

## COM in plain C, Part 5


By [Jeff Glatt](#)

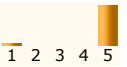
Posted : **22 May 2006**

Updated : **22 May 2006**

Views : **25,612**

Add a connectable object (source/sink).

27 votes for this Article. 

Popularity: [6.62](#) Rating: **4.63** out of 5  1 2 3 4 5

 [Download source files - 246 Kb](#)

### Contents

- [Wrapping a callback function in a COM object](#)
- [IConnectionPointContainer and IConnectionPoint objects](#)
- [An example C app](#)
- [Adding support for script languages](#)
- [Another example C app](#)
- [A VBScript example](#)
- [Multiple types of callback objects](#)
- [Multiple callback objects](#)

### Introduction

Often, it's convenient for a DLL function we call to "callback" into one of our own functions so that we can do some additional tasks at a particular point, or be notified that something has happened. For example, consider the standard C library function `qsort`. The fourth argument passed to `qsort` is a pointer to some function we provide to compare two items. In this way, when we call `qsort`, `qsort` allows us to determine the order that items are sorted, while `qsort` itself does the work of actually reordering the items.

We can't just provide any arbitrary function to `qsort`. The guy who wrote `qsort` specified exactly what arguments would be passed to our "compare" callback function, what our function returns, and of course, specified exactly what the purpose of our callback function is. Furthermore, `qsort` determines when our callback function is actually called (because it is `qsort` itself that calls our function). The guy who designed `qsort` mandated that our compare callback function **must** be defined as so:

```
int (__cdecl *compare)(const void *elem1, const void *elem2);
```

Let's create our own version of `qsort`, which we'll call `Sort`. (Actually, we'll cheat to make this simpler, and just do a bubble sort.) And let's assume that we'll put this in a DLL named `ISort.dll`.

Instead of passing `Sort` a pointer to the compare function each time, let's design this so that the app will first call a separate `SetCompare` function to allow our DLL to store the pointer in some global. Let's also put a function in the DLL called `UnsetCompare` that an app can call to clear that function pointer. So here's our DLL source code (that we'll put in a file called `ISort.c`):

```

// A global to store the pointer to the app's Compare function
int (STDMETHODCALLTYPE *CompareFunc)(const void *, const void *);

void STDMETHODCALLTYPE SetCompare(
    int (STDMETHODCALLTYPE *compare)(const void *, const void *))
{
    // Save the compare function ptr in a global
    CompareFunc = compare;
}

void STDMETHODCALLTYPE UnsetCompare(void)
{
    CompareFunc = 0;
}

HRESULT STDMETHODCALLTYPE Sort(void *base,
    DWORD numElems, DWORD sizeElem)
{
    void *hi;
    void *p;
    void *lo;
    void *tmp;

    // Has the app set its Compare function pointer yet?
    if (!CompareFunc) return(E_FAIL);

    // Do the (bubble) sort
    if ((tmp = GlobalAlloc(GMEM_FIXED, sizeElem)))
    {
        hi = ((char *)base + ((numElems - 1) * sizeElem));
        lo = base;

        while (hi > base)
        {
            lo = base;
            p = ((char *)base + sizeElem);
            while (p <= hi)
            {
                if ((*CompareFunc)(p, lo) > 0) lo = p;
                (char *)p += sizeElem;
            }

            CopyMemory(tmp, lo, sizeElem);
            CopyMemory(lo, hi, sizeElem);
            CopyMemory(hi, tmp, sizeElem);
            (char *)hi -= sizeElem;
        }

        GlobalFree(tmp);
        return(S_OK);
    }

    return(E_OUTOFMEMORY);
}

```

Now, let's write an app that uses our DLL to sort an array of five **DWORDs**. First, the app calls **SetCompare** to specify its compare callback function (which we'll name **Compare**). Then, the app calls **Sort**. Finally, when our app is finished using **Sort**, and we wish to make sure that any additional calls to **Sort** do not accidentally cause it to call our **Compare** function, we call **UnsetCompare**.

```

// An array of 5 DWORDs to be sorted
DWORD Array[5] = {2, 3, 1, 5, 4};

// Our Compare function
int STDMETHODCALLTYPE Compare(const void *elem1, const void *elem2)
{
    // Do a compare of the two elements. We know that
    // we passed an array of DWORD values to Sort()

    if (*(DWORD *)elem1 == *(DWORD *)elem2) return 0;
    if (*(DWORD *)elem1 < *(DWORD *)elem2) return -1;
    return 1;
}

int main(int argc, char **argv)
{
    // Give ISort.dll our Compare function ptr
    SetCompare(Compare);

    // Sort of array of 5 DWORDs
    Sort(&Array[0], 5, sizeof(DWORD));

    UnsetCompare();

    return 0;
}

