Template meta-programming: Avoiding saying a type before it is complete

devblogs.microsoft.com/oldnewthing/20230929-00

September 29, 2023

Raymond Chen

As we noted last time, C++/WinRT's `get_strong()` will produce a broken strong reference if destruction has already begun. The C++ standard library's `enable_shared_from_this` solves this problem by saving a weak reference in the object itself. But if you tried the analogous trick in versions of C++/WinRT prior to 2.0.211028.7, you got a weird compiler error.

```
struct Me : winrt::implements<Me, winrt::IInspectable>
{
    winrt::weak_ref<Me> m_weakSelf;
};

// msvc
type_traits(1173,28): error C2139: 'Me': an undefined class is not allowed as an
argument to compiler intrinsic type trait '__is_base_of'
test.h(16,8): message : see declaration of 'Me'
base.h(1963,45): message : see reference to variable template 'const bool
is_base_of_v<winrt::Windows::Foundation::IUnknown, Me>' being compiled
base.h(4119,24): message : see reference to alias template instantiation 'winrt::
impl::com_ref<T>' being compiled
with
[
    T=Me
]
Test.cpp(19,25): message : see reference to class template instantiation 'winrt::
weak_ref<D>' being compiled
with
[
    D=Me
]

// gcc
type_traits: In instantiation of 'struct std::is_base_of<winrt::Windows::Foundation::
IUnknown, Me>':
type_traits:3282:69:   required from 'constexpr const bool std::is_base_of_v<winrt::
Windows::Foundation::IUnknown, Me>'
required from 'struct winrt::weak_ref<Me>'
required from here
type_traits:1447:38: error: invalid use of incomplete type 'struct Me'
|     : public integral_constant<bool, __is_base_of(_Base, _Derived)>
|                                      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: forward declaration of 'struct Me'
| struct Me : winrt::implements<Me, winrt::IInspectable>
|        ^^
type_traits: In instantiation of 'constexpr const bool std::is_base_of_v<winrt::
Windows::Foundation::IUnknown, Me>':
required from 'struct winrt::weak_ref<Me>'
required from here
type_traits:3282:69: error: 'value' is not a member of 'std::is_base_of<winrt::
Windows::Foundation::IUnknown, Me>'
|   inline constexpr bool is_base_of_v = is_base_of<_Base, _Derived>::value;
|                                                                     ^~~~~

// clang
type_traits:3345:68: error: incomplete type 'Me' used in type trait expression
  inline constexpr bool is_base_of_v = __is_base_of(_Base, _Derived);
                                                                  ^
note: in instantiation of variable template specialization 'std::is_base_of_v<winrt::
Windows::Foundation::IUnknown, Me>' requested here
```

```
        std::is_base_of_v<winrt::Windows::Foundation::IUnknown,T>, int, int> v);
             ^
note: in instantiation of template class 'winrt::weak_ref<Me>' requested here
    winrt::weak_ref<Me> m_weakSelf;
                        ^
note: definition of 'Me' is not complete until the closing '}'
struct Me : winrt::implements<Me, winrt::IInspectable>
       ^
```

In this case, clang is the one that pinpoints the problem: “Definition of `Me` is not complete until the closing brace.”

The problem is that `weak_ref<T>` requires that `T` be a complete type because it has a constructor that relies on the completeness of `T`:

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    weak_ref(impl::com_ref<T> const& object)
    {
        ⟦ implementation elided ⟧
    }

    ⟦ other members ⟧
};
```

The `impl::com_ref` template type is an internal C++/WinRT helper. We can peek at its definition:

