Peter Kohout

Unread,
Jan 5, 2002, 10:19:44 AM

I recently came across this article. I am really amazed at the concept and since I have looking around for ways to wrap the WndProc into some sort of useful framework, think that this is the way to go. I am totally baffled by the use of assembly language in the following line

```
thunk.m_relproc = (int)proc - ((int)this+sizeof(_WndProcThunk));
```

I think it has something to do with the memory lay out of the class which contains both static and virtual functions. I have tried to email the author of the article a number of times but am not getting any response. If anyone could explain to me how this piece of code works or just point me in the right direction I would be very grateful.

Thanking in advance

Peter

Thunking WndProcs in ATL

By Fritz Onion
Published in C++ Report, March 1999 issue.

With the recent addition of Wizard support for ATL window classes, developers are turning to ATL for more than COM servers. While MFC remains the library of choice for full application development, ATL provides an attractive alternative when a simple set of window wrapper classes is all that is needed. In this column I will look at how ATL's window classes encapsulate window handles, providing basic window functionality through a set of C++ classes. In particular, I will look at how ATL dispatches messages to C++ classes from window procedures through an interesting technique involving an assembly language thunk.
There are two pieces of sample code available for download to which you may want to refer while reading the column. The first is a very simple application that uses ATL's window classes to bring up a frame window that displays some text in the client area. The second is a similar application that instead uses the Window class we will build in this column. Its implementation is essentially identical to ATL's window classes, but does not use the layered template hierarchy of ATL to simplify the discussion of the window encapsulation techniques. Both samples can be downloaded at http://www.develop.com/hp/onion/cpprep/AtlWindows.zip.

The Goal

Our primary goal in building a C++ class that wraps the functionality of a window handle is to allow users of the class to create customized windows by simply deriving from our class. This means that the details of defining window procedures and individual message handling must be masked behind a normal-looking C++ class. Our goal will be to build such a class using the same techniques that ATL's window classes employ. Although we will be looking at the techniques used by the ATL window classes, we will use our own window class simply called Window to actually present the code. This is primarily because the ATL classes make extensive use of templates (hence the 'T' in ATL) and trying to understand its layered template hierarchy on top of this interesting window encapsulation technique is more, I suspect, than the average reader would care to digest in one sitting. A complete example of building a custom window with ATL's classes is provided in one of the previously mentioned samples.
Having said that, let's take a look at how we would like to be able to use our Window class to create a custom window. As a simple example, our custom window will simply write the text "Foo" in its client area, and terminate our application when destroyed.

```
class FooWindow : public Window
```

```
{
	public:
	virtual void OnPaint()
	{
		PAINTSTRUCT ps;
		HDC dc = BeginPaint(&ps);
		RECT r;
		GetClientRect(&r);
		DrawText(hdc, "Foo", 3, &r, DT_CENTER);
		EndPaint(&ps);
	}
	virtual void OnDestroy()
	{
		PostQuitMessage(0);
	}
};
```

And the code to actually create our simple window would look like:

```
RECT r; SetRect(&r, 0,0,500,500);
FooWindow wnd;
wnd.Create(NULL, r);
...
```

Notice that there is no reference to a window class or a window procedure anywhere in the code. That doesn't mean they aren't there, but rather that they are implicitly created and managed for us through our C++ class. The allure of this approach is its simplicity for the developer. Without even having to know what a window procedure is, a C++ developer can create a custom window.

The first step of putting this class together is trivial - adding support for manipulating the window handle. Add an HWND data member, and provide all of the SDK functions that manipulate HWNDs as member functions, implicitly passing our HWND data member as a parameter:

```
class Window
{
	HWND m_hWnd;
public:
	Window() : m_hWnd(0) {}
	BOOL GetClientRect(LPRECT lpRect) const { return ::GetClientRect(m_hWnd, lpRect); }
	HDC BeginPaint(LPPAINTSTRUCT lpPaint) { return ::BeginPaint(m_hWnd, lpPaint); }
	void EndPaint(LPPAINTSTRUCT lpPaint) { ::EndPaint(m_hWnd, lpPaint); }
	BOOL ShowWindow(int nCmdShow) { return ::ShowWindow(m_hWnd, nCmdShow); }
	// Continue with all HWND related SDK functions
};
```

The remaining tasks are all related to messages and are not quite as easily solved.