```

## Wrapping a callback function in a COM object

COM also has a facility to allow callbacks, but as you may have feared, it's not quite as straight-forward as our example above. In the directory named *ISort*, you'll find the files to a COM object that implements our above Sort DLL.

First of all, we have to wrap our *Sort* function inside some COM object. After all, we're going to turn *ISort.dll* into a COM component. Let's define an *ISort* object, using the `DECLARE_INTERFACE_` macro Microsoft provides. Like all COM objects, its VTable begins with the *QueryInterface*, *AddRef*, and *Release* functions. We won't bother adding any *IDispatch* function to it. (Our *ISort* is not useable by script languages such as VBScript). And then, we'll add the *Sort* function as the first extra function. Here's the definition that we'll put in *ISort.h*:

```

#undef INTERFACE
#define INTERFACE ISort
DECLARE_INTERFACE_ (INTERFACE, IUnknown)
{
    STDMETHODCALLTYPE (QueryInterface) (THIS_ REFIID, void **) PURE;
    STDMETHODCALLTYPE_ (ULONG, AddRef) (THIS) PURE;
    STDMETHODCALLTYPE_ (ULONG, Release) (THIS) PURE;
    STDMETHODCALLTYPE (Sort) (THIS_ void *, DWORD, DWORD) PURE;
};

```

So our app is going to get an instance of our *ISort* object and call its *Sort* function. You've now had plenty of experience writing C/C++ apps that get a COM object and call its functions. That's nothing new.

And of course, we need to run *GUIDGEN.EXE* to generate a GUID for our *ISort* object and its VTable. We'll give them names of `CLSID_ISort` and `IID_ISort` respectively, and put them in *ISort.h*.

```
// ISort object's GUID

// {619321BA-4907-4596-874A-AEFF082F0014}

DEFINE_GUID(CLSID_ISort, 0x619321ba, 0x4907, 0x4596,
    0x87, 0x4a, 0xae, 0xff, 0x8, 0x2f, 0x0, 0x14);

// ISort VTable's GUID

// {4C9A7D40-D0ED-45ea-9520-1CB9095973F8}

DEFINE_GUID(IID_ISort, 0x4c9a7d40, 0xd0ed, 0x45ea,
    0x95, 0x20, 0x1c, 0xb9, 0x9, 0x59, 0x73, 0xf8);
```

And like we typically do, we'll need to add at least one, extra, private data member to our `ISort` object -- a reference count member. So, we'll define a `MyRealISort` inside `ISort.c` that adds this extra private member to our `ISort`:

```
typedef struct {
    ISortVtbl *lpVtbl;    // ISort's VTable

    DWORD      count;     // ISort's reference count
} MyRealISort;
```

By now, all of this should be quite familiar to you.

All that's left is to provide some way for an app to give us a pointer to its `Compare` function so our `Sort` function can call it. So, do we simply add `SetCompare` and `UnsetCompare` functions to our `ISort` VTable? Nope.

Here's where it starts to get complicated. Microsoft needed some standard way to allow an app to provide its callback functions to a COM object. Microsoft decided that an app should wrap its callback functions in a COM object. What particular COM object? That depends.

Remember how it was the author of `qsort` who determined what arguments were passed to the callback, and what it returned? Well, since we're the guys writing our `ISort` object, we get to make up our own COM object that the app must use to wrap any callback functions we require. So, let's just define our own COM object that we'll call an `ICompare`. Of course, we must start its VTable with the three functions `QueryInterface`, `AddRef`, and `Release`. So, what do we want next? Let's see... Do we want to support script languages supplying us a compare function (such as a VBScript function)? If so, we'd put some `IDispatch` functions next. No, let's skip that, and for now, only allow a C/C++ app to provide us with a compare callback.

