# Thoughts on creating a tracking pointer class, part 13: Restoring the strong exception guarantee

devblogs.microsoft.com/oldnewthing/20250827-00/?p=111518

August 27, 2025

Last time, we created a tracking pointer class based on `std::shared_ptr`. We found a problem with the move-assignment operator: It didn't satisfy the strong exception guarantee:

```
    trackable_object&
        operator=(trackable_object&& other) {
        set_target(nullptr);
        m_tracker = other.transfer_out();
        set_target(owner());
        return *this;
    }
```

If an exception occurs after we set the target to `nullptr` we exit with all the tracking pointers expired, which violates the rule that an exception leaves the object state unchanged.

To fix this, we cannot make any irreversible changes until we have passed the point where the last exception could be raised. The exception occurs in the call to `new_tracker()` inside `transfer_out()`. When that happens, the exchange into `other.m_tracker` does not occur, so `m_tracker` is safely unchanged. So we just need to delay expiring the old tracking pointers until after we have successfully transferred out.

```
    trackable_object&
        operator=(trackable_object&& other) {
        auto inbound = other.transfer_out();
        set_target(nullptr);
        m_tracker = inbound;
        set_target(owner());
        return *this;
    }
```

We can code-golf this by using `std::exchange` to replace the `m_tracker` while saving the old value, and then updating the target of that tracker manually.

```
    trackable_object&
        operator=(trackable_object&& other) {
        auto old = std::exchange(m_tracker, other.transfer_out());
        *old = nullptr;
        set_target(owner());
        return *this;
    }
```

And another iteration of code golfing to inline the result:

```
    trackable_object&
        operator=(trackable_object&& other) {
        *std::exchange(m_tracker, other.transfer_out()) = nullptr;
        set_target(owner());
        return *this;
    }
```

We noted last time that the constructors are also potentially-throwing. Many C++ algorithms and classes are significantly more efficient if they know that move operations cannot throw, so making the move constructor and move assignment operator potentially-throwing could end up begin quite expensive. And you probably expect operations like `vector::insert` and `std::sort` to move elements rather than copy them. Furthermore, many collection operations (such as `vector::insert` and `vector::erase`) leave the vector in an "unspecified" state if a move assignment throws an exception.[1]

With the throwing move assignment operator, we have to be careful to consider the state of the trackable object if `transfer_out()` fails. In that case, we have already disconnected the trackers, so a failure to copy nevertheless breaks tracking pointers, which violates the strong exception guarantee.

To fix that, we don't abandon the old tracking pointers until we are sure we can get new ones.

Next time, we'll make the constructors and move-assignment operations non-throwing, though it comes at a cost.

[1] Another side effect is that it prevents trackable objects from being nothrow-swappable, since swapping is based on move operations. We could add a custom `swap` method and a custom overload of `std::swap`, but that also creates the onus on the derived class to

provide the same customizations on itself so that it can forward the methods into `trackable_object`.