The layout of a COM object

devblogs.microsoft.com/oldnewthing/20040205-00

February 5, 2004

Raymond Chen

The Win32 COM calling convention specifies the layout of the virtual method table (vtable) of an object. If a language/compiler wants to support COM, it must lay out its object in the specified manner so other components can use it.

It is no coincidence that the Win32 COM object layout matches closely the C++ object layout. Even though COM was originally developed when C was the predominant programming language, the designers saw fit to "play friendly" with the up-and-coming new language C++.

The layout of a COM object is made explicit in the header files for the various interfaces. For example, here's IPersist from objidl.h, after cleaning up some macros.

```
typedef struct IPersistVtbl
{
    HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
        IPersist * This,
        /* [in] */ REFIID riid,
        /* [iid_is][out] */ void **ppvObject);
    ULONG ( STDMETHODCALLTYPE *AddRef )(
        IPersist * This);
    ULONG ( STDMETHODCALLTYPE *Release )(
        IPersist * This);
    HRESULT ( STDMETHODCALLTYPE *GetClassID )(
        IPersist * This,
        /* [out] */ CLSID *pClassID);
} IPersistVtbl;
struct IPersist
{
    const struct IPersistVtbl *lpVtbl;
};
```

This corresponds to the following memory layout:

p →	lpVtbl →	QueryInterface
		AddRef

	Release
	GetClassID

What does this mean?

A COM interface pointer is a pointer to a structure that consists of just a vtable. The vtable is a structure that contains a bunch of function pointers. Each function in the list takes that interface pointer (p) as its first parameter ("this").

The magic to all this is that since your function gets p as its first parameter, you can "hang" additional stuff onto that vtable:

p →	lpVtbl →	QueryInterface
	… other stuff …	AddRef
		Release
		GetClassID

The functions in the vtable can use offsets relative to the interface pointer to access its other stuff.

If an object implements multiple interfaces but they are all descendants of each other, then a single vtable can be used for all of them. For example, the object above is already set to be used either as an IUnknown or as an IPersist, since IUnknown is a subset of IPersist.

On the other hand, if an object implements multiple interfaces that are not descendants of each other, then you get multiple inheritance, in which case the object is typically laid out in memory like this:

p →	lpVtbl →	→ →	QueryInterface (1)
q →	lpVtbl →	QueryInterface (2)	AddRef (1)
	… other stuff …	AddRef (2)	Release (1)
		Release (2)	…
		…	

If you are using an interface that comes from the first vtable, then the interface pointer is p. But if you're using an interface that comes from the second vtable, then the interface pointer is q.

Hang onto that diagram, because tomorrow we will learn about those mysterious “adjustor thunks”.

Raymond Chen

Follow