C++, C, VBScript, Windows, Visual Studio, COM, Dev

COM in plain C, Part 3

By Jeff Glatt

COM collections in C

Posted : **7 May 2006**
Updated : **7 May 2006**
Views : **23,876**

- Download source files - 174 Kb

Contents

Introduction

Sometimes, we may need to maintain a list of items. For example, let's say that our COM component is designed to operate some hardware PCI card we've designed. And, let's say that a user can install several of these cards into one computer, and we want our component to control all of the available cards, letting an application retrieve information about each card, and access each card individually.

In other words, when our component runs, it needs to query what cards are in the system, and make available to an application, a list of information about all of them. For the sake of argument, let's assume that the only information we need to make available is the "name" of each card, where the first card in the system will be named "Port 1", the second card will be named "Port 2", etc.

Since we don't know ahead of time how many cards will be in a system, the best approach would be to create some struct that can be linked to others of its kind, in a linked list. For example, maybe we'll define an `IENUMITEM` struct to hold information about a card:

```
typedef struct _IENUMITEM {
   // To link several IENUMITEMs into a list.

   struct _IENUMITEM  *next;
   // This item's value (ie, its port name).

   char               *value;
} IENUMITEM;
```

And if we have three ports, our linked list of `IENUMITEM`s would look something like this (where we normally would `GlobalAlloc` the `IENUMITEM`s, but I'll just statically declare/initialize them below for quick edification):

```
IENUMITEM Port1 = {&Port2, "Port 1"};
IENUMITEM Port2 = {&Port3, "Port 2"};
IENUMITEM Port3 = {0, "Port 3"};
```

In COM lingo, we refer to a group of related items as a "collection". So, the linked list of three items above would be our collection.

But there's one problem with our IENUMITEM above. It has a char * member, which is not automation compatible. We could change this to a BSTR, which is automation compatible. Better yet, let's put a VARIANT right in the IENUMITEM. This has the advantage of making an IENUMITEM generic (i.e., it can store any kind of automation data type). And as we'll see later, we have to return an item's value to an application, using a VARIANT. So, since we need to deal with a VARIANT anyway, let's store it in our IENUMITEM that way. Here's the new definition of an IENUMITEM:

```
typedef struct _IENUMITEM {
   struct _IENUMITEM  *next;
   VARIANT            *value;
} IENUMITEM;
```

And, here is how we may allocate one, and set its value to "Port 1" (error checking omitted):

```
IENUMITEM  *enumItem;

enumItem = (IENUMITEM *)GlobalAlloc(GMEM_FIXED, sizeof(IENUMITEM));
enumItem->next = 0;
enumItem->value.vt = VT_BSTR;
enumItem->value.bstrVal = SysAllocString(L"Port 1");
```

Defining a collection object

Remember that some script languages do not have the concept of a pointer. Such a language can't possibly walk the above list on its own, because the first member of each IENUMITEM is a pointer to the next IENUMITEM. So, we need to provide an object that helps the application (using our COM object) walk this list, fetching each item's value.

Because Microsoft left it up to the Visual Basic programmers (ugh) to come up with this object, they chose to base it upon an IDispatch. In other words, the VTable for our object must start with the three IUnknown functions as all COM objects do (QueryInterface, AddRef, and Release), and then must be followed immediately with the four standard IDispatch functions (GetTypeInfoCount, GetTypeInfo, GetIDsOfNames, and Invoke). Our object must then have three more functions. In our IDL file, when we define the VTable (i.e., the interface) for this object we'll create, we must name these three extra functions Count, Item, and _NewEnum. In our actual VTable for the object, we can use any name we want for the pointers (although I'll stick to those names). Why? Because no one is ever going to directly call them. No one will ever even know that we're going to add these three functions to our VTable. These functions must be called only indirectly via the Invoke function of our object -- even if you're using a language like C that **could** have called them directly if only Microsoft's Visual Basic programmers had designed this to accommodate more powerful languages. That's the terrible price we all pay for letting Visual Basic programmers design this. Furthermore, in our IDL file, we must assign the ID DISPID_VALUE to the Item function, and DISPID_NEWENUM to the _NewEnum function. We **must**. The Count function can be assigned any positive number of our choosing for its ID. Does there appear to be any sense of consistency or logic to this whatsoever? No? Remember -- Visual Basic programmers.

We'll also need to generate a new GUID for this object's VTable, for use in our IDL file (i.e., the type library).

Let's look at the definition of this object. We can call it anything we like. I'll choose to call it an ICollection:

```
// Our ICollection VTable's GUID

// {F69902B1-20A0-4e99-97ED-CD671AA87B5C}

DEFINE_GUID(IID_ICollection, 0xf69902b1, 0x20a0, 0x4e99, 0x97,
            0xed, 0xcd, 0x67, 0x1a, 0xa8, 0x7b, 0x5c);

// Our ICollection's VTable

#undef  INTERFACE
#define INTERFACE ICollection
DECLARE_INTERFACE_ (INTERFACE, IDispatch)
{
   // IUnknown functions

   STDMETHOD  (QueryInterface) (THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)  (THIS) PURE;
   STDMETHOD_ (ULONG, Release) (THIS) PURE;
   // IDispatch functions

   STDMETHOD_ (ULONG, GetTypeInfoCount) (THIS_ UINT *) PURE;
   STDMETHOD_ (ULONG, GetTypeInfo) (THIS_ UINT, LCID,
                                    ITypeInfo **) PURE;
   STDMETHOD_ (ULONG, GetIDsOfNames) (THIS_ REFIID, LPOLESTR *,
               UINT, LCID, DISPID *) PURE;
   STDMETHOD_ (ULONG, Invoke) (THIS_ DISPID,
               REFIID, LCID, WORD, DISPPARAMS *,
               VARIANT *, EXCEPINFO *, UINT *) PURE;
   // Extra functions

   STDMETHOD (Count) (THIS_ long *);
   STDMETHOD (Item) (THIS_ long, VARIANT *);
   STDMETHOD (_NewEnum) (THIS_ IUnknown **);
};
```

