Thoughts on creating a tracking pointer class, part 11: Repairing assignment

devblogs.microsoft.com/oldnewthing/20250825-00/?p=111497

August 25, 2025

Last time, we made sure that tracking pointers to const objects couldn't be converted into tracking pointers to non-const objects, but I noted that fixing this introduced a new problem.

We fixed the problem by introducing two new constructors that allow construction of either a `tracking_ptr<const T>` or `tracking_ptr<T>` from `tracking_ptr<T>`. If the destination is a `tracking_ptr<T>`, then the copy or move construction from `tracking_ptr<T>` merely overrides the copy or move construction inherited from the base class, so there is no redeclaration conflict.

The problem is that in the case of `tracking_ptr<T>`, the new constructors are copy and move constructors since they construct from another instance of the same type. And if you declare a move constructor, then the copy and move assignment operators are implicitly declared as deleted.

So we need to bring them back.

```
template<typename T>
struct tracking_ptr : tracking_ptr_base<std::remove_cv_t<T>>
{
private:
    using base = tracking_ptr_base<std::remove_cv_t<T>>;
    using MP = tracking_ptr<std::remove_cv_t<T>>;

public:
    T* get() const { return this->tracked; }

    using base::base;
    tracking_ptr(MP const& other) : base(other) {}
    tracking_ptr(MP&& other) : base(std::move(other)) {}

    tracking_ptr& operator=(tracking_ptr const&) = default;
    tracking_ptr& operator=(tracking_ptr&&) = default;
};
```

But now we have the reverse problem: If you declare a copy or move assignment, then the copy and move *constructors* are implicitly declared as deleted.

So we have to bring those back too:

```
template<typename T>
struct tracking_ptr : tracking_ptr_base<std::remove_cv_t<T>>
{
private:
    using base = tracking_ptr_base<std::remove_cv_t<T>>;
    using MP = tracking_ptr<std::remove_cv_t<T>>;

public:
    T* get() const { return this->tracked; }

    using base::base;
    tracking_ptr(tracking_ptr const& other) : base(other) {}
    tracking_ptr(tracking_ptr&& other) : base(std::move(other)) {}
    tracking_ptr(MP const& other) : base(other) {}
    tracking_ptr(MP&& other) : base(std::move(other)) {}

    tracking_ptr& operator=(tracking_ptr const&) = default;
    tracking_ptr& operator=(tracking_ptr&&) = default;
};
```

And now we have the double-definition problem we saw last time: In the case of `tracking_ptr<T>` where `T` is non-const, we have two declarations for the same copy constructor (and two for the same move constructor), which is not allowed.

There's another problem: In the case of assigning a `tracking_ptr<const T>` to a `tracking_ptr<T>`, we actually perform it in two steps: First we convert the `tracking_ptr<const T>` to a `tracking_ptr<T>`, and then we assign the `tracking_ptr<T>` to its destination. This creates a temporary `tracking_ptr<T>` that gets linked into the chain, and then unlinked. Can we avoid that inefficiency and just assign it directly?

It turns out the same trick works for both problems.

```
template<typename T>
struct tracking_ptr : tracking_ptr_base<std::remove_cv_t<T>>
{
private:
    using base = tracking_ptr_base<std::remove_cv_t<T>>;
    using Source = std::conditional_t<std::is_const_v<T>,
        base, tracking_ptr<std::remove_cv_t<T>>>;

public:
    T* get() const { return this->tracked; }

    using base::base;
    tracking_ptr(Source const& other) : base(other) {}
    tracking_ptr(Source&& other) : base(std::move(other)) {}

    tracking_ptr& operator=(Source const& other) {
        static_cast<base&>(*this) = other;
        return *this;
    }
    tracking_ptr& operator=(Source&& other) {
        static_cast<base&>(*this) = std::move(other);
        return *this;
    }
};
```

If creating a `tracking_ptr<const T>`, then we accept assignment or construction from either `tracking_ptr<T>` or `tracking_ptr<const T>`. But if creating a `tracking_ptr<T>` where `T` is non-const, then we accept assignment or construction only from another `tracking_ptr<T>`. This is expressed in the definition of `Source`, which says that tracking pointers to const things can accept the base type, which means that it will accept any type of tracking pointer to that thing (either to a const or non-const thing). But if it's a tracking pointer to a non-const thing, then it accepts only tracking pointers to the same non-const thing.

We also have to write out the copy and move assignment operators. We could use `= default` in the case where the `Source` is equal to `tracking_ptr<T>`, but if dealing with a tracking pointer to a const thing, the `Source` is the `base`, and the compiler doesn't know how to default-assign that. So we just write it out explicitly, which works for both cases.

So are we done? I guess.

But wait.

Recall that the complexity of moving a trackable object is linear in the number of tracking pointers because we have to update all the tracking pointers to point to the new location of the moved object. But we can get the cost down to *O*(1) if we are willing to make some concessions. We'll look at this alternate design next time.