Let's add an extra function to `ICompare`. We'll call it `Compare`. That will be the app's compare callback. (So now, you see how the app has to wrap its callback in our own defined COM object.) Here's the definition of our `ICompare` (which we'll add to `ISort.h`):

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

We do need to run `GUIDGEN.EXE` to create a GUID for our `ICompare`'s VTable. (We won't need a GUID for the object itself). We'll give it the name of `DIID_Compare`. (Traditionally, Microsoft prepends a *D* to the names of GUIDs intended for VTables that wrap callbacks. We'll just follow the tradition.)

```
// ICompare VTable's GUID  
  
// {4115B8E2-1823-4bbc-B10D-3D33AAA12ACF}  
  
DEFINE_GUID(DIID_ICompare, 0x4115b8e2, 0x1823, 0x4bbc,  
    0xb1, 0xd, 0x3d, 0x33, 0xaa, 0xa1, 0x2a, 0xcf);
```

Let's go back to our app. Now, we have to put a COM object inside our app code. Specifically, we have to create an `ICompare` object, and have our `Compare` callback be its extra function. Because we'll need only one `ICompare` object, we'll simply declare it static. We'll also have to `#include ISort.h` because that contains the definition of the `ICompare` object, its VTable, and its GUID. So, here's the code we insert to replace the `Compare` function in our above app code:

```

#include "ISort.h"

// Here's our app's ICompare object. We need only one of these,
// so we'll just declare it static. That way, we don't have to
// allocate it, free it, nor maintain any reference count.
static ICompare    MyCompare;

// This is ICompare's QueryInterface. It returns a pointer
// to our ICompare object.
HRESULT STDMETHODCALLTYPE QueryInterface(ICompare *this,
    REFIID vTableGuid, void **ppv)
{
    // Since this is an ICompare object, we must recognize
    // ICompare VTable's GUID, which is defined for us in
    // ISort.h. Our ICompare can also masquerade as an IUnknown.
    if (!IsEqualIID(vTableGuid, &IID_IUnknown) &&
        !IsEqualIID(vTableGuid, &DIID_ICompare))
    {
        *ppv = 0;
        return(E_NOINTERFACE);
    }

    *ppv = this;

    // Normally, we'd call our AddRef function here. But since
    // our ICompare isn't allocated, we don't need to bother
    // with reference counting.

    return(NOERROR);
}

ULONG STDMETHODCALLTYPE AddRef(ICompare *this)
{
    // Our one and only ICompare isn't allocated. Instead, it's
    // statically declared above (MyCompare). So we'll just
    // return 1.

    return(1);
}

ULONG STDMETHODCALLTYPE Release(ICompare *this)
{
    // Our one and only ICompare isn't allocated, so we
    // never need worry about freeing it.

    return(1);
}

// This is the extra function for ICompare. It is called
// Compare. The ISort object will call this function when
// we call ISort's Sort function.
long STDMETHODCALLTYPE Compare(ICompare *this,
    const void *elem1, const void *elem2)
{
    // Do a compare of the two elements. We know that
    // we passed an array of DWORD values to Sort().

    if (*(DWORD *)elem1 == *(DWORD *)elem2) return 0;
    if (*(DWORD *)elem1 < *(DWORD *)elem2) return -1;
    return 1;
}

```

Notice that our `Compare` function is there. It's simply wrapped inside an `ICompare` object now. (And the first argument passed to it is now a pointer to our `ICompare` object, which will be our statically declared `MyCompare`, in this case.)

Now, we're left with the really convoluted job. How does our app give its `ICompare` object to our `ISort` object (inside `ISort.dll`)? I really wish I could say that this is going to be simple and easy-to-understand, but unfortunately, due to some dubious design decisions by the MS guys who devised this whole scheme, it's going to be a painfully twisted trip through the house of horrors.

## IConnectionPointContainer and IConnectionPoint objects

To let an app give its `ICompare` object to our `ISort` object, we have to add some things to our `ISort` object. Specifically, we have to add two sub-objects to it. (Hopefully, in the last chapter, you have read and understand how to create an object that has sub-objects.)