```
template <typename T>
using com_ref = std::conditional_t<
    std::is_base_of_v<Windows::Foundation::IUnknown, T>,
    T,
    com_ptr<T>>;
```

In words, a `com_ref<T>` is just a `T` if `T` is itself a projected type. Otherwise, it’s a `com_ptr<T>`.

Now, when you write `weak_ref<T>`, this instantiates the template, and the compiler generates declarations for all of the members. One of those members is the second constructor that takes a `impl::com_ref<T>` as a parameter. And that’s where the error occurs, because in order to know what `impl::com_ref<T>` means, the compiler has to see whether `T` has `winrt::Windows::Foundation::IUnknown` as a base class, and that requires that `T` be a complete type.

We can solve this problem by templating the second constructor.

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    template<typename U>
    weak_ref(U&& object)
    {
        ⟦ implementation elided ⟧
    }

    ⟦ other members ⟧
};
```

This defers the instantiation of the second constructor until somebody tries to call it, which will (we hope) happen after `T` is a complete type.

But wait, we're not done yet.

For one thing, the original constructor specified its parameter as `impl::com_ref<T>`, which means that if the caller passed a braced list, that braced list is used to construct a `impl::com_ref<T>`. But we don't get that effect with the forwarding reference. Forwarding references received braced lists as a `initializer_list<X>` for some type `X`. In order to tell the compiler, "If you see a braced list, please treat it as the constructor parameters to a `impl::com_ref<T>`," we specify a default type:

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    template<typename U = impl::com_ref<T>>
    weak_ref(U&& object)
    {
        ⟦ implementation elided ⟧
    }

    ⟦ other members ⟧
};
```

Now, the original implementation assumed that `object` was a `com_ref<T>`, so if the inbound parameter isn't one, we need to convert it.

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    template<typename U = impl::com_ref<T>>
    weak_ref(U&& objectArg)
    {
        impl::com_ref<T> const& object = objectArg;

        ⟦ implementation elided ⟧
    }

    ⟦ other members ⟧
};
```

There's a subtlety here, though. The conversion from `objectArg` to `object` is an explicit conversion, but parameter conversions use only implicit conversions. We can use SFINAE to limit ourselves to types that support implicit conversion to `com_ref<T>`.

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    template<typename U = impl::com_ref<T>,
             typename = std::enable_if_t<
                std::is_convertible_v<
                    U&&,
                    impl::com_ref<T> const&>>>
    weak_ref(U&& objectArg)
    {
        impl::com_ref<T> const& object = objectArg;

        ⟦ implementation elided ⟧
    }

    ⟦ other members ⟧
};
```

This revision also solves another problem: Without SFINAE, our templated `weak_ref` constructor would also be used as the copy and move constructor, rather than using the compiler-generated default versions. Fortunately, the SFINAE rejects `weak_ref const&` and `weak_ref&&` (since neither is convertible to `com_ref<T>`), so adding the SFINAE also magically restores the copy and move constructors.

Finally, to avoid code size explosion due to each specialization producing a different conversion, we factor the original constructor into a helper that takes only a `com_ref<>`. That way, the bulk of the code is shared among all the constructors.

```
template <typename T>
struct weak_ref
{
    weak_ref(std::nullptr_t = nullptr) noexcept {}

    template<typename U = impl::com_ref<T>,
             typename = std::enable_if_t<
                std::is_convertible_v<
                    U&&,
                    impl::com_ref<T> const&>>>
    weak_ref(U&& object)
    {
        from_com_ref(static_cast<impl::com_ref<T> const&>(object));
    }

    ⟦ other members ⟧

    template<typename U>
    void from_com_ref(U&& object)
    {
        ⟦ implementation elided ⟧
    }
};
```

Again, the `from_com_ref` method is templated so that it is not instantiated immediately, since it is not allowed to say that its parameter is a `com_ref<T>`. Instead, we secretly accept a `com_ref<T>` by using a parlor trick from some time ago: We accept a templated parameter even though in practice, only one type ever gets passed.

What this all means is that starting in C++/WinRT version 2.0.211028.7, you can have a class hold a weak reference to itself.

```
struct Me : winrt::implements<Me, winrt::IInspectable>
{
    winrt::weak_ref<Me> m_weakSelf = get_weak();
};
```