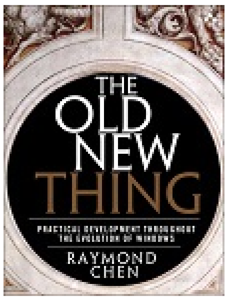
Thoughts on creating a tracking pointer class, part 2: Using a std::list

devblogs.microsoft.com/oldnewthing/20250812-00/?p=111454

August 12, 2025



Last time, we worked out how we want a proposed "tracking pointer" to work. Now we can try implementing it.

One idea is to give each trackable object a list of all the tracking pointers that are tracking it, so it can update them as the object moves.

The tracking pointer is a pointer to a `std::list` node (in the form of an iterator to it). This works because list iterators are invalidated only when the list items itself is removed from the list. Other modifications to the list do not invalidate iterators. The node itself is just a pointer to the tracked object.

When a tracking pointer destructs, it unregisters itself from the list of active tracking pointers.

We start with the tracking pointer:

```
template<typename T>
using tracker_list = std::list<T*>;

template<typename T>
using tracker_node_ptr = tracker_list&ltT>::iterator;

template<typename T>
struct tracking_ptr
{
    ~tracking_ptr() {
        if (m_node->p) {
            m_node->p->trackable_object<T>::detach(this);
        }
    }

    ⟦ more to come ⟧

    T* get() const noexcept { return m_node->p; }

private:
    friend struct trackable_object<T>;

    tracking_ptr(tracker_node_ptr<T> const& node) noexcept :
        m_node(node) {}

    tracker_node_ptr<T> node;
};
```

One thing to watch out for is the case where the object that derives from `trackable_object` has its own `detach` method. We want the `trackable_object`‘s `detach` method, so we apply an explicit qualifier to find the correct `detach` method.

Meanwhile, this is what the tracked object has to do:

```
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 tracking_ptr<T>(
            m_trackers.push_back(outer()));
    }

private:
    friend struct tracking_ptr<T>;

    tracker_list<T> m_trackers;

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

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

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

When a trackable object is constructed normally, we do nothing special, so it has an empty list of trackers.

When a trackable object is copied, we override the implicit copy constructor (which would by default copy the list) by telling it that we want to construct it fresh, which means that the tracking list is empty. The copy has a separate trackable lifetime.

When a trackable object is moved, then the new object adopts all the tracking pointers that had tracked the old object by moving them into its own `m_trackers`. It then updates all of those tracking pointers to point to the new object rather than the old object.

To create a new tracking pointer, we add a node to the linked list (which holds a pointer to the tracked object) and wrap that node inside a `tracking_ptr`.

When a trackable object destructs, we update all of the tracking pointers so that they produce `nullptr`. This lets them know that the tracking pointer has expired.

But wait, that’s not going to work: The tracking pointer holds an iterator to the linked list node, but the node is about to be destructed when the list destructs.

Okay, so our tracking pointer can’t be a list node, because those disappear when the list destructs. The `std::list` does not have an `extract` methods, so there’s no way to extend a node’s lifetime beyond that of the list to which it belongs.

We’ll continue the story next time.