How can I produce a C-style array from a Windows Runtime asynchronous operation?

devblogs.microsoft.com/oldnewthing/20211203-00

December 3, 2021

Raymond Chen

C-style arrays fall through the cracks of the Windows Runtime. We spent the past few days working around the inability to pass them with transfer semantics. Today we look at another hole, namely the inability for them to be the product of an asynchronous operation.

There is no facility in the Windows Runtime for an `IAsyncOperation<T[]>`. An asynchronous operation can produce a primitive type, a structure type, or a reference type, but not an array.

As with the inability to transfer ownership of a C-style array, we can work around this by producing an `IBuffer`, assuming that the underlying type of the array has no destructor. It suffers from the same awkwardness of getting the data into and out of the buffer, as well as limiting yourself to languages that support raw pointers.

You might try wrapping the array inside a `PropertyValue` and using `PropertyValue.GetInt32Array` to retrieve it. However, this returns a copy of the underlying array, which can be a problem if the array is large and you're trying to avoid copies.

You could create your own wrapper type whose method for producing the C-style array is destructive:

```
namespace Sample
{
    runtimeclass WidgetIndicesHolder
    {
        Int32[] DetachIndexArray();
    }

    runtimeclass Widget
    {
        // The "indices" array will be very large.
        Windows.Foundation.IAsyncOperation<WidgetIndicesHolder>
            GetIndicesAsync();
    }
}
```

The consuming code would do this:

```
auto holder = co_await widget.GetIndicesAsync();
auto indices = holder.DetachIndexArray();
```

Another idea is to use the role reversal technique we came up with when dealing with ownership transfer and have the operation complete with a delegate, and then you call the delegate to get the C-style array.

```
namespace Sample
{
    delegate Int32[] WidgetIndicesProducer();

    runtimeclass Widget
    {
        // The "indices" array will be very large.
        Windows.Foundation.IAsyncOperation<WidgetIndicesProducer>
            GetIndicesAsync();
    }
}
```

In this case, the consuming code would look like this:

```
auto producer = co_await widget.GetIndicesAsync();
auto indices = producer();
```

Before digging into the implementation, let's compare these two strategies.

That's a trick question. The two strategies are effectively the same!

The similarly is clearer if I rewrite the delegate in terms of what it looks like at the ABI layer:

```
namespace Sample
{
    runtimeclass WidgetIndicesProducer
    {
        Int32[] Invoke();
    }

    runtimeclass Widget
    {
        // The "indices" array will be very large.
        Windows.Foundation.IAsyncOperation<WidgetIndicesProducer>
            GetIndicesAsync();
    }
}
```

A delegate is just a class with a single `Invoke` method whose parameters are the delegate parameters and whose return type is the delegate return type.[1] The language projection exposes the `Invoke` method as if it were a function call.

As a result, all that we did was rename `WidgetIndicesHolder` to `WidgetIndices-Producer` and `DetachIndexArray` to `Invoke`.

So let's use the delegate version, since it involves less typing, and it also allows us to reuse the `WidgetIndicesProducer` delegate that we used to transfer a C-style array into a Windows Runtime class.

On the consuming side, we can avoid the `producer` temporary by invoking the returned delegate immediately.

```
auto indices = (co_await widget.GetIndicesAsync())();
```

On the producing side, we wrap our return value inside the same sort of delegate we used when we looked at transferring ownership into a class: We move the C-style array into the delegate, and then move it out on request.

```
namespace winrt::Sample::implementation
{
    struct Widget : WidgetT<Widget>
    {
        IAsyncOperation<WidgetIndicesProducer>
        GetIndicesAsync()
        {
            auto result = co_await CalculateIndicesAsync();
            co_return WidgetIndicesProducer(
                [result = std::move(result)]() mutable
                { return std::move(result); });
        }
    };
}
```

We can take advantage of the delegate constructor that takes a lambda and avoid having to repeat the name of the delegate:

```
namespace winrt::Sample::implementation
{
    struct Widget : WidgetT<Widget>
    {
        IAsyncOperation<WidgetIndicesProducer>
        GetIndicesAsync()
        {
            auto result = ... calculate the indices ...
            co_return
                [result = std::move(result)]() mutable
                { return std::move(result); };
        }
    };
}
```

Note that it is essential that the C-style array be captured by value into the delegate. The `[&]` capture is definitely wrong, because the delegate is going to outlive the call to `GetIndicesAsync`.

Bonus chatter: One customer tried this:

```
namespace Sample
{
    runtimeclass Widget
    {
        // The "indices" array will be very large.
        Windows.Foundation.IAsyncAction GetIndicesAsync(out Int32[] indices);
    }
}
```

The idea here is that the caller passes in a variable to receive the indices, but the indices don't actually show up until the `IAsyncAction` completes. The intended calling usage would be something like

```
winrt::com_array<int32_t> indices;
co_await widget.GetIndicesAsync(indices);
// use the indices
```

Breaking it down a bit more:

```
winrt::com_array<int32_t> indices;
auto action = widget.GetIndicesAsync(indices);
// indices not yet ready
co_await action;
// okay, now we have indices
```

This doesn't work because the `[out]` parameters are valid only for the lifetime of the call. And the call is done when it returns an `IAsyncAction`.

The `indices` may no longer exist by the time the operation completes. For example, during the "indices not yet ready" comment, the caller might decide to go off and do something else, and that other thing might throw an exception, causing everything to unwind and the `indices` to disappear. Or maybe the caller did a `wait_for()` to wait for the indices with a timeout, and if the operation times out, it just gives up. Or maybe the `co_await` threw an exception when trying to register the continuation.

The implementation of `GetIndicesAsync` doesn't know that any of these things have happened, and it will happily write to an already-destroyed object, which is a great source of memory corruption.

For garbage-collected languages, it's even worse, because even in the absence of errors, garbage collection may run while the `IAsyncAction` is pending. The garbage collector might move the `out` parameter or even destroy it completely if the variable is not used after

the `await`.

And certainly it's not going to work for marshalled calls, because the server-side implementation receives a server-side `indices` variable which is marshalled back to the client side when the `GetIndicesAsync` function returns an `IAsyncAction`. COM doesn't know that "Oh, wait, don't marshal it back yet. I'm going to update it some more later."

So don't do that. It doesn't work and corrupts memory.

[1] This isn't strictly true, but it's true enough for the purpose of this discussion. Another difference is that delegates derive directly from `IUnknown` rather than from `IInspectable`.

Raymond Chen

Follow