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

COM in plain C, Part 4

By **Jeff Glatt**

Make a COM object with multiple interfaces, in C.

Posted : **16 May 2006**
Updated : **16 May 2006**
Views : **23,354**

22 votes for this Article.
Popularity: 6.33 Rating: **4.71** out of 5

Download source files - 194 Kb

Contents

Introduction

Sometimes, a COM object may have what is called *multiple interfaces*. Saying that a COM object has multiple interfaces is just another way of saying that the object is composed of several "sub-objects". Each sub-object is itself a full COM object, with its own `lpVtbl` member (pointer to its own VTable), and its own VTable set of functions (including its own `QueryInterface`, `AddRef`, and `Release` functions).

An object may have numerous kinds of sub-objects. For example, let's say that our object manages one audio card. An audio card can have several input and output jacks upon it. Let's say that we want to have a sub-object for each input/output jack. For example, let's assume we have a Line In jack, a Mic In jack, and a Speaker Out jack. We, therefore, may have three sub-objects called `ILineIn`, `IMicIn`, and `ISpeakerOut`. Each individual object may have functions that specifically control its individual jack. For example, maybe the Mic In jack has a setting whereby the jack can be switched between accepting a low impedance, or a high impedance, microphone. So, our `IMicIn` object may have a `SetImpedance` function. But the `ILineIn` and `ISpeakerOut` objects do not need such a function because that setting isn't pertinent to those jacks. Maybe the `ILineIn` and/or `ISpeakerOut` objects may need other functions that are specific to those jacks. Let's say that the `ILineIn` has a function called `Mute` to mute the signal. Our `ISpeakerOut` has a function called `SetVolume` to set its volume. Here is how we may define these three objects, using the macro that Microsoft has provided to define a COM object:

```
#undef  INTERFACE
#define INTERFACE  IMicIn
DECLARE_INTERFACE_ (INTERFACE, IUnknown)
{
   STDMETHOD  (QueryInterface) (THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)  (THIS) PURE;
   STDMETHOD_ (ULONG, Release) (THIS) PURE;
   STDMETHOD  (SetImpedance)   (THIS_ long) PURE;
};

#undef  INTERFACE
#define INTERFACE  ILineIn
DECLARE_INTERFACE_ (INTERFACE, IUnknown)
{
   STDMETHOD  (QueryInterface) (THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)  (THIS) PURE;
   STDMETHOD_ (ULONG, Release) (THIS) PURE;
   STDMETHOD  (Mute)           (THIS_ long) PURE;
};

#undef  INTERFACE
#define INTERFACE  ISpeakerOut
DECLARE_INTERFACE_ (INTERFACE, IUnknown)
{
   STDMETHOD  (QueryInterface) (THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)  (THIS) PURE;
   STDMETHOD_ (ULONG, Release) (THIS) PURE;
   STDMETHOD  (SetVolume)      (THIS_ long) PURE;
};
```

Notice that each sub-object's VTable begins with its own `QueryInterface`, `AddRef`, and `Release` functions, as all COM objects must. After this, we list any extra functions that are in the object's VTable. So, `IMicIn` has its `SetImpedance` function, `ILineIn` has its `Mute` function, and `ISpeakerOut` has its `SetVolume` function. (For demonstration, I've defined each of these to be passed one argument -- a `long` value. In the case of `SetImpedance`, this may be the impedance value. For `Mute`, this may be a 1 for mute or 0 for un-mute. For `SetVolume`, this may be the volume level.)

> **Note:** I haven't added any `IDispatch` functions to these sub-objects (and have not specified that their VTable is based upon `IDispatch`). Instead, I've omitted the `IDispatch` functions, and specified that the VTable is based upon `IUnknown`. Therefore, none of these sub-objects' functions are directly callable by script languages such as VBScript or JScript. That's OK, because those script languages are not designed to be able to use multiple interface objects anyway.

