COM in plain C

By **Jeff Glatt**, 28 Mar 2006

Download source files - 17.6 Kb

Contents

Introduction

There are numerous examples that demonstrate how to use/create COM/OLE/ActiveX components. But these examples typically use Microsoft Foundation Classes (MFC), .NET, C#, WTL, or at least ATL, because those frameworks have pre-fabricated "wrappers" to give you some boilerplate code. Unfortunately, these frameworks tend to hide all of the low level details from a programmer, so you never really do learn how to use COM components per se. Rather, you learn how to use a particular framework riding on top of COM.

If you're trying to use plain C, without MFC, WTL, .NET, ATL, C#, or even any C++ code at all, then there is a dearth of examples and information on how to deal with COM objects. This is the first in a series of articles that will examine how to utilize COM in plain C, without any frameworks.

With standard Win32 controls such as a Static, Edit, Listbox, Combobox, etc., you obtain a handle to the control (i.e., an `HWND`) and pass messages (via `SendMessage`) to it in order to manipulate it. Also, the control passes messages back to you (i.e., by putting them in your own message queue, and you fetch them with `GetMessage`) when it wants to inform you of something or give you some data.

Not so with an OLE/COM object. You don't pass messages back and forth. Instead, the COM object gives you some pointers to certain functions that you can call to manipulate the object. For example, one of Internet Explorer's objects will give you a pointer to a function you can call to cause the browser to load and display a web page in one of your windows. One of Office's objects will give you a pointer to a function you can call to load a document. And if the COM object needs to notify you of something or pass data to you, then you will be required to write certain functions in your program, and provide (to the COM object) pointers to those functions so the object can call those functions when needed. In other words, you need to create your own COM object(s) inside your program. Most of the real hassle in C will involve defining your own COM object. To do this, you'll need to know the minute details about a COM object -- stuff that most of the pre-fabricated frameworks hide from you, but which we'll examine in this series.

> **In conclusion**, you call functions in the COM object to manipulate it, and it calls functions in your program to notify you of things or pass you data or interact with your program in some way. This scheme is analogous to calling functions in a DLL, but as if the DLL is also able to call functions inside your C program -- sort of like with a "callback". But unlike with a DLL, you don't use `LoadLibrary()` and `GetProcAddress()` to obtain the pointers to the COM object's functions. As we'll soon discover, you instead use a different operating system function to get a pointer to an object, and then use that object to obtain pointers to its functions.

A COM object and its VTable

Before we can learn how to use a COM object, we first need to learn what it is. And the best way to do that is to create our own COM object.

But before we do that, let's examine a C `struct` data type. As a C programmer, you should be quite familiar with `struct`. Here's an example definition of a simple struct (called "`IExample`") that contains two members -- a `DWORD` (accessed via the member name "`count`"), and an 80 `char` array (accessed via the member name "`buffer`").

```
struct IExample {
   DWORD   count;
   char    buffer[80];
};
```

Let's use a `typedef` to make it easier to work with:

```
typedef struct {
   DWORD   count;
   char    buffer[80];
} IExample;
```

And here's an example of allocating an instance of the above struct (error checking omitted), and initializing its members:

```
IExample * example;

example = (IExample *)GlobalAlloc(GMEM_FIXED, sizeof(IExample));
example->count = 1;
example->buffer[0] = 0;
```

Did you know that a struct can store a pointer to some function? Hopefully, you did, but here's an example. Let's say we have a function which is passed a `char` pointer, and returns a `long`. Here's our function:

```
long SetString(char * str)
```

```
{
   return(0);
}
```

Now we want to store a pointer to this function inside `IExample`. Here's how we define `IExample`, adding a member ("`SetString`") to store a pointer to the above function (and I'll use a `typedef` to make this more readable):

```
typedef long SetStringPtr(char *);

typedef struct {
   SetStringPtr * SetString;
   DWORD          count;
   char           buffer[80];
} IExample;
```

And here's how we store a pointer to `SetString` inside our allocated `IExample`, and then call `SetString` using that pointer:

```
example->SetString = SetString;

long value = example->SetString("Some text");
```

OK, maybe we want to store pointers to two functions. Here's a second function:

```
long GetString(char *buffer, long length)
{
   return(0);
}
```

Let's re-define `IExample`, adding another member ("`GetString`") to store a pointer to this second function:

```
typedef long GetStringPtr(char *, long);

typedef struct {
   SetStringPtr * SetString;
   GetStringPtr * GetString;
   DWORD          count;
   char           buffer[80];
} IExample;
```

And here we initialize this member:

```
example->GetString = GetString;
```

But let's say we don't want to store the function pointers directly inside of `IExample`. Instead, we'd rather have an array of function pointers. For example, let's define a second struct whose sole purpose is to store our two function pointers. We'll call this a `IExampleVtbl` struct, and define it as so:

```
typedef struct {
   SetStringPtr * SetString;
   GetStringPtr * GetString;
} IExampleVtbl;
```

Now, we'll store a pointer to the above array inside of `IExample`. We'll add a new member called "`lpVtbl`" for that purpose (and of course, we'll remove the `SetString` and `GetString` members since they've been moved to the `IExampleVtbl` struct):

```
typedef struct {
   IExampleVtbl * lpVtbl;
   DWORD          count;
   char           buffer[80];
} IExample;
```

So here's an example of allocating and initializing a `IExample` (and of course, a `IExampleVtbl`):

```
// Since the contents of IExample_Vtbl will never change, we'll
// just declare it static and initialize it that way. It can
// be reused for lots of instances of IExample.
static const IExampleVtbl IExample_Vtbl = {SetString, GetString};

IExample * example;

// Create (allocate) a IExample struct.
example = (IExample *)GlobalAlloc(GMEM_FIXED, sizeof(IExample));

// Initialize the IExample (ie, store a pointer to
// IExample_Vtbl in it).
example->lpVtbl = &IExample_Vtbl;
example->count = 1;
example->buffer[0] = 0;
```

And to call our functions, we do:

```
char buffer[80];

example->lpVtbl->SetString("Some text");
example->lpVtbl->GetString(buffer, sizeof(buffer));
```

