# Thoughts on creating a tracking pointer class, part 7: Non-modifying trackers, second try

devblogs.microsoft.com/oldnewthing/20250819-00/?p=111488

August 19, 2025

Last time, we tried to add non-modifying trackers to our tracking pointers implementation. I noted at the end that our attempt was wrong.

The problem is in the code we didn't change:

```
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;
    }
}
```

The `static_cast` is a downcast from a `tracking_node` to its derived `tracking_ptr<T>`. But the derived class might not be `tracking_ptr<T>`! It could be a `tracking_ptr<const T>`.

To fix this, we need to use a consistent type for the derived class. We can do this by renaming our original `tracking_ptr` to `tracking_ptr_base`, which will serve as the consistent derived class, and then move the `get()` method to a `tracking_ptr` that is derived from `tracking_ptr_base`.

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

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

    tracking_ptr_base(tracking_ptr_base const& other) noexcept :
        tracking_node(copy_node(other)),
        tracked(other.tracked) { }

    ~tracking_ptr_base() = default;

    tracking_ptr_base& operator=(tracking_ptr_base const& other) noexcept {
        tracked = other.tracked;
        if (tracked) {
            join(trackers(tracked));
        } else {
            disconnect();
        }
        return *this;
    }

    tracking_ptr_base& operator=(tracking_ptr_base&& 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_node copy_node(tracking_ptr_base const& other) noexcept
    {
        if (other.tracked) {
            return tracking_node(as_join{},
                                 trackers(other.tracked));
        } else {
            return tracking_node(as_solo{});
        }
    }

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

protected:
    T* tracked;
};
```

```
template<typename T>
struct tracking_ptr : tracking_ptr_base<std::remove_cv_t<T>>
{
public:
    T* get() const { return this->tracked; }

    using tracking_ptr::tracking_ptr_base::
                        tracking_ptr_base;
};

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

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

    ⟦ ... ⟧

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

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 still requires that the original object be itself mutable. But if all you have is a const reference to a trackable object, surely you should be allowed to create a non-modifying tracking pointer to it, right?

We'll do that next time.