Also, remember that the above macro also automatically defines the object itself as having one member, `lpVtbl` -- a pointer to its VTable:

```
typedef struct {
   IMicInVtbl  *lpVtbl;
} IMicIn;

typedef struct {
   ILineInVtbl  *lpVtbl;
} ILineIn;

typedef struct {
   ISpeakerOutVtbl  *lpVtbl;
} ISpeakerOut;
```

Embedding sub-objects in our object

There are a few rules concerning the sub-objects. One of the sub-objects is considered to be the "base object", and its VTable pointer must be the very first member of our object itself. For example, let's say that we want our `IMicIn` sub-object to be the base object. We could define our object (which we'll call an `IAudioCard`) as so, embedding the `IMicIn` sub-object first, and embedding the other sub-objects after:

```
typedef struct {
   IMicIn       mic;     // Our IMicIn (base) sub-object.

   ILineIn      line;    // Our ILineIn sub-object.

   ISpeakerOut  speaker; // Our ISpeakerOut sub-object.

} IAudioCard;
```

Remember that the very first member of an `IMicIn` is its `lpVtbl` (i.e., pointer to its VTable). And since our `IMicIn` is embedded directly inside of our `IAudioCard` object at the very start of the struct, this means the very first member inside of our `IAudioCard` object is effectively a pointer to `IMicIn`'s VTable. This is how `IMicIn` becomes the base object. So, `IAudioCard` does indeed begin with a pointer to a VTable (because that's what `IMicIn` begins with), and its VTable does in fact begin with the three functions `QueryInterface`, `AddRef`, and `Release`.

Another rule is that every sub-object's VTable must have its own GUID. So, we would need to run *GUIDGEN.EXE* to create GUIDs for the `IMicIn`, `ILineIn`, and `ISpeakerOut` VTables. And, we would need a GUID for our `IAudioCard` object itself.

How an app obtains the base object

Typically, an app would get hold of the base object (in this case, our `IMicIn`) by calling our `IClassFactory`'s `CreateInstance`, passing the GUID for our `IAudioCard`. (Or to get our base object, maybe the app would call some extra function we've added to another one of our objects, like how we returned our collection object or `IEnumVARIANT` objects in the previous part of this series.) Our `IClassFactory`'s `CreateInstance` would then return a pointer to our base object.

```
// How an app gets IAudioCard's base object. Error-checking omitted!

// Include the .H file that defines the VTables of
// our IAudioCard sub-objects, and has all of the
// needed GUIDs.
#include "IAudioCard.h"

// First, we need to get IAudioCard's base object. Let's assume we
// do this by calling CoCreateInstance. We must pass the GUID for the
// DLL containing the IAudioCard object. Let's assume that this GUID
// has been given the name CLSID_IAudioCard in IAudioCard.h. We must
// also pass the IAudioCard object's GUID which we'll assume is
// given the name IID_IAudioCard. CoCreateInstance returns a pointer
// to IAudioCard's base object, which we store in our audioCard variable.
// Note that we've declared our audioCard variable as an IUknown *. This
// is because we don't really know what kind of object that base object
// is. (Well, it's going to be an IMicIn object. But if this DLL was
// written by someone else, we may not have been told what kind of
// object the base object really is). All we know is that it definitely
// has a QueryInterface function, since all COM objects start with that one.
IUnknown    *audioCard;
CoCreateInstance(&CLSID_IAudioCard, 0, CLSCTX_ALL, &IID_IAudioCard,
     (void **)&audioCard);
// "audioCard" now contains a pointer to the base object (whatever it
// is -- in this case, it will be an IMicIn, but we'd typically
// follow up with a call to QueryInterface, passing IID_IMicIn
// if we specifically want the IMicIn).
```

How an app obtains a sub-object from the base object

The way that an app gets hold of one of our `IAudioCard`'s sub-objects is by calling the base object's (`IMicIn`'s) `QueryInterface` function, passing the VTable GUID of the sub-object it wants. (And now, you see the other purpose of the `QueryInterface` function, besides allowing an app to verify what kind of object it has. `QueryInterface` is also used to obtain sub-objects of a multiple interface object.) What this means is that the base object's `QueryInterface` function should be able to recognize when an app passes the GUID of any of the other sub-objects' VTables, locate that sub-object, and return (to the app) a pointer to that sub-object.

