What's the difference between VARIANT and VARIANTARG?

devblogs.microsoft.com/oldnewthing/20171221-00

December 21, 2017

Raymond Chen

One of my colleagues asked me, "What's the difference between `VARIANT` and `VARIANTARG`?"

If you look at the definitions in the `oaidl.h` header file, you'll see that `VARIANTARG` is just an alias for `VARIANT`.

```
typedef VARIANT VARIANTARG;

typedef VARIANT *LPVARIANTARG;
```

"Why have two names for the same thing?"

The two structures are physically identical, but the rules surrounding them are different.

This is mentioned rather opaquely in the documentation for `VARIANT`:

> **VARIANTARG** describes arguments passed within DISPPARAMS, and **VARIANT** to specify variant data that cannot be passed by reference.
>
> When a variant refers to another variant by using the VT_VARIANT | VT_BYREF vartype, the variant being referred to cannot also be of type VT_VARIANT | VT_BYREF. VARIANTs can be passed by value, even if VARIANTARGs cannot.

The first sentence says that you use `VARIANTARG` as part of a `DISPPARAMS`, which is the structure used to pass parameters (also known as "arguments") to methods of dispatch interfaces.

The second sentence is not relevant to the discussion. It says that only one level of pointer chasing is allowed. You can't send the method on a wild goose chase where you pass a variant that says "The real data is over there, in that other variant", and then have the second variant say, "Ha ha, fooled, you. The real data is over there in that other other variant."

The third sentence starts to hint at the underlying issue. It says that `VARIANT` s can be passed by value, but `VARIANTARG` s cannot.

Interesting, but no real insight as to why you can pass `VARIANT` by value but not `VARIANTARG` .

There's another MSDN page titled VARIANT and VARIANTARG. Maybe that'll help us get to the bottom of the mystery.

> The **VARIANT** type cannot have the **VT_BYREF** bit set.

Aha, that's the difference. The `VARIANTARG` structure is allowed to say, "Hey, I don't contain the data you want, but you can look over there for it." For example, it could set its variant type to `VT_BYREF | VT_I4` to say, "There is an integer, but it's not stored in the `lVal` member. Instead, you have to go to the `plVal` member, which is a pointer to the integer you want."

This explains why `VARIANT` can be copied, but `VARIANTARG` cannot: If you try to copy a `VARIANTARG` that uses `VT_BYREF` , you are just copying the raw pointer to the data, not the data itself. You have no control over the memory being pointed to, so you have no way to prevent it from being freed.

Using `VT_BYREF` is allowed in a `DISPPARAMS` because the caller assumes the responsibility of keeping the pointed-to data valid for the duration of the method call. That's just one of the basic ground rules of programming, specifically the stability requirement. The caller has to wait for the method call to return before it can free the memory pointed to by the `VARIANTARG` .

Okay, so what if you're implementing a method and you want to make a copy of the `VARIANTARG` ? How do you deal with the `VT_BYREF` ?

This is where the `VariantCopyInd` function comes into play. This function takes a `VARIANTARG` , possibly with `VT_BYREF` , and converts it into a `VARIANT` , with all `VT_BYREF` removed. It does this by chasing the pointer one level and copying the value back into the `VARIANT` . For example, if the `VARIANTARG` were a `VT_BYREF | VT_I4` , then the `VariantCopyInd` function would follow the `plVal` pointer, read the integer stored there, and copy it to the output `VARIANT` 's `lVal` member, resulting in a simple `VT_I4` .

The "Ind" therefore stands for "Indirect". The `VariantCopyInd` function indirects through the pointer hiding inside the `VT_BYREF` .

Well, that was a strange bit of spelunking.

Raymond Chen

Follow