When you have a SAFEARRAY, you need to know what it is a SAFEARRAY *of*

devblogs.microsoft.com/oldnewthing/20130125-00

January 25, 2013

Raymond Chen

A customer had a problem with `SAFEARRAY`, or more specifically, with `CComSafeArray`.

```
CComSafeArray<VARIANT> sa;
GetAwesomeArray(&sa);


LONG lb = sa.GetLowerBound();
LONG ub = sa.GetUpperBound();


for (LONG i = lb; i <= ub; i++) {
 CComVariant item = sa.GetAt(i);
 … use the item …
}
```

The `GetAt` method returns a `VARIANT&`, and when it is copy-constructed into `item`, the `DISP_E_BADVARTYPE` exception is raised.

On the other hand, if the offending line is changed to

```
CComQIPtr<IAwesome> pAwesome = sa.GetAt(i).punkVal;
```

then the problem goes away.

Your initial reaction to this code would be that there is an off-by-one error in the loop control, but it turns out that there isn't because `SAFEARRAY` uses inclusive upper bounds rather than exclusive.

The first step in debugging this is seeing what is in the bad variant that makes the copy constructor think it's not a valid variant type.

Inspecting in the debugger shows that the variant returned by `GetAt` has a valid `punk`, but the `vt` is `0x1234`. Well, that's not a valid variant type, so that's the proximate cause of the problem.

How did an invalid variant type get into your `SAFEARRAY`?

At this point the customer realized that maybe their code to create the array was faulty, so they offered to share it.

```
void GetAwesomeArray(SAFEARRAY **ppsa)
{
 SAFEARRAY *psa = SafeArrayCreateVector(VT_UNKNOWN, 0, m_count);
 for (LONG i = 0; i < m_count; i++) {
  CComPtr<IAwesome> spAwesome;
  CreateAwesomeThing(i, &spAwesome);
  SafeArrayPutElement(psa, &i, spAwesome);
 }
 *ppsa = psa;
}
```

Okay, now all the pieces fell into place.

The `GetAwesomeArray` function is creating an array of `VT_UNKNOWN`, but the code fragment that calls `GetAwesomeArray` treats it as an array of `VT_VARIANT`.

Your array of `IUnknown*` is being misinterpreted as an array of `VARIANT`. That explains all the symptoms: The `vt` is wrong, because it's really just the low-order word of the first `IUnknown*`. Ignoring the `vt` and going straight for the `punk` seems to work because that's where the *second* `IUnknown*` happens to be. (Or third, if you are compiling as 32-bit.)

In other words, it's as if you did a `reinterpret_cast<VARIANT&>(punkArray[0])`.

If you had used regular C-style arrays or a C++ collection, then the compile-time type checking would have told you that you mismatched the producer and consumer. But since you went through a `SAFEARRAY`, that compile-time type information is lost, since a `SAFEARRAY` is a polymorphic array. It now becomes your job to keep track of what you have an array *of*, and its dimensions and bounds.

You can keep track of this information via documentation, "This function returns a 1-dimensional `SAFEARRAY` of `VT_IUNKNOWN`, with lower bound 0 and variable upper bound." Or you can check at runtime, by calling `SafeArrayGetVartype` to see what the base type is, and `SafeGetDim` to see how many dimensions the array has, and `SafeArrayGetLBound` and `SafeArrayGetUBound` to obtain the upper and lower bounds for those dimensions.

The code above seemed not to be sure which model it wanted to use. It trusted the base type and the dimension, but checked the upper and lower bounds.

Anyway, assuming we are going with the "keep track via documentation" approach, the solution for the original problem is to have the producer and consumer agree on exactly what kind of `SAFEARRAY` is being handed around. Either produce an array of `VT_UNKNOWN` and

consume it as a `CComSafeArray<IUnknown*>` or produce an array of `VT_VARIANT` and consume it as a `CComSafeArray<VARIANT>`.