How to clone a Windows Runtime vector in the face of possible concurrent modification, part 3

devblogs.microsoft.com/oldnewthing/20230714-00

July 14, 2023

Raymond Chen

Last time, we cloned a Windows Runtime vector in the face of possible concurrent modification, but we ran into trouble with `std::vector<bool>`.

As I noted some time ago, the C++ language defines a specialization `std::vector<bool>` which represents a packed bit array, rather than defining a separate type like `std::bitvector`. This has made a lot of people very angry and has been widely regarded as a bad move.

In our case, the specialization of `std::vector<bool>` breaks our `clone_as_vector` function, since it needs a `winrt::array_view<bool>`, which needs a C-style array of `bool` objects, not a packed bit array.

We'll have to detect the `bool` case in our function and substitute a `std::unique_ptr<bool[]>`.

```
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;
    uint32_t expected;
    uint32_t actual;
    do {
        expected = v.Size();
        if constexpr (std::is_same_v<T, bool>) {
            temp = std::make_unique<bool[]>(expected + 1);
            actual = v.GetMany(0,
                winrt::array_view(temp.get(), expected + 1));
        } else {
            temp.resize(expected + 1, winrt_empty_value<T>());
            actual = v.GetMany(0, temp);
        }
    } while (actual > expected);
    if constexpr (std::is_same_v<T, bool>) {
        return std::vector(temp.get(), temp.get() + actual);
    } else {
        temp.erase(temp.begin() + actual, temp.end());
        return temp;
    }
}
```

If the value type of the vector is a `bool`, then the `temp` changes to a `std::unique_ptr<bool[]>`, and that means that we have to change how we resize the array (namely by replacing it with a new allocation) and how we generate the `array_view<bool>` (by using the pointer + size constructor).

After the loop has captured the elements, we convert our C-style array of `bool` into a `std::vector<bool>`.

Now that I looked at it some more, it seems that there is barely any shared code at all. May as well just make it two functions glued together.

```
template<typename V>
auto clone_as_vector(V const& v)
-> std::vector<decltype(v.GetAt(0))>
{
    using T = decltype(v.GetAt(0));
    uint32_t expected;
    uint32_t 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,
                winrt::array_view(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, winrt_empty_value<T>());
            actual = v.GetMany(0, temp);
        } while (actual > expected);
        temp.erase(temp.begin() + actual, temp.end());
        return temp;
    }
}
```

Whew, we took care of the pesky `std::vector<bool>`.

Next time, we'll look at the potential infinite loop and whether it offers a denial of service attack.