This shouldn't look totally unfamiliar. First, I ran *GUIDGEN.EXE* to create another GUID. I pasted it above, and gave it the variable name `IID_ICollection`. That's the GUID for our `ICollection` object's VTable.

Next, we define our `ICollection` object's VTable. We're using the same macro we used when we defined our `IExample2` VTable in the last article. And as mentioned earlier, it starts with the three `IUnknown` functions, followed by the four `IDispatch` functions, just like `IExample2` did when we added support for scripting languages. These functions are going to perform the same duties as the respective functions did in `IExample2`.

Finally, we add the three extra functions `Count`, `Item`, and `_NewEnum`. Later, we'll examine what they do.

Remember that the above macro automatically defines our `ICollection` object as so:

```
typedef struct {
   ICollectionVtbl  *lpVtbl;
} ICollection;
```

In other words, it is defined to have only one data member -- a pointer to our `ICollection`'s VTable. This member is named `lpVtbl`, of course. But, we need to add an extra `DWORD` member to keep a reference count (just like we did to `IExample2`, and consequently defined a `MyRealIExample2` to accommodate any extra data members). So, we'll define a `MyRealICollection` as so:

```
typedef struct {
   ICollectionVtbl  *lpVtbl;
   DWORD            count;
} MyRealICollection;
```

Now, let's take a look at how we'll define the VTable (interface) in our IDL file:

```
[uuid(F69902B1-20A0-4e99-97ED-CD671AA87B5C), oleautomation, object]
interface ICollection : IDispatch
{
   [propget, id(1)]
   HRESULT Count([out, retval] long *);
   [propget, id(DISPID_VALUE)]
   HRESULT Item([in] long, [out, retval] VARIANT *);
   [propget, id(DISPID_NEWENUM), restricted]
   HRESULT _NewEnum([out, retval] IUnknown **);
};
```

Note that we've used the new GUID created for our `ICollection` VTable.

We use the `oleautomation` keyword because our functions will accept only automation compatible data types, and return such.

Finally, we use the `object` keyword to indicate that this VTable belongs to some object. It's not going to be an object that an application gets hold of via our `IClassFactory`. (Later, we'll see how an application gets hold of one of our `ICollection` objects.) In fact, an application is never going to be told that our `ICollection` object is anything except a standard `IDispatch` object (i.e., our `ICollection` object is going to masquerade as an ordinary `IDispatch` as far as an application or script engine is concerned). Since our object is going to masquerade as an ordinary `IDispatch`, there's no need to specifically define our `ICollection` object itself inside of our IDL file (unlike we had to do with `IExample2` when we listed what interfaces were in it, and which interface was the default). All scripting languages and applications already know all about an ordinary `IDispatch` object, so there's no need to put any information about this `IDispatch`-impersonator (`ICollection`) object itself in our IDL file. We need to define only its VTable as above, and mark that VTable as belonging to some object.

Note that, on the interface line, we do specify that this VTable contains `IDispatch` functions in the expected place and order. So, it certainly can masquerade as an `IDispatch`.

We, then, list the three extra functions. Notice that they are all defined as `propget`. Also notice that the `Item` function has an ID (DISPID) of `DISPID_VALUE`, and the `_NewEnum` function has an ID of `DISPID_NEWENUM`. (We also use the `restricted` keyword on that latter one, because an object browser is not supposed to show the `_NewEnum` function. It is supposed to be a function that only a script engine would call internally, but an actual script would never call.) For the `Count` function, we can choose any positive ID number, so I arbitrarily chose 1. (It doesn't matter that `IExample2`'s `Buffer` function also has an ID of 1. These VTables are used in two different objects, so their IDs do not need to be unique between the two VTables).

Later, we'll actually get into writing these functions.

Before we change our `IExample2` sources, let's just copy the entire *IExample2* directory to a new directory named *IExample3*. We'll also rename all of the files to reflect this new directory (i.e., *IExample2.h* becomes *IExample3.h*, and *IExample2.c* becomes *IExample3.c*, etc.). And while we're at it, let's edit the files in this new directory, renaming our `IExample2` object to an `IExample3`, to distinguish it from our previous sources. All we do is search and replace every instance of "IExample2" with "IExample3". And, don't forget to run *GUIDGEN.EXE* to make new GUIDs for `IExample3`, its VTable, and the type library. Replace the old GUIDs in *IExample3.h*, and update the UUIDs in *IExample3.idl*. After all, we don't want our new DLL (which we'll name *IExample3.dll*) to conflict with our previous *IExample2.dll*. I've done all this for you, and put the resulting files in an *IExample3* directory.

