The puzzle of trying to put an object into a std::optional

devblogs.microsoft.com/oldnewthing/20241113-00

November 13, 2024

The C++ standard library template type `std::optional<T>` has one of two states. It could be empty (not contain anything), or it could contain a `T`.

Suppose you start with an empty `std::optional<T>`. How do you put a `T` into it?

One of my colleagues tried to do it in what seemed to be the most natural way: Use the assignment operator.

```
struct Doodad
{
    Doodad();
    ~Doodad();
    std::unique_ptr<DoodadStuff> m_stuff;
};

struct Widget
{
    std::optional<Doodad> m_doodad;

    Widget()
    {
        if (doodads_enabled()) {
            // I guess we need a Doodad too.
            Doodad d;
            m_doodad = d;
        }
    }
};
```

Unfortunately, the assignment failed to compile:

```
Widget.cpp: error C2679: binary '=': no operator found which takes a right-hand
operand of type 'Doodad' (or there is no acceptable conversion)
```

I asked for the rest of the error message, because the details will explain what the compiler tried to do (and why it couldn't). It's long, but we'll walk through it.

```
    optional(617,1):
    could be 'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<Doodad> &)'
        Widget.cpp(100,9):
        'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<Doodad> &)': cannot convert argument 2 from 'Doodad' to 'const
std::optional<Doodad> &'
            Widget.cpp(100,27):
            Reason: cannot convert from 'Doodad' to 'const std::optional<Doodad>'
            Widget.cpp(100,27):
            No user-defined-conversion operator available that can perform this
conversion, or the operator cannot be called
    optional(283,28):
    or       'std::optional<Doodad> &std::optional<Doodad>::operator =
(std::nullopt_t) noexcept'
        Widget.cpp(100,9):
        'std::optional<Doodad> &std::optional<Doodad>::operator =(std::nullopt_t)
noexcept': cannot convert argument 2 from 'Doodad' to 'std::nullopt_t'
            Widget.cpp(100,27):
            No user-defined-conversion operator available that can perform this
conversion, or the operator cannot be called
    optional(321,28):
    or       'std::optional<Doodad> &std::optional<Doodad>::operator =
(std::optional<_Ty2> &&) noexcept(<expr>)'
        Widget.cpp(100,9):
        'std::optional<Doodad> &std::optional<Doodad>::operator =(std::optional<_Ty2>
&&) noexcept(<expr>)': could not deduce template argument for 'std::optional<_Ty2>
&&' from 'Doodad'
    optional(307,28):
    or       'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<_Ty2> &) noexcept(<expr>)'
        Widget.cpp(100,9):
        'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<_Ty2> &) noexcept(<expr>)': could not deduce template argument for
'const std::optional<_Ty2> &' from 'Doodad'
    optional(292,28):
    or       'std::optional<Doodad> &std::optional<Doodad>::operator =(_Ty2 &&)
noexcept(<expr>)'
        Widget.cpp(100,9):
        'std::optional<Doodad> &std::optional<Doodad>::operator =(_Ty2 &&)
noexcept(<expr>)': could not deduce template argument for '__formal'
            optional(288,33):
            'std::enable_if_t<false,int>' : Failed to specialize alias template
    Widget.cpp(100,9):
    while trying to match the argument list '(std::optional<Doodad>, Doodad)'
```

The compiler is showing its work. It's showing you all the possible overloaded assignment operators and explained why each one failed. The way to understand what went wrong is to look for the overload you intended to use and see why the compiler rejected it. Let's take them one at a time.

```
could be 'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<Doodad> &)'
    cannot convert argument 2 from 'Doodad' to 'const std::optional<Doodad> &'
    Reason: cannot convert from 'Doodad' to 'const std::optional<Doodad>'
    No user-defined-conversion operator available that can perform this conversion,
or the operator cannot be called
```

The first assignment operator available is the one where you assign a `std::optional<Doodad>` to another `std::optional<Doodad>`. This one failed because you passed a `Doodad`, not a `std::optional<Doodad>`, and there was no eligible conversion.

Okay, what's next?

```
or 'std::optional<Doodad> &std::optional<Doodad>::operator =(std::nullopt_t)
noexcept'
    'std::optional<Doodad> &std::optional<Doodad>::operator =(std::nullopt_t)
noexcept': cannot convert argument 2 from 'Doodad' to 'std::nullopt_t'
    No user-defined-conversion operator available that can perform this conversion,
or the operator cannot be called
```

This is the emptying assignment, where you can assign a `std::nullopt` to the optional to return it to the empty state. This is not what we wanted either, so we're not surprised that it failed.

Onward.

```
or 'std::optional<Doodad> &std::optional<Doodad>::operator =(std::optional<_Ty2> &&)
noexcept(<expr>)'
    'std::optional<Doodad> &std::optional<Doodad>::operator =(std::optional<_Ty2> &&)
noexcept(<expr>)': could not deduce template argument for 'std::optional<_Ty2> &&'
from 'Doodad'
```

This is the case of move-assigning a `std::optional<T2>` to a `std::optional<T1>`. This is also not what we were trying to do, so the fact that it failed is expected.