One good thing is that Microsoft has already defined these two sub-objects (and their VTable GUIDs) in some include files that ship with your compiler. The two sub-objects are called `IConnectionPointContainer` and `IConnectionPoint`. All we need to do is add them to our `ISort` object. Since we've already defined a `MyRealISort`, we'll just embed these two sub-objects inside it. And since we know that an app is going to give us a pointer to an `ICompare` object, let's also add a place to store that inside `MyRealISort`.

```
typedef struct {
    ISortVtbl          *lpVtbl;
    DWORD              count;
    IConnectionPointContainer container;
    IConnectionPoint   point;
    ICompare            *compare;
} MyRealISort;
```

Note that our base object is our `ISort` itself. We could notate it like so:

```
typedef struct {
    ISort          iSort;
    DWORD          count;
    IConnectionPointContainer container;
    IConnectionPoint   point;
    ICompare          *compare;
} MyRealISort;
```

But since an `ISort` has only its one `lpVtbl` member, both definitions amount to the same thing.

As you'll remember from the last chapter, the base object's `QueryInterface` must recognize the GUIDs of all of its sub-object's VTables, as well as its own VTable. So our `ISort QueryInterface` must look like so:

```

HRESULT STDMETHODCALLTYPE QueryInterface(ISort *this,
    REFIID vTableGuid, void **ppv)
{
    // Because an IConnectionPointContainer
    // is a sub-object of our ISort, we
    // must return a pointer to this sub-object
    // if the app asks for it. Because
    // we've embedded our IConnectionPointContainer
    // object inside of our
    // MyRealISort, we can get that sub-object
    // very easily using pointer arithmetic.

    if (IsEqualIID(vTableGuid, &IID_IConnectionPointContainer))
        *ppv = ((unsigned char *)this +
            offsetof(MyRealISort, container));

    else if (IsEqualIID(vTableGuid, &IID_IUnknown) ||
        IsEqualIID(vTableGuid, &IID_ISort))
        *ppv = this;

    else
    {
        *ppv = 0;
        return(E_NOINTERFACE);
    }

    this->lpVtbl->AddRef(this);

    return(NOERROR);
}

```

But wait. Did we forget to also check for our `IConnectionPoint` sub-object's VTable GUID, and return a pointer to that? No, we didn't forget. Unfortunately, the MS guys who thought up this design kind of broke the rules. The `IConnectionPoint` sub-object is not gotten by calling the base object's `QueryInterface` (nor the `QueryInterface` of our `IConnectionPointContainer` sub-object). In a moment, we'll see how it's gotten.

The `IConnectionPointContainer` sub-object can serve two purposes.

First of all, by passing an `IConnectionPointContainer` VTable GUID to our `ISort`'s `QueryInterface` function, an app can determine whether our `ISort` accepts callback functions. Incidentally, in COM documentation, such an object (i.e., one that provides an `IConnectionPointContainer`) is described as *sourcing events*. If `QueryInterface` returns a pointer to an `IConnectionPointContainer`, then the object can source events. (If not, then a 0 is returned.)

Secondly, the `IConnectionPointContainer` has a function to get an `IConnectionPoint` sub-object. That function is called `FindConnectionPoint`.

So, let's write the functions of our `IConnectionPointContainer` sub-object. Because it's a sub-object of `ISort`, and `ISort` is the base object, our `IConnectionPointContainer`'s `QueryInterface`, `AddRef`, and `Release` functions simply delegate to `ISort`'s `QueryInterface`, `AddRef`, and `Release`.



```

STDMETHODIMP QueryInterface_Connect(IConnectionPointContainer *this,
                                    REFIID vTableGuid, void **ppv)
{
    // Because this is a sub-object of our ISort (ie, MyRealISort) object,
    // we delegate to ISort's QueryInterface. And because we embedded the
    // IConnectionPointContainer directly inside of MyRealISort, all we need
    // is a little pointer arithmetic to get our ISort.

    return(QueryInterface((ISort *)((char *)this -
                                   offsetof(MyRealISort, container)), vTableGuid, ppv));
}

STDMETHODIMP_(ULONG) AddRef_Connect(IConnectionPointContainer *this)
{
    // Because we're a sub-object of ISort, delegate to its AddRef()

    // in order to increment ISort's reference count.

    return(AddRef((ISort *)((char *)this - offsetof(MyRealISort, container))));
}

STDMETHODIMP_(ULONG) Release_Connect(IConnectionPointContainer *this)
{
    // Because we're a sub-object of ISort, delegate to its Release()

    // in order to decrement ISort's reference count.

    return(Release((ISort *)((char *)this - offsetof(MyRealISort, container))));
}

```

For now, we're going to use a stub for the `EnumConnectionPoints` function.

```

STDMETHODIMP EnumConnectionPoints(IConnectionPointContainer *this,
                                  IEnumConnectionPoints **enumPoints)
{
    *enumPoints = 0;
    return(E_NOTIMPL);
}

```

The real work is in `FindConnectionPoint`. The app passes us a handle to where it wants us to return `ISort`'s `IConnectionPoint` sub-object. Since we've embedded it directly inside `MyRealISort`, it's easy to locate.

