Behind C++/WinRT: How does C++/WinRT get the list of implemented interfaces?

devblogs.microsoft.com/oldnewthing/20220325-00

March 25, 2022

Raymond Chen

Last time, we looked at how C++/WinRT decides which interfaces are implemented. The `is_interface` type is just part of the larger template metaprogramming infrastructure inside C++/WinRT.

Today, we'll reverse-engineer the `interface_list<...>` template type, which holds a list of interfaces inside the `...`.

```
template <typename...> struct interface_list;

template <>
struct interface_list<>
{
    template <typename T, typename Predicate>
    static constexpr void* find(const T*, const Predicate&) noexcept
    {
        return nullptr;
    }
};
```

The base case is an empty interface list. In that case, there are no interfaces, and any attempt to find one always fails.

```
template <typename First, typename ... Rest>
struct interface_list<First, Rest...>
{
    template <typename T, typename Predicate>
    static constexpr void* find(const T* obj, const Predicate& pred) noexcept
    {
        if (pred.template test<First>())
        {
            return to_abi<First>(obj);
        }
        return interface_list<Rest...>::find(obj, pred);
    }
    using first_interface = First;
};
```

The recursive case is an interface list of at least one element. That element is called `First`, and is given the member type name `first_interface` for later extraction. The rest of the elements are called `Rest`.

The `interface_list<...>` has a `find` method that looks for the first interface that satisfies a given predicate. From the usage pattern, you can see that the expected pattern for the predicate is

```
class Predicate
{
public:
    template<typename I> bool test() const noexcept
    {
        return "true" if the interface "I" passes the test
    }
};
```

You can infer this because

- The `test` method must be public because we call it from outside the class.
- We call the `test<First>()` method, so the template parameter is the interface to test.
- The return value is used by the `if` statement, so it should be a `bool` (or at least be `bool`-like).
- We invoke it on a const `pred` object, so the method must be `const`.
- The containing function is `noexcept` so `test()` should be `noexcept` as well.

In the base case, there are no interfaces, so clearly there are no matches, so it just returns `nullptr`.

In the recursive case, it asks the predicate to test the first interface. If so, the we return `to_abi<First>(obj)`. Otherwise we call oureslves recursively to test the other interfaces.

Template metaprogramming traditionally involves a lot of recursion.[1]

The next helper is this guy:

```
template <typename, typename> struct interface_list_append_impl;

template <typename... T, typename... U>
struct interface_list_append_impl<interface_list<T...>, interface_list<U...>>
{
    using type = interface_list<T..., U...>;
};
```

The `interface_list_append_impl` class appends two interface lists. It extracts the interfaces from two interface lists (calling them `T...` and `U...`) and then creates a new interface list called `type` whose interfaces are the concatenation of `T...` and `U...`.

To concatenate two interface lists, you would say

```
using result = interface_list_append_impl<list1, list2>::type;
```

Two patterns common to template metaprogramming are the "type" pattern and the "value" pattern. You create a class whose job is to perform some template metaprogramming task that produces a type or value. In order to express a type, you define a nested type called `type` which is aliased to to the produced type. And to express a value, you define a `static const` (or `constexpr` starting in C++11) member called `value` which has the produced value.

The first complex operation on interface lists is filtering:

```
template <template <typename> class, typename...>
struct filter_impl;
```

This weird declaration says that `filter_impl` is a templated class whose first template parameter is another template (which takes a single type template parameter), followed by zero or more additional types.

Before implementing the `filter_impl`, we give it a prettier name:

```
template <template <typename> class Predicate, typename... T>
using filter = typename filter_impl<Predicate, unwrap_implements_t<T>...>::type;
```

This is another common extension to the "type" and "value" patterns: Starting in C++11, you can define custom templated types or values that extract the type or value from the templated class. Here, we make `filter<...>` a typing-saver for `filter_impl<...>::type`, with the extra wrinkle that the types that come after the predicate pass through `unwrap_implements_t`. (We'll look at `unwrap_implements_t` some other time. The focus here is on the interface list template metaprogramming.)

This `using` statement also teaches us that the first template parameter is a predicate, seeing as it's named `Predicate`.

Okay, now we start implementing `filter_impl`:

```
template <template <typename> class Predicate>
struct filter_impl<Predicate>
{
    using type = interface_list<>;
};
```

The base case is filtering an empty list of types, which is easy: You get an empty interface list.

Next comes the recursive case:

```
template <template <typename> class Predicate, typename T, typename... Rest>
struct filter_impl<Predicate, T, Rest...>
{
    using type = typename interface_list_append_impl<
        std::conditional_t<
        Predicate<T>::value,
        interface_list<winrt::impl::uncloak<T>>,
        interface_list<>
        >,
        typename filter_impl<Predicate, Rest...>::type
    >::type;
};
```

We test the predicate by looking for `Predicate<T>::value` and treating it as a `bool`. Therefore the `Predicate` is expected to follow the value pattern:

```
template<typename I>
class Predicate
{
    static constexpr bool value =
        "true" if interface "I" satisfies the predicate;
}
```

`std::conditional_t` is a type-trait helper that is called like this:

```
std::conditional_t<condition, if_true, if_false>
```

If the condition is `true`, then the `std::conditional_t` resolves to the type `if_true.` Otherwise, it resolves to `if_false`.