One more thing. Let's say we've decided that our functions may need to access the "`count`" and "`buffer`" members of the struct used to call them. So, what we'll do is always pass a pointer to that struct as the first argument. Let's rewrite our functions to accommodate this:

```
typedef long SetStringPtr(IExample *, char *);
typedef long GetStringPtr(IExample *, char *, long);

long SetString(IExample *this, char * str)
{
   DWORD  i;

   // Let's copy the passed str to IExample's buffer
   i = lstrlen(str);
   if (i > 79) i = 79;
   CopyMemory(this->buffer, str, i);
   this->buffer[i] = 0;

   return(0);
}

long GetString(IExample *this, char *buffer, long length)
{
   DWORD  i;

   // Let's copy IExample's buffer to the passed buffer
   i = lstrlen(this->buffer);
   --length;
   if (i > length) i = length;
   CopyMemory(buffer, this->buffer, i);
   buffer[i] = 0;

   return(0);
}
```

And let's pass a pointer to the `IExample` struct when calling its functions:

```
example->lpVtbl->SetString(example, "Some text");
example->lpVtbl->GetString(example, buffer, sizeof(buffer));
```

If you've ever used C++, you may be thinking *"Wait a minute. This seems strangely familiar."* It should. What we've done above is to recreate a C++ class, using plain C. The `IExample` struct is really a C++ class (one that doesn't inherit from any other class). A C++ class is really nothing more than a struct whose first member is always a pointer to an array -- an array that contains pointers to all the functions inside of that class. And the first argument passed to each function is always a pointer to the class (i.e., struct) itself. (This is referred to as the hidden "`this`" pointer.)

At its simplest, a COM object is really just a C++ class. You're thinking *"Wow! `IExample` is now a COM object? That's all there is to it?? That was easy!"* Hold on. `IExample` is getting closer, but there's much more to it. It's not that easy. If it were, this wouldn't be a "Microsoft technology", now would it?

First of all, let's introduce some COM technobabble. You see that array of pointers above -- the `IExampleVtbl` struct? COM documentation refers to that as an **interface** or **VTable**.

One requirement of a COM object is that the first three members of our VTable (i.e., our `IExampleVtbl` struct) must be called `QueryInterface`, `AddRef`, and `Release`. And of course, we have to write those three functions. Microsoft has already determined what arguments must be passed to these functions, what they must return, and what calling convention they use. We'll need to `#include` some Microsoft include files (that either ship with your C compiler, or you download the Microsoft SDK). We'll re-define our `IExampleVtbl` struct as so:

```
#include <windows.h>
#include <objbase.h>
#include <INITGUID.H>

typedef HRESULT STDMETHODCALLTYPE QueryInterfacePtr(IExample *, REFIID, void **);
typedef ULONG STDMETHODCALLTYPE AddRefPtr(IExample *);
typedef ULONG STDMETHODCALLTYPE ReleasePtr(IExample *);

typedef struct {
   // First 3 members must be called QueryInterface, AddRef, and Release
   QueryInterfacePtr  *QueryInterface;
   AddRefPtr          *AddRef;
   ReleasePtr         *Release;
   SetStringPtr       *SetString;
   GetStringPtr       *GetString;
} IExampleVtbl;
```

Let's examine that `typedef` for `QueryInterface`. First of all, the function returns an `HRESULT`. This is defined simply as a `long`. Next, it uses `STDMETHODCALLTYPE`. This means that arguments are not passed in registers, but rather, on the stack. And this also determines who does cleanup of the stack. In fact, for a COM object, we should make sure that all of our functions are declared with `STDMETHODCALLTYPE`, and return a `long` (`HRESULT`). The first argument passed to `QueryInterface` is a pointer to the object used to call the function. Aren't we turning `IExample` into a COM object? Yes, and that's what we're going to pass for this argument. (Remember we decided that the first argument we pass to any of our functions will be a pointer to the struct used to call that function? COM is simply enforcing, and relying upon, this design.)

Later, we'll examine what a `REFIID` is, and also talk about what that third argument to `QueryInterface` is for. But for now, note that `AddRef` and `Release` also are passed that same pointer to our struct we use to call them.

OK, before we forget, let's add `HRESULT STDMETHODCALLTYPE` to `SetString` and `GetString`:

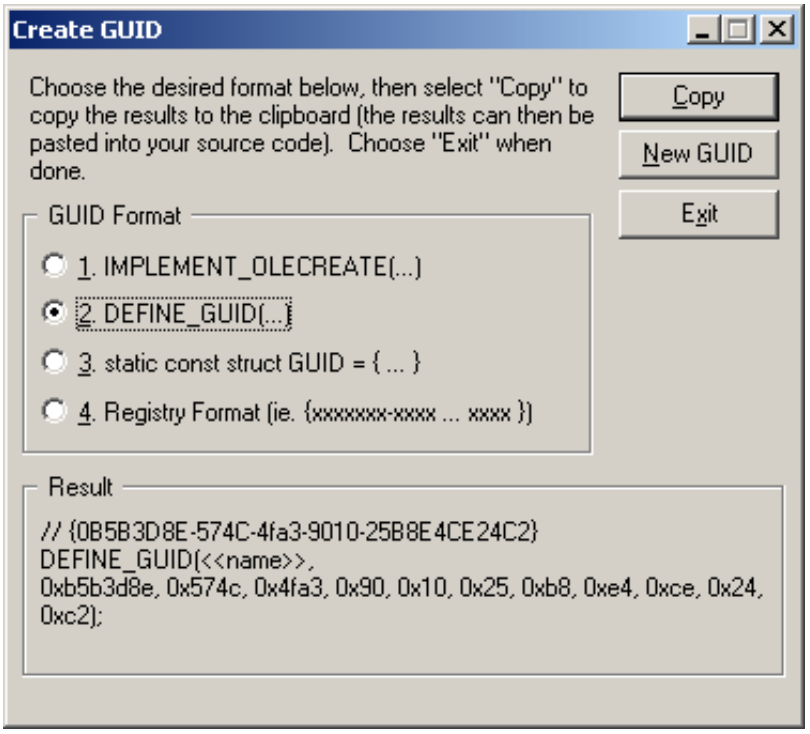
```
typedef HRESULT STDMETHODCALLTYPE SetStringPtr(IExample *, char *);
typedef HRESULT STDMETHODCALLTYPE GetStringPtr(IExample *, char *, long);

HRESULT STDMETHODCALLTYPE SetString(IExample *this, char * str)
{
   ...

   return(0);
}

HRESULT STDMETHODCALLTYPE GetString(IExample *this, char *buffer, long value)
{
   ...

   return(0);
}
```

> **In conclusion**, a COM object is basically a C++ class. A C++ class is just a struct that always starts with a pointer to its VTable (an array of function pointers). And the first three pointers in the VTable will always be named `QueryInterface`, `AddRef`, and `Release`. What additional functions may be in its VTable, and what the name of their pointers are, depends upon what type of object it is. (You determine what other functions you want to add to your COM object.) For example, Internet Explorer's browser object will undoubtedly have different functions than some object that plays music. But all COM objects begin with a pointer to their VTable, and the first three VTable pointers are to the object's `QueryInterface`, `AddRef`, and `Release` functions. The first argument passed to an object's function is a pointer to the object (struct) itself. That is the law. Obey it.