Some helper functions

We need to construct our list of port names. Let's write a helper function to do that. We'll arbitrarily assume we have three ports, and so create three `IENUMITEM`s. We'll store the head of this list in a global variable `PortsList`. And, we also need a helper function to free the list when we're done with it.

```
IENUMITEM  *PortsList;

// This is just a helper function to free

// up our PortsList. Called when our DLL unloads.

void freePortsCollection(void)
{
   IENUMITEM *item;

   item = PortsList;

   // Is there another item in the list?

   while ((item = PortsList))
   {
      // Get the next item *before* we delete this one

      PortsList = item->next;

      // If the item's value is an object, we need to Release()

      // it. If it's a BSTR, we need to SysFreeString() it.

      // VariantClear does this for us.

      VariantClear(&item->value);

      // Free the IENUMITEM.

      GlobalFree(item);
   }
}

// This is just a helper function to initialize our Ports list.

// Called when our DLL first loads.

HRESULT initPortsCollection(void)
{
   IENUMITEM *item;

   // Add a "Port 1" IENUMITEM to our list.

   if ((PortsList = item =
       (IENUMITEM *)GlobalAlloc(GMEM_FIXED,
       sizeof(IENUMITEM))))
   {
      item->next = 0;
      item->value.vt = VT_BSTR;
      if ((item->value.bstrVal = SysAllocString(L"Port 1")))
      {
         // Add a "Port 2" IENUMITEM to our list.

         if ((item->next = (IENUMITEM *)GlobalAlloc(
                              GMEM_FIXED, sizeof(IENUMITEM))))
         {
            item = item->next;
            item->value.vt = VT_BSTR;
            if ((item->value.bstrVal = SysAllocString(L"Port 2")))
            {
               // Add a "Port 3" IENUMITEM to our list.

               if ((item->next = (IENUMITEM *)GlobalAlloc(
                                    GMEM_FIXED, sizeof(IENUMITEM))))
               {
                  item = item->next;
                  item->next = 0;
                  item->value.vt = VT_BSTR;
                  if ((item->value.bstrVal =
                       SysAllocString(L"Port 3")))
                     return(S_OK);
               }
            }
         }
      }
   }

   // Error
```

We're also going to add a second global variable named `CollectionTypeInfo` to hold a `ITypeInfo` for our `ICollection`... uh, the `IDispatch` object. (We'll talk about why we need this, later). So, we need to add that global variable, and then let's write two helper functions -- one to initialize the variable to zero, and one to Release the `ITypeInfo`:

```
// Our ICollection's ITypeInfo. We need only one of these so

// we can make it global

ITypeInfo    *CollectionTypeInfo;

// This helper function initializes our ICollection TypeInfo.

// It's called when our DLL is loading.

void initCollectionTypeInfo(void)
{
   // We haven't yet created the ITypeInfo for our ICollection

   CollectionTypeInfo = 0;
}

// This helper function just Release()'s our ICollection TypeInfo.

// It's called when our DLL is unloading.

void freeCollectionTypeInfo(void)
{
   if (CollectionTypeInfo)
       CollectionTypeInfo->lpVtbl->AddRef(CollectionTypeInfo);
}
```

Now, we need to modify our `DllMain` to call these helper functions:

```
BOOL WINAPI DllMain(HINSTANCE instance, DWORD fdwReason, LPVOID lpvReserved)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
      {
         MyTypeInfo = 0;

         // Initialize our ICollection stuff

         initCollectionTypeInfo();

         // Initialize our Ports list

         if (initPortsCollection())
         {
            MessageBox(0, "Can't allocate the PortsList", "ERROR", MB_OK);
            return(0);
         }

         OutstandingObjects = LockCount = 0;

         MyIClassFactoryObj.lpVtbl = (IClassFactoryVtbl *)&IClassFactory_Vtbl;

         DisableThreadLibraryCalls(instance);
         break;
      }

      case DLL_PROCESS_DETACH:
      {
         // Free our Ports list

         freePortsCollection();

         // Free our ICollection ITypeInfo

         freeCollectionTypeInfo();

         if (MyTypeInfo) MyTypeInfo->lpVtbl->Release(MyTypeInfo);
      }
   }

   return(1);
}
```

Our collection object's functions

Now, let's write the actual functions for our `ICollection` (actually, our `MyRealICollection`). Rather than put this code in *IExample3.c*, let's create a separate source file for it, called *PortNames.c*. And, we'll put the definition of our `ICollection` VTable and its GUID in a separate *PortNames.h* file. (We can also put our above helper functions in *PortNames.c*.)

The `IUnknown` functions (`QueryInterface`, `AddRef`, and `Release`) and `IDispatch` functions (`GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, and `Invoke`) are almost the same as the respective functions for our `IExample3` object. So rather than reproduce the code here, I'll refer you to the file *PortNames.c* (in the directory *IExample3*).

One difference is that our `ICollection` functions are passed an `ICollection` object pointer (instead of an `IExample3` object pointer), of course. And, `ICollection`'s `Release` function is slightly different (since unlike `IExample3`'s `Release`, there is no buffer member to free).

The other key difference is the call to `GetTypeInfoOfGuid`. Note that we're passing the GUID for our `ICollection` VTable (instead of the GUID for our `IExample3` VTable, like we do in *IExample3.c*). Here's the deal. When we get an `ITypeInfo` for `IExample3` (by calling `loadMyTypeInfo` in *IExample3.c*), we pass the GUID of `IExample3`'s VTable to the OLE function `GetTypeInfoOfGuid`. The implication of this is that the default `ITypeInfo` Microsoft creates for us is good for getting information **only** about the functions in our `IExample3` VTable. **It cannot be used to get information about the functions in another object's VTable.** But, we **do** need an `ITypeInfo` that will provide us with information about our `ICollection`'s functions. So now, we have to call `GetTypeInfoOfGuid` again, but this time we pass the GUID for our

ICollection object's VTable (i.e., the new GUID I created). This will return a second ITypeInfo (which we store in that global variable we added, named CollectionTypeInfo). This second ITypeInfo can be used with ICollection's IDispatch functions to get information about ICollection's functions. And it can be used with DispInvoke and DispGetIDsOfNames in ICollection's Invoke and GetIDsOfNames functions to do almost all of the work for us -- just like we did with IExample3's ITypeInfo.

