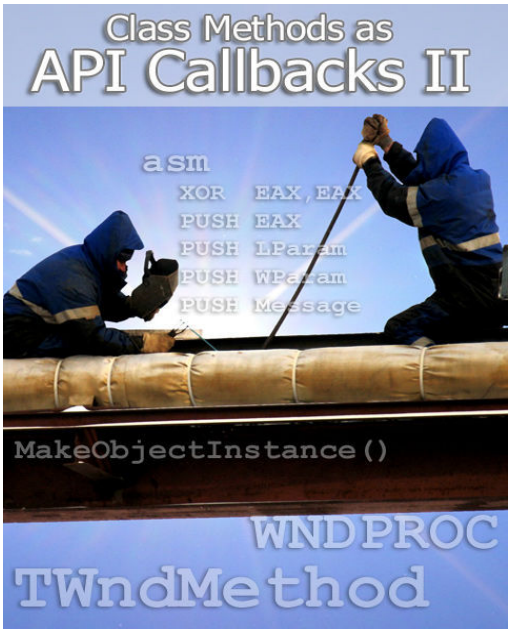
Using Class Methods as API Callbacks, Part II

By Remy Lebeau

Versions: C++Builder 2007, 2006, V6, V5, V4, V3, V1



In Part I of this series, I introduced some commonly used ways of using class methods as API callbacks. I also gave a brief overview of the VCL's `MakeObjectInstance()` function. This month, I will delve further into the inner workings of how `MakeObjectInstance()` actually works. This knowledge leads into Part III of this series, where I will introduce a new proxy system I have developed for using class methods as API callbacks in a much more flexible manner than `MakeObjectInstance()` offers.

Why is MakeObjectInstance() so important?

`MakeObjectInstance()` is at the core of Borland's message-handling architecture for `HWND`s that are created by the VCL. By using a few select x86 Assembly instructions, Borland's proxy code effectively delegates a call from a seemingly non-class function into a call to a class method, with the appropriate `this` pointer value passed to it. The proxy does this by manipulating the call stack directly. This approach is far more efficient than other frameworks, such as MFC, which use messy lookup tables to accomplish the same result (the ATL, by comparison, uses a proxy system similar to Borland's for handling `WNDPROC`-oriented proxies).

`MakeObjectInstance()` is widely used by the VCL, so Borland has optimized it to maximize its memory usage while reducing its memory allocations. I am not going to focus on that optimization in this article. It does not pertain to this discussion, and the code I will describe in Part III is more flexible than `MakeObjectInstance()` and thus does not make use of the same optimizations. I will instead focus on what a single proxy instance contains internally and how it does its work in general. If you want to know what Borland optimized, you can study the VCL source code on your own. It is available as an installable option in any C++Builder/Delphi version. As I develop my own proxy code further, I may implement my own optimizations at a later time.

Diving In

When calling `MakeObjectInstance()`, you pass it a pointer to a non-static class method of an existing object instance. The class method must match the signature of the VCL's `TWndMethod` type. `MakeObjectInstance()` returns a `void*` pointer to a `WNDPROC` proxy that you can then pass around and call like a normal non-class function pointer (to actually call it yourself, you will have to type-cast the `void*` pointer first). Refer to **Listing A** for an example.

Normally, declaring and using a pointer-to-class-method can be a little tricky in C++, especially when different class types are involved. Such pointers are class-specific, and calling class methods using them requires the use of special operators. Refer to **Listing B** for an example.

To support multiple pointer-to-class-method types generically in a single function using standard C++, you must use C++ templates instead. However, `MakeObjectInstance()` is implemented in Pascal (as is most of the VCL), so templates are out of the question. Fortunately, Borland has implemented a com-

piler extension—the `__closure` keyword—that makes it very easily to accomplish what is needed.

In a nutshell, a closure is a special structure containing two pointers – one to the memory address of the class method's implementation code, and the other to the object instance. When calling a method via a closure, the compiler generates machine code that sets up the call as if the method had been called by your code normally. As long as a class method matches the signature of the closure type, it does not matter which class the method comes from. Also, methods from different classes can be interchanged at will within the same closure instance (which is how VCL event handlers are able to work the way they do).

