void relink() noexcept {
        next->prev = this;
        prev->next = this;
    }

    void reset() noexcept {
        prev = next = this;
    }

    void disconnect() noexcept {
        unlink();
        reset();
    }

    void join(tracking_node& other) noexcept {
        if (this != &other) {
            unlink();
            next = std::exchange(other.next, this);
            prev = &other;
            relink();
        }
    }
```

```
    void displace(tracking_node& other) noexcept {
        if (this != &other) {
            unlink();
            next = std::exchange(other.next, &other);
            prev = std::exchange(other.prev, &other);
            relink();
        }
    }
};
```

The `tracking_node` is the element that represents a node in the linked list. There are three constructors, depending on how you want the node to be related to other nodes.

- “solo”: The node forms its own circularly linked list.
- “join”: The node joins the circularly linked list that another node already belongs to.
- “displace”: The node joins the circularly linked list that another node already belongs to, and then kicks that other node out of the list. It’s called “displace” because the new node displaces the old node from its spot in the circularly linked list.

We don’t need to split “join” into “join_before” and “join_after” because the order of elements in the circularly linked list is not significant.

In addition to constructing a node in solo, join, or displace mode, you can also take an existing node and tell it to leave whatever circular linked list it currently belongs to and go solo or join or displace another node.[1]

Now we can build a `tracking_ptr` out of the tracking node.

```
template<typename T> struct trackable_object;

template<typename T>
struct tracking_ptr : private tracking_node
{
    T* get() const { return tracked; }

    tracking_ptr() noexcept :
        tracking_node(as_solo{}),
        tracked(nullptr) {}

    tracking_ptr(tracking_ptr& other) noexcept :
        tracking_node(as_join{}, other),
        tracked(other.tracked) {}

    ~tracking_ptr() = default;

    tracking_ptr& operator=(tracking_ptr& other) noexcept {
        tracked = other.tracked;
        tracking_node::join(other);
        return *this;
    }

    tracking_ptr& operator=(tracking_ptr&& other) noexcept {
        tracked = std::exchange(other.tracked, nullptr);
        tracking_node::displace(other);
        return *this;
    }

private:
    friend struct trackable_object<T>;

    static tracking_node& trackers(T* p) noexcept {
        return p->trackable_object<T>::m_trackers;
    }

    tracking_ptr(T* p) noexcept :
        tracking_node(as_join{}, trackers(p)),
        tracked(p) { }

    T* tracked;
};
```

A freshly constructed tracking pointer tracks nothing: It is a circular linked list with itself as the only element, and it holds no pointer.

Copying a tracking pointer joins the new tracking pointer to the circular linked list of the source and tracks the same object as the source.

Moving a tracking pointer makes the new tracking pointer displace the old tracking pointer in the circular linked list, the new tracking pointer tracks what the old tracking pointer tracked, and the old tracking pointer no longer tracks anything.

Copy-assigning and move-assigning a tracking pointer is the same as copy-constructing or move-constructing, except that we join or displace the other node using an existing node rather than a newly-constructed on. Note that we don't need to do `if (this != &other)` because the assignments to `tracked` end up being nops if we are self-assigning.

The last constructor is the one that gets the party started: It's the one that the `trackable_object<T>` uses to create the first tracking pointer. We create the new tracking pointer by joining the trackable object's anchor node (which we'll see soon) and tracking the object `p`.

Destroying a tracking pointer unlinks it from its tracking list. The `tracking_node` destructor already does this, so we have nothing special to do.

Finally, we get to the `trackable_object<T>`.

```
template<typename T>
struct trackable_object
{
    trackable_object() noexcept :
        m_trackers(tracking_node::as_solo{}) {}

    ~trackable_object()
        { set_target(nullptr); }

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

    // Move constructor: Transfers trackers
    trackable_object(trackable_object&& other) noexcept :
        m_trackers(tracking_node::as_displace{},
                   other.m_trackers) {
        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_trackers.displace(other.m_trackers);
        set_target(owner());
        return *this;
    }

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

private:
    friend struct tracking_ptr<T>;

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

    tracking_node m_trackers;

    void set_target(T* p) noexcept
    {
        for (tracking_node* n = m_trackers.next;
             n != &m_trackers; n = n->next) {
            static_cast<tracking_ptr<T>*>(n)->tracked = p;
        }
    }
};
```

Constructing a trackable object from scratch or copying from another object gives you an empty list of trackers. Moving from another object transfers the tracking list to the new object and becomes the target of all the tracking pointers.

The copy assignment operator is a nop because copying doesn't affect tracking pointers.

The move assignment operator first orphans all of its existing tracking pointers by telling them that they no longer have a target. Then it takes over the tracking pointers of the source object and sets itself as the target of those taken-over tracking pointers.

Creating a tracking pointer is done by constructing a `tracking_ptr` from a pointer to the tracked object. That constructor creates a tracking pointer which joins the other tracking pointers by adding itself to the circular linked list anchored at `m_trackers`.

The last interesting method is `set_target` which updates the target pointer in all of the tracking pointers.

Now we can take our trackable object out for a spin.

```
struct S : trackable_object<S>
{
    S() = default;
    S(S const&) = default;
    S(S&&) = default;
    S& operator=(S const&) = default;
    S& operator=(S&&) = default;
};

void test()
{
    S s1;
    // make a tracking pointer for s1
    auto ptr = s1.track();

    S s2 = std::move(s1);
    // ptr now points to s2

    auto ptr2 = ptr;
    // ptr2 also points to s2

    auto ptr3 = s1.track();
    // ptr3 tracks s1

    s1 = std::move(s2);
    // ptr now tracks s1
    // ptr2 now tracks s1
    // ptr3 now tracks nothing
}
```

One nice thing about this design is that all the tracking pointer operations are noexcept because they never allocate memory. A downside is that when an object moves, we have to update all of the tracking pointers that were tracking the moved-from and moved-to

objects, so the cost is proportional to the number of tracking pointers involved.

There is a problem you may have noticed: You can't copy from a tracking pointer because we need to update the source's links so that the destination pointer can join the circular linked list. We'll fix that next time.

[1] If we were feeling fancy, we could be tempted to write `disconnect`, `join`, and `transfer` by destructing the object and then re-constructing it in place.

```
void disconnect() noexcept {
    this->~tracking_node();
    new (this) tracking_node(as_solo{});
}

void join(tracking_node& other) noexcept
{
    if (this != &other) {
        this->~tracking_node();
        new (this) tracking_node(as_join{}, other);
    }
}

void transfer(tracking_node& other) noexcept
{
    if (this != &other) {
        this->~tracking_node();
        new (this) tracking_node(as_transfer{}, other);
    }
}
```

For this trick to work, many requirements must be met. A practical one is that the constructor is `noexcept`, so we don't have to worry about getting stuck after we destruct the old object and fail to construct the new one. And one requirement of [basic.life]/8 regarding the creation of an object memory formerly occupied by another object also requires that none of the object's non-static data members is `const` or a reference. But unfortunately, we run afoul of another requirement, which is that the objects involved be the *most derived object*. This doesn't work for us because the `tracking_node` is a base class of `tracking_ptr`, which renders it ineligible for "memory recycling" tricks like this. We would have to use `std::launder` to access the new object, but that's so cumbersome it's not worth it.