More on harmful overuse of std::move

devblogs.microsoft.com/oldnewthing/20240603-00

June 3, 2024

Raymond Chen

Some time ago, I wrote about harmful overuse of `std::move`. Jonathan Duncan asked,

> Is there some side-effect or other reason I can't see return std::move(name); case isn't possible to elide? Or is this just a case of the standards missing an opportunity and compilers being bound to obey the standards?

In the statement `return std::move(name);`, what the compiler sees is `return f(...);` where `f(...)` is some mysterious function call that returns an rvalue. For all it knows, you could have written `return object.optional_name().value();`, which is also a mysterious function call that returns an rvalue. There is nothing in the expression `std::move(name)` that says, "Trust me, this rvalue that I return is an rvalue of a local variable from this very function!"

Now, you might say, "Sure, the compiler doesn't know that, but what if we made it know that?" Make the function `std::move` a magic function, one of the special cases where the core language is in cahoots with the standard library.

This sort of in-cahoots-ness is not unheard of. For example, the compiler has special understanding of `std::launder`, so that it won't value-propagate memory values across it, and the compiler has special understanding of memory barriers, so that it won't optimize loads and stores across them.

So why not add `std::move` to the list of functions that the compiler has special understanding of? Technically, this is already permitted by the standard, because the standard requires that any specialization of a templated standard library function "meets the standard library requirements for the original template," so you can't write a specialization of `std::move` that, say, returns a *copy* of the object. However, I think it's still legal for the specialization to send angry email to your boss[1] before returning the rvalue reference.

Okay, so we add a new clause to the standard that says that specializations of `std::move` are disallowed.

This does leave in the lurch alternate implementations of `std::move`. For example, the Windows Implementation Library (WIL) has its own implementation of `std::move` called `wistd::move`. It does this because some of the components that use WIL operate under a constraint that C++ exceptions are disallowed, which means that they cannot `#include <memory>`. But it would also mean that `wistd::move` is no longer a drop-in replacement for `std::move`: The compiler would recognize `std::move` as special, but not `wistd::move`.

Okay, so we tell those people, "Oh, stop being such a stick in the mud. Come on in, the water's fine! Use `std::move`!"

If we operated naïvely, we would say, "Sure you can return the `std::move` of a local variable, and we'll reuse the return value slot." But that would be wrong, because that would be move-constructing an object from another object that resides at the same address, which is not something that happens in normal C++, and I suspect that a lot of move constructors don't handle that case. (Not that I expect them to.)

So the C++ language would have to disavow the move constructor at all. It could say that if the `return` statement takes the form `return std::move(name)` where `name` is the name of a local variable eligible for NRVO, then the `std::move` may be elided.

And maybe to accommodate those people who are afraid of exception-infested waters, you could expand the rule to say that if the compiler can determine that the returned value is an rvalue to a local variable that is eligible for NRVO, then it can be rewritten as returning that local variable via NRVO (while still preserving any other observable behaviors of the relevant expression).

I mean, you *could* do this. Maybe you can even write up a proposal and see what the language committee thinks.

Oh wait, somebody already wrote that proposal! Stop Forcing `std::move` to Pessimize, which was presented to the C++ standard committee in November 2023, and the response was "Weak consensus, needs more work".

Bonus viewing: CppCon 2018: Arthur O'Dwyer "Return Value Optimization: Harder Than It Looks".

[1] More practical examples would be "doing performance logging" or "doing debug logging" rather than "sending angry email to your boss".