How to clone a Windows Runtime vector in the face of possible concurrent modification, part 4

devblogs.microsoft.com/oldnewthing/20230718-00

July 18, 2023

Raymond Chen

So far, we've looked at cloning a Windows Runtime vector in the face of possible concurrent modification, using C++/WinRT as our projection. Let's extend our solution to C++/CX.

The initial translation to C++/CX is straightforward. It's pretty much a literal translation. We can take advantage of the fact that the default constructor for hat pointers is a null pointer, so we don't have to use `winrt_empty_value()`.

```
template<typename V>
auto clone_as_vector(V const& v)
-> std::vector<decltype(v->GetAt(0))>
{
    using T = decltype(v->GetAt(0));
    std::conditional_t<
        std::is_same_v<T, bool>,
        std::unique_ptr<bool[]>,
        std::vector<T>> temp;
    unsigned expected;
    unsigned actual;
    do
    {
        expected = v->Size;
        if constexpr (std::is_same_v<T, bool>) {
            temp = std::make_unique<bool[]>(expected + 1);
            actual = v->GetMany(0, Platform::
                ArrayReference<T>(temp.get(), expected + 1));
        }
        else {
            temp.resize(expected + 1);
            actual = v->GetMany(0, Platform::
                ArrayReference<T>(temp.data(), expected + 1));
        }
        }
    } while (actual > expected);
    if constexpr (std::is_same_v<T, bool>) {
        return std::vector(temp.get(), temp.get() + actual);
    } else {
        temp.resize(actual);
        return temp;
    }
}

template<typename V>
auto CloneVector(V const& v)
-> Windows::Foundation::
    Collections::IVector<decltype(v->GetAt(0))>^
{
    return ref new Platform::Collections::
        Vector<decltype(v->GetAt(0))>(
            clone_as_vector(v));
}
```

Or if you prefer the total separation version:

```
template<typename V>
auto clone_as_vector(V const& v)
-> std::vector<decltype(v->GetAt(0))>
{
    using T = decltype(v->GetAt(0));
    unsigned expected;
    unsigned actual;
    if constexpr (std::is_same_v<T, bool>) {
        std::unique_ptr<bool[]> temp;
        do
        {
            expected = v->Size;
            temp = std::make_unique<bool[]>(expected + 1);
            actual = v->GetMany(0, Platform::
                ArrayReference<T>(temp.get(), expected + 1));
        } while (actual > expected);
        return std::vector(temp.get(), temp.get() + actual);
    } else {
        std::vector<T>> temp;
        do
        {
            expected = v->Size;
            temp.resize(expected + 1);
            actual = v->GetMany(0, Platform::
                ArrayReference<T>(temp.data(), expected + 1));
        } while (actual > expected);
        temp.erase(temp.begin() + actual, temp.end());
        return temp;
    }
}
```

However, there's an addition wrinkle to C++/CX: C++/WinRT uses the `==` operator to implement `IndexOf`, but C++/CX lets you pass a custom equality comparer. (The default comparer is `std::equal_to<T>`, which uses `==`.) Therefore, to capture the full generality of `Vector`, we allow the caller of `CloneVector` to specify an equality comparer.

Since we want to let the `V` be deduced, we resort to using type deduction from the future.[1]

```
template<typename E = void, typename V>
auto CloneVector(V const& v)
-> Windows::Foundation::
    Collections::IVector<decltype(v->GetAt(0))>^
{
    using T = decltype(v->GetAt(0));
    using ActualE = std::conditional_t<
        std::conditional_t<std::is_same_v<E, void>,
            std::equal_to<T>, E>;

    return ref new Platform::Collections::
        Vector<T, ActualE>(clone_as_vector(v));
}
```

There's another customization point that we haven't bothered with: `std::vector` lets you provide a custom allocator. I omitted that because custom allocators are rather esoteric, and besides, the `Platform::Collections::Vector` requires the provided `std::vector` to use the standard allocator.

Bonus chatter: You can see the implementation of the C++/CX collection types in the header file `collection.h` that comes with the Visual C++ compiler. Other parts of the C++/CX implementation can be found in `vccorlib.h`.

I'm not smart. I just know how to read.

[1] An alternative to type deduction from the future is overloading:

```
template<typename E, typename V>
auto CloneVector(V const& v)
-> Windows::Foundation::
    Collections::IVector<decltype(v->GetAt(0))>^
{
    using T = decltype(v->GetAt(0));
    return ref new Platform::Collections::
        Vector<T, E>(clone_as_vector(v));
}

template<typename V>
auto CloneVector(V const& v)
-> Windows::Foundation::
    Collections::IVector<decltype(v->GetAt(0))>^
{
    using T = decltype(v->GetAt(0));
    return CloneVector<std::equal_to<T>, V>(v);
}
```

The risk here is that somebody might want `E = V`.

```
// Error: Ambiguous call to overloaded function.
CloneVector<V>(v);
```

This is trying to create a clone whose equality comparer is `V`, but the compiler reports this as an ambiguous call because it matches both templates.

This could happen if `V` is a tree-like structure, so each element is a collection of other instances of itself.