The difference between assignment and attachment with ATL smart pointers

devblogs.microsoft.com/oldnewthing/20091120-00

November 20, 2009

Raymond Chen

Last time, I presented a puzzle regarding a memory leak. Here's the relevant code fragment:

```
CComPtr<IStream> pMemoryStream;
CComPtr<IXmlReader> pReader;
UINT nDepth = 0;
//Open read-only input stream
pMemoryStream = ::SHCreateMemStream(utf8Xml, cbUtf8Xml);
```

The problem here is assigning the return value of `SHCreateMemStream` to a smart pointer instead of attaching it.

The `SHCreateMemStream` function creates a memory stream and returns a pointer to it. That pointer has a reference count of one, in accordance with COM rules that a function which produces a reference calls `AddRef`, and the responsibility is placed upon the recipient to call `Release`. The assignment operator for `CComPtr<T>` is a copy operation: It `AddRef`s the pointer and saves it. You're still on the hook for the reference count of the original pointer.

```
ATLINLINE ATLAPI_(IUnknown*) AtlComPtrAssign(IUnknown** pp, IUnknown* lp)
{
        if (lp != NULL)
                lp->AddRef();
        if (*pp)
                (*pp)->Release();
        *pp = lp;
        return lp;
}
template <class T>
class CComPtr
{
public:
        ...
        T* operator=(T* lp)
        {
                return (T*)AtlComPtrAssign((IUnknown**)&p, lp);
        }
```

Observe that assigning a `T*` to a `CComPtr<T>` AddRefs the incoming pointer and `Release`s the old pointer (if any). When the `CComPtr<T>` is destructed, it will release the pointer, undoing the `AddRef` that was performed by the assignment operator. In other words, assignment followed by destruction has a net effect of zero on the pointer you assigned. The operation behaves like a copy.

Another way of putting a pointer into a `CComPtr<T>` is with the `Attach` operator. This is a transfer operation:

```
        void Attach(T* p2)
        {
                if (p)
                        p->Release();
                p = p2;
        }
```

Observe that there is no `AddRef` here. When the `CComPtr<T>` is destructed, it will perform the `Release`, which doesn’t undo any operation performed by the `Attach`. Instead, it releases the reference count held by the original pointer you attached.

Let’s put this in a table, since people seem to like tables:

Operation	Behavior	Semantics
Attach()	Takes ownership	Transfer semantics
operator=()	Creates a new reference	Copy semantics

You use the `Attach` method when you want to assume responsibility for releasing the pointer (ownership transfer). You use the assignment operator when you want the original pointer to continue to be responsible for its own release (no ownership transfer).

There is also a `Detach` method which is the opposite of `Attach`: Detaching a pointer from the `CComPtr<T>` means "I am taking over responsibility for releasing this pointer." The `CComPtr<T>` gives you its pointer and then forgets about it; you're now on your own.

The memory leak in the code fragment above occurs because the assignment operator has copy semantics, but we wanted transfer semantics, since we want the smart pointer to take the responsibility for releasing the pointer when it is destructed.

```
pMemoryStream.Attach(::SHCreateMemStream(utf8Xml, cbUtf8Xml));
```

The `CComPtr<T>::operator=(T*)` method is definitely one of the more dangerous methods in the `CComPtr<T>` repertoire, because it's so easy to assign a pointer to a smart pointer without giving it a moment's thought. (Another dangerous method is the `T** CComPtr<T>::operator&()`, but at least that has an assertion to try to catch the bad usages. Even nastier is the secret QI'ing assignment operator.) I have to say that there is merit to Ben Hutchings' recommendation simply not to allow a simple pointer to be assigned to a smart pointer, precisely because the semantics are easily misunderstood. (The boost library, for example, follows Ben's recommendation.)

Here's another exercise based on what you've learned:

> Application Verifier told us that we have a memory leak, and we traced it back to the function `GetTextAsInteger`.
>
> ```
> BSTR GetInnerText(IXMLDOMNode *node)
> {
>     BSTR bstrText = NULL;
>     node->get_text(&bstrText);
>     return bstrText;
> }
> DWORD GetTextAsInteger(IXMLDOMNode *node)
> {
>     DWORD value = 0;
>     CComVariant innerText = GetInnerText(node);
>     hr = VariantChangeType(&innerText, &innerText, 0, VT_UI4);
>     if (SUCCEEDED(hr))
>     {
>         value = V_UI4(&innerText);
>     }
>     return value;
> }
> ```
>
> Obviously, the problem is that we passed the same input and output pointers to `VariantChangeType`, causing the output integer to overwrite the input `BSTR`, resulting in the leak of the `BSTR`. But when we fixed the function, we still got the leak:
>
> ```
> DWORD GetTextAsInteger(IXMLDOMNode *node)
> {
>     DWORD value = 0;
>     CComVariant innerText = GetInnerText(node);
>     CComVariant textAsValue;
>     hr = VariantChangeType(&innerText, &textAsValue, 0, VT_UI4);
>     if (SUCCEEDED(hr))
>     {
>         value = V_UI4(&textAsValue);
>     }
>     return value;
> }
> ```
>
> Is there a leak in the `VariantChangeType` function itself?

Hint: It is in fact explicitly documented that the output parameter to `VariantChangeType` can be equal to the input parameter, which results in an in-place conversion. There was nothing wrong with the original call to `VariantChangeType`.

Raymond Chen

Follow