The Problems

To make our Window class a reality, we are tasked with two problems. The first, and most difficult is, how do we forward messages from a window procedure onto the class instance? This is difficult, because messages are received by a window procedure through a mechanism that knows nothing about our C++ class. A window procedure receives four parameters:

```
LRESULT WndProc(HWND hWnd, UINT uMsg,
WPARAM wParam, LPARAM lParam);
```

The last three parameters are the message itself, and the first parameter is the handle of the window to which the message was issued. If we are to make the window procedure transparent

to the user of our Window class, we must register a generic window procedure that somehow passes all incoming messages onto the correct instance of our Window class.

The only link we have is the HWND parameter - somehow we must be able to retrieve the instance of our Window class that wraps a given window handle. Once we have the class, we can simply invoke the corresponding message function on the class, which brings us to our second problem.

Assuming we can retrieve the correct instance of our Window class within a window procedure, how then do we issue the message to the class? The obvious solution is to add a virtual function for every possible message (as our sample use of the Window class implied). Within our window procedure, we would invoke the appropriate virtual function based on the incoming message.

The user of our class would then simply override the subset of messages she cared about, and we would let the virtual function mechanism of C++ take care of the rest.

Unfortunately, the obvious solution is probably not the best. Because there are a large number of messages (over 150) we are going to have a rather large vtable if we make each message a virtual function. This vtable would be replicated for each derivative class, incurring more overhead than we probably ought to for a simple wrapper class. So the second of our two problems is, how can we translate messages into method calls on our Window class without incurring the overhead of a large vtable? If we can solve each of these problems, we will have a viable window wrapper class.

Message Cracking

Let's deal with the second (and simpler) of the two problems first. We basically want to provide virtual function capabilities to a class without the overhead of a full vtable. MFC solves this problem with its message map mechanism, which basically implements partially filled vtables, trading time (performing a linear search on the table) for space (only overridden functions are added).

The approach ATL takes is to use a single virtual function to map the incoming message onto the correct class definition, and then asks the class itself to transform the message ID into a method call on itself. The function returns a Boolean to indicate whether it handled the message or not, allowing the caller to perform some default handling if was not handled.

To add this capability to our Window class, let's add the single virtual function (with the same signature as ATL's) which must be overridden by derived classes to perform message handling.

```
class Window
{
public:
	virtual bool ProcessWindowMessage(HWND hWnd, UINT uMsg,
	WPARAM wParam, LPARAM lParam,
	LRESULT& lResult, DWORD dwMsgMapID = 0)
	{
		// No messages are handled at this level
		return false;
	}
};
```

And an example of its use might look like:

```
class FooWindow : public Window
{
public:
	virtual bool ProcessWindowMessage(HWND hWnd, UINT uMsg,
	WPARAM wParam, LPARAM lParam,
	LRESULT& lResult, DWORD dwMsgMapID = 0)
	{
		BOOL bHandled;
```

```
		if(uMsg == WM_PAINT)
			lResult = OnPaint(uMsg, wParam, lParam, bHandled);
		if(uMsg == WM_DESTROY)
			lResult = OnDestroy(uMsg, wParam, lParam, bHandled);
		return bHandled;
	}
	LRESULT OnPaint(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
	{
		// Drawing code would go here
		bHandled = true;
		return 0;
	}
	LRESULT OnDestroy(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
	{
		PostQuitMessage(0);
		bHandled = true;
		return 0;
	}
};
```

There are, of course, a number of macros to generate much of this code for you in ATL (and, in fact, wizard support in Visual C++ 6.0). Using ATL's macros, the above code would be written as follows:

```
class FooWindow : public Window
{
public:
	BEGIN_MSG_MAP(CFooWindow)
		MESSAGE_HANDLER(WM_PAINT, OnPaint)
		MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
	END_MSG_MAP()
	// The handler functions remain unchanged
};
```

We now have most of the functionality of our original virtual function model, without the unwieldy vtable. The one piece that's missing is the inheritance of message handlers from base classes. If we derive from a class that has implemented a handler for the WM_PAINT message, we would expect to inherit that painting functionality as we would have if the handlers were virtual functions.

ATL forces you to be explicit about inheriting message handling functionality from base classes. You must explicitly call the base class' implementation of ProcessWindowsMessage within your implementation; typically at the end of the function so that any messages handled by both your class and its base class will use your class' handler.

