Using perfect (and imperfect) forwarding to simplify C++ wrapper classes

devblogs.microsoft.com/oldnewthing/20230104-00

January 4, 2023

Raymond Chen

There may be cases where you have a C++ class that wants to wrap another C++ class that is contained as a member. In C++/WinRT, this can happen if your C++ class wants to act like an `IVector`, but with some bonus methods or extra functionality.

```
// MIDL

runtimeclass ItemCollection : [default] IVector<Item>
{
    String ToJson();
}
```

The implementation of this will probably consist of an internal `IVector` that forwards all of the `ItemCollection`'s `IVector` methods, plus one additional method for producing JSON.

```
// C++/WinRT

namespace winrt::Contoso::implementation
{
    struct ItemCollection : ItemCollectionT<ItemCollection>
    {
        Windows::Foundation::Collections::IIterator<Contoso::Item> First()
        {
            return m_items.First();
        }

        Contoso::Item GetAt(uint32_t index)
        {
            return m_items.GetAt(index);
        }

        uint32_t Size()
        {
            return m_items.Size();
        }

        bool IndexOf(Contoso::Item const& value, uint32_t& index)
        {
            return m_items.IndexOf(value, index);
        }

        uint32_t GetMany(uint32_t startIndex, array_view<Contoso::Item> items)
        {
            return m_items.GetMany(startIndex, items);
        }

        // And our bonus method
        hstring ToJson();

    private:
        Windows::Foundation::Collections::IVector<Contoso::Item> m_items;
    };
}
```

It's annoying that there's so much boilerplate to do the method forwarding, and that we have to keep looking up the parameters and return types so that each forwarder has the correct signature. Fortunately, we can use perfect forwarding to write most of them for us:

```
namespace winrt::Contoso::implementation
{
    struct ItemCollection : ItemCollectionT<ItemCollection>
    {
        template<typename...Args> decltype(auto) First(Args&&... args)
        {
            return m_items.First(std::forward<Args>(args)...);
        }

        template<typename...Args> decltype(auto) GetAt(Args&&... args)
        {
            return m_items.GetAt(std::forward<Args>(args)...);
        }

        template<typename...Args> decltype(auto) Size(Args&&... args)
        {
            return m_items.Size(std::forward<Args>(args)...);
        }

        template<typename...Args> decltype(auto) IndexOf(Args&&... args)
        {
            return m_items.IndexOf(std::forward<Args>(args)...);
        }

        template<typename...Args> decltype(auto) GetMany(Args&&... args)
        {
            return m_items.GetMany(std::forward<Args>(args)...);
        }

        // And our bonus method
        hstring ToJson();

    private:
        Windows::Foundation::Collections::IVector<Contoso::Item> m_items;
    };
}
```

Using perfect forwarding means that we don't have to remember the types and number of parameters for each of the methods, or what the methods return. Furthermore, if there are multiple overloads of a method, a single perfect forwarder covers them all!

```
template<typename...Args> decltype(auto) Name(Args&& args)
{
    return m_widget.Name(std::forward<Args>(args)...);
}
```

This forwarder forwards both the property setter and getter to the `m_widget`.

We can take some shortcuts here, because we know that the parameters to C++/WinRT Windows Runtime methods are safe to pass through as lvalues, so we can get rid of the `std::forward`. (Indeed, our original wrapper methods didn't try to preserve rvalue-ness.)

We also know that the return value from C++/WinRT Windows Runtime methods are not C++ references, so we can simplify the `decltype(auto)` to `auto`. This gives us

```
namespace winrt::Contoso::implementation
{
    struct ItemCollection : ItemCollectionT<ItemCollection>
    {
        template<typename...Args> auto First(Args&&... args)
        {
            return m_items.First(args...);
        }

        template<typename...Args> auto GetAt(Args&&... args)
        {
            return m_items.GetAt(args...);
        }

        template<typename...Args> auto Size(Args&&... args)
        {
            return m_items.Size(args...);
        }

        template<typename...Args> auto IndexOf(Args&&... args)
        {
            return m_items.IndexOf(args...);
        }

        template<typename...Args> auto GetMany(Args&&... args)
        {
            return m_items.GetMany(args...);
        }

        // And our bonus method
        hstring ToJson();

    private:
        Windows::Foundation::Collections::IVector<Contoso::Item> m_items;
    };
}
```

Finally, we can take advantage of C++20 abbreviated function templates, which now makes the forwarder small enough to fit on one line:

```
namespace winrt::Contoso::implementation
{
    struct ItemCollection : ItemCollectionT<ItemCollection>
    {
        auto First(auto&&... args) { return m_items.First(args...); }
        auto GetAt(auto&&... args) { return m_items.GetAt(args...); }
        auto Size(auto&&... args) { return m_items.Size(args...); }
        auto IndexOf(auto&&... args) { return m_items.IndexOf(args...); }
        auto GetMany(auto&&... args) { return m_items.GetMany(args...); }

        // And our bonus method
        hstring ToJson();

    private:
        Windows::Foundation::Collections::IVector<Contoso::Item> m_items;
    };
}
```

Raymond Chen

Follow