Inside C++/WinRT: IReference

devblogs.microsoft.com/oldnewthing/20221215-00

December 15, 2022

Raymond Chen

Last time, we looked at how to create an `IReference<T>` in C++/WinRT. How did I know the answer? By reading the source code and reverse-engineering what operations were valid and what they produced.

Let's look at the available constructors:

```
IReference(IReference const& other) = default;
IReference(IReference&& other) = default;
```

The copy and move constructors are implicitly declared, but I'm declaring them explicitly so they show up on the list.

```
IReference(std::nullptr_t = nullptr) noexcept {}
```

This constructor serves as a default constructor, or you can construct from `nullptr`. Either way creates an empty `IReference`. This constructor exists for all interface types, so it's nothing surprising.

```
IReference(void* ptr, take_ownership_from_abi_t) noexcept
    : Windows::Foundation::IInspectable(
        ptr, take_ownership_from_abi) {}
```

The `take_ownership_from_abi` tag type constructor creates an `IReference` that takes ownership of an ABI pointer. This constructor also exists for all interface types, so nothing special is happening yet.

```
IReference(T const& value)
    : IReference<T>(
        impl::reference_traits<T>::make(value))
{}
```

This is a conversion constructor which takes the underlying value type, calls the `make` method from some class we haven't yet studied, and uses that to initialize the `IReference`. We'll look at that class later.

The last constructor is

```
IReference(std::optional<T> const& value)
    : IReference(
        value ? IReference(value.value()) : nullptr)
{}
```

This is a conversion constructor that takes a `std::optional<T>` and produces the corresponding `std::IReference<T>`. If the `optional` has a value, then we wrap it inside an `IReference` using the value conversion constructor we saw above. Otherwise, we initialize from `nullptr`, which means "no value".

One thing to observe is that these last two constructors work with CTAD, so you can omit the template specialization:

```
// compile deduces IReference<double> from double
double value = 42.0;
auto ref = IReference(value);

// compile deduces IReference<double> from std::optional<double>
std::optional<double> value(42.0);
auto ref = IReference(value);
```

The last member function is the `std::optional` conversion:

```
operator std::optional<T>() const
{
    if (*this)
    {
        return this->Value();
    }
    else
    {
        return std::nullopt;
    }
}
```

This converts an `IReference<T>` to a `std::optional<T>` by producing the value if the wrapped pointer is non-null, or producing an empty `optional` if the wrapped pointer is null.

All that's left is studying the `impl::reference_traits`.

```
template <typename T>
struct reference_traits
{
    static auto make(T const& value)
    { return winrt::make<impl::reference<T>>(value); }
    using itf = Windows::Foundation::IReference<T>;
};
```

This is the unspecialized template traits class which `make`s an `IReference<T>` by creating an instance of the private `impl::reference` class. We'll make a note to come back to that later.

What follows is a series of specializations for various fundamental types. For example,

```
template <>
struct reference_traits<uint8_t>
{
    static auto make(uint8_t value)
    { return Windows::Foundation::PropertyValue::CreateUInt8(value); }
    using itf = Windows::Foundation::IReference<uint8_t>;
};
```

To create an `IReference<uint8_t>`, the `make()` method uses the `PropertyValue::CreateUInt8()` method. Repeat for the other special-purpose factory methods of `PropertyValue`.

Okay, so now we're left with that `impl::reference` class:

```
template <typename T>
struct reference :
    implements<reference<T>,
        Windows::Foundation::IReference<T>,
        Windows::Foundation::IPropertyValue>
{
    reference(T const& value) : m_value(value)
    { }

    T Value() const { return m_value; }

    Windows::Foundation::PropertyType Type() const noexcept
    {
        return Windows::Foundation::PropertyType::OtherType;
    }

    static constexpr bool IsNumericScalar() noexcept
    {
        return std::is_arithmetic_v<T> || std::is_enum_v<T>;
    }

    uint8_t GetUInt8() const
    {
        return to_scalar<uint8_t>();
    }

    ⟦ repeat for the other Get(IntegralType) methods ⟧

    float GetSingle() { throw hresult_not_implemented(); }

    ⟦ repeat for the other Get(NonIntegralType) methods ⟧

private:
    template <typename To>
    To to_scalar() const
    {
        if constexpr (IsNumericScalar()) {
            return static_cast<To>(m_value);
        } else {
            throw hresult_not_implemented();
        }
    }

    T m_value;
};
```

The `impl::reference<T>` type is used for custom enumerations and custom structures. In both cases, the `IReference<T>::Value()` produces the wrapped value.

The remaining methods provide the implementation of `IPropertyValue`, which we noted last time is one of the hidden requirements of `IReference<T>`.

If used for enumerations, `IsNumericScalar()` reports `true`, and all of the `Get(IntegralType)` methods return the underlying integral value, cast to the requested integral type. On the other hand, for structures, `IsNumericScalar()` reports `false`, and all of the `Get(IntegralType)` methods throw `hresult_not_implemented()`.

For the floating point and other non-integral types, the methods always throw `hresult_not_implemented()`.

It's from reverse-engineering the C++/WinRT implementation of `IReference` that we inferred the rules for its use. Being able to reverse-engineer the proper use of a library from reading its source code is a key under-appreciated software developer skill. Until you can infer the proper way of working with unfamiliar code, your career growth options will be limited. You'll be one of those people who can only do things they are taught, without the ability to learn new things on their own.

Raymond Chen

Follow