Why does COM require output pointers to be initialized even on failure?

devblogs.microsoft.com/oldnewthing/20091231-00

December 31, 2009

Raymond Chen

One of the rules of COM is that if a parameter is marked as an output pointer, then you have to initialize the thing it points to, even if your function failed and you have nothing to return. For example, we saw the problems that can occur if you forget to set the output pointer to `NULL` in the `IUnknown::QueryInterface` method. Why does COM have this rule? Doesn't it know that a failure is a failure? Because there are failures and there are sort-of failures. A function can return an error code despite having partially succeeded. For example, if a function receives a buffer that is too small to hold all the data that is available, it might fill the buffer as much as possible, and then return an error like `ERROR_MORE_DATA` to say, "Hey, I gave you what I could, but there's more out there." The COM enumerator pattern specifies that if the caller asks for more items than there are remaining in the enumeration, then the `IEnumXXX::Next` method should return as many items as it can and return `S_FALSE`. The COM library itself doesn't know the details and semantics of every method and interface. It doesn't know that your `IFlipper::Frob` method has a special return value of `E_PENDING` which means "I started the operation and here is a provisional result, but the final result will come later." Why does COM even care about your method calls anyway? After all, isn't a COM method just a C++ virtual function? I'm just calling a function in some other part of my program; why does COM get involved? COM indeed doesn't get involved, except when it needs to, such as when the object is being marshalled between apartments or processes. In those cases, a COM method call does not go directly to the C++ virtual method of the target object but rather goes through a proxy which packages up the input parameters and ships them to the final destination, a process known as marshalling. On the other side, a receiver proxy unpacks the parameters (unmarshals) and calls the destination object. When the destination object completes executing the method, the process repeats but this time with the output parameters: The output parameters are packaged up and sent back to the calling thread, where they are unpackaged and returned to the original caller. (Parameters which are in/out end up being marshalled in both directions.) The code that does the marshalling and unmarshalling doesn't know any special behaviors of your method. It doesn't know that when the method returns `E_PENDING` then the first and third output parameters contain values that should be returned to the caller, but the second does not. It just packages up all the output parameters and sends them back. A related scenario is the method where one input parameter controls whether another input parameter is ignored. In those cases, you still can’t pass garbage as the second input parameter even though it is ignored, because the COM marshaller doesn’t know the special semantics of the first parameter with respect to the interpretation of the second. Now, you might say, “Well, I know that I’m not using any of the cases where the COM marshaller gets involved, so I’m going to break these rules because there is nobody around to enforce them!”

Except that you yourself are relying on this behavior; you just don’t realize it. More about that next time.

Raymond Chen

Follow