How can I transfer ownership of a C-style array to a Windows Runtime component?

devblogs.microsoft.com/oldnewthing/20211201-00

December 1, 2021

Raymond Chen

Suppose you have a large C-style array, and you want to transfer ownership of that array to a Windows Runtime component. For concreteness, let's say that we have this:

```
namespace Sample
{
    runtimeclass Widget
    {
        // The "indices" array will be very large.
        void SetIndices(Int32[] indices);
    }
}
```

You might be tempted to do something like this:

```
void SetWidgetIndices(Widget const& widget)
{
  winrt::com_array<int32_t> indices = CalculateIndices();
  widget.SetIndices(std::move(indices));
}
```

Using a `std::move` means that you are fine with the method stealing the resources out of the object, and you no-so-secretly hope that it will do so.

But it won't.

If you look at the rules for passing C-style arrays across the Windows Runtime ABI boundary, you'll see that a parameter declared as `T[] v` uses the *PassArray* pattern. In that pattern, ownership of the data remains with the caller, and the recipient must make a copy if it wants to access it beyond the end of the method.

So that's not going to work.

The *FillArray* pattern doesn't work either. That is for asking the method to fill a preallocated array, which is not what we're doing here.

And the last pattern, *ReceiveArray* doesn't work, because that is for transferring ownership from the method back to the caller.

So we're stuck. How can we do this without incurring a copy of a large block of data?

One option is to express the data in the form of an `IVector` instead of a C-style array. Since `IVector` is an interface, the recipient can just `AddRef` the interface and continue using it later. The major downside of this is that it costs you a lot of performance, since access to each element of an `IVector` is a virtual method call.[1]

Another option is to express the data in the form of a byte buffer, `IBuffer`. However, this works only for types that have no destructor (like integers). Furthermore, getting the data into and out of the buffer is a bit awkward, since you have to do some casting of the byte buffer to get it into the form you want.

```
auto data = reinterpret_cast<int32_t*>(m_buffer.data());
auto size = m_buffer.Length() / sizeof(int32_t);
auto view = winrt::array_view(data, data + size);
// access the data via the view
```

It's also a problem for languages which do not have raw pointer types.

It occurred to me that there's still a third option, but you have to change your point of view: Since the only ownership-transferring operation is from the method to its caller, reverse the roles so that the caller can "return" the array to the method.

```
namespace Sample
{
    // The "indices" array will be very large.
    delegate Int32[] WidgetIndicesProducer();

    runtimeclass Widget
    {
        void SetIndices(WidgetIndicesProducer producer);
    }
}
```

To provide the indices, you actually provide a callback that generates the indices and returns them via the *ReceiveArray* pattern.

```
void SetWidgetIndices(Widget const& widget)
{
  winrt::com_array<int32_t> indices = CalculateIndices();
  widget.SetIndices(
    [&] { return std::move(indices); });
}
```

The `[&]` capture assumes that the lambda will be called back before the `indices` variable is destructed. A safer version would be to store the indices inside the lambda itself.

```
void SetWidgetIndices(Widget const& widget)
{
  widget.SetIndices(
    [indices = CalculateIndices()]() mutable
    { return std::move(indices); });
}
```

You could simplify this to

```
void SetWidgetIndices(Widget const& widget)
{
  widget.SetIndices([] { return CalculateIndices(); });
}
```

but note that this changes the order of evaluation, since `CalculateIndices()` is called from inside the call to `SetIndices()`.

Yes, it's awkward, but at least it's a workaround. You can make it slightly less awkward with a wrapper function:

```
void SetWidgetIndices(
    Widget const& widget,
    winrt::com_array<int32_t>&& indices)
{
  widget.SetIndices(
    [indices = std::move(indices)]() mutable
    { return std::move(indices); });
}
```

Next time, we'll look at the implementation side of this method.

[1] You can use `IVector::GetMany()` to slurp out the elements, but that's still a copy operation, which we are trying to avoid.

Raymond Chen

Follow