Thoughts on creating a tracking pointer class, part 10: Proper conversion

devblogs.microsoft.com/oldnewthing/20250822-00/?p=111494

August 22, 2025

Last time, we added the ability to convert tracking pointers to non-const objects into tracking pointers to const objects, but we noted that there's a problem.

The problem is that our change accidentally enabled the reverse conversion: From const to non-const.

We want to be able to convert non-const to const, but not vice versa, so let's require the source to be non-const.

```
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)) {}
};
```

The conversion operators now require a tracking pointer to a non-const object (which to reduce typing we call `MP` for *mutable pointer*). The const-to-const version is inherited from the base class.

Inheriting the constructors is particularly convenient because it avoids redefinition conflicts. If we didn't have inherited constructors, we would have started with

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

public:
    tracking_ptr(tracking_ptr const& other);
    tracking_ptr(MP const& other);

    tracking_ptr(tracking_ptr&& other);
    tracking_ptr(MP&& other);
};
```

But this doesn't work with `tracking_ptr<Widget>` because you now have pairs of identical constructors since the "non-const-to-`T`" versions are duplicates of the copy and move constructor when `T` is itself non-const. Substituting `T` = `Widget`, we get

```
template<typename T>
struct tracking_ptr
{
private:
    using MP = tracking_ptr<Widget>;

public:
    tracking_ptr(tracking_ptr<Widget> const& other);
    tracking_ptr(tracking_ptr<Widget> const& other);

    tracking_ptr(tracking_ptr<Widget>&& other);
    tracking_ptr(tracking_ptr<Widget>&& other);
};
```

And the compiler complains that you declared the same constructor twice. You would have to use SFINAE to remove the second one.

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

public:
    tracking_ptr(tracking_ptr const& other);

    template<typename = std::enable_if<std::is_const_v<T>>>
    tracking_ptr(MP const& other);

    tracking_ptr(tracking_ptr&& other);

    template<typename = std::enable_if<std::is_const_v<T>>>
    tracking_ptr(MP&& other);
};
```

On the other hand, redeclaring an inherited constructor overrides it, so we can just declare our constructors and not worry about conflicts.

But wait, our attempt to fix this problem introduced a new problem. We'll look at that next time.