Keep going.

```
or 'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<_Ty2> &) noexcept(<expr>)'
    'std::optional<Doodad> &std::optional<Doodad>::operator =(const
std::optional<_Ty2> &) noexcept(<expr>)': could not deduce template argument for
'const std::optional<_Ty2> &' from 'Doodad'
```

This is the copy-assignment version of the above, so we can skip this one, too.

```
or 'std::optional<Doodad> &std::optional<Doodad>::operator =(_Ty2 &&)
noexcept(<expr>)'
    'std::optional<Doodad> &std::optional<Doodad>::operator =(_Ty2 &&)
noexcept(<expr>)': could not deduce template argument for '__formal'
        'std::enable_if_t<false,int>' : Failed to specialize alias template
    while trying to match the argument list '(std::optional<Doodad>, Doodad)'
```

This is the final catch-all case of assigning an arbitrary object to an optional. This is the one we were hoping to use, but somehow it failed because of a "could not deduce template argument" from `std::enable_if_t<false, int>`, and that leading `false` tells us that an `enable_if` precondition failed. Let's look at the precondition.

```
template <class _Ty2 = _Ty,
        enable_if_t<
            conjunction_v<
                negation<
                    is_same<optional, _Remove_cvref_t<_Ty2>>
                >,
                negation<
                    conjunction<is_scalar<_Ty>, is_same<_Ty, decay_t<_Ty2>>>
                >,
                is_constructible<_Ty, _Ty2>,
                is_assignable<_Ty&, _Ty2>
            >,
            int> = 0>
    _CONSTEXPR20 optional& operator=(_Ty2&& _Right) noexcept(⟦...⟧)
```

Let's work on simplifying this template metaprogramming. In our case, `_Ty2` is `Doodad&`, so `std::decay_t<_Ty2>` is `std::decay_t<Doodad&>`, which is `Doodad`. From its name, it's highly likely that the internal template `_Remove_cvref_t` is `std::remove_cv_t`+`std::remove_reference_t`, but if you don't trust your intuition, you can look it up for yourself:

```
template<class _Ty>
using _Remove_Cvref_t _MSVC_KNOWN_SEMANTICS = remove_cv_t<remove_reference_t<_Ty>>;
```

Applying it to the case where `_Ty2` is `Doodad&` results in `remove_cv_t<remove_reference_t<Doodad&>>` which is `remove_cv_t<Doodad>` which is just `Doodad`. Plugging all that back into the `enable_if`, as well as `_Ty = Doodad` (since `_Ty` is the template parameter to `optional` itself) gives us this:

```
        enable_if_t<
            conjunction_v<
                negation<
                    is_same<optional, Doodad>
                >,
                negation<
                    conjunction<is_scalar<Doodad>, is_same<Doodad, Doodad>>
                >,
                is_constructible<Doodad, Doodad&>,
                is_assignable<Doodad&, Doodad&>
            >,
            int> = 0>
    _CONSTEXPR20 optional& operator=(Doodad;& _Right) noexcept(⟦...⟧)
```

Now we can interpret the expression. The operator is enabled if…

```
!is_same<optional, Doodad> &&
!(is_scalar<Doodad> && is_same<Doodad, Doodad>) &&
is_constructible<Doodad, Doodad&> &&
is_assignable<Doodad&, Doodad&>
```

(It so happens that these are precisely the conditions spelled out in the C++ language specification. I doubt this is a coincidence.)

The first clause says “you are not assigning from a `std::optional<Doodad>`“, which is true. We are assigning from a `Doodad`. The purpose of this clause is to remove this overload from consideration in favor of the other overload that specifically is for optional-to-optional assignment.

The second clause says “you are not trying to assign a scalar that is the same type of the optional.” I think this is to remove this overload from consideration in favor of converting the source scalar to an `optional<_Ty>` and assigning that. Regardless, it doesn’t apply here, so we pass that test too.

The next test is to see whether a `Doodad` can be constructed from a `Doodad&`, and in the case of a `Doodad`, it turns out that this is not true because the `Doodad` contains a `unique_ptr`, which makes it non-copyable.

Okay, so we can fix that by using `std::move` to move the `Doodad` on the stack into the optional, right?

```
Doodad d;
m_doodad = std::move(d);
// or even
m_doodad = Doodad();
```

Unfortunately, this fails in basically the same way. But how can that be?

It’s because `Doodad` is not move-assignable, even though all of its members are movable!

The requirements for an implicitly-defined move-assignment operator are that the type have no user-declared copy constructors, move constructors, copy assignment operators, or destructors. Our `Doodad` has a destructor, so that removes the implicitly-defined move-assignment operator.

Bonus reading: Implicit Move Must Go.

So our `Doodad` is not movable, not copyable.

One solution is to make our `Doodad` movable. This means investigating the class invariants and verifying that memberwise `std::move` preserves them. This can get tricky if, for example, the `Doodad` allowed pointers to itself to escape. If you've done the analysis and confirmed that memberwise `std::move` is correct behavior, you can add

```
Doodad(Doodad&&) = default;
Doodad& operator=(Doodad&&) = default;
```

to ask for the compiler to generate a default move constructor and default move assignment operator.

But maybe you study the `Doodad` and conclude that it is not movable for whatever reason. What else can you do?

We'll look at our options next time.