So, if an app wants to get our `ISpeakerOut` object, it must first get our `IAudioCard`'s base object. Then, the app must call that base object's `QueryInterface`, passing the GUID of our `ISpeakerOut`'s VTable. `QueryInterface` will return a pointer to our `ISpeakerOut` object.

```
// How an app gets IAudioCard's ISpeakerOut sub-object.

// Error-checking omitted!


IUnknown    *audioCard;
ISpeakerOut *speakerOut;

// Get IAudioCard's base object.

CoCreateInstance(&CLSID_IAudioCard, 0, CLSCTX_ALL, &IID_IAudioCard,
     (void **)&audioCard);

// Now that we've got the base object (whatever it is) in "audioCard", let's

// call its QueryInterface, asking for its ISpeakerOut sub-object. We call the

// base object's QueryInterface, passing the GUID of ISpeakerOut's VTable,

// which we'll assume has the name IID_ISpeakerOut. QueryInterface will

// store this pointer in "speakerOut".

audioCard->lpVtbl->QueryInterface(audioCard, &IID_ISpeakerOut,
     &speakerOut);

// Now that we've got the ISpeakerOut, we can Release the base object.

// NOTE: If the base object happened to be the ISpeakerOut, then both

// "speakerOut" and "audioCard" are the same pointer (object). But

// CoCreateInstance has done an AddRef on it for us. And QueryInterface

// has also done an AddRef on it for us. So the Release below undoes

// only one of the AddRef's. Our ISpeakerOut isn't going away, because

// we still need to do one more Release on it.

audioCard->lpVtbl->Release(audioCard);
```

How an app obtains an object from another sub-object

Another rule is that a base object can't be deleted until all of the other sub-objects have been `Release`d by the app. If the app is holding onto any pointers to sub-objects, the base object must stick around (even if the app has `Release`'d its hold on the base object). Why? Well, this has to do with the next rule.

As usual, every sub-object's `QueryInterface` function must be able to recognize being passed its own GUID. (I.e., `IMicIn`'s `QueryInterface` must recognize its own VTable GUID, and `ISpeakerOut` must recognize its own VTable GUID, and `ILineIn` must recognize its own VTable GUID.) But, each sub-object's `QueryInterface` must also recognize its base object's GUID, and be able to return a pointer to that base object.

For example, an app can pass our `IAudioCard` VTable's GUID to `ISpeakerOut`'s `QueryInterface` function. `ISpeakerOut`'s `QueryInterface` must recognize this and return a pointer to the base object. What this should tell you is that each sub-object needs to be able to locate the base object. And of course, this means that the base object must exist while any other sub-object exists.

```
// How an app gets IAudioCard's base object from

// its ISpeakerOut sub-object. Error-checking omitted!


// Assume that we've got a pointer to the ISpeakerOut sub-object

// in our variable "speakerOut". To get the base object, we call

// ISpeakerOut's QueryInterface, passing the GUID of IAudioCard's

// VTable. Here, QueryInterface stores that pointer in our

// variable "audioCard".


IUnknown    *audioCard;

speakerOut->lpVtbl->QueryInterface(speakerOut,
                &IID_IAudioCard, &audioCard);

// Note: We must audioCard->lpVtbl->Release(audioCard) when

// we're done with the base object.
```

In fact, every sub-object must be able to recognize the VTable GUIDs of every other sub-object, and locate/return a pointer to that other sub-object.

For example, an app can pass our `ILineIn` VTable's GUID to `ISpeakerOut`'s `QueryInterface` function. `ISpeakerOut`'s `QueryInterface` must recognize this, and return a pointer to the `ILineIn` sub-object.

```
// How an app gets the ILineIn sub-object from the ISpeakerOut sub-object.

// Error-checking omitted!


// Assume that we've got a pointer to the ISpeakerOut sub-object

// in our variable "speakerOut". To get the ILineIn sub-object, we call

// ISpeakerOut's QueryInterface, passing the GUID of ILineIn's

// VTable. Here, QueryInterface stores that pointer in our

// variable "lineIn".


ILineIn    *lineIn;

speakerOut->lpVtbl->QueryInterface(speakerOut, &IID_ILineIn,
    &lineIn);

// Note: We must lineIn->lpVtbl->Release(lineIn) when

// we're done with the ILineIn.
```

Delegation

So, we must ensure that the `QueryInterface` of every sub-object can recognize the GUIDs of every other sub-object's VTable, and our `IAudioCard`'s VTable, and return a pointer to the appropriate sub-object.

