# On the importance of managing the stream pointer when manipulating marshal data

devblogs.microsoft.com/oldnewthing/20220620-00

June 20, 2022

Raymond Chen

Managing the stream pointer is an important part of dealing with marshal data because the marshaled data for an individual object is combined with the marshaled data for other objects into a giant stream, which has a recursive structure. For example, consider a structure like this:

```
struct Int32AndTwoThings
{
    int32_t value;
    IThing1* thing1;
    IThing2* thing2;
};
```

If COM needs to marshal this structure, it needs to marshal the value and the two things, and the memory stream would consist of the integer, followed by two serialized things:

| | |
|---|---|
| value | Int32And-TwoThings |
| thing1 | |
| thing2 | |

Each of the things might themselves be complex objects with sub-things:

| | | |
|---|---|---|
| value | | Int32And-TwoThings |
| sub-thing 1.1 | thing1 | |
| sub-thing 1.2 | | |
| sub-thing 2.1 | thing2 | |
| sub-thing 2.2 | | |

And so on.

Once you realize this, the rules for managing the stream pointer become self-evident.

To generate the marshal data, COM starts with the root object `Int32AndTwoThings` and asks it to generate its marshal data. This structure writes out the integer value and then recursively asks `thing1` and `thing2` to generate their marshal data. Next, `thing1` consists of two sub-things, so it asks each of those two sub-things to generate *their* marshal data. When `thing1` is done, the same thing happens with `thing2`. Each block of marshal data is appended to the stream-so-far, resulting in the diagram above.

For unmarshaling, the objects are processed in the same order, but this time instead of writing the marshal data, the unmarshalers *read* the data. The root object `Int32AndTwoThings` reads its 32-bit integer, leaving the stream pointer at the start of where `thing1` wrote its marshal data.

| | | | |
|---|---|---|---|
| → | value | | Int32AndTwoThings |
| | sub-thing 1.1 | thing1 | |
| | sub-thing 1.2 | | |
| | sub-thing 2.1 | thing2 | |
| | sub-thing 2.2 | | |

Next comes a recursive call to unmarshal `thing1`, and it in turn starts by asking sub-thing 1.1 to unmarshal. The stream pointer is exactly at the same place it was when sub-thing 1.1 generated its marshal data, so sub-thing 1.1 gets back the data it wrote and can unmarshal itself, leaving the stream pointer just past the end of the sub-thing 1.1 marshal data:

| | | | |
|---|---|---|---|
| | value | | Int32AndTwoThings |
| → | sub-thing 1.1 | thing1 | |
| | sub-thing 1.2 | | |
| | sub-thing 2.1 | thing2 | |
| | sub-thing 2.2 | | |

This process continues until the stream is consumed. It works provided that (1) the components are unmarshaled in the same order that they are marshaled, and (2) each component leaves the stream pointer just past the last byte it generated or consumed.

When it comes time to destroy the marshal data, we walk through the components recursively the same way that we did when unmarshaling. It's just that instead of consuming the data, we destroy it. It's important to finish with the stream pointer just past the last byte destroyed so that each component is destroying the bytes that it originally generated.

A common mistake in custom marshalers is to forget to position the stream pointer past the marshal data. You might think, "Well, I have nothing to destroy, since none of the things I wrote require any special cleanup; they're just integers and strings. So I'll just return `S_OK` without doing anything."

It's fine that you have nothing to clean up, but by neglecting to update the stream pointer, you cause all the objects that come after you in the stream to operate on effectively corrupt marshal data. This usually manifests itself as crashes (if you're lucky) or memory leaks (if you're not) because the subsequent objects are unable to perform proper cleanup. And this type of memory leak is very hard to track down: You marshal an object, and somehow the reference to the object simply vanishes into thin air. The culprit is totally unrelated to the object that was leaked, and is almost certainly long gone. Something to keep in mind when you're tracking down a mysterious leak of a COM object, and you find an outstanding reference coming from deep inside COM's marshaling infrastructure. That reference is being captured in the marshal data, and you need to track down whether the marshal data was leaked, or if something went wrong during the `ReleaseMarshalData`.

Next time, we'll implement an object that does a shallow copy, and which demonstrates this recursive nature of marshaling.

Raymond Chen

**Follow**