C++/WinRT implementation inheritance: Notes on winrt::implements, part 2

devblogs.microsoft.com/oldnewthing/20250220-00

February 20, 2025

Some time ago, we investigated how C++/WinRT decides which interfaces your class implements when you use the `implements` template.

I promised to talk about `unwrap_implements_t` at some point in the future, so I guess now's the time.

```
template <typename T, typename = std::void_t<>>
struct unwrap_implements
{
    using type = T;
};

template <typename T>
struct unwrap_implements<T,
    std::void_t<typename T::implements_type>>
{
    using type = typename T::implements_type;
};

template <typename T>
using unwrap_implements_t =
    typename unwrap_implements<T>::type;
```

The `std::void_t` template is a helper for SFINAE: it expands to `void` if all if its template arguments can be evaluated. The first usage of it is in the definition of the basic case:

```
template <typename T, typename = std::void_t<>>
struct unwrap_implements
{
    using type = T;
};
```

Here, we use `std::void_t<>`. Since all of the template arguments can be evaluated (all zero of them), this is the same as just `void`. I'm not sure why the code uses the longer formulation instead of just writing `void`; maybe it's just to parallel the usage of `std::void_t` argument in the partial specialization.

The partial specialization uses `std::void_t<typename T::implements_type>`, so it is testing whether the type `T` has a member type named `implements_type`. If so, then this partial specialization succeeds, and the `type` is whatever `implements_type` was. The `implements` template creates an `implements_type` member type, so this succeeds when `T` derives from `implements` (possibly through a chain of intermediate classes).

Therefore, the result of `unwrap_implements<T>::type` is the `implements` if `T` derives (eventually) from `implements`; otherwise, it's just `T` itself.

The last part is just making `unwrap_implements_t<T>` a shorthand for `unwrap_implements<T>::type`.

I talked through this whole thing step by step, but after some practice, you recognize this pattern fairly quickly, and you read it as "if (SFINAE condition) then (partial specialization thing) else (fallback thing)."

Next time, we'll build on this to understand the allowable inheritance structures for `winrt::implements`.