How are we going to accomplish this?

Since we've already determined that a base object must recognize all of the different VTable GUIDs (including its own), and return a pointer to the appropriate sub-object, our base object's `QueryInterface` already does all the work of locating/returning any sub-object. So, all that the `QueryInterface` of any other sub-object needs to do is call its base object's `QueryInterface`. In other words, the sub-object's `QueryInterface` "passes the buck" to the base object. After all, if the sub-object can get a pointer to its base object, it can call its base's `QueryInterface`, just like an app would.

And to keep the base object from disappearing before all of the other sub-objects are released, whenever the base returns a sub-object, we will have to increment its reference count. This reference count will be complementarily decremented each time the app `Release`s a sub-object. In this way, the base's reference count will not hit zero until all outstanding pointers to both the base, as well as all other sub-objects, have been `Release`d. But note that if an app calls some sub-object's `AddRef` function, we must increment the base's reference count too. It must be so to keep the base's reference count synchronized with how many times we expect the app to `Release` the sub-objects. So, a sub-object's `AddRef` and `Release` functions will also call the base's `AddRef` and `Release` functions, respectively. Thus, the base's reference count is incremented each time one of the other sub-objects is `AddRef`ed, and decremented each time one of the sub-objects is `Release`d.

When a sub-object calls its base's `QueryInterface`, `AddRef`, and `Release`, we refer to this as *delegation*. The sub-object is delegating (to its base) some of the work the sub-object itself is supposed to be doing.

QueryInterface, AddRef, and Release of our base object

As shown earlier, we can define our `IAudioCard` to actually embed the three sub-objects right inside of it. Because we've chosen the `IMicIn` as the base object, it must be first. We must also add a reference count member to our object.

```
typedef struct {
   IMicIn      mic;     // Our IMicIn (base) sub-object.

   ILineIn     line;    // Our ILineIn sub-object.

   ISpeakerOut speaker; // Our ISpeakerOut sub-object.

   long        count;   // The reference count.

} IAudioCard;
```

As soon as our `IAudioCard` object is allocated (presumably by our `IClassFactory`'s `CreateInstance`), all three sub-objects are simultaneously allocated (because they exist directly inside of the `IAudioCard`). So, we can initialize the sub-objects right then and there, stuffing their VTables into their respective `lpVtbl` members.

For example, our `IClassFactory`'s `CreateInstance` may do something like this:

```
IAudioCard  *thisObj

// Allocate the IAudioCard (and its 3 embedded sub-objects).

thisobj = (IAudioCard *)GlobalAlloc(GMEM_FIXED, sizeof(IAudioCard));

// Store IMicIn's VTable in its lpVtbl member.

thisobj->mic.lpVtbl = &IMicIn_Vtbl;

// Store ILineIn's VTable in its lpVtbl member.

thisobj->line.lpVtbl = &ILineIn_Vtbl;

// Store ISpeakOut's VTable in its lpVtbl member.

thisobj->speaker.lpVtbl = &ISpeakerOut_Vtbl;
```

After this, `CreateInstance` would call the base object's (`IMicIn`'s) `QueryInterface` to check that the app passed our `IID_IAudioCard` GUID, and to fill in the app's pointer with the base (`IMicIn`) object.

When an app asks our base object's `QueryInterface` for a sub-object, we can simply use pointer arithmetic to locate it within our `IAudioCard` object. For example, here's how `IMicIn`'s `QueryInterface` locates the `ISpeakerOut` child object (assuming "`this`" is a pointer to `IMicIn`):

```
// Is the app asking for our ISpeakerOut sub-object? If it

// passed ISpeakerOut's VTable GUID, that's the case.

if (IsEqualIID(vTableGuid, &IID_ISpeakerOut))

   // Just locate the ISpeakerOut object embedded directly

   // inside of IAudioCard

   *ppv = ((unsigned char *)this + offsetof(IAudioCard, speaker));
```

QueryInterface, AddRef, and Release of our sub-objects

Because each sub-object also exists directly inside of our `IAudioCard`, the sub-object can also use pointer arithmetic to locate the base (`IMicIn`) object. For example, `ISpeakerOut`'s `QueryInterface` can call `IMicIn`'s `QueryInterface` like so (assuming that "`this`" is the `ISpeakerOut`):

```
IMicIn  *base;

base = (IMicIn *)((unsigned char *)this -
                  offsetof(IAudioCard, speaker));
base->lpVtbl->QueryInterface(base, tableGuid, ppv);
```

Another way to add a sub-object to our object

Another way we can choose to add sub-objects to our object is to have our base object's `QueryInterface` allocate and initialize the other sub-object when the app first asks for that sub-object. The sub-object will not actually exist until the app asks for it.