A GUID

Let's continue on our journey to make `IExample` a real COM object. We have yet to actually write our `QueryInterface`, `AddRef`, and `Release` functions. But before we can do that, we must talk about something called a Globally Universal Identifier (GUID). Ack. What's that? It's a 16 byte array that is filled in with a unique series of bytes. And when I say unique, I do mean unique. One GUID (i.e., 16 byte array) **cannot** have the same series of bytes as another GUID... anywhere in the world. Every GUID ever created has a unique series of 16 bytes.

And how do you create that series of 16 unique bytes? You use a Microsoft utility called *GUIDGEN.EXE*. It either ships with your compiler, or you get it with the SDK. Run it and you see this window:



As soon as you run *GUIDGEN*, it automatically generates a new GUID for you, and displays it in the Result box. Note that what you see in your Result box will be different than the above. After all, every single GUID generated will be different than any other. So you had better be seeing something different than I see. Go ahead and click on the "New GUID" button to see some different numbers appear in the Result box. Click all day and entertain yourself by seeing if you ever generate the same series of numbers more than once. You won't. And what's more, nobody else will ever generate **any** of those number series you generate.

You can click on the "Copy" button to transfer the text to the clipboard, and paste it somewhere else (like in your source code). Here is what I pasted when I did that:

```
// {0B5B3D8E-574C-4fa3-9010-25B8E4CE24C2}
DEFINE_GUID(<<name>>, 0xb5b3d8e, 0x574c, 0x4fa3,
            0x90, 0x10, 0x25, 0xb8, 0xe4, 0xce, 0x24, 0xc2);
```

The above is a macro. A `#define` in one of the Microsoft include files allows your compiler to compile the above into a 16 byte array.

But there is one thing that we must do. We must replace `<<name>>` with some C variable name we want to use for this GUID. Let's call it `CLSID_IExample`.

```
// {0B5B3D8E-574C-4fa3-9010-25B8E4CE24C2}
DEFINE_GUID(CLSID_IExample, 0xb5b3d8e, 0x574c, 0x4fa3,
      0x90, 0x10, 0x25, 0xb8, 0xe4, 0xce, 0x24, 0xc2);
```

Now we have a GUID we can use with `IExample`.

We also need a GUID for `IExample`'s VTable ("interface"), i.e., our `IExampleVtbl` struct. So go ahead and click on *GUIDGEN.EXE*'s New GUID button, and copy/paste it somewhere. This time, we're going to replace `<<name>>` with the C variable name `IID_IExample`. Here's what I pasted/edited:

```
// {74666CAC-C2B1-4fa8-A049-97F3214802F0}
DEFINE_GUID(IID_IExample, 0x74666cac, 0xc2b1, 0x4fa8,
      0xa0, 0x49, 0x97, 0xf3, 0x21, 0x48, 0x2, 0xf0);
```

> **In conclusion**, every COM object has its own GUID, which is an array of 16 bytes that are different from any other GUID. A GUID is created with the *GUIDGEN.EXE* utility. A COM

object's VTable (i.e., interface) also has a GUID.

QueryInterface(), AddRef(), and Release()

Assume we want to allow another program to get hold of some `IExample` struct (i.e., COM object) we create/initialize, so the program can call our functions. (We won't yet examine the details of how another program gets hold of our `IExample`. We'll discuss that later).

Besides our own COM object, there may be lots of other COM components installed upon a given computer. (And again, we'll defer discussing how to install our COM component.) And different computers may have different COM components installed. How does that program determine if our `IExample` COM object is installed, and distinguish it from all of the other COM objects?

Remember that each COM object has a totally unique GUID, as does our `IExample` object. And our VTable for `IExample` has a GUID too. What we need to do is tell the developer writing that program what the GUIDs for our `IExample` object and its VTable are. Typically, you do that by giving him an include (*.H*) file with the above two GUID macros you got from *GUIDGEN.EXE*. OK, so the other program knows `IExample`'s and its VTable's GUIDs. What does it do with them?

That's where our `QueryInterface` function comes in. Remember that every COM object must have a `QueryInterface` function (as well as `AddRef` and `Release`). The other program is going to pass our `IExample` VTable GUID to our `QueryInterface` function, and we're going to check it to make sure it is indeed the `IExample` VTable's GUID. If it is, then we'll return something to let the program know that it indeed has an `IExample` object. If the wrong GUID is passed, we're going to return some error that and let it know that what it has isn't an `IExample` object. So, all of the COM objects on the computer will return an error if their `QueryInterface` is passed the `IExample` VTable's GUID, except our own `QueryInterface`.

That second argument passed to `QueryInterface` is the GUID we need to check. The third argument is (a handle) where we will return the same object pointer passed to us, if the GUID matches the `IExample` VTable's GUID. If not, we'll zero out that handle. In addition, `QueryInterface` returns the `long` value `NOERROR` (i.e., `#define`'d as 0) if the GUID matches, or some non-zero error value (`E_NOINTERFACE`) if not. So, let's look at `IExample`'s `QueryInterface`:

```
HRESULT STDMETHODCALLTYPE QueryInterface(IExample *this,
                         REFIID vTableGuid, void **ppv)
{
   // Check if the GUID matches IExample
   // VTable's GUID. Remember that we gave the
   // C variable name IID_IExample to our
   // VTable GUID. We can use an OLE function called
   // IsEqualIID to do the comparison for us.
   if (!IsEqualIID(riid, &IID_IExample))
   {
      // We don't recognize the GUID passed
      // to us. Let the caller know this,
      // by clearing his handle,
      // and returning E_NOINTERFACE.
      *ppv = 0;
      return(E_NOINTERFACE);
   }

   // It's a match!

   // First, we fill in his handle with
   // the same object pointer he passed us. That's
   // our IExample we created/initialized,
   // and he obtained from us.
   *ppv = this;
```

```
   // Now we call our own AddRef function,
   // passing the IExample.
   this->lpVtbl->AddRef(this);

   // Let him know he indeed has a IExample.
   return(NOERROR);
}
```

Now let's talk about our `AddRef` and `Release` functions. You'll notice we called `AddRef` in `QueryInterface`... if we really did have a `IExample`.