The app also passes us the VTable GUID we defined for our `ICompare` (callback) object. That's how we know that the app really does mean to provide us with the correct object that we require. We don't want to return a `IConnectionPoint` if the app isn't prepared to give us an `ICompare` object.

```

STDMETHODIMP FindConnectionPoint(IConnectionPointContainer *this,
    REFIID vTableGuid, IConnectionPoint **ppv)
{
    // Is the app asking us to return an IConnectionPoint object it can use
    // to give us its ICompare object? The app asks this by passing us
    // ICompare VTable's GUID (which we defined in ISort.h).
    if (IsEqualIID(vTableGuid, &DIID_ICompare))
    {
        MyRealISort *iSort;

        // The app obviously wants to give its ICompare object to our
        // ISort. In order to do that, we need to give the app a standard
        // IConnectionPoint. This is easy to do since we embedded both
        // our IConnectionPointContainer and IConnectionPoint inside of our
        // ISort. All we need is a little pointer arithmetic.

        iSort = (MyRealISort *)((char *)this - offsetof(MyRealISort, container));
        *ppv = &iSort->point;

        // Because we're giving the app a pointer to our IConnectionPoint,
        // and our IConnectionPoint is a sub-object of ISort, we need to
        // increment ISort's reference count. The easiest way to do this is
        // to call our IConnectionPointContainer's AddRef, because all we do
        // there is delegate to our ISort's AddRef.

        AddRef_Connect(this);

        return(S_OK);
    }

    // We don't support any other app callback objects for our ISort.
    // All we've defined, and support, is an ICompare object. Tell
    // the app we don't know anything about the GUID he passed to us, and
    // do not give him any IConnectionPoint object.

    *ppv = 0;
    return(E_NOINTERFACE);
}

```

That takes care of our `IConnectionPointContainer`'s functions. Now we need to write our `IConnectionPoint`'s functions.

Since `IConnectionPoint` is also a sub-object of `ISort` (just like `IConnectionPointContainer`), its `QueryInterface`, `AddRef`, and `Release` will delegate to `ISort`'s `QueryInterface`, `AddRef`, and `Release`. I won't bother reproducing those functions since they are almost identical to `IConnectionPointContainer`'s.

For now, we're going to use a stub for the `EnumConnections` function.

The `GetConnectionInterface` function simply copies `ICompare`'s VTable GUID to a buffer that the app passes. Later, we'll see the use for this.

The `GetConnectionPointContainer` function returns a pointer to the `IConnectionPointContainer` sub-object that created this `IConnectionPoint` object. What this implies is that, as long as the app is holding onto some `IConnectionPoint` sub-object that our `IConnectionPointContainer` gives to the app, our `IConnectionPointContainer` must stick around. This is no problem here since we've embedded both our `IConnectionPointContainer` and `IConnectionPoint` inside our `MyRealISort` (and our `MyRealISort` is going to stick around until all its sub-objects and base object are `Released`).

The above three functions are fairly trivial, so I'll simply refer you to the source code in *Sort.c*.

The real work is in the *Advise* and *Unadvise* functions. This is how an app actually gives us its *ICompare*. *Advise* essentially does the same task as our *SetCompare* function did. And *Unadvise* does what our *UnsetCompare* did.

