In C++/WinRT, how do I create or consume an IReference that wraps a particular value?

devblogs.microsoft.com/oldnewthing/20221214-00

December 14, 2022

Raymond Chen

Let's look at a variety of Windows Runtime methods that accept and receive `IReference<T>` for various types `T`.

```
namespace Contoso
{
    enum CustomEnum
    {
        Circle,
        Square,
    };

    struct CustomStruct
    {
        Int32 Value1;
        Double Value2;
    };

    runtimeclass References
    {
        // Producers
        Windows.Foundation.IReference<Double> M1();
        Windows.Foundation.IReference<CustomEnum> M2();
        Windows.Foundation.IReference<CustomStruct> M3();

        // Consumers
        void M4(Windows.Foundation.IReference<Double> value);
        void M5(Windows.Foundation.IReference<CustomEnum> value);
        void M6(Windows.Foundation.IReference<CustomStruct> value);
    }
}
```

Okay, let's look at the first set of methods, the producers.

```
namespace winrt::Contoso::implementation
{
    using Windows::Foundation::IReference;
    using Windows::Foundation::PropertyValue;

    struct References : ReferencesT<References>
    {
        IReference<double> M1()
        {
            // Group 1
            return PropertyValue::CreateDouble(42.0)
                .as<IReference<double>>()

            // Group 2
            return IReference<double>(42.0);
            return IReference(42.0);
            return 42.0;

            // Group 3
            return IReference<double>(nullptr);
            return nullptr;
        }
    };
}
```

One way to produce an `IReference<T>` is to ask another object to do it for you, in this case, the `Windows.Foundation.PropertyValue`. This works for the basic arithmetic types, strings, and a few foundational structures, and arrays of those things.

Group 2 shows a simpler alternative: Construct the `IReference<T>` from the desired value. The first version spells everything out explicitly. The second version uses class template argument deduction (commonly abbreviated as CTAD) to avoid having to specify the `<T>`, and the third version uses the fact that the `IReference<T>` is implicitly convertible from the value, so you don't need to say anything at all!

Dirty little secret: Group 2 is not doing anything different Group 1. It's just a very convenient shorthand.

The third group shows how to return the null reference, indicating that there is no value at all. If you like to type, you can spell out the null reference constructor the long way, but you are more likely to use the implicit conversion from `nullptr`.

Usually, you aren't hard-coding a value or non-value, but instead are checking if a value exists, and either returning a reference to that value, or a null reference.

```
struct References : ReferencesT<References>
{
    std::optional<double> m_value;

    IReference<double> M1()
    {
        if (m_value) {
            // Group 1
            return PropertyValue::CreateDouble(m_value.value())
                .as<IReference<double>>()

            // Group 2
            return IReference<double>(m_value.value());
            return IReference(m_value.value());
            return m_value.value();
        } else {
            // Group 3
            return IReference<double>(nullptr);
            return nullptr;
        }
    }
};
```

For enumerations and structures, you can't use the `PropertyValue` trick because the `PropertyValue` does not have a factory method for enumerations and structures. You have to use Group 2, which detects that you're using a custom enumeration or structure and provides a corresponding custom implementation of `IPropertyValue`.

```
struct References : ReferencesT<References>
{
    std::optional<CustomEnum> m_enumValue;
    std::optional<CustomStruct> m_structValue;

    IReference<CustomEnum> M2()
    {
        if (m_value) {
            // Group 1
            // return PropertyValue::CreateEnum(m_value.value())
            //     .as<IReference<CustomEnum>>()

            // Group 2
            return IReference<CustomEnum>(m_value.value());
            return IReference(m_value.value());
            return m_value.value();
        } else {
            // Group 3
            return IReference<CustomEnum>(nullptr);
            return nullptr;
        }
    }

    IReference<CustomEnum> M3()
    {
        if (m_value) {
            // Group 1
            // return PropertyValue::CreateStruct(m_value.value())
            //     .as<IReference<CustomStruct>>()

            // Group 2
            return IReference<CustomStruct>(m_value.value());
            return IReference(m_value.value());
            return m_value.value();
        } else {
            // Group 3
            return IReference<CustomStruct>(nullptr);
            return nullptr;
        }
    }
};
```