Remember that we're allocating the `IExample` on behalf of the other program. He's simply gaining access to it. And it's our responsibility to free it when the other program is done using it. How do we know when that is?

We're going to use something called "reference counting". If you look back at the definition of `IExample`, you'll see that I put a `DWORD` member in there (`count`). We're going to make use of this member. When we create a `IExample`, we'll initialize it to 0. Then, we're going to increment this member (by 1) every time `AddRef` is called, and decrement it by 1 every time `Release` is called.

So, when our `IExample` is passed to `QueryInterface`, we call `AddRef` to increment its `count` member. When the other program is done using it, the program will pass our `IExample` to our `Release` function, where we will decrement that member. And if it's 0, we'll free `IExample` then.

This is another important rule of COM. If you get hold of a COM object created by someone else, you must call its `Release` function when you're done with it. We certainly expect the other program to call our `Release` function when it is done with our `IExample` object.

Here then are our `AddRef` and `Release` functions:

```
ULONG STDMETHODCALLTYPE AddRef(IExample *this)
{
   // Increment the reference count (count member).
   ++this->count;

   // We're supposed to return the updated count.
   return(this->count);
}

ULONG STDMETHODCALLTYPE Release(IExample *this)
{
   // Decrement the reference count.
   --this->count;

   // If it's now zero, we can free IExample.
   if (this->count == 0)
   {
      GlobalFree(this);
      return(0);
   }

   // We're supposed to return the updated count.
   return(this->count);
}
```

There's one more thing we're going to do. Microsoft has defined a COM object known as an `IUnknown`. What's that? An `IUnknown` object is just like `IExample`, except its VTable contains **only** the `QueryInterface`, `AddRef`, and `Release` functions (i.e., it doesn't contain additional functions like our `IExample` VTable has `SetString` and `GetString`). In other words, an `IUnknown` is the bare minimum COM object. And Microsoft created a special GUID for an `IUnknown` object. But you

know what? Our `IExample` object can also masquerade as an `IUnknown` object. After all, it has the `QueryInterface`, `AddRef`, and `Release` functions in it. Nobody needs to know it's really an `IExample` object if all they care about are just those first three functions. We're going to change one line of code so that we report success if the other program passes us either our `IExample` GUID or an `IUnknown` GUID. And by the way, Microsoft's include files give the `IUnknown` GUID the C variable name `IID_IUnknown`:

```
// Check if the GUID matches IExample's GUID or IUnknown's GUID.
if (!IsEqualIID(vTableGuid, &IID_IExample) &&
        !IsEqualIID(vTableGuid, &IID_IUnknown))
```

> **In conclusion**, for our own COM object, we allocate it on behalf of some other program (which gains access to the object and uses it to call our functions). We're responsible for freeing the object. We use reference counting in conjunction with our `AddRef` and `Release` functions to accomplish this safely. Our `QueryInterface` allows other programs to verify they have the object they want, and also allows us to increment the reference count. (Actually, the `QueryInterface` primarily serves a different purpose that we'll examine later. But at this point, it will suffice to think of its purpose this way.)

So, is `IExample` now a real COM object? Yes it is! Great! Not too hard! We're done!

Wrong! We still have to package this thing into a form that another program can use (i.e., a Dynamic Link Library), and write code to do a special install routine, and examine how the other program gets hold of our `IExample` we create (and that will involve us writing more code).

An IClassFactory object

Now we need to look at how a program gets hold of one of our `IExample` objects, and ultimately, we have to write more code to realize this. Microsoft has devised a standardized method for this. It involves us putting a second COM object (and its functions) inside our DLL. This COM object is called an `IClassFactory`, and it has a specific set of functions already defined in Microsoft's include files. It also has its own GUID already defined, and given the C variable name of `IID_IClassFactory`.

Our `IClassFactory`'s VTable has five specific functions in it, which are `QueryInterface`, `AddRef`, `Release`, `CreateInstance`, and `LockServer`. Notice that the `IClassFactory` has its own `QueryInterface`, `AddRef`, and `Release` functions, just like our `IExample` object. After all, our `IClassFactory` is a COM object too, and the VTable of all COM objects must start with those three functions. (But to avoid a name conflict with `IExample`'s functions, we'll preface our `IClassFactory`'s function names with "class", such as `classQueryInterface`, `classAddRef`, and `classRelease`. As long as `IClassFactory`'s VTable defines its first three members as `QueryInterface`, `AddRef`, and `Release`, that's OK.)

The really important function is `CreateInstance`. The program calls our `IClassFactory`'s `CreateInstance` whenever the program wants us to create one of our `IExample` objects, initialize it, and return it. In fact, if the program wants several of our `IExample` objects, it can call `CreateInstance` numerous times. OK, so that's how a program gets hold of one of our `IExample` objects. *"But how does the program get hold of our* `IClassFactory` *object?"*, you may ask. We'll get to that later. For now, let's simply write our `IClassFactory`'s five functions, and make its VTable.