In this case, if the first interface in the interface list passes the test, then the `std::conditional_t` produces a single-element `interface_list` consisting of the uncloaked version of the interface. (We'll look at uncloaking later.) Otherwise, it produces a zero-element interface list.

This interface list is then combined via `interface_list_append_impl` with a recursive call to process the rest of the list, and we extract the resulting type.

What this ultimately does is produce a type that is an `interface_list` where the elements of the list are the interfaces which satisfy the predicate.

Next comes an alternate way of invoking the filter:

```
template <template <typename> class Predicate, typename ... T, typename ... Rest>
struct filter_impl<Predicate, interface_list<T...>, Rest...>
{
    using type = typename interface_list_append_impl<
        filter<Predicate, T...>,
        filter<Predicate, Rest...>
    >::type;
};
```

This is a convenience helper which lets you apply a filter to an existing `interface_list`, as well as optional additional types. It extracts the interfaces from the `interface_list` and then filters them, and then also filters any additional types. (Since this is a recursive call, those additional types might themselves be `interface_list` types, in which case the inner types will be extracted from those types as well.)

And another convenience helper that will lead us to the answer to the puzzle we solved earlier this week:

```
template <template <typename> class Predicate, typename D, typename ... I, typename
... Rest>
struct filter_impl<Predicate, winrt::implements<D, I...>, Rest...>
{
    using type = typename interface_list_append_impl<
        filter<Predicate, I...>,
        filter<Predicate, Rest...>
    >::type;
};
```

If the first type after the predicate is an `implements`, it extracts the second and subsequent template parameters from the `implements` template type and filters those, and then continues recursively with any remaining types.

We're getting close to our goal of understanding the puzzle:

```
template <typename T>
using implemented_interfaces = filter<is_interface, typename T::implements_type>;
```

This defines a helper type `implements_interfaces`: Given a type, it pulls out the `implements_type` and filters it with the `is_interface` predicate.

The `implements_type` is a type that the `implements<>` template defines so it can find itself later:

```
template <typename D, typename... I>
struct implements : impl::producers<D, I...>,
                    impl::base_implements<D, I...>::type
{
...
public:

    using implements_type = implements;
 ...
};
```

Note that we are taking advantage of C++ injected class names which lets us write `implements` instead of the longer `mplements<D, I...>`. (The reason we have an explicit `implements_type` subtype is that other classes in C++/WinRT also define an `implements_type`, so this type name allows uniform access to the implemented thing.)

The `is_interface` class we discussed last time is the final piece of the puzzle: The interfaces `I...` passed to `implements<D, I...>` are filtered through `is_interface` to produce an `interface_list<...>` which holds all of the interfaces that are implemented.

The error that we got was complaining that there was no definition for `first_interface`, and from the definition at the start of this article, we see that the definition is present only if the interface list is nonempty.

Putting it all together, we see that the error occurred because none of the `I...` types was recognized as an interface by `is_interface`, so the interface list ended up empty, and in order to `make` an object, it has to implement at least one interface. (Because without any interfaces, what you have isn't even a COM object.)

Bonus chatter: C++ template metaprogramming is like programming in Prolog without a debugger.

[1] The introduction of fold expressions in C++17 provides an opportunity to convert recursion to iteration. The trick is to rephrase your loop in terms of a fold expression. In this case, we could have written it as

```
template<typename I, typename T, typename Predicate>
void find_interface_helper(
    const T* obj, const Predicate& pred,
    bool& found, void*& result)
    noexcept
{
    if (!found && pred.template test<I>()) {
        found = true;
        result = to_abi<I>(obj);
    }
}

template <typename...Interfaces>
struct interface_list
{
    template <typename T, typename Predicate>
    static constexpr void* find(const T* obj, const Predicate& pred) noexcept
    {
        bool found = false;
        void* result = nullptr;
        (find_interface_helper<Interfaces>(obj, pred, found, result), ...);
        return result;
    }
};
```

but it's not clear that the effort was worth it. Fold expressions don't have a way to `break` out of the loop, so we need a `found` variable that we use to cause the remaining calls to have no effect once a candidate is found.

Aha, maybe we can fold using a short-circuiting operator. The short circuiting lets us "break" out of a "loop".

```
template<typename I, typename T, typename Predicate>
bool find_interface_helper(
    const T* obj, const Predicate& pred,
    void*& result) noexcept
{
    if (pred.template test<I>()) {
        result = to_abi<I>(obj);
        return true;
    }
    return false;
}

template <typename...Interfaces>
struct interface_list
{
    template <typename T, typename Predicate>
    static constexpr void* find(const T* obj, const Predicate& pred) noexcept
    {
        void* result = nullptr;
        (find_interface_helper<Interfaces>(obj, pred, result) || ...);
        return result;
    }
};
```

And now we can inline the `find_interface_helper`:

```
template <typename...Interfaces>
struct interface_list
{
    template <typename T, typename Predicate>
    static constexpr void* find(const T* obj, const Predicate& pred) noexcept
    {
        void* result = nullptr;
        ([&] {
            if (pred.template test<Interfaces>()) {
                 result = to_abi<Interfaces>(obj);
                return true;
            }
            return false;
        }() || ...);
        return result;
    }
};
```

Raymond Chen

Follow