Notice that ICollection's IDispatch functions use this new ITypeInfo, whereas IExample3's IDispatch functions use IExample3's ITypeInfo. They are not the same ITypeInfo, and cannot be used interchangeably.

All that's left is to write the three extra functions, Count, Item, and _NewEnum.

The Count function is simple. It is passed a pointer to a long. Count fills in that pointer with the total number of items in our list. For example, above we have three ports (IENUMITEM structs) in our list, so we would return a 3.

Here's our Count function:

```
STDMETHODIMP Count(ICollection *this, long *total)
{
   DWORD     count;
   IENUMITEM *item;

   // Count how many items in the list by just walking it all

   // the way to the last IENUMITEM, incrementing a count for

   // each item.

   count = 0;
   item = (IENUMITEM *)&PortsList;
   while ((item = item->next)) ++count;

   // Return the total.

   *total = count;

   return(S_OK);
}
```

The Item function is also simple. It is passed a long that tells us which item is being queried (where 0 is the first item, 1 is the second item, etc.). It is also passed a VARIANT into which we copy the value of that item.

Here's our Item function:

```
STDMETHODIMP Item(ICollection *this, long index, VARIANT *ret)
{
   IENUMITEM  *item;

   // Assume we have nothing to return.

   ret->vt = VT_EMPTY;

   // Locate to the item that the caller wants.

   item = (IENUMITEM *)PortsList;
   while (item && index--) item = item->next;

   // Any more items left?

   if (item)
   {
      // Copy the item's value to the VARIANT that the caller supplied.

      // If what we're returning to the caller is an object, we must AddRef()

      // it on the caller's behalf. The caller is expected to Release() it

      // when done with it. If what we're returning is a BSTR, then we must

      // SysAllocString a copy of it. Caller is expected to SysFreeString it.

      // Other datatypes are simply copied to the caller's VARIANT as is.

      // VariantCopy() does all this for us. It also returns S_OK if all

      // went well.

      return(VariantCopy(ret, &item->value));
   }

   // If no more items, return S_FALSE.

   return(S_FALSE);
}
```

As you'll note from the comment above, the OLE function `VariantCopy` does half the work for us.

For now, we'll gloss over the `_NewEnum` function. We'll stick in a dummy one that returns `E_NOTIMPL`.

Once we have all the `ICollection` functions written, we can statically declare its VTable:

```
static const ICollectionVtbl ICollectionVTable =
      {Collection_QueryInterface,
       Collection_AddRef,
       Collection_Release,
       GetTypeInfoCount,
       GetTypeInfo,
       GetIDsOfNames,
       Invoke,
       Count,
       Item,
       _NewEnum};
```

How an application obtains our collection object

Let's consider how an application gets hold of one of our `MyRealICollection` objects. The easiest thing to do is to add another (extra) function to our `IExample3` object. The application will call this new function to allocate and receive one of our `MyRealICollection` objects. (But we're going to lie to the application and tell it that it's just an ordinary `IDispatch`.)

We need to change the definition of `IExample3`'s VTable (in *IExample3.h*), to add this new function, which I'll arbitrarily call `GetPorts`. I'll define it as being passed a handle to an `IDispatch`, which is where we'll return the pointer to our newly allocated `MyRealICollection`... er, `IDispatch`. Yeah, that's it. It's just an `IDispatch`. Wink, wink. Here's our updated `IExample3` VTable:

```
// IExample3's VTable

#undef  INTERFACE
#define INTERFACE IExample3
DECLARE_INTERFACE_ (INTERFACE, IDispatch)
{
    // IUnknown functions

    STDMETHOD  (QueryInterface) (THIS_ REFIID, void **) PURE;
    STDMETHOD_ (ULONG, AddRef) (THIS) PURE;
    STDMETHOD_ (ULONG, Release) (THIS) PURE;
    // IDispatch functions

    STDMETHOD_ (ULONG, GetTypeInfoCount)(THIS_ UINT *) PURE;
    STDMETHOD_ (ULONG, GetTypeInfo) (THIS_ UINT, LCID, ITypeInfo **) PURE;
    STDMETHOD_ (ULONG, GetIDsOfNames) (THIS_ REFIID, LPOLESTR *,
                UINT, LCID, DISPID *) PURE;
    STDMETHOD_ (ULONG, Invoke) (THIS_ DISPID, REFIID, LCID,
                WORD, DISPPARAMS *, VARIANT *, EXCEPINFO *, UINT *) PURE;
    // Extra functions

    STDMETHOD  (SetString) (THIS_ BSTR) PURE;
    STDMETHOD  (GetString) (THIS_ BSTR *) PURE;
    STDMETHOD  (GetPorts) (THIS_ IDispatch **) PURE;  // <--- Added GetPorts here

};
```

Notice that I added `GetPorts` to the end of the VTable. Also notice, I'm specifying that `GetPorts` will fill in an `IDispatch` pointer (even though it will really be our `MyRealICollection`). It's an `IDispatch`. Honest. Would I lie?