Making the VTable is easy. Unlike our `IExample` object's `IExampleVtbl`, we don't have to define our `IClassFactory`'s VTable struct. Microsoft has already done that for us by defining a `IClassFactoryVtbl` struct in some include file. All we need to do is declare our VTable and fill it in with pointers to our five `IClassFactory` functions. Let's create a static VTable using the variable name `IClassFactory_Vtbl`, and fill it in:

```
static const IClassFactoryVtbl IClassFactory_Vtbl = {classQueryInterface,
classAddRef,
classRelease,
classCreateInstance,
classLockServer};
```

Likewise, creating an actual `IClassFactory` object is easy because Microsoft has already defined that struct too. We need only one of them, so let's declare a static `IClassFactory` using the variable name `MyIClassFactoryObj`, and initialize its `lpVtbl` member to point to our above VTable:

```
static IClassFactory MyIClassFactoryObj = {&IClassFactory_Vtbl};
```

Now, we just need to write those above five functions. Our `classAddRef` and `classRelease` functions are trivial. Because we never actually allocate our `IClassFactory` (i.e., we simple declare it as a static), we don't need to free anything. So, `classAddRef` will simply return a 1 (to indicate that there is always one `IClassFactory` hanging around). And `classRelease` will do likewise. We don't need to do any reference counting for our `IClassFactory` since we don't have to free it.

```
ULONG STDMETHODCALLTYPE classAddRef(IClassFactory *this)
{
   return(1);
}

ULONG STDMETHODCALLTYPE classRelease(IClassFactory *this)
{
   return(1);
}
```

Now, let's look at our `QueryInterface`. It needs to check if the GUID passed to it is either an `IUnknown`'s GUID (since our `IClassFactory` has the `QueryInterface`, `AddRef`, and `Release` functions, it too can masquerade as an `IUnknown` object) or an `IClassFactory`'s GUID. Otherwise, we do the same thing as we did in `IExample`'s `QueryInterface`.

```
HRESULT STDMETHODCALLTYPE classQueryInterface(IClassFactory *this,
                          REFIID factoryGuid, void **ppv)
{
   // Check if the GUID matches an IClassFactory or IUnknown GUID.
   if (!IsEqualIID(factoryGuid, &IID_IUnknown) &&
       !IsEqualIID(factoryGuid, &IID_IClassFactory))
   {
      // It doesn't. Clear his handle, and return E_NOINTERFACE.
      *ppv = 0;
      return(E_NOINTERFACE);
   }

   // It's a match!

   // First, we fill in his handle with the same object pointer he passed us.
   // That's our IClassFactory (MyIClassFactoryObj) he obtained from us.
   *ppv = this;

   // Call our IClassFactory's AddRef, passing the IClassFactory.
   this->lpVtbl->AddRef(this);

   // Let him know he indeed has an IClassFactory.
   return(NOERROR);
}
```

Our `IClassFactory`'s `LockServer` can be just a stub for now:

```
HRESULT STDMETHODCALLTYPE classLockServer(IClassFactory *this, BOOL flock)
{
   return(NOERROR);
}
```

There's one more function to write -- `CreateInstance`. This is defined as follows:

```
HRESULT STDMETHODCALLTYPE classCreateInstance(IClassFactory *,
                          IUnknown *, REFIID, void **);
```

As usual, the first argument is going to be a pointer to our `IClassFactory` object (`MyIClassFactoryObj`) which was used to call `CreateInstance`.

We use the second argument only if we implement something called *aggregation*. We won't get into this now. If this is non-zero, then someone wants us to support aggregation, which we're not going to do, and we will indicate that by returning an error.

The third argument will be the `IExample` VTable's GUID (if someone indeed wants us to allocate, initialize, and return a `IExample` object).

The fourth argument is a handle where we'll return the `IExample` object we create.

So let's dive into our `CreateInstance` function (named `classCreateInstance`):

```
HRESULT STDMETHODCALLTYPE classCreateInstance(IClassFactory *this,
        IUnknown *punkOuter, REFIID vTableGuid, void **ppv)
{
   HRESULT          hr;
   struct IExample *thisobj;

   // Assume an error by clearing caller's handle.
   *ppv = 0;

   // We don't support aggregation in IExample.
   if (punkOuter)
      hr = CLASS_E_NOAGGREGATION;
   else
   {
      // Create our IExample object, and initialize it.
      if (!(thisobj = GlobalAlloc(GMEM_FIXED,
                     sizeof(struct IExample))))
         hr = E_OUTOFMEMORY;
      else
      {
         // Store IExample's VTable. We declared it
         // as a static variable IExample_Vtbl.
         thisobj->lpVtbl = &IExample_Vtbl;

         // Increment reference count so we
         // can call Release() below and it will
         // deallocate only if there
         // is an error with QueryInterface().
         thisobj->count = 1;

         // Fill in the caller's handle
         // with a pointer to the IExample we just
         // allocated above. We'll let IExample's
         // QueryInterface do that, because
         // it also checks the GUID the caller
         // passed, and also increments the
         // reference count (to 2) if all goes well.
         hr = IExample_Vtbl.QueryInterface(thisobj, vTableGuid, ppv);

         // Decrement reference count.
```

```
         // NOTE: If there was an error in QueryInterface()
         // then Release() will be decrementing
         // the count back to 0 and will free the
         // IExample for us. One error that may
         // occur is that the caller is asking for
         // some sort of object that we don't
         // support (ie, it's a GUID we don't recognize).
         IExample_Vtbl.Release(thisobj);
      }
   }

   return(hr);
}
```

That takes care of implementing our `IClassFactory` object.

Packaging into a DLL

In order to facilitate another program getting hold of our `IClassFactory` (and to call its `CreateInstance` function to obtain some `IExample` objects), we'll package our above source code into a Dynamic Link Library (DLL). This tutorial does not discuss how to create a DLL per se, so if you're unfamiliar with that, then you should first read a tutorial about DLLs.

Above, we've already written all the code for our `IExample` and `IClassFactory` objects. All we need to do is paste this into our source for the DLL.

But there's still more to do. Microsoft also dictates that we must add a function to our DLL called `DllGetClassObject`. Microsoft has already defined what arguments it is passed, what it should do, and what it should return. A program is going to call our `DllGetClassObject` to obtain a pointer to our `IClassFactory` object. (Actually, as we'll see later, the program is going to call an OLE function named `CoGetClassObject`, which in turn calls our `DllGetClassObject`.) So, this is how the program gets hold of our `IClassFactory` object -- by calling our `DllGetClassObject`. Our `DllGetClassObject` function must perform this job. Here's how it's defined:

```
HRESULT PASCAL DllGetClassObject(REFCLSID objGuid,
        REFIID factoryGuid, void **factoryHandle);
```

The first argument passed is going to be the GUID for our `IExample` object (not its VTable's GUID). We need to check this to make sure that the caller definitely intended to call our DLL's `DllGetClassObject`. Note that every COM DLL has a `DllGetClassObject` function in it, so again, we need that GUID to distinguish our `DllGetClassObject` from every other COM DLL's `DllGetClassObject`.