For example, suppose our FooWindow class inherited from a BarWindow class whose message handling functionality we wanted to inherit. Our implementation of ProcessWindowsMessage would look like:

```
class FooWindow : public BarWindow
{
public:
	virtual bool ProcessWindowMessage(
		HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam, LRESULT& lResult,
		DWORD dwMsgMapID = 0
	)
	{
		BOOL bHandled;
		if(uMsg == WM_PAINT)
			lResult = OnPaint(uMsg, wParam, lParam, bHandled);
```

```
                if(uMsg == WM_DESTROY)
                        lResult = OnDestroy(uMsg, wParam, lParam, bHandled);
                return bHandled ? true : BarWindow::ProcessWindowMessage (...);
        }
};
```

The same code using ATL's macros would now look like:

```
class FooWindow : public BarWindow
{
public:
BEGIN_MSG_MAP(CFooWindow)
MESSAGE_HANDLER(WM_PAINT, OnPaint)
MESSAGE_HANDLER(WM_DESTROY, OnDestroy)
CHAIN_MSG_MAP(BarWindow)
END_MSG_MAP()
};
```

Wndproc Thunking

We now return to the first (and harder) of our two problems: how do we forward messages from a window procedure onto the class instance? MFC solves this problem by maintaining a global associative array mapping HWNDs to C++ class instances. Every time a window is created from a wrapper class, the class instance and the HWND are added to the global array. This works fine for single-threaded applications, but with multiple threads, you need to either protect access to the global array (adding contention and speed degradation) or store the array not globally but locally for each thread (in thread local storage). MFC chose the latter of these two solutions.

Unfortunately, storing this relationship in TLS means that passing instances of C++ classes wrapping window handles between threads can cause problems (because the mapping is only in one thread). ATL prides itself on being a very thread-savvy library, so an alternative approach which introduces no contention and does not use TLS is desired. To understand its solution, let's re-state our original problem. How do we retrieve the C++ class associated with a window handle from within a window procedure?
The trick is not to store the association between the window handle and the C++ class at all, but rather to change the window procedure to accept the C++ class as a parameter instead of the window handle. When our Window class calls its Create method to create the underlying window handle, we will cache this pointer in some well-known place, and create the window handle with a special 'start-up' window procedure. Once the window handle is created, our 'start-up' window procedure will be called with an initial create message. In our 'start-up' window procedure, we create a sequence of assembly language instructions (the thunk) that replaces the HWND parameter of the window procedure with the physical address of the pointer (retrieved from the well-known place), and then jumps to the 'real' window procedure with the altered stack. We then replace the window procedure of our window with the address of our thunk code. In our 'real' window procedure, we grab the C++ class by simply casting the HWND parameter into our Window class, and call our virtual function ProcessWindowMessage ().

Now, let's step through the code that actually does this. First, we need to register a window class implicitly, so we will add a Register () function to our Window class and call it from within our Create () method. The Register () function will register a generic window class (with a synthetically created name). We will cache this window class as an ATOM in our Window class.

```
class Window
{
        ATOM m_WndClassAtom;
public:
        Window() : m_WndClassAtom(0) {}
```

```
	void Register()
	{
		if (m_WndClassAtom != 0) return; // Already have a class
		// Synthetic window class naming
		TCHAR buf[255];
		wsprintf(buf, "ATL:%8.8X", (DWORD)this);
		// See if class is already registered
		WNDCLASSEX wc;
		m_WndClassAtom = ::GetClassInfoEx(...,buf, &wc);
		if (m_WndClassAtom == 0) // Not registered
		{
			wc.lpfnWndProc = StartWindowProc;
			wc.lpszClassName = buf;
			// Remaining members populated with
			// standard values
			m_WndClassAtom = ::RegisterClassEx(&wc);
		}
	}
};
```

The actual window creation is where we begin to hook things up. In our Create function, we first register our window class using our new Register function. Next, we need to store our class instance in some well-known place so that it can later be retrieved by our 'start-up' window procedure.

ATL's solution to this is to cache it in its global _Module object. We mentioned that we were going to avoid global storage in our solution, but keep in mind this is only temporary - as soon as our window procedure gets called, we will extract the pointer from the _Module object, and there will be no more global state. Because it is global, the actual caching and retrieval of the '**this**' pointer is performed on a thread-relative basis, and access is protected with a critical section. Once we cache ourselves with the _Module object, we simply create the window, using our newly registered window class which associates the 'start-up' window procedure with our window.

```
bool Window::Create(HWND hWndParent,...)
{
	Register();
	// Cache our this pointer with the _Module
	_Module.AddCreateWndData(..., this);
	// Create the HWND
	LPCTSTR wndClass = MAKELONG(m_WndClassAtom, 0);
	m_hWnd = ::CreateWindow(wndClass, ...);
	return m_hWnd != NULL;
}
```

Now we need to define the actual thunk code that will be used to alter the stack of our main window procedure. ATL defines the thunk code as a structure:

```
#pragma pack(push,1)
struct _WndProcThunk
{
	DWORD m_mov;
	DWORD m_this;
	BYTE m_jmp;
	DWORD m_relproc;
};
#pragma pack(pop)
```

The packing alignment must be set to one byte so that the instructions lay out in memory properly. This structure will contain the code for two assembly instructions which perform the actual thunk.

