Thoughts on creating a tracking pointer class, part 6: Non-modifying trackers

devblogs.microsoft.com/oldnewthing/20250818-00/?p=111486

August 18, 2025

Let's add non-modifying trackers to our tracking pointers implementation. That is, a tracking pointer that gives you only read-only access to the tracked object.

The idea is that tracking pointers and non-modifying tracking pointers all share the same circular doubly linked list. The only difference is what kind of pointer comes out of `get`.

First, we'll introduce a nickname `MT` meaning "mutable (non-const) version of `T`" and have `tracking_ptr` use it instead of `T`, with the exception of the `get()` method, which returns the (possibly-const) original type `T`. (Actually, we may as well also remove volatility while we're at it. Completing volatility support will be left as a pointless exercise.)

```
template<typename T>
struct tracking_ptr : private tracking_node
{
private:
    using MT = std::remove_cv_t<T>;

public:

    T* get() const { return tracked; }

    ⟦ ... other public members as before ... ⟧

private:
    friend struct trackable_object<MT>;

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

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

    ⟦ ... ⟧

    MT* tracked;
};
```

Next, we add a `ctrack()` method to the `trackable_object` to produce a non-modifying tracking pointer.

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

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

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

private:
    friend struct tracking_ptr<T>;
    friend struct tracking_ptr<const T>;

    ⟦ ... ⟧
};
```

Okay, now we can have an object give away a non-modifying tracking pointer to itself by using `ctrack()` instead of `track()`.

But wait, this code is wrong.

We'll continue our investigation next time.