The second argument is going to be the GUID of an `IClassFactory`.

The third argument is a handle to where the program expects us to return a pointer to our `IClassFactory` (if the program did indeed pass `IExample`'s GUID, and not some other COM object's GUID).

```
HRESULT PASCAL DllGetClassObject(REFCLSID objGuid,
        REFIID factoryGuid, void **factoryHandle)
{
   HRESULT  hr;

   // Check that the caller is passing
   // our IExample GUID. That's the COM
   // object our DLL implements.
   if (IsEqualCLSID(objGuid, &CLSID_IExample))
```

```
   {
      // Fill in the caller's handle
      // with a pointer to our IClassFactory object.
      // We'll let our IClassFactory's
      // QueryInterface do that, because it also
      // checks the IClassFactory GUID and does other book-keeping.
      hr = classQueryInterface(&MyIClassFactoryObj,
                           factoryGuid, factoryHandle);
   }
   else
   {
      // We don't understand this GUID.
      // It's obviously not for our DLL.
      // Let the caller know this by
      // clearing his handle and returning
      // CLASS_E_CLASSNOTAVAILABLE.
      *factoryHandle = 0;
      hr = CLASS_E_CLASSNOTAVAILABLE;
   }

   return(hr);
}
```

We're almost done with what we need to create our DLL. There's just one more thing. It's not really the program that loads our DLL. Rather, the operating system does so on behalf of the program when the program calls `CoGetDllClassObject` (i.e., `CoGetClassObject` locates our DLL file, does a `LoadLibrary` on it, uses `GetProcAddress` to get our above `DllGetClassObject`, and calls it on behalf of the program). And unfortunately, Microsoft didn't work out any way for the program to tell the OS when the program is done using our DLL and the OS should unload (`FreeLibrary`) our DLL. So we have to help out the OS to let it know when it is safe to unload our DLL. We must provide a function called `DllCanUnloadNow` which will return `S_OK` if it's safe to unload our DLL, or `S_FALSE` if not.

And how will we know when it is safe?

We're going to have to do more reference counting. Specifically, every time we allocate an object for a program, we're going to have to increment a count. Each time the program calls that object's `Release` function, and we free that object, we'll decrement that same count. Only when the count is zero will we tell the OS that our DLL is safe to unload, because that's when we know for sure that the program isn't using any of our objects. So, we'll declare a static `DWORD` variable named `OutstandingObjects` to maintain this count. (And of course, when our DLL is first loaded, this needs to be initialized to 0.)

So, where is the most convenient place to increment this variable? In our `IClassFactory`'s `CreateInstance` function, after we actually `GlobalAlloc` the object and make sure everything went OK. So, we'll add a line in that function, right after the call to `Release`:

```
static DWORD OutstandingObjects = 0;

HRESULT STDMETHODCALLTYPE classCreateInstance(IClassFactory *this,
        IUnknown *punkOuter, REFIID vTableGuid, void **ppv)
{
   ...

         IExampleVtbl.Release(thisobj);

         // Increment our count of outstanding objects if all
         // went well.
         if (!hr) InterlockedIncrement(&OutstandingObjects);;
      }
   }

   return(hr);
}
```

And where is the most convenient place to decrement this variable? In our `IExample`'s `Release` function, right after we `GlobalFree` the object. So we add a line after `GlobalFree`:

```
InterlockedDecrement(&OutstandingObjects);
```

But there's more. (Do the messy details **never** end with Microsoft?) Microsoft has decided that there should be a way for a program to lock our DLL in memory if it desires. For that purpose, it can call our `IClassFactory`'s `LockServer` function, passing a 1 if it wants us to increment a count of locks on our DLL, or 0 if it wants to decrement a count of locks on our DLL. So, we also need a second static `DWORD` reference count which we'll call `LockCount`. (And of course, this also needs to be initialized to 0 when our DLL loads.) Our `LockServer` function now becomes:

```
static DWORD LockCount = 0;

HRESULT STDMETHODCALLTYPE
        classLockServer(IClassFactory *this, BOOL flock)
{
   if (flock) InterlockedIncrement(&LockCount);
   else InterlockedDecrement(&LockCount);

   return(NOERROR);
}
```

Now we're ready to write our `DllCanUnloadNow` function:

```
HRESULT PASCAL DllCanUnloadNow(void)
{
   // If someone has retrieved pointers to any of our objects, and
   // not yet Release()'ed them, then we return S_FALSE to indicate
   // not to unload this DLL. Also, if someone has us locked, return
   // S_FALSE
   return((OutstandingObjects | LockCount) ? S_FALSE : S_OK);
}
```

If you download the example project, the source file for our DLL (*IExample.c*) is in the directory *IExample*. Also supplied are Microsoft Visual C++ project files that create a DLL (*IExample.dll*) from this source.

Our C++/C include file

As mentioned earlier, in order for a program written in C++/C to use our `IExample` DLL, we need to give that program's author our `IExample`'s, and its VTable's, GUIDs. We'll put those GUID macros in an include (*.H*) file which we can distribute to others, and also include in our DLL source. We also need to put the definition of our `IExampleVtbl`, and `IExample`, structs in this include file, so the program can call our functions via the `IExample` we give it.

Up to now, we defined our `IExampleVtbl`, and `IExample`, structs as so:

```
typedef HRESULT STDMETHODCALLTYPE QueryInterfacePtr(IExample *, REFIID, void **);
typedef ULONG STDMETHODCALLTYPE AddRefPtr(IExample *);
typedef ULONG STDMETHODCALLTYPE ReleasePtr(IExample *);
typedef HRESULT STDMETHODCALLTYPE SetStringPtr(IExample *, char *);
typedef HRESULT STDMETHODCALLTYPE GetStringPtr(IExample *, char *, long);

typedef struct {
   QueryInterfacePtr  *QueryInterface;
   AddRefPtr          *AddRef;
   ReleasePtr         *Release;
```

```
   SetStringPtr       *SetString;
   GetStringPtr       *GetString;
} IExampleVtbl;

typedef struct {
   IExampleVtbl *lpVtbl;
   DWORD         count;
   char          buffer[80];
} IExample;
```

There is one problem with the above. We don't want to let the other program know about our "`count`" and "`buffer`" members. We want to hide them from the program. A program should never be allowed to directly access our object's data members. It should know only about the "`lpVtbl`" member so that it can call our functions. So, as far as the program is concerned, we want our `IExample` to be defined as so:

```
typedef struct {
   IExampleVtbl *lpVtbl;
} IExample;
```

Furthermore, although the `typedef`s for the function definitions make things easier to read, if you have a lot of functions in your object, this could get verbose and error-prone.