The mov instruction will change the incoming HWND parameter on the stack (located at esp+4) to be the '**this**' pointer of the C++ class instance associated with that window. The jmp instruction performs a relative jump to the actual window procedure with the newly altered stack.

```
mov dword ptr [esp+4], pThis
jmp WndProc
```

Next, we need a way to populate the thunk with the correct pointer values relative to the class which is creating the window handle. To simplify this procedure, we can define a class which contains our thunk structure, and provide it with an initialization function. The initialization function will need the real window procedure address and the '**this**' pointer of the class instance. We will then add an instance of this thunk class to our Window class, because each instance of our class will need its own thunk.

```
class CWndProcThunk
{
public:
	_WndProcThunk thunk;
	void Init(WNDPROC proc, void* pThis)
	{
		thunk.m_mov = 0x042444c7;// mov dword ptr [esp+4]
		thunk.m_this = (DWORD)pThis;
		thunk.m_jmp = 0xe9; // jmp WndProc
		thunk.m_relproc =
		(int)proc - ((int)this+sizeof(_WndProcThunk));
	}
};
class Window
{
	CWndProcThunk m_thunk;
	...
};
```

Our 'start-up' window procedure is the next critical piece. The purpose of the 'start-up' window procedure will only be to hook up the thunk, and then get out of the way. In order to initialize the thunk, it will retrieve the C++ class instance pointer from the _Module. When it retrieves '**this**' pointer, it is assuming that no other C++ class has been cached since our class called the Create method.

This is a valid assumption relative to a single thread, because a window procedure is invoked immediately after the creation of a window handle. Because the _Module object caches the classes on a per-thread basis, there is no danger of retrieving the wrong class. We will also need the address of our 'real' window procedure, which is retrieved by calling the GetWindowProc method of our Window class. We will define this function shortly. The last task of our 'start-up' window procedure is to set the thunk code as the new window procedure for our window handle.

This is accomplished by calling SetWindowLong and passing in the thunk structure of our local CWndProcThunk class. The window procedure needs to be accessible independent of any instances of our class, so we make it a static member function of the Window class.

```
class Window
{
	static LRESULT CALLBACK StartWindowProc(
		HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam
```

```
	)
	{
		Window* pThis = _Module.ExtractCreateWndData();
		pThis->m_hWnd = hWnd;
		pThis->m_thunk.Init(pThis->GetWindowProc(), pThis);
		WNDPROC pProc = (WNDPROC)&(pThis->m_thunk.thunk);
		::SetWindowLong(hWnd, GWL_WNDPROC, (LONG)pProc);
		return pProc(hWnd, uMsg, wParam, lParam);
	}
...
};
```

Now everything is in place except our real window procedure, and that's easy. We simply define a window procedure which casts the HWND parameter to a Window pointer (this is what the thunk did for us), take that Window pointer, call the virtual function ProcessWindowMessage, and let the message propagation handle it from there. If the call to ProcessWindowMessage fails to handle the message, we defer to DefWindowProc for standard handling.

Finally, we add a method to retrieve this function pointer.

```
class Window
{
	static LRESULT CALLBACK WindowProc(
		HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam
	)
	{
	Window* pThis = (Window*)hWnd;
	LRESULT lRes;
	if (!pThis->ProcessWindowMessage(pThis->m_hWnd, uMsg, wParam, lParam, lRes, 0))
		return ::DefWindowProc(pThis->m_hWnd, uMsg, wParam, lParam);
	else
		return 0;
	}
	virtual WNDPROC GetWindowProc() { return WindowProc; }
...
};
```

Conclusions

ATL provides a lightweight set of classes that encapsulate the functionality and message propagation of window handles. The technique for mapping window procedure messages onto C++ class method invocation involves re-defining the window procedure using an assembly language thunk which alters the parameters on the call stack and forwards the call to the real window procedure.

It is important to be aware of this technique if you are using ATL's window classes, because some usual assumptions about window procedures and window handles may no longer hold. For example, each window handle created from the same C++ class will have a different window procedure address, even though semantically they map onto the same window procedure.
In general, however, ATL's window classes provide a clean solution to mapping messages onto method calls, with very little overhead.