# C++/WinRT implementation inheritance: Notes on winrt::implements, part 3

devblogs.microsoft.com/oldnewthing/20250221-00

February 21, 2025


In C++/WinRT, the `implements` template type starts like this:

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

The `producers` template type generates the vtables for the COM interfaces. But that's not what we're looking at today. Today we're going to look at the `base_implements` part.

The `base_implements` type is defined as follows:

```
template <typename D, typename Dummy = std::void_t<>, typename... I>
struct base_implements_impl
    : impl::identity<root_implements<D, I...>> {};

template <typename D, typename... I>
struct base_implements_impl<D, std::void_t<typename nested_implements<I...>::type>,
I...>
    : nested_implements<I...> {};

template <typename D, typename... I>
using base_implements = base_implements_impl<D, void, I...>;
```

We learned from last time that this uses SFINAE: to implement an "if then else" pattern. In this case, it's saying "If `nested_implements<I...>::type` exists, then derive from `nested_implements<I...>`. Otherwise, derive from `impl::identity<root_implements<D, I...>>`."

```
template <typename T>
struct identity
{
    using type = T;
};
```

Okay, so `identity<T>::type` is just `T`. This is basically a copy of `std::type_identity`. C++/WinRT supports C++17, but `std::type_identity` didn't show up until C++20, so C++/WinRT provides its own copy.

Applying this to `base_implements_impl` simplifies it to "If `nested_implements<I...>::type` exists, then derive from `nested_implements<I...>`. Otherwise, derive from `root_implements<D, I...>`."

So what is `nested_implements`?

```
template <typename...>
struct nested_implements
{};

template <typename First, typename... Rest>
struct nested_implements<First, Rest...>
    : std::conditional_t<is_implements_v<First>,
    impl::identity<First>, nested_implements<Rest...>>
{
    static_assert(
        !is_implements_v<First> ||
        !std::disjunction_v<is_implements<Rest>...>,
        "Duplicate nested implements found");
};
```

This is a recursively-defined `nested_implements`. In the base case, `nested_implements<>` is an empty class. Otherwise, we peel off the first template parameter and see if it derives from `implements`. If so, then we use it. Otherwise, we recurse on the remaining parameters.

So `nested_implements` searches through the template parameters and takes the first one that derives from `implements`. Otherwise, it's an empty class.

But wait, there's extra work done in the `static_assert`. First, let's translate it from C++ template-ese to pseudo-code. The `std::disjunction` takes the logical OR of its arguments, so the second part expands to `!(is_implements_v<Rest1> || is_implements_v<Rest2> || ...)`, which says "None of the `Rest` is an `implements`."

Now combine this with the first part, and we get "Either `First` is not an `implements`, or none of the `Rest` is an `implements`." If you transform this to an implication relation, you get "If `First` is an `implements`, then none of the `Rest` is an `implements`."

During the recursion, `First` progresses through all of the interface arguments, so the assertion verifies that at most one of the interface arguments supports `implements`.

Okay, so unwinding back to `base_implements`, we had previously determined that the definition was "If `nested_implements<I...>::type` exists, then derive from `nested_implements<I...>`. Otherwise, derive from `impl::identity<root_implements<D, I...>>`."

Combining this with our discovery that `nested_implements` takes the first interface that is an `implements`, we see that the result is that `base_implements` is

- If none of the interfaces is an `implements`, then use `root_implements`.
- If exactly one of the interfaces is an `implements`, then use it. (It will provide the `root_implements` so we don't have to.)
- If more than one of the interfaces is an `implements`, then raise a compile-time error.

From all this, you can figure out the legal inheritance structures for `winrt::implements`: The `implements` must form a single chain of inheritance, possibly passing through other non-`implements` classes along the way. You cannot inherit (directly or indirectly) from multiple `implements`. The innermost `implements` provides the `root_implements`.

| | | | |
|---|---|---|---|
| | A | | `A : implements<A, I1, B, I2>, X1` |
| | ↓ | | |
| | `implements` | X1 | |
| | ↓ | | |
| `produce<A, I1>` | B | `produce<A, I2>` | `B : implements<B, C, I3>` |
| | ↓ | | |
| | `implements` | | |
| | ↓ | | |
| | C | `produce<B, I3>` | `C : D, X2` |
| | ↓ | | |
| | D | X2 | `D : implements<D, I4, I5>` |
| | ↓ | | |
| `root_implements` | `produce<D, I4>` | `produce<D, I5>` | |

Next time, we'll look at how you can employ base classes and inheritance in your Windows Runtime implementation classes while still adhering to these restrictions.