Finally, there is the problem that the above is a C definition. It really doesn't make things easy for a C++ program which wants to use our COM object. After all, even though we've written `IExample` in C, our `IExample` struct is really a C++ class. And it's a lot easier for a C++ program to use it defined as a C++ class than a C struct.

Instead of defining things as above, Microsoft provides a macro we can use to define our VTable and object in a way that works for both C and C++, and hides the extra data members. To use this macro, we must first define the symbol `INTERFACE` to the name of our object (which in this case is `IExample`). And prior to that, we must `undef` that symbol to avoid a compiler warning. Then, we use the `DECLARE_INTERFACE_` macro. Inside of the macro, we list our `IExample` functions. Here's what it will look like:

```
#undef  INTERFACE
#define INTERFACE   IExample
DECLARE_INTERFACE_ (INTERFACE, IUnknown)
{
   STDMETHOD  (QueryInterface)  (THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)   (THIS) PURE;
   STDMETHOD_ (ULONG, Release)  (THIS) PURE;
   STDMETHOD  (SetString)       (THIS_ char *) PURE;
   STDMETHOD  (GetString)       (THIS_ char *, DWORD) PURE;
};
```

This probably looks a bit bizarre.

When defining a function, `STDMETHOD` is used whenever the function returns an `HRESULT`. Our `QueryInterface`, `SetString`, and `GetString` functions return an `HRESULT`. `AddRef` and `Release` do not. Those latter two return a `ULONG`. So that's why we instead use `STDMETHOD_` (with an ending underscore) for those two. Then, we put the name of the function in parentheses. If the function doesn't return an `HRESULT`, we need to put what type it returns, and then a comma, before the function name. After the function name, we list the function's arguments in parentheses. `THIS` refers to a pointer to our object (i.e., `IExample`). If the only thing passed to the function is that pointer, then you simply put `THIS` in parentheses. That's the case for the `AddRef` and `Release` functions. But the other functions have additional arguments. So, we must use `THIS_` (with an ending underscore). Then we list the remaining arguments. Notice that there is **no** comma between `THIS_` and the remaining arguments. But there is a comma in between each of the remaining arguments. Finally, we

put the word `PURE` and a semicolon.

To be sure, this is a weird macro, and it's this way mostly to define a COM object so that it works both for a plain C compiler as well as a C++ compiler.

"But where's the definition of our `IExample` *struct?"*, you may ask. This macro is very weird indeed. It causes the C compiler to automatically generate the definition of a `IExample` struct that contains only the "`lpVtbl`" member. So just by defining our VTable this way, we automatically get a definition of `IExample` suitable for some other programmer.

Paste our two GUID macros into this include file, and we're all set. I did that to create the file *IExample.h*.

But as you know, our `IExample` really has two more data members. So what we're going to have to do is define a "variation" of our `IExample`, inside of our DLL source file. We'll call it a "`MyRealIExample`", and it will be the real definition of our `IExample`:

```
typedef struct {
   IExampleVtbl *lpVtbl;
   DWORD         count;
   char          buffer[80];
} MyRealIExample;
```

And we'll change a line in our `IClassFactory`'s `CreateInstance` so that we allocate a `MyRealIExample` struct:

```
if (!(thisobj = GlobalAlloc(GMEM_FIXED, sizeof(struct MyRealIExample))))
```

The program doesn't need to know that we're actually giving it an object that has some extra data members inside it (which are for all practical purposes, hidden from that program). After all, both of these structs have the same "`lpVtbl`" member pointing to the same array of function pointers. But now, our DLL functions can get access to those "hidden" members just by typecasting a `IExample` pointer to a `MyRealIExample` pointer.

The Definition (DEF) file

We also need a DEF file to expose the two functions `DllCanUnloadNow` and `DllGetClassObject`. Microsoft's compiler also wants them to be defined as `PRIVATE`. Here's our DEF file, which must be fed to the linker:

```
LIBRARY IExample
EXPORTS
DllCanUnloadNow   PRIVATE
DllGetClassObject PRIVATE
```

Install the DLL, and register the object

We've now completed everything we need to do in order to make our *IExample.dll*. We can go ahead and compile *IExample.dll*.

But that's not the end of our job. Before any other program can use our `IExample` object (i.e., DLL), we need to do two things:

1. Install our DLL somewhere that can be found by the computer running the program.

2. Register our DLL as a COM component.

We need to create an install program that will copy *IExample.DLL* to a well-chosen location. For example, perhaps we'll create a "*IExample*" directory in the *Program Files* directory, and copy the DLL there. (Of course, our installer should do version checking, so that if there is a later version of our DLL already installed there, we don't overwrite it with an earlier version.)

We then need to register this DLL. This involves creating several registry keys.

We first need to create a key under **HKEY_LOCAL_MACHINE\Software\Classes\CLSID**. For the name of this new key, we must use our `IExample` object's GUID, but it must be formatted in a particular, text string format.

If you download the example project, the directory *RegIExample* contains an example installer for *IExample.dll*. The function `stringFromCLSID` demonstrates how to format our `IExample` GUID into a text string suitable for creating a registry key name with it.

Note: This example installer does not copy the DLL to some well-chosen location before registering it. Rather, it allows you to pick out wherever you've compiled *IExample.dll* and register it in that location. This is just for convenience in developing/testing. A production quality installer should copy the DLL to a well-chosen location, and do version checking. These needed enhancements are left for you to do with your own installer.

Under our "GUID key", we must create a subkey named **InprocServer32**. This subkey's default value is then set to the full path where our DLL has been installed.

We must also set a value named **ThreadingModel** to the string value "both", if we don't need to restrict a program to calling our DLL's functions only from a single thread. Since we don't use global data in our `IExample` functions, we're thread-safe.

After we run our installer, *IExample.dll* is now registered as a COM component on our computer, and some program can now use it.

Note: The directory *UnregIExample* contains an example uninstaller for *IExample.dll*. It essentially removes the registry keys that `RegIExample` created. A production quality uninstaller should also remove *IExample.dll* and any directories created by the installer.