If your value is being kept in a `std::optional` already, then you can take advantage of the implicit conversions between `IReference<T>` and `std::optional<T>`, added in version 2.0.210930.8. An empty `std::optional` corresponds to a null `IReference`, and a `std::optional` with a value corresponds to a non-null `IReference`.

```
struct References : ReferencesT<References>
{
    std::optional<double> m_value;
    std::optional<CustomEnum> m_enumValue;
    std::optional<CustomStruct> m_structValue;

    IReference<CustomEnum> M1()
    {
        return m_value;
    }

    IReference<CustomEnum> M2()
    {
        return m_enumValue;
    }

    IReference<CustomEnum> M3()
    {
        return m_structValue;
    }
};
```

Furthermore, thanks to the magic of the curiously recurring template pattern (CRTP), your methods can just return `std::optional` and let the projection do the work of converting it to an `IReference`. And since the return type now matches the variable type, you can use `auto` and save yourself a lot of typing.

```
struct References : ReferencesT<References>
{
    std::optional<double> m_value;
    std::optional<CustomEnum> m_enumValue;
    std::optional<CustomStruct> m_structValue;

    auto M1() { return m_value; }
    auto M2() { return m_enumValue; }
    auto M3() { return m_structValue; }
};
```

When receiving a value as a parameter, you check if the parameter is null to detect the null reference, and if it's not, you use the `Value()` method to extract the value:

```
struct References : ReferencesT<References>
{
    void M4(IReference<double> const& value)
    {
        if (value) {
            ConsumeValue(value.Value());
        } else {
            NoValue();
        }
    }

    void M5(IReference<CustomEnum> const& value)
    {
        if (value) {
            ConsumeValue(value.Value());
        } else {
            NoValue();
        }
    }

    void M6(IReference<CustomStruct> const& value)
    {
        if (value) {
            ConsumeValue(value.Value());
        } else {
            NoValue();
        }
    }
};
```

And again, you can take advantage of the implicit conversion and CRTP and just operate with `std::optional`.

```
struct References : ReferencesT<References>
{
    void M4(std::optional<double> const& value)
    {
        ConsumeValue(value.value_or(42.0));
    }

    void M5(std::optional<CustomEnum> const& value)
    {
        ConsumeValue(value.value_or(CustomEnum::Square));
    }

    void M6(std::optional<CustomStruct> const& value)
    {
        ConsumeValue(value.value_or({});
    }
};
```

Bonus chatter: If you're going to retain the `IReference` that is passed to a method, you should re-wrap it inside your own `IReference` or `std::optional` because the passed-in `IReference` may come with strings attached. For example, it may have come from JavaScript, which is a single-threaded language, and that means that you can't use the passed-in `IReference` from a background thread. The caller might also try pulling some sneaky tricks, like having the `Value()` method return a different value each time you call it!

```
struct References : ReferencesT<References>
{
    IReference<double> m_savedValue;

    void M4(IReference<double> const& value)
    {
        // Re-wrap the IReference to avoid shenanigans.
        m_savedValue = value ? value.Value() : nullptr;
    }
};
```

Another way to wrap the value is inside a `std::optional`.

```
struct References : ReferencesT<References>
{
    std::optional<double> m_savedValue;

    void M4(IReference<double> const& value)
    {
        m_savedValue = value;
    }
};
```

Bonus bonus chatter: A customer who wanted to produce an `IReference` didn't realize that C++/WinRT came with pre-made implementations, so they tried to create their own:

```
// Don't do this.
template<typename Enum>
struct enum_reference :
    implements<enum_reference,
        IReference<Enum>>
{
    enum_reference(Enum value)
        : m_value(value) {}
    Enum Value() { return m_value; }
private:
    Enum const m_value;
};

IReference<CustomEnum> M1()
{
    return winrt::make<enum_reference>
        (CustomEnum::Circle);
}
```

This works according to the letter of the `IReference` interface, but it misses some finer points which are required in practice.

For example, in practice, objects which implement `IReference<Enum>` also need to implement `IPropertyValue` so that weakly-typed languages can use `IPropertyValue::Type()` to peek inside and realize that they have a numeric type, even if that numeric type is a custom enumeration. They also need to implement `IReference<T>` where `T` is the underlying type of the enumeration (either `int32_t` for plain enumerations or (either `uint32_t` for flags enumerations) so that the receipient can represent it as a `Number` type.

In general, you should use the C++/WinRT implementations, which have already been carefully written to address these subtleties.

Next time, we'll look at the C++/WinRT implementation of `IReference<T>` to see how it works. (In fact, that's how I figured out the rules for `IReference<T>`: Read the implementation and then reverse-engineer the rules that allow the implementation to work.)

Raymond Chen

Follow