Thoughts on creating a tracking pointer class, part 1: Concept art

devblogs.microsoft.com/oldnewthing/20250811-00/?p=111451

August 11, 2025

Suppose you have a C++ object, and you want to be able to create a “tracking pointer” that points to the object and follows it as it moves.

```
// Concept art - final product may differ in significant ways

struct Widget : trackable_object<Widget>
{
    ⟦ ... ⟧
};

Widget w;

tracking_ptr<Widget> p(w);

assert(p.get() == &w);
p->Toggle(); // same as w.Toggle()

Widget moved = std::move(w);
assert(p.get() == &moved);
p->Toggle(); // same as moved.Toggle()
```

The first thing to note is that this is inherently a single-threaded concept. You can’t do this in a multithreaded way because you have no way to prevent an object from moving while you are accessing it.

Okay, so let’s assume that all uses (including destruction) of the `tracking_ptr` and the tracked object are restricted to a single thread.

It’s clear that this will require cooperation from the tracked object, so it can update all the existing tracking pointers when it is moved-from.

Before we come up with an implementation, let’s figure out what the rules are.

When an object is constructed from scratch, there are no initial tracking pointers to it.

When an object is destructed, then all pointers that were tracking it become expired.

When an object is copy-constructed from another object, there are no initial tracking pointers to it. Any tracking pointers that point to the source object still point to the source object.

When an object is move-constructed from another object, any tracking pointers that pointed to the source object now track the new object. Nothing is tracking the source object any more.

When an object is copy-assigned, then any pointers that were tracking the source object continue to track the source object. But what about any pointers that were tracking the destination object? Interesting question. Let's keep going and see what other questions come up.

When an object is move-assigned, any pointers that were tracking the source object now track the destination object, and the source object is now untracked. But what about pointers that were tracking the destination object? It's that same question again.

How you choose to resolve the question about pointers that were tracking an overwritten object depends on how you conceptualize the assignment operator. One way of thinking about it is that the assignment operator transfers the information to the destination object, but the destination object retains its identity. Somebody moved all their furniture into your apartment and made it look just like their old apartment. But it's still your apartment!

Another way of thinking about it is that the assignment operator also obliterates the old identity, as if the assignment was just an optimized version of "destroy the destination object, and then move-construct a new object in its place." If somebody take your bag of groceries and throws out all your groceries and puts their groceries in it, then it's not really your bag of groceries any more. The important thing about the bag of groceries was not the bag, but the groceries!

The original problem formulation was to "follow an object as it moves", but that is technically nonsense in C++. In C++, objects don't move. Their *contents* move. Since we are tracking the contents on move-construction, it seems that the intent was to track the contents and not the object. So if new contents move into an existing object, the tracking pointers for the old contents should expire, since those contents are now gone. That's the model we'll use, though we'll make notes about how we could implement the alternate interpretation.

Bonus reading: Providing a stable memory address, a similar problem, but for the special case where there is only one tracking pointer.

Bonus chatter: The swap pattern sort of leans toward breaking any pre-existing tracking pointers to a moved-to object.

```
Widget widget; // leftover widget lying around
widget = std::move(a);
a = std::move(b);
b = std::move(widget);
```

This pattern takes a pre-existing `Widget` object, say one left over from an earlier step, and uses it as a temporary object in the swap pattern. If any leftover tracking pointers to `widget` continued to track it after `a` was moved into it, then those tracking pointers are now accidentally tracking the unrelated Widget `b`.

Now, that was a somewhat weak argument for orphaning tracking pointers to a moved-into object, but a much stronger case can be made by looking at methods like `std::vector::erase`: If you have a vector `v` of, say, two trackable objects, and then you do `v.erase(v.begin());` to erase the first element, this operation accomplishes the erasure by move-assigning the second element over the first element. But presumably you want erasing an element to orphan any tracking pointers to it, rather than having them start tracking the object the got moved into its reused memory.