# If you have to create a Windows Runtime Vector from C++/WinRT, do it as late as possible

devblogs.microsoft.com/oldnewthing/20240522-00

May 22, 2024

Raymond Chen

The Windows Runtime Vector is a language interoperability type, allowing your collection to be created from (say) C++, and manipulated by C#, JavaScript, Rust, Python, or any other language for which a Windows Runtime projection has been written.

When implementing a function that returns a C++/WinRT `IVector`, I often see this pattern:

```
winrt::IVector<winrt::hstring> GetNamesOfItems()
{
    auto v = winrt::multi_threaded_vector<winrt::hstring>();
    for (auto&& item : GetItems()) {
        v.Append(item.Name());
    }
    return v;
}
```

The `multi_threaded_vector<T>` function creates an object that implements `IVector<winrt::hstring>` and returns that interface. We then use the `IVector<T>::Append()` method to append a bunch of objects to that vector before returning it.

The gotcha is that `IVector` is a COM interface, so all method calls are virtual, which means that the compiler can't optimize the method calls (unless it uses advanced techniques like devirtualization). Also, COM method calls must adhere to the Windows Runtime ABI, so C++-specific concepts like rvalue references cannot be used. The parameter to `v.Append()` is always copied; there is no concept of a "move" operation at the ABI.

It's better to operate with `std::vector` as long as possible, and convert it to `IVector` as a final step.

```
winrt::IVector<winrt::hstring> GetNamesOfItems()
{
    auto v = std::vector<winrt::hstring>();
    for (auto&& item : GetItems()) {
        v.push_back(item.Name());
    }
    return winrt::multi_threaded_vector(std::move(v));
}
```

This time, we are pushing elements onto the back of a `std::vector`, and the compiler can inline that operation. Furthermore, `push_back` has an overload that accepts rvalue references, so the `winrt::hstring` returned by `item.Name()` can be moved into the vector rather than copied.

Staying with a `std::vector` also opens you up to other things. For example, you can sort them:

```
    std::sort(v.begin(), v.end());
```

This operation is not available for `IVector` because `std::sort` requires a random-access iterator, but `IVector`'s iterator does not meet the qualifications because the dereferencing oeprator returns a value, not a reference. There is no way to get a reference to a Windows Runtime vector element.

And since we have a `std::vector`, we can use other tricks, like resizing the vector and writing to it, thereby removing the reallocation code from the inner loop.

```
winrt::IVector<winrt::hstring> GetNamesOfItems()
{
    auto v = std::vector<winrt::hstring>();
    auto items = GetItems();
    v.resize(items.size());
    std::transform(items.begin(), items.end(), v.begin(),
        [](auto&& item) { return item.Name(); });
    return winrt::multi_threaded_vector(std::move(v));
}
```

Next time, we'll see a special case of this that lets you avoid the vector, too.