# Thoughts on creating a tracking pointer class, part 5: Copying our tracking pointer

devblogs.microsoft.com/oldnewthing/20250815-00/?p=111484

August 15, 2025


Our previous attempt to create a tracking pointer class had the perhaps-undesirable property that copying a const tracking pointer is not permitted. This requirement arose because copying a tracking pointer causes the copy to be linked into the circular linked list of the source, and that means modifying the link pointers in the source. Can we work around this?

The secret here is realizing that it's not important where the copy gets inserted into the circular linked list. The order of items in the list is not significant. So don't use the source pointer as the insertion point for the copy. Instead, use the source's tracking anchor node, which is non-const.

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

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

private:
    ⟦ ... as before ... ⟧

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

    T* tracked;
```

Switching to the anchor node does have a complication that we need to check for a null pointer in the source before we try to get the target's trackers. If the source tracks a null pointer, then that means that we are copying an expired tracking pointer, so the copy should also be empty. We use the helper function trick to choose a constructor at runtime.

Note that this solution takes advantage of const/non-const aliasing. We are modifying a const object through a non-const path. This type of aliasing is normally a compiler's nightmare:

```
bool test(int const& v, int* p)
{
    auto old_value = v;
    *p = 42;
    return old_value == 99;
}
```

The compiler cannot optimize this to

```
bool test(int const& v, int* p)
{
    *p = 42;
    return v == 99;
}
```

because of the possibility that somebody called it like this:

```
int v = 99;
test(v, &v);
```

**Bonus chatter**: The Itanium processor had special instructions to allow compilers to perform this optimization in the mainline path, with a fallback if it turned out that `v == &p` after all.

```
        // no "alloc" needed - lightweight leaf function
        // on entry, r32 = address of v, r33 = p
        ld4.a   r31 = [r32]             // fetch v into r31 in advance
        mov     r30 = 42                // r30 = constant 42
        mov     r29 = 99 ;;             // r29 = constant 99
        st4     [r33] = r30 ;;          // write 42 to *p
        ld4.c.nc r31 = [r32]            // reload v if necessary
        cmp.eq  p6, p7 = r31, r29 ;;    // v == 99?
(p6)    mov     ret0 = 1                // result is 1 if true
(p7)    mov     ret0 = 0                // result is 0 if false
        br.ret.sptk.many rp             // return
```

Next time, we'll add a `ctrack` method for creating a tracking pointer that produces a `const T`. For the times you want to let somebody track an object, but not modify it.