And I have to make the same change in the IDL file for our `IExample3` VTable:

```
[uuid(CFADB388-9563-4591-AABB-BE7794AEC17C), dual,
            oleautomation, hidden, nonextensible]
interface IExample3VTbl : IDispatch
{
   [helpstring("Sets the test string.")]
   [id(1), propput] HRESULT Buffer([in] BSTR);
   [helpstring("Gets the test string.")]
   [id(1), propget] HRESULT Buffer([out, retval] BSTR *);
   [helpstring("Gets the enumeration for our hardware ports.")]
   [id(2), propget] HRESULT Ports([out,
           retval] IDispatch **);  // <--- Added GetPorts here

};
```

Notice, I made this newly added function a `propget`, just like `Buffer`. This is so that a script can use an ordinary assignment instruction to get our `MyRealICollection`... duh!... `IDispatch` object. The member is called "`Ports`" as far as the script is concerned. Never mind that we don't actually have a `Ports` data member in our `IExample3` object. This is sort of a phony member. But the script doesn't need to know that.

And, I arbitrarily gave it a DISPID of 2.

Don't forget that we need to add this function to our statically declared `IExample3Vtbl` in *IExample3.c*:

```
static const IExample3Vtbl IExample3_Vtbl = {QueryInterface,
AddRef,
Release,
GetTypeInfoCount,
GetTypeInfo,
GetIDsOfNames,
Invoke,
SetString,
GetString,
GetPorts}; // <--- Added GetPorts here
```

So, we need to write the `GetPorts` function:

```
static HRESULT STDMETHODCALLTYPE GetPorts(IExample3 *this, IDispatch **portsObj)
{
   // Create an IDispatch to enumerate our port names.

   // Caller is responsible for Release()'ing

   // it. NOTE: We're really returning a MyRealICollection,

   // but the caller doesn't know that.

   // He thinks we're returning an IDispatch,

   // which is ok because a MyRealICollection's

   // VTable starts with the 3 IUnknown functions

   // followed by the 4 IDispatch functions, just

   // like a real IDispatch object's VTable

   if (!(*portsObj = allocPortsCollection()))
      return(E_OUTOFMEMORY);
}
```

The above simply calls another helper function (named `allocPortsCollection`) we'll put in *PortNames.c*. This helper function does the work of `GlobalAlloc`'ing a `MyRealICollection` and initializing it. It's fairly similar to how our `IClassFactory`'s `CreateInstance` `GlobalAlloc`s an `IExample3` and initializes it. We even bump up the count of outstanding objects because our `MyRealICollection`... er, `IDispatch` object is going to be given to some application (which is expected to later `Release` it).

```
IDispatch * allocPortsCollection(void)
{
   MyRealICollection *collection;

   // Allocate a MyRealICollection

   if ((collection = (MyRealICollection *)GlobalAlloc(
                     GMEM_FIXED, sizeof(MyRealICollection))))
   {
      // Store its VTable

      collection->lpVtbl = (ICollectionVtbl *)&ICollectionVTable;

      // AddRef it

      collection->count = 1;

      // Indicate another outstanding object since we'll be returning it

      // to an application which is expected to Release() it

      InterlockedIncrement(&OutstandingObjects);
   }

   // Return it as if it were an IDispatch (which it can be used as)

   return((IDispatch *)collection);
}
```

We're done. You can compile the *IExample3.dll*. To register it, just take `IExample2`'s register utility (*RegIExample2*) and replace every "IExample2" with "IExample3". After all, there is nothing about `IExample3` which requires us to register it any differently than `IExample2`. Likewise, to un-register it, modify `IExample2`'s un-register utility (*UnregIExample2*).

A VBScript example

Let's write a VBScript example that uses our collection to display the port names. I've written such an example (*IExample3.vbs*), and placed it in the *IExample3* directory.

Of course, the VBScript needs to first call `CreateObject` to get one of our `IExample3` objects. If you installed it properly, it should have a ProdID of "`IExample3.object`". Now that the script has our `IExample3`, it can simply access that phony "`Ports`" member to get one of our `MyRealICollection`... damn!... `IDispatch` objects to use. Here, we assign it to a variable named "`coll`".

```
Set coll = myObj.Ports
```

Next, we can call the `Count` function to determine how many port names there are. Actually, because our type library defined this function as `propget`, the script can use an assignment.

```
count = coll.Count
```

Now, we loop around, calling the `Item` function to get each port name, and display it:

```
For i = 0 To count - 1
   MsgBox coll.Item(i)
Next
```

And, that's it.

A C example

For a C/C++ application to use our collection object, it needs to call our `Count` and `Item` functions indirectly, through the `Invoke` function. Hold on to your seats because this is going to be a bumpy ride. The MS Visual Basic programmers designed the `IDispatch` functions to be passed arguments, and return values, such that those folks could easily add COM support to VB and get it out the door quickly. But they didn't give any concern to how easy it would be to utilize the `IDispatch` functions from any other language. It's a major pain in the butt from C/C++.

In the directory *IExample3App*, is an example C application that does just what the VBScript above does. It obtains our `Ports` collection object, and uses it to display the names of all ports. I'm not going to discuss how the C app gets our `IExample3` object. That is exactly the same as how it gets an `IExample2` object (except we `#include IExample3.h`, and use the `IExample3` object's GUID).