Let's look at *Advise*:

```
STDMETHODIMP Advise(IConnectionPoint *this, IUnknown *obj, DWORD *cookie)
{
    HRESULT      hr;
    MyRealISort  *iSort;

    // Get our MyRealISort that this IConnectionPoint sub-object belongs
    // to. Because this IConnectPoint sub-object is embedded directly
    // inside its MyRealISort, all we need is a little pointer arithmetic.
    iSort = (MyRealISort *)((char *)this - offsetof(MyRealISort, point));

    // We allow only one ICompare for our ISort, so see if the app already
    // called our Advise(), and we got one. If so, let the app know that it
    // is trying to give us more ICompares than we allow.
    if (iSort->compare) return(CONNECT_E_ADVISELIMIT);

    // Ok, we haven't yet gotten the one ICompare we allow from the app. Get
    // the app's ICompare object. We do this by calling the QueryInterface
    // function of the app object passed to us. We pass ICompare VTable's
    // GUID (which we defined in ISort.h).
    //
    // Save the app's ICompare pointer in our ISort compare member, so we
    // can get it when we need it.
    hr = obj->lpVtbl->QueryInterface(obj, &IID_ICompare, &iSort->compare);

    // We need to return (to the app) some value that will clue our Unadvise()
    // function below how to locate this app ICompare. The simplest thing is
    // to just use the app's ICompare pointer as that return value.
    *cookie = (DWORD)iSort->compare;

    return(hr);
}
```

Actually, the app doesn't pass a pointer to its *ICompare* object. That would be too easy, straight-forward, and understandable, and obviously that wasn't the goal for the MS programmers who designed this thing. Rather, the app passes us some app object from which we can request the app to give us its *ICompare*. Our *Advise* function is expected to call that object's *QueryInterface* function, passing the *ICompare* VTable's GUID to request the app's *ICompare*. I guess that MS thought this would be a good way to double-check that the app really is giving us an *ICompare* (except, don't you know that if an app is so badly written, it can't figure out what it should be doing with *Advise*, then its *QueryInterface* will probably return the wrong object too?). Our course, the app's *QueryInterface* will automatically do an *AddRef* on the *ICompare* it gives to us, so we are expected to *Release* the app's *ICompare* when done with it.

Our *Advise* function stores the app's *ICompare* pointer in our *MyRealISort*'s *compare* member, so we can access it when we need it, and finally *Release* it.

*Advise* must return some *DWORD* value, of our own choosing, to the app. The app is expected to store this

**DWORD** value, and later pass it to our **Unadvise** function. We can choose any **DWORD** value we want, but this value should somehow help our **Unadvise** function when the app later wants us to give back (i.e., **Release**) its **ICompare**. Indeed, the app later calls our **Unadvise** when the app no longer wants us to call its callback functions, and to instead **Release** its **ICompare**. So, let's look at **Unadvise**:

```
STDMETHODIMP Unadvise(IConnectionPoint *this, DWORD cookie)
{
    MyRealISort *iSort;

    // Get the MyRealISort that this IConnectionPoint sub-object belongs
    // to. Because this IConnectPoint sub-object is embedded directly
    // inside its MyRealISort, all we need is a little pointer arithmetic.
    iSort = (MyRealISort *)((char *)this - offsetof(MyRealISort, point));
    // Use the passed value to find wherever we stored his ICompare pointer.
    // Well, since we allow only one ICompare for our ISort, we already
    // know we stored it in our ISort->compare member. And Advise()
    // returned that pointer as the "cookie" value. So we already got the
    // ICompare right now.
    //
    // Let's just make sure the cookie he passed is really the pointer we
    // expect.

    if (cookie && (ICompare *)cookie == iSort->compare)
    {
        // Release the app's ICompare
        ((ICompare *)cookie)->lpVtbl->Release((ICompare *)cookie);

        // We no longer have the app's ICompare, so clear the ISort
        // compare member.
        iSort->compare = 0;

        return(S_OK);
    }
    return(CONNECT_E_NOCONNECTION);
}
```

The rest of the code in *Sort.c* is almost exactly the same as in our very first chapter's *IExample.c*. The only difference is that, after our **IClassFactory**'s **CreateInstance** allocates our **MyRealISort**, we must initialize the sub-objects inside it, and initialize any other private data members.

You can compile our COM object into *ISORT.DLL*. To register it, simply take the register utility for **IExample** (in the directory *RegIExample*), and replace all instances of **IExample** with **ISort**.

## An example C app

In the directory *ISortApp* is an example C app that uses our **ISort** DLL. Previously, we discussed the **ICompare** functions, so all that's really new here is how the app calls our **Advise** function to give us its **ICompare** object, and then later calls **Unadvise** to ask us to release it. This is a fairly trivial example, so you can peruse the comments in *ISortApp.c*.

