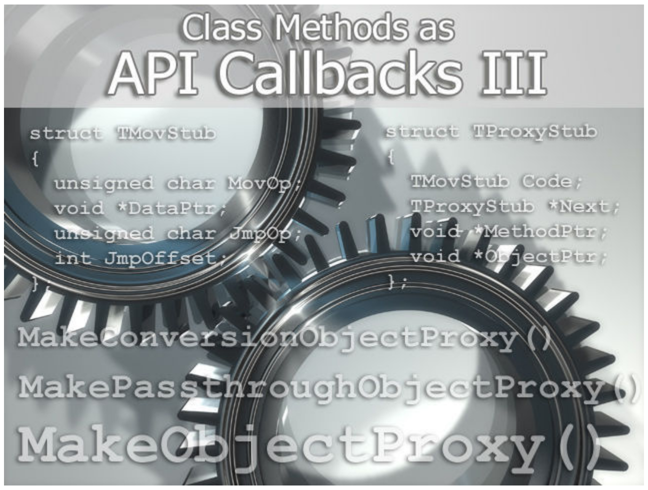
Using Class Methods as API Callbacks, Part III

By Remy Lebeau

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



For several months now, I have been working on a new set of proxy code that abstracts out Borland's `MakeObjectInstance()` proxy architecture so it can be used with any non-static class method of any signature, even virtual methods. The key to this is the invocation wrapper function that the proxy uses to invoke a class method. The wrapper function is now a user-defined value on a per-proxy basis. This way, you can custom tailor your own proxies if the ones I provide do not suit your needs.

Foreword

At the time of this writing, my new code is not quite ready for release yet. I am still working out some technical issues with it. So there will be a Part IV added to this series, which will provide the code and focus on demonstrating how to use it in your projects.

Optimizations and thread safety

In Part II of this series, I mentioned that Borland optimizes its memory usage inside of `MakeObjectInstance()`, and that I would not be using the same optimization in my code. In a nutshell, the proxy created by `MakeObjectInstance()` is relatively small in size and is fixed-length. When allocating executable memory, the smallest amount that can be allocated at a time is 4K, so `MakeObjectInstance()` fills each memory block with as many proxy instances as it can fit and then reuses them as needed. This cuts down on the number of allocations needed during the process's lifetime.

Since then, after a few design changes in my proxy stub, I ended up implementing the same optimization in my code after all. However, I have taken it a couple of steps further.

A major limitation of `MakeObjectInstance()` is that it is not thread-safe. So I have added an optional `CRITICAL_SECTION` around my memory blocks. It is not enabled by default. If you plan to use proxies in multiple threads, then you can define `OBJPROXY_MULTITHREADED` in your project options to enable it.

`MakeObjectInstance()` also never frees memory pages it allocates. I have implemented a `CompactObjectProxyMem()` function to release any memory blocks that are cached but no longer in use. If you use proxies for short periods, you can use this to free available memory if you know you won't need it anymore.

Introducing the new proxy functions

I have made several types of passthrough proxies and conversion proxies. They support the standardized `__cdecl` and `__stdcall` calling conventions that most C/C++ compilers implement, as well as both Borland's and Microsoft's `__fastcall` calling conventions (Borland supports Microsoft's `__fastcall` convention via its own `__msfastcall` compiler extension). The core of my new proxy system is the `MakeObjectProxy()` function. It allocates the actual proxy stub. Two other functions—`MakePassthroughObjectProxy()` and `MakeConversionObjectProxy()`—call `MakeObjectProxy()` internally. Refer to **Listing A** for declarations.

MakeObjectProxy()

As input, `MakeObjectProxy()` accepts an object pointer, the memory address of the class method's implementation code, the memory address of the wrapper function that will invoke the class method, and the CPU register used to pass the proxy's data to the wrapper function. Most of my proxies use the `ECX` register to pass proxy data to the invocation wrapper function, just like Borland's `MakeObjectInstance()` proxy does. However, some proxies need to use `ECX` for other purposes, so the CPU register is user-defined for flexibility. This is especially useful if you want to implement your own custom proxies.