The main point of interest begins where we call our `IExample3`'s `GetPorts` function to get our `MyRealICollection`... *cough*... `IDispatch` object. That starts where I put the following comment (in *IExample3App.c*):

```
// STUDY THIS
```

Peruse all of that code and read the comments. They detail the steps you need to jump though to use `IDispatch` functions from C/C++. Then, take an MS Visual Basic programmer out to lunch to "thank" him, and slip lots of hot pepper into his food when he isn't looking.

An IEnumVARIANT object

If we look at the following lines in our `Item` function, something may alarm us:

```
// Locate to the item that the caller wants.

item = (IENUMITEM *)PortsList;
while (item && index--) item = item->next;
```

Every time someone calls our `Item` function, we have to start at the head of the list and search to the desired item. Assume we have 30,000 items in the list. Say, an application asks us to fetch item 28,000. We have to walk through 27,000 items before we get to the desired item. Now, let's say the application calls `Item` again to request item 29,000. Even though it's only one more link away, we start all over from the head of the list and walk though the 28,000 items. Obviously, this isn't very efficient.

We could perhaps add another data member to our MyRealICollection. This member would store the "position" where we last left off in the list. Microsoft thought about this, and then decided, instead of monkey-ing around with the collection object (which really isn't designed to be efficiently called from C/C++ anyway, thanks to VB developers), a second standard COM object, called an IEnumVARIANT, would be defined. The main purpose of an IEnumVARIANT is to store the current position that an application has "read from" within the list. But given how inefficient and troublesome it is for a C/C++ app to call our collection's Item function indirectly via Invoke, MS decided to have some functions in the IEnumVARIANT that an app can **directly** call to do what we previously did with the collection's Item function... and more. Specifically, an IEnumVARIANT has four functions in it (plus the three IUnknown functions, of course) named Next, Skip, Reset, and Clone.

- The IEnumVARIANT's Next function makes our collection object's Item function superfluous. With a single call to Next, an app can return the values of several items at once (by supplying a count of how many items to return, and an array of VARIANTs to store all the values). So, an app could call Next to read four items. On the next call to Next, our IEnumVARIANT will automatically start reading at the fifth item in the list, and return however many values the app requests.
- The IEnumVARIANT's Reset simply resets the position back to the start of the list.
- The IEnumVARIANT's Skip sets the position to a particular point (i.e., it's analogous to seeking within a disk file, except, here it sets the position within our list).
- The IEnumVARIANT's Clone allocates (and returns to the app) another IEnumVARIANT object whose position is the same as the IEnumVARIANT being cloned. This is used in case an app wants to nest loops where it's necessary to remember more than one position in a particular list.

Microsoft has already defined (in an include file that ships with your compiler) an IEnumVARIANT object and its VTable (i.e., Microsoft has already decided what functions are in the VTable, what those functions are passed, and what they return). So, we don't need to do that. But like with all the objects we've created so far, we need to add a couple extra data members to our IEnumVARIANT. So once again, we'll define a MyRealIEnumVariant that has these extra members.

But before we modify our IExample3 sources, let's again make a new directory named *IExample4*. We'll do that thing where we copy the sources to this new directory, and rename/edit them. Search and replace "IExample3" with "IExample4". Run *GUIDGEN.EXE* to create new GUIDs, and put them in *IExample4.h*, *PortNames.h*, and *IExample4.idl*. Once again, I've done this for you, and created an *IExample4* directory with the new files.

In *PortNames.c*, we'll add the definition of our MyRealIEnumVariant object, write all its functions, and statically declare its VTable. This all done starting at the comment:

```
//============================================================

//================== IEnumVARIANT functions ==================

//============================================================
```

In fact, you should now be quite familiar with what its QueryInterface, AddRef, and Release functions do. And the other four functions are rather trivial, so you can peruse the source code comments to get details about those functions.

The real question is *"How does an app get hold of one of my IEnumVARIANT objects?"*. Remember before, how we ignored our collection object's _NewEnum function, and just had it return E_NOTIMPL? Well, guess what. That's the function an app calls to get one of our IEnumVARIANT objects, so now we have to write some real code for it.

In other words, to get one of our IEnumVARIANTs, the app must first get our IExample4 object, call our IExample4's GetPorts function to get our collection object, and then call our collection's _NewEnum function to get an IEnumVARIANT. It's not the short way home, but that's how it works. Some bad news: a C/C++ app can't directly call our collection's _NewEnum function. Just like with our collection's Count and Item functions, a C/C++ app has to indirectly call _NewEnum through Invoke. Blech. The good news is that, once our C/C++ app has the IEnumVARIANT, the collection object can be Release()d and we're done with that latter aberration.

So, let's examine our collection's _NewEnum function:

```
STDMETHODIMP _NewEnum(ICollection *this, IUnknown **enumObj)
{
   IEnumVARIANT *enumVariant;

   if (!(enumVariant = allocIEnumVARIANT())) return(E_OUTOFMEMORY);
   *enumObj = (IUnknown *)enumVariant;
   return(S_OK);
}
```

This just calls our helper function named allocIEnumVARIANT which allocates and initializes our IEnumVARIANT (actually, a MyRealIEnumVariant) much like how our IClassFactory's CreateInstance allocates our IExample4 object, or our IExample4's GetPorts function allocates our collection object. There's really nothing new here.