An example C program

Now we're ready to write a C program that uses our `IExample` COM object. If you download the example project, the directory *IExampleApp* contains an example C program.

First of all, the C program `#include`s our *IExample.h* include file, so it can reference our `IExample` object's, and its VTable's, GUIDs.

Before a program can use any COM object, it must initialize COM, which is done by calling the function `CoInitialize`. This need be done only once, so a good place to do it is at the very start of the program.

Next, the program calls `CoGetClassObject` to get a pointer to *IExample.dll*'s `IClassFactory` object. Note that we pass the `IExample` object's GUID as the first argument. We also pass a pointer to our variable `classFactory` which is where a pointer to the `IClassFactory` will be returned to us, if all goes well.

Once we have the `IClassFactory` object, we can call its `CreateInstance` function to get a `IExample` object. Note how we use the `IClassFactory` to call its `CreateInstance` function. We get the function via `IClassFactory`'s VTable (i.e., its `lpVtbl` member). Also note that we pass the `IClassFactory` pointer as the first argument. Remember that this is standard COM.

Note that we pass `IExample`'s VTable GUID as the third argument. And for the fourth argument, we pass a pointer to our variable `exampleObj` which is where a pointer to an `IExample` object will be returned to us, if all goes well.

Once we have an `IExample` object, we can `Release` the `IClassFactory` object. Remember that a program must call an object's `Release` function when done with the object. The `IClassFactory` is an object, just like `IExample` is an object. Each has its own `Release` function, which must be called when we're done with the object. We don't need the `IClassFactory` any more. We don't want to obtain any more `IExample` objects, nor call any of the `IClassFactory`'s other functions. So, we can `Release` it now. Note that this does not affect our `IExample` object at all.

So next, we call the `IClassFactory`'s `Release` function. Once we do this, our `classFactory` variable no longer contains a valid pointer to anything. It's garbage now.

But we still have our `IExample` pointer. We haven't yet `Released` that. So next, we decide to call some of `IExample`'s functions. We call `SetString`. Then we follow up with a call to `GetString`. Note how we use the `IExample` pointer to call its `SetString` function. We get the function via `IExample`'s VTable. And also notice that we pass the `IExample` pointer as the first argument. Again, standard COM.

When we're finally done with the `IExample`, we `Release` it. Once we do this, our `exampleObj` variable no longer contains a valid pointer to anything.

Finally, we must call `CoUninitialize` to allow COM to clean up some internal stuff. This needs to be done once only, so it's best to do it at the end of our program (but only if `CoInitialize` succeeded).

> There's also a function called `CoCreateInstance` that can be used to replace the call to `CoGetClassObject` (to get the DLL's `IClassFactory`), and then the call to the `IClassFactory`'s `CreateInstance`. `CoCreateInstance` itself calls `CoGetClassObject`, and then calls the `IClassFactory`'s `CreateInstance`. `CoCreateInstance` directly returns our `IExample`, bypassing the need for us to get the `IClassFactory`. Here's an example use:

```
if ((hr = CoCreateInstance(&CLSID_IExample, 0,
        CLSCTX_INPROC_SERVER, &IID_IExample, &exampleObj)))
   MessageBox(0, "Can't create IExample object",
              "CoCreateInstance error",
              MB_OK|MB_ICONEXCLAMATION);
```

An example C++ program

The directory *IExampleCPlusApp* contains an example C++ program. It does exactly what the C example does. But, you'll note some important differences. First, because the macro in *IExample.h* defines `IExample` as a C++ class (instead of a struct), and because C++ handles classes in a special way, the C++ program calls our `IExample` function in a different format.

In C, we get an `IExample` function by directly accessing the VTable (via the `lpVtbl` member), and we always pass the `IExample` as the first argument.

The C++ compiler knows that a class has a VTable as its first member, and automatically accesses its `lpVtbl` member to get a function in it. So, we don't have to specify the `lpVtbl` part. Also, the C++ compiler automatically passes the object as the first argument.

So whereas in C, we code:

```
classFactory->lpVtbl->CreateInstance(classFactory, 0,
                      &IID_IExample, &exampleObj);
```

in C++, we instead code:

```
classFactory->CreateInstance(0, IID_IExample, &exampleObj);
```

Note: We also omit the `&` on the `IID_IExample` GUID. The GUID macro for C++ doesn't require that it be specified.

Modifying the code

To create your own object, make a copy of the *IExample* directory. Delete the *Debug* and *Release* sub-directories, and the following files:

```
IExample.dsp
IExample.dsw
IExample.ncb
IExample.opt
IExample.plg
```

In the remaining files (*IExample.c*, *IExample.h*, *IExample.def*), search and replace the string *IExample* with the name of your own object, for example `IMyObject`. Rename these files per your new object name (i.e., *IMyObject.c*, etc.).

Create a new Visual C++ project with your new object's name, and in this directory. For the type of project, choose "Win32 Dynamic-Link Library". Create an empty project. Then add the above three files to it.

Make sure you use *GUIDGEN.EXE* to generate your own GUIDs for your object and its VTable. **Do not use the GUIDs that I generated.** Replace the GUID macros in the *.H* file (and remember to replace the `<<name>>` part of the GUID macro too).

Remove the functions `SetString` and `GetString` in the *.C* and *.H* files, and add your own functions instead. Modify the `INTERFACE` macro in the *.H* file to define the functions you added.

Change the data members of `MyRealIExample` (i.e., `MyRealIMyObject`, whatever) to what you want.

Modify the installer to change the first three strings in the source.

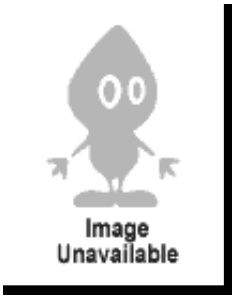
In the example programs, search and replace the string *IExample* with the name of your object.

What's next?

Although a C or C++ program, or a program written in most compiled languages, can use our COM object, we have yet to add some support that will allow most interpreted languages to use our object, such as Visual Basic, VBscript, JScript, Python, etc. This will be the subject of Part II of this series.

License

About the Author



Jeff Glatt

United States

No Biography provided

Comments and Discussions

220 messages have been posted for this article Visit **http://www.codeproject.com/Articles/13601/COM-in-plain-C** to post and view comments on this article, or click **here** to get a print view with messages.