You may be thinking to yourself: "`MakeObjectInstance()` is given the memory addresses of the method code and the object instance, so the proxy should be able to just assign the `this` pointer and jump into the method, right?" That is exactly what it does. But how does it actually do that?

Inside __fastcall

Keep in mind that `MakeObjectInstance()` is specifically designed to work with class methods that match the signature of `TWndMethod`. The first thing to look at is the calling convention being used. Borland's `__fastcall` calling convention (another compiler extension) is the C++ analogue of Delphi's `Register` calling convention, which uses CPU registers (like its name suggests)—specifically `EAX`, `EDX`, and `ECX`, in that order—to accept the first three 32-bit-sized parameters from the caller. Subsequent parameters, or any parameters that are larger than 32 bits (`double`, `__int64`, etc), are passed on the stack instead, pushed in left-to-right order, and the called method pops the values from the stack when it exits. A return value, if any, is stored in the `EAX` register upon exit. I will touch on this more in Part III of this series.

When calling a non-static class method, the object's `this` pointer is passed as a hidden first parameter. For the `__fastcall` convention, the `this` pointer is passed to the class method via the `EAX` register. `TWndMethod` has an explicit second parameter, which is a reference to a `TMessage` structure. A reference is a 32-bit value, so it is passed to the class method via the `EDX` register. `TWndMethod` does not use the `ECX` register, so the proxy makes use of it for its own internal purpose.

Listing A: Using MakeObjectInstance()

```
typedef void __fastcall (__closure
*TWndMethod)(TMessage&);

__fastcall TMyClass::TMyClass()
{
  m_WndProc =
    MakeObjectInstance(&WndProc);
  m_DefProc = (WNDPROC)
    ::SetWindowLong(hSomeWnd,
      GWL_WNDPROC, (LONG) m_WndProc);
}

__fastcall TMyClass::~TMyClass()
{
  ::SetWindowLong(hSomeWnd, GWL_WNDPROC,
    (LONG) m_DefProc);
  FreeObjectInstance(m_WndProc);
}

void __fastcall TMyClass::
  WndProc(TMessage &Message)
{
  // process Message as needed …
}
```

Listing B: Example C++ method pointer usage

```
typedef void (TMyClass::*TMyMethodPtr)(int);

void TMyClass::DoSomethingWith(int Value)
{
  // use Value as needed …
}

void DoSomething()
{
  TMyClass myObject;
  TMyMethodPtr method =
    &TMyClass::DoSomethingWith;
  (myObject.*method)(12345);
}
```

Inside __stdcall

The next thing to look at is the `WNDPROC` signature that `MakeObjectInstance()` is wrapping:

```
typedef LRESULT WINAPI (*WNDPROC)(
  HWND, UINT, WPARAM, LPARAM);
```

The `__stdcall` calling convention uses only the stack for passing parameters. They are pushed on the stack in right-to-left order, and the called method pops the values from the stack when it exits. A return value, if any, is stored in the `EAX` register upon exit. I will touch on this more in Part III of this series.

Converting from one to the other

Examining the two signatures, there are some similarities, and there are also some differences, in how they behave. So, the proxy has to do some clever conversion to make a TWndMethod class method pointer act as a WNDPROC function pointer. In C++, a simple wrapper function might look something like the following:

```
LRESULT WINAPI WndProcWrapper(
  HWND hWnd, UINT uMsg, WPARAM wParam,
  LPARAM lParam)
{
  TMessage Msg;
  Msg.Message = (int) uMsg;
  Msg.WParam = (int) wParam;
  Msg.LParam = (int) lParam;
  Msg.Result = 0;

  // this effectively calls
  // (object->*method)(Msg)…
  WndMethodPtr(Msg);

  return Msg.Result;
}
```

This code, in fact, is exactly what the MakeObjectInstance() proxy is doing. However, the implementation is not written in C++, but instead in x86 Assembly. And, there is still the question of where the proxy gets the TWndMethod class method pointer so it knows which object to call into. It is time to look inside the implementation of the proxy itself.

Looking behind the curtain

The proxy that MakeObjectInstance() generates is not actually a function, in the normal sense. It is a block of executable memory that is dynamically allocated at run-time using the Win32 API VirtualAlloc() function. The PAGE_EXECUTE_READWRITE flag is used so the memory block can directly contain machine code that the CPU can run. The memory block holds a specially designed 19-byte structure (OK, it is a bit larger than that—I am simplifying the details slightly). **Listing C** shows what this structure would look like in C++ terms.