But notice that _NewEnum asks the app to pass a handle where we return an IUnknown object -- not an IEnumVARIANT. Yes, it's true that we're really returning our IEnumVARIANT, but it is masquerading as an IUnknown object, and it can do that because its VTable starts with the three IUnknown functions.

"But why masquerade? Didn't you just say that _NewEnum is used to get hold of our IEnumVARIANT?"

Yes... in a roundabout manner. [Cue scary monster movie soundtrack.]

In a previous article, I alluded to the fact that a COM object could actually have many VTables inside of it. We say that such an object has "multiple interfaces". Microsoft decided that _NewEnum should be able to pass back an object that could have multiple interfaces, and the IEnumVARIANT may be just one of many VTables in it (and not even the first VTable in the object). So, what an app is expected to do is take this IUnknown object we give it, and call that object's QueryInterface, passing the GUID for an IEnumVARIANT (IID_IEnumVARIANT, which is defined in Microsoft's include files for us). And then, QueryInterface will return a pointer to the IEnumVARIANT VTable (i.e., essentially, the IEnumVARIANT embedded inside of whatever object that IUnknown object really is -- because you just know it's something else masquerading as an IUnknown).

In our case, the app is going to call our IEnumVARIANT's QueryInterface, asking us for an IEnumVARIANT. And, we're just going to return the same pointer again. Totally unnecessary. Inefficient. Illogical. But, that's sometimes how COM is when you've got parts of it that are designed by MS VB developers (who had a vested interest in making the whole IDispatch thing a lot like how VB internally calls its own built-in functions, thus minimizing their work, and foisting this design on everyone else regardless of how inconvenient and unwieldy that can be), and other programmers who wanted to use things like multiple interfaces (which a VBScript can't even directly use -- talk about a comedy of errors).

Anyway, we've got our IEnumVARIANT support done, so we can compile *IExample4.dll*. To register it, once again, you can modify *RegIExample2.c*, searching and replacing "IExample2" with "IExample4".

Another VBScript example

A VBScript can absolutely not call our collection's _NewEnum function (because this may return an object with multiple interfaces that need to be QueryInterfaced -- and a VBScript can't deal with such an object). So, a VBScript cannot obtain our IEnumVARIANT and call its functions.

Does this mean an IEnumVARIANT is useless to a VBScript? Nope. The VBScript engine can itself use our IEnumVARIANT. When does the engine do that? When the script uses a For Each loop to enumerate the items in our collection. In the directory *IExample4* is a VBScript named *IExample4.vbs* which uses such a For Each loop. This is a slightly different way for the script to do the same thing that *IExample3.vbs* did, but it's more efficient under-the-hood (because the engine uses our IEnumVARIANT's Next function, instead of the VBScript using our collection object's Item function). And, it's slightly less coding on the part of the script because the engine retrieves the item's value on behalf of the script. Here's how it's done:

```
set myObj = CreateObject("IExample4.object")
Set coll = myObj.Ports
For Each elem In coll
   MsgBox elem
Next
```

The first two lines are the same as *IExample3.vbs* (except that we use the ProdID for *IExample4.dll* now).

But the loop is different. When the VBScript engine executes that `For Each` line, it gets hold of our `IEnumVARIANT` (by calling our collection's `_NewEnum`, and doing a `QueryInterface` for `IID_IEnumVARIANT`). It stores away this `IEnumVARIANT` internally so it can be used on subsequent iterations of the loop. Then, it calls our `IEnumVARIANT`'s `Next` function, asking for an item's value to be returned. Of course, the first time `Next` is called, we return the port name for the first item (i.e., the string "Port 1"). The VB engine stuffs this string into the variable "`elem`". This is all done in that one VBScript instruction. Now, the script simply displays the value of "`elem`" (which is the string "Port 1"). On the next iteration of the loop, the VB engine once again calls our `IEnumVARIANT`'s `Next` function, asking for one more item's value. This is the second time `Next` has been called, so of course, we return the second item's value, which is the string "Port 2". The VB engine now updates the `elem` variable to this new value. And the script displays "Port 2". This continues on until the VB engine calls our `IEnumVARIANT`'s `Next`, and we have no more items to return. At that point, `Next` returns `S_FALSE` (instead of `S_OK`) to the engine. And, the engine drops out of the loop (and `Release`s our `IEnumVARIANT`).

Another C example

In the directory *IExample4App*, there is a C example that demonstrates how to get and use our `IEnumVARIANT`. We still have to mess with the collection object and its `Invoke`. But at least, the loop is more efficient. And the whole thing is less involved than using the collection object's `Item` function.

A more generic approach

If you look at the `IEnumVARIANT` and the collection functions in *PortName.c*, you'll see that these are hard-wired to work upon only our list of port names (i.e., `PortsList`). But with a small bit of retooling, we can rewrite those functions to work upon any linked list of `IENUMITEM`s we throw at them. In other words, we can make these functions generic, so if our component needs to maintain several different types of lists, we can more easily provide further collection and `IEnumVARIANT` objects that reuse these same functions without any further modification. So, let's separate just the `IEnumVARIANT` and the collection functions into a new, separate source file named *IEnumVariant.c*. The only stuff we'll leave in *PortNames.c* is the code that specifically accesses our `PortsList`.

But first, we'll do that thing where we create a new *IExample5* directory, copy the files there, and rename/edit them. You should know the routine by now. I've done the work, and created an *IExample5* directory.