Note: The base object cannot be implemented this way. It must be embedded.

To facilitate this, we'll add an extra member to our `IAudioCard` for each sub-object. This member will be a pointer to the sub-object. For example, our `IAudioCard` object may look like this:

```
typedef struct {
   IMicIn      mic;      // Our base (IMicIn) object.

                         // It must be directly embedded.

   ILineIn     *line;    // A pointer to the ILineIn object.

   ISpeakerOut *speaker; // A pointer to the ISpeakerOut object.

   long        count;    // The reference count.

} IAudioCard;
```

When the `IAudioCard` itself is allocated, we would zero out those pointers because the sub-objects are not yet created. In `IMicIn`'s `QueryInterface`, when an app asks for one of the sub-objects, we would `GlobalAlloc` and initialize it then, and then stuff the pointer to it into its respective `IAudioCard` member. Then, we'd return that pointer to the app. The next time the app calls our `QueryInterface` asking for that same object, we'd simply return the same pointer we stored.

But since a sub-object needs to be able to find the base object, we'll need to add an extra pointer member to each sub-object. That extra member will store a pointer to its base. So, our `ISpeakerOut` object may look like this:

```
typedef struct {
   ISpeakerOutVtbl  *lpVtbl;   // Our ISpeakerOut's VTable. Must be first.

   // NOTE: The sub-objects do not need their own reference count.

   // Instead, they increment/decrement the base's count.

   IMicIn           *base;     // A pointer to the base object.

} ISpeakerOut;
```

Immediately after IMicIn's QueryInterface allocates the ISpeakerOut object, IMicIn will stuff a pointer to itself into the base member of the ISpeakerOut sub-object.

For example, IMicIn's QueryInterface may return an ISpeakerOut sub-object as so:

```
// Is the app asking for our ISpeakerOut sub-object? If it

// passed ISpeakerOut's VTable GUID, that's the case.

if (IsEqualIID(vTableGuid, &IID_ISpeakerOut))
{
   IAudioCard   *myObj;

   // Because our IMicIn is the base object, "this" is effectively

   // pointing to our IAudioCard too.

   myObj = (IAudioCard *)this;

   // If we've already allocated the ISpeakerOut, then our

   // IAudioCard->speaker member points to it. We just need

   // to return that pointer.

   if (!myObj->speaker)
   {
      // We didn't allocate the ISpeakerOut yet. Let's do so now,

      // and save the pointer in our IAudioCard->speaker member.

      if (!(myObj->speaker = (ISpeakerOut *)GlobaAlloc(GMEM_FIXED,
           sizeof(ISpeakerOut))))
         return(E_OUTOFMEMORY);

      // Set the base member.

      myObj->speaker->base = this;

      // Set ISpeakerOut's VTable into its lpVtbl member.

      myObj->speaker->lpVtbl = &ISpeakerOut_Vtbl;
   }

   // Return the ISpeakerOut sub-object.

   *ppv = myObj->speaker;
}
```

When the app finally Releases our IAudioCard object and all its sub-objects, we'll need to check those pointer members (in IAudioCard's Release) and GlobalFree any existing sub-objects (right before we GlobalFree our IAudioCard itself).

Now, when our ISpeakerOut's QueryInterface needs to call its base's QueryInterface, all the ISpeakerOut needs to do is (assuming "this" is a pointer to ISpeakerOut):

```
this->base->lpVtbl->QueryInterface(this->base, tableGuid, ppv);
```

The choice is up to you whether you want to embed a sub-object directly inside of your object, or just put a pointer to the sub-object inside the object and then allocate the sub-object separately. Allocating the sub-objects separately means that large sub-objects don't need to exist until/unless the app actually needs them. So, if a particular sub-object has lots of private data members, this won't consume memory needlessly. On the other hand, embedding the sub-objects saves having to add an extra pointer member to every sub-object, and if it's likely that the app is going to ask for those sub-objects anyway, they may as well exist as soon as their parent object is created.