Because COM event connections can be fairly complicated, I've supplied a second example of an object that uses callbacks. In the directory *IExampleEvts* is the source to another COM DLL. This DLL implements an object (**IExampleEvts**) that has an extra function called **DoSomething**. This function has the ability to call any of up to five different app callbacks. So we have defined an **IFeedback** object to wrap all five of those callback functions. Its definition is in *IExampleEvts.h*. (A callback object can contain numerous extra, callback functions.

And each function can have a different purpose, take different arguments, and return different values.)

Take the time to peruse these examples and fully understand what is going on. Things are going to get more complicated as we go on from here.

## Adding support for script languages

In order to support script languages, we need to add some `IDispatch` functions. Let's take our `IExampleEvts` COM object and adapt it for use with script languages.

First, let's create a new directory named `IExampleEvts2`, and copy the original sources there. Then, we'll rename the files, replacing all instances of `IExampleEvts` with `IExampleEvts2`, and all instances of `IFeedback` with `IFeedback2`. I've done this for you already. I also ran `GUIDGEN.EXE` and made new GUIDs for the objects and their VTables. Since we're going to also need a type library, I created a new GUID for that too.

We need to add `IDispatch` functions to both our `IExampleEvts2` VTable and our `IFeedback2` VTable. You can compare the original `IExampleEvts.h` with the new `IExampleEvts2.h` to see the changes. Note that both VTables now have the five standard `IDispatch` functions added, and also the macros now specify that both VTables are based on an `IDispatch` (instead of `IUnknown`).

In `IExampleEvts2.c`, I've added the five `IDispatch` functions to our `IExampleEvts2` object and its statically declared VTable. I also added a static variable `MyTypeInfo` to hold an `ITypeInfo` that Microsoft's `GetTypeInfoOfGuid` creates for us (and which we use with `DispInvoke`). This is same thing we did when we added support to our `IExample2` object back in chapter 2. There's nothing new to any of these changes so far.

We do have to change our `DoSomething` function. No longer can we call any callback function directly (referencing it via the `lpVtbl` of the object). This is because a script engine will not actually have function pointers in its VTable for those extra callback functions. Instead, we must call it indirectly using the callback object's (`IFeedback`'s) `Invoke` function. We must pass the appropriate DISPID of the function we wish to call.

The only new thing appears in our `IExampleEvts2.IDL` file (i.e., the file that `MIDL.EXE` compiles into our `.TLB` type library file). Notice how we've defined `IFeedback2`'s VTable (which we arbitrarily label as `IFeedback2Vtbl`). This is similar to how we define our `IExampleEvts2` VTable, except that we don't use the `dual` keyword on `IFeedback2`'s VTable. Script engines can't utilize a dual VTable for a callback object. I've arbitrarily chosen to use DISPID 1 for the first callback function (`Callback1`), 2 for the second callback function, etc.

The other new thing is how we define our `IExampleEvts2` object itself as so:

```
coclass IExampleEvts2
{
    [default] interface IExampleEvts2Vtbl;
    [source, default] interface IFeedback2Vtbl;
}
```

The first interface line should look familiar. It's the same as how we've always marked an object's VTable in any IDL file. But the second interface line tells anyone using our type library that `IExampleEvts2` supports a callback object whose VTable is described by `IFeedback2Vtbl`. The "source" keyword indicates this is a callback VTable. The "default" keyword means that this VTable is the one a script engine should use for any callback object associated with our `IExampleEvts2` object.

You can compile our COM object into `IExampleEvts2.DLL`. To register it, simply take the register utility for `IExample2` (in the directory `RegIExample2`) and replace all instances of `IExample2` with `IExampleEvts2`.

## Another example C app

In the directory `IExampleEvts2App` is an example C app that uses our `IExampleEvts2` DLL. Compare this with the original source in `IExampleEvtsApp`. Notice that we've added the five standard `IDispatch` functions to our `IFeedback2` object.

