How can I force a copy of a C++ value?

devblogs.microsoft.com/oldnewthing/20240308-00

March 8, 2024

Raymond Chen

Last time, we saw a case where we wanted to pass a copy of an object to an operator that accepted objects by reference. We solved this by forcing a copy and using the copy.

```
// Original
co_await m_pendingAction;

// Alternative 1: Copy to a local
auto pendingAction = m_pendingAction;
co_await pendingAction;

// Alternative 2: Copy to a temporary
co_await winrt::IAsyncAction(m_pendingAction);
```

Our problem was motivated by the `co_await` operator, but this problem afflicts function calls, too.

```
void consume(Widget const& widget, std::function<void()> callback);

auto widget = std::make_unique<Widget>();
consume(*widget, [&] {
    widget.reset();
});
```

In the above contrived example, we create a `widget` and pass it by reference to the `consume` method, but in the callback, we destroy the widget! You probably aren't crazy enough to do this on purpose, but it might happen by accident due to reentrancy or a complex interaction between components. Again, the solution is to pass a copy.

```
// Pass a copy of the widget to consume in case the
// original gets destroyed.

// Alternative 1: Copy to a local
auto widget_copy = *widget;
consume(widget_copy, [&] {
    widget.reset();
});

// Alternative 2: Copy to a temporary
consume(Widget(*widget), [&] {
    widget.reset();
});
```

If only there were a convenient way to make a temporary copy of something. The first alternative requires you to name a local variable, and the lifetime of that local variable will extend to the of the block, which may be longer than you were expecting.

The second alternative creates the temporary inline, but you have to spell out the type name, and that type name can get quite unwieldy.

Maybe there's a third way that gives us the inline temporary but also avoids typing out the type name. Here's a start:

```
template<typename T>
std::decay_t<T> copy_of(T&& t)
{ return std::forward<T>(t); }

co_await copy_of(m_pendingAction);

consume(copy_of(*widget), [&] {
    widget.reset();
});
```

The `copy_of` takes an arbitrary reference and returns a copy of whatever it refers to. There are some sneaky bits here.

We can't declare the parameter as `copy_of(T const& t)` because `T` might not have a copy constructor from `T const&`.

We use `std::decay_t` in case `T` has a cv-qualifier. (And to get rid of any reference declarators on `T` that may have been introduced by the forwarding reference.)

Technically, the `std::forward<T>` is not necessary starting in C++17-maybe-20-question-mark?[1] thanks to P18250R0 which introduced implicit moves from rvalue reference parameters into return values. Basically, it says that if you `return` a parameter that is an rvalue reference, then the compiler is allowed to move from that parameter instead of its normal behavior of treating it as an lvalue. Nonetheless, it doesn't hurt to be explicit about it, particularly if you aren't confident that everyone will be using a C++17 compiler that incorporates P18250R0, or if you want your code to work downlevel to C++11.

There's also a problem if `T`'s copy constructor or move copy constructor is explicit. We'll have to use an explicit conversion.

And I forgot to put a `constexpr` on the function, so it can be compile-time evaluated; and I forgot to put a `noexcept` clause, which probably should follow the `noexcept(noexcept)` idiom, leaving us with

```
template<typename T>
std::decay_t<T> copy_of(T&& t)
noexcept(noexcept(
    std::decay_t<T>(std::forward<T>(t))
    ))
{ return
    std::decay_t<T>(std::forward<T>(t))
; }
```

But wait, we're not done yet.

In P0849R8, C++23 added "`auto` decay copy" as a core language feature, so starting in C++23, you can write `auto(x)` to make a copy of `x`, removing the need for the `copy_of` function entirely.

```
// Await a copy because Cancel()
// can modify it while awaiting.
co_await auto(m_pendingAction);

// Consume a copy because the lambda modifies it
// from the callback.
consume(auto(*widget), [&] {
    widget.reset();
});
```

It took over a decade, but we finally got there.

¹ P1825R0 was submitted as a C++17 defect report, which makes it retroactive to C++17, but it naturally takes time for compilers to implement the required changes, so the confidence level doesn't really kick in until C++20.