In fact, you could even mix the techniques, having some sub-objects embedded and others allocated separately.

An example object with multiple interfaces

Let's create an object that has multiple interfaces. We'll call this object an `IMultInterface`, and we'll create a DLL named *IMultInterface.DLL* containing our object. Our object will contain a base object that we'll call an `IBase`, and two additional sub-objects called `ISub1` and `ISub2`, respectively. For the sake of illustration, we'll embed `ISub1` directly inside of our `IMultInterface` (and of course, our `IBase` must be embedded too at the start), but we'll allocate `ISub2` separately.

In the directory *IMultInterface*, you'll find the source files.

Rather than put the functions of all three sub-objects into one source file, we'll put them in three separate files, named *IBase.c*, *Sub1.c*, and *Sub2.c*.

IMultInterface.h contains the definitions of the three sub-objects, plus all the GUIDs.

Just for illustration, I've added an extra function to `IBase` called `Sum`. An app passes two `long`s to this function, and it returns a sum of those two values (in a pointer to a `long` supplied by the app).

I've added an extra function to `ISub1` called `ShowMessage`. An app passes a string to this function, and it displays a message box.

I've added three extra functions to `ISub2` called `Increment`, `Decrement`, and `GetValue`. The first function increments a `long` member of `ISub2`. The second function decrements that `long` member. And, the last function retrieves its current value.

I've defined our `IMultInterface` object in a separate include file named *IMultInterface2.h*. Why? Because an app is going to `#include` our *IMultInterface.h* file, and we don't want the app to know anything about the actual, internal structure of our `IMultInterface`. So, we've put that latter information in a separate *.H* file that will be `#include`d only by our *IBase.c*, *ISub1.c*, and *ISub2.c* files.

I've put our `IClassFactory` into *IBase.c*. If you peruse *IBase.c*, you'll notice that this base object is almost exactly like the `IExample` object from our very first chapter. In fact, a lot of the code is copied verbatim from *IExample.c*, with comments removed from the duplicate code. What comments remain in *IBase.c* concern what needed to be done to add `ISub1` and `ISub2` as sub-objects to `IMultInterface`. There really isn't much new here. The biggest change is to the base object's `QueryInterface` and `Release` functions. `IBase`'s `QueryInterface`, `AddRef`, and `Release` functions are renamed to prepend **IBase_** to them, and also removed `static` from them. Since `ISub1` and `ISub2`'s code will be in separate files, and they need to call the base object's `QueryInterface`, `AddRef`, and `Release`, we need to make those latter functions global, and avoid name conflicts.

Also, there is some minor retooling to our `IClassFactory`'s `CreateInstance`.

After you compile this code into the DLL *IMultInterface.dll*, you can register it by modifying the installer we created in our very first chapter (whose source is in the *RegIExample* directory). Simply replace all instances of `IExample` with `IMultInterface`. You can create an uninstaller by doing the same to *UnregIExample*.

An example C app that uses our object

In the directory *IMultInterfaceApp*, is an example C app that uses our `IMultIterface` object (i.e., its three sub-objects). Peruse the comments in the code to see how the app gets the base object, and can get any sub-object from another sub-object.

What's next?

As mentioned earlier, an object with multiple interfaces can't be directly used by script languages such as VBScript or JScript. Why? Because in order to get a sub-object, the script would have to be able to call the `QueryInterface` function. And to call `QueryInterface`, the script must reference the pointer to the VTable (our object's `lpVtbl` member). But VBScript and JScript have no concept of how to access a pointer.

So, does this mean that multiple interface objects are totally useless to VBScript or JScript? Not necessarily. The VBScript and JScript engines themselves could call our object's `QueryInterface` on behalf of the script. Is there any particular instance when the engine may do that? Yes -- when hooking up "event sinks", which will be the topic of the next chapter.

License

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

9 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/com_in__c4.aspx** to post and view comments on this article, or click **here** to get a print view with messages.