Instead of simply declaring a global variable that is the list itself, we'll wrap the list in another struct we'll call a `IENUMLIST`, as so:

```
typedef struct {
   struct _IENUMITEM *head;
   DWORD             count;
} IENUMLIST;
```

The `head` member is where the list is stored. And we've added a `count` field that will be incremented each time we create another collection or `IEnumVARIANT` object that uses this particular list.

Now, we change our `PortsList` global to this new struct:

```
IENUMLIST    PortsList;
```

The key to making our collection and `IEnumVARIANT` functions more generic is adding an extra data member to our `MyRealIEnumVariant` and `MyRealICollection` objects. We'll add a member to our `MyRealICollection` that holds a pointer to the `IENUMLIST` it should operate upon. And, we'll write a new

helper function that will allocate a MyRealICollection object. This helper function will be passed that IENUMLIST pointer, and will store it in this new data member added to our MyRealICollection. I've written such a function (allocICollection) and put it in *IEnumVariant.c*. It's passed a pointer to whatever IENUMLIST we want our collection object to operate upon.

We'll also add a data member to our MyRealIEnumVariant. It will do the same thing as the new data member added to our MyRealICollection (i.e., hold a pointer to the IENUMLIST that our IEnumVARIANT operates upon).

Other changes are trivial, but what we're left with in *PortNames.c* is just a little bit of code to create and delete our PortsList, and to create a collection object specifically to wrap PortsList.

To create another list, and add support for a collection and IEnumVARIANT objects, all we need do is create another source file, much like what is in *PortNames.c*. In fact, let's do this.

Let's assume that we want to create a list of network cards in a system. And for each network card, we want to provide two pieces of information: the name of the network card, and its MAC address. We'll put the code in *NetCards.c* and *NetCards.h*.

We want to return more than one piece of information per item. (I.e., each IENUMITEM will pertain to a single network card. And for each network card, we want to let the script know the card's name and its MAC address.) The best way to approach this is to create a "sub-object" which we'll arbitrarily call a INetwork object. We'll put two functions in this object, Name and Address. The Name function will return a BSTR of the network card's name, and the Address function will return a BSTR of the MAC address.

We're going to create an INetwork object for each network card in the computer. Then, we're going to create an IENUMITEM in our list for that card. We'll stuff the INetwork pointer into the IENUMITEM VARIANT's punkVal field, and set the vt field to VT_DISPATCH. This list of IENUMITEMs (containing INetwork objects) is created in allocNetworkObjectsCollection (in *NetCards.c*).

In order to be accessible from VBScript, we'll need to include the IDispatch functions in our INetwork's VTable. Of course, this means that we'll need an ITypeInfo for our INetwork's VTable. So, we'll have to generate a new GUID for its VTable. And, we'll have to call GetTypeInfoOfGuid, passing that new GUID, to get an ITypeInfo for it. We'll save this in a global variable named NetTypeInfo. All this code is in *NetCards.c*. This code should look remarkably similar to what you saw with our collection and IExample5 objects, because those also have IDispatch functions and need their ITypeInfo objects.

And to make our INetwork's extra functions (i.e., Name and Address) directly callable from C/C++, we'll need to declare its VTable as "dual" in our IDL file. We'll also have to include its VTable definition in *IExample5.h* so a C/C++ app knows exactly what order and arguments those extra functions involve. I've added the definition of our INetwork's VTable to both *IExample5.h* and *IExample5.idl*. Note that it looks remarkably similar to our IExample5 object. Both contain IDispatch functions. Both are declared dual. Their only differences are in their extra functions. But like our IEnumVARIANT object, our INetwork object itself doesn't need to be declared in our IDL. Only its VTable needs to be declared. After all, our INetwork is going to look like a standard IDispatch object to an app, except that its extra functions **will** be in its VTable, and a C/C++ app can call them directly.

By giving our IENUMLIST a count field, we can determine when all the collection and IEnumVARIANT objects are done with its list, and therefore delete the list whenever we want. (I.e., unlike the previous example, we aren't restricted to deleting the list only when our DLL terminates.) In fact, you'll notice that we don't actually create our list of INetwork objects until an app actually asks us for our collection of network cards. And then, we delete the list of INetwork objects as soon as the last collection/IEnumVARIANT using that list is Release()d.

In the *IExample5* directory is a VBScript example that does a For Each loop to gain access to each one of our INetwork objects, and then calls the Name and Address functions to display the card's name and MAC address.

Add/Remove items

Sometimes, we may want to give a script/app the ability to add or remove items from our list. The traditional

approach for this is to put `Add` and `Remove` functions in our collection object. Because each list may contain different kinds of items, requiring different kinds of data, you'll have to define another collection object specific to a particular list. You'll define its VTable, and put these extra two functions (`Add` and `Remove`) in it. The `Add` function will have to be written so that the script/app can pass in whatever data is needed to create the new item. (And, the function may first search the list to see if there's already a matching item, so that duplicate items are avoided). The `Remove` function should be passed some arg(s) that allow you to locate the desired item to delete.

Of course, you'll need to generate a GUID for this new collection's VTable. And, you'll need to create an `ITypeInfo` for it by passing its GUID to `GetTypeInfoOfGuid`, and storing the `ITypeInfo` in a global that is used by `Invoke`, `GetTypeInfo`, and `GetIDsOfNames`.

The good news is that you can use many of the same functions (in *IEnumVariant.c*) that our original collection (`ICollection`) and `IEnumVARIANT` uses. So, there is not as much new code to write as you may expect.

License

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

6 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/com_in_c3.aspx** to post and view comments on this article, or click **here** to get a print view with messages.