Although `IExampleEvts2.IDL` did not define the `IFeedback2` VTable as "dual", we've nevertheless included the

extra functions (`Callback1` through `Callback5`) in our actual statically declared VTable (`IFeedback2_Vtbl`). In this way, we can fool `DispInvoke` into using this VTable as if it really was declared as dual. Also, notice that the type library we load is `IExampleEvts2`'s type library, because that is where the `IFeedback2` VTable is described. And the `TypeInfo` we ask `GetTypeInfoOfGuid` to create is for an `IFeedback2` VTable. We use this `TypeInfo` with our `IFeedback2`'s `Invoke` function.

## A VBScript example

In the `IExampleEvts2` directory is a VBScript example named `IExampleEvts2.vbs` as so:

```
Set myobj = WScript.CreateObject("IExampleEvts2.object", "label_")
myobj.DoSomething()

sub label_Callback1()
    Wscript.echo "Callback1 is called"
end sub
```

The first line uses the Windows Scripting Shell's `CreateObject` to get an instance of our `IExampleEvts2` object (which should have been registered with the ProdID of `IExampleEvts2.object`). The reason we use the Script Shell's `CreateObject` instead of the VB engine's `CreateObject`, is because the former allows us to pass a second argument -- a string that is prepended to the name of any callback function. Here, we've specified the string `label`. So for example, when our `IExampleEvts2 DoSomething` calls the script engine's `IFeedback2`'s `Invoke` and passes a DISPID of 1 (i.e., for `Callback1`), then the VBScript function that gets called is `label_Callback1`.

In fact, we've provided a VB function by that name, which simply displays a message box.

## Multiple types of callback objects

An object can support numerous types of callback objects. For example, if we wanted, we could have our `ISort` object support being given both an `ICompare` and `IFeedback2` callback objects. In this case, we would need a separate `IConnectionPoint` object for each. In that case, we'd need to put both the `ICompare` and `IFeedback2` VTables in our IDL file, and reference them both in our object like so:

```
coclass IExampleEvts2
{
    [default] interface IExampleEvts2VTbl;
    [source, default] interface IFeedback2VTbl;
    [source] interface ICompareVTbl;
}
```

But note that only one source interface can be marked as default, and a script engine can use only the default one (i.e., a script can't provide us any `ICompare` callback functions).

Furthermore, we have to provide real code for our `IConnectionPointContainer`'s `EnumConnectionPoints` function. This function should return another object that can enumerate each of the different types of `IConnectPoints` that we support. (The object it returns is a standard enumerator object such as the enumerator we used in our Collections chapter.)

Because callback objects are usually used in conjunction with scripting (and script engines can't support multiple types of callback objects per a single coclass object), I'm going to delve further into multiple types of callback objects.

## Multiple callback objects

In our example objects, we allowed an app (or script engine) to provide us only one `IFeedback2` object (which we stored in our `MyRealIExampleEvts2`'s `feedback` member). But theoretically, an app may provide us with as many `IFeedback2` callback objects as it desires. In our `IConnectionPoint`'s `Advise`, we simply refused to accept more than one `IFeedback2` from an app/engine. If you wish to allow an app/engine to provide more `IFeedback2`s, then instead of a `feedback` member, you may wish to define another struct that can be linked into a list, and also has a member to store an `IFeedback2` pointer. You would `GlobalAlloc` one of these

structs in `Advise`, and link them into a list whose head would be stored in some `MyRealIExampleEvs2` member.

You would also need to modify `DoSomething` to loop through the entire list, and call each `IFeedback2`'s `Invoke`.

But again, since the primary use of callback objects is for scripting, and most uses will involve only one callback object, we'll skip developing an example to illustrate.

## What's next?

In the next article, we'll examine how to turn a C app into an ActiveX script host (i.e., how to run scripts from your C app, and how a script calls C functions in your app, and your app calls functions in the script, to control your app and exchange data with a script).

## License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](#)

## About the Author

**Jeff Glatt**

Location:  United States

## Discussions and Feedback



**18 messages** have been posted for this article. Visit

[http://www.codeproject.com/KB/COM/com\\_in\\_c5.aspx](http://www.codeproject.com/KB/COM/com_in_c5.aspx) to post and view comments on this article, or click [here](#) to get a print view with messages.

[PermaLink](#) | [Privacy](#) | [Terms of Use](#)

Last Updated: 22 May 2006

Editor: [Smitha Vijayan](#)

Copyright 2006 by Jeff Glatt  
Everything else Copyright © [CodeProject](#), 1999-2008  
Web19 | [Advertise on the Code Project](#)