The memory block contains three x86 Assembly instructions in it: CALL NEAR PTR [offset], POP ECX, and JMP [offset], in that order. The CALL instruction is stored at the very beginning of the memory block so it is the first thing the CPU sees when the proxy stub is executed. It contains an offset to the POP ECX instruction. The JMP instruction, which immediately follows the POP ECX instruction in memory, contains an offset to the actual code that performs the class method invocation.

Listing C: Proxy stub

```
#pragma pack(push, 1)
struct TProxyStub
{
  unsigned char CallOp;
  int CallOffset;
  TWndMethod Method;
  unsigned char PopEcxOp;
  unsigned char JmpOp;
  int JumpOffset;
};
#pragma pack(pop)
```

Wait—the code to call the class method is not inside the proxy itself? No, it is not. The actual logic to set up and call the class method is contained inside a wrapper function that is implemented within the VCL source code. The proxy is just a general-purpose stub to call the wrapper function, which then does the actual work. I utilize that fact in the new proxy system I will describe in Part III of this series.

The odd part of the proxy stub is the CALL instruction. As you can see in **Listing C**, the TWndMethod pointer immediately follows the CALL instruction in the proxy's memory block. Borland uses the CALL instruction in a roundabout manner to pass the TWndMethod pointer to the invocation wrapper function. To briefly summarize, a CALL instruction essentially just pushes the next instruction's memory address (a.k.a., the return address) from the CPU's EIP register onto the stack and then jumps execution to the specified offset. The pushed address is where the called code is supposed to return to when finished (usually by calling the RET instruction). In this case, however, the CALL is actually pushing the memory address of the TWndMethod pointer onto the stack, and then jumping to the POP ECX instruction. That instruction then pops the TWndMethod pointer off of the stack and puts it into the ECX register. The subsequent JMP instruction then jumps to the invocation wrapper function, where the real fun begins.

Inside the wrapper function

Why does the proxy go to all this effort? The invocation wrapper function that is implemented by the VCL—StdWndProc()—is a WNDPROC callback. The

CALL/POP/JMP trio allows execution to flow into StdWndProc() as if the OS had called it directly, even though it really did not. However, the CALL/POP pair loads the target TWndMethod pointer into the ECX register first, which would not have happened otherwise.

So what does StdWndProc() do? Essentially, it does the same as the WndProcWrapper() method shown earlier, but is implemented in x86 Assembly. Here's the code:

```
function StdWndProc(Window: HWND; Message,
  WParam: Longint; LParam: Longint):
Longint; stdcall; assembler;
asm
  XOR     EAX,EAX
  PUSH    EAX
  PUSH    LParam
  PUSH    WParam
  PUSH    Message
  MOV     EDX,ESP
  MOV     EAX,[ECX].Longint[4]
  CALL    [ECX].Pointer
  ADD     ESP,12
  POP     EAX
end;
```

Remember that the TWndMethod pointer is stored in the ECX register. Upon entry, the call stack is identical to what it was when the proxy was first entered. A standard stack frame is established (handled by the compiler), copies of the WNDPROC parameters on the stack are re-pushed on the stack in the layout of a TMessage structure, whose memory address is then stored in the EDX register (the second parameter of __fastcall). The TWndMethod's object pointer is then stored in the EAX register (the first parameter of __fastcall). Finally, a CALL to the TWndMethod's implementation code is made. The class method is now running at this point!

Upon exit of the class method, execution returns to StdWndProc(). It then copies the TMessage's Result value to the EAX register (the return value) and pops the TMessage from the stack. At this point, the call stack is back to the same as when StdWndProc() was first entered, so the standard stack frame is cleaned up (again, handled by the compiler) and execution jumps to the original return address of the caller that invoked the proxy. The OS has just called a class method without ever realizing it!

Conclusions

In Part III of this series, I will introduce the new proxy system I have designed, and show examples of its use in C++ code. I am taking the MakeObjectInstance() system to a whole new level that has never before been explored by Borland development tool users. There are some really neat tricks available now for API users to take advantage of, so stay tuned.

Contact Remy at remy@lebeausoftware.org.