MakePassthroughObjectProxy()

`MakePassthroughObjectProxy()` creates a pass-through proxy. This type of proxy is used when the source and destination signature types exactly match each other, other than the omission of the hidden `this` parameter from the source signature type.

A pass-through proxy is very flexible because it preserves the caller's parameters on the call stack and registers, allowing the class method to use them as-is. This is especially important for `__cdecl` functions that use variable argument lists, and `__fastcall` functions that use stack-based parameters. No special logic has to be implemented to handle parameters. The only modification performed is to inject the target object's `this` pointer into the class method invocation.

MakeConversionObjectProxy()

`MakeConversionObjectProxy()` creates a conversion proxy. This type of proxy is used when the source and destination signature types differ only in calling conventions, but otherwise match each other in parameters and return types. Extra work is needed to manipulate the call stack and registers of the source type to operate within the semantics of the destination type. I have provided conversion proxies that handle conversions between the four supported calling conventions.

MakeWndMethodObjectProxy()

For custom proxies, you need to call `MakeObjectProxy()` directly, passing your own wrapper functions to it. The VCL's `MakeObjectInstance()` function is a good example of such a proxy. Not only does it convert from `__stdcall` to Borland's `__fastcall`, but it also manipulates the call stack to accept four input parameters from the OS and put them into a single `TMessage` structure that is then passed to any class method that matches Borland's `TWndMethod` signature. I have included a `MakeWndMethodObjectProxy()` function as a direct replacement to the `MakeObjectInstance()` function to demonstrate this concept within my proxy system.

Listing A: Proxy creation functions

```
enum OBJPROXY_CONVENTION
{
  OBJPROXY_CDECL,
  OBJPROXY_STDCALL,
  #if defined(
    OBJPROXY_SUPPORTS_BORLAND_FASTCALL)
  OBJPROXY_BORLAND_FASTCALL_0,
  OBJPROXY_BORLAND_FASTCALL_1,
  OBJPROXY_BORLAND_FASTCALL_2,
  #endif
  #if defined(
    OBJPROXY_SUPPORTS_MICROSOFT_FASTCALL)
  OBJPROXY_MICROSOFT_FASTCALL_0,
  OBJPROXY_MICROSOFT_FASTCALL_1,
  OBJPROXY_MICROSOFT_FASTCALL_2,
  #endif
  OBJPROXY_MAX_CONVENTIONS
};

enum OBJPROXY_CPUREGISTER
{
  OBJPROXY_EAX,
  OBJPROXY_EBX,
  OBJPROXY_ECX,
  OBJPROXY_EDX,
  OBJPROXY_MAX_REGISTERS
};

void* MakeObjectProxy(void *Object,
  void *Method, void *ProxyCode,
  OBJPROXY_CPUREG CPURegister =
    OBJPROXY_ECX);
void FreeObjectProxy(void *Proxy);
void CompactObjectProxyMem(void);
void* MakePassthroughObjectProxy(
  void *Object, void *Method,
  OBJPROXY_CONVENTION Convention);
void* MakeConversionObjectProxy(
  void *Object, void *Method,
  OBJPROXY_CONVENTION SrcConvention,
  OBJPROXY_CONVENTION DestConvention);
```

Template helpers

You may notice that my functions are expecting pointer-to-class-method addresses to be passed as `void*` pointers. C++ does not allow a pointer-to-class-method to be converted to a `void*` (the compiler will report errors if you try), so I have provided extra wrappers around the functions using C++ templates to help get around this issue.

Looking at the new proxies

I have implemented many ready-to-use wrapper functions for invoking class methods. Refer to **Table 1** for the supported combinations.

Altogether, there are approximately two-dozen proxies implemented. There are too many details to explain exactly how each individual one is implemented. The source code I will provide in the next part of this series will be fully commented, and will include examples of how to use them in your own code.

The new proxy stub

The new proxy stub I have implemented differs from the proxy stub that the `MakeObjectInstance()` function uses. The following code shows the new proxy stub in C++ terms:

```
#pragma pack(push, 1)
struct TMovStub
{
   unsigned char MovOp;
   void *DataPtr;
   unsigned char JmpOp;
   int JmpOffset;
};

struct TProxyStub
{
   TMovStub Code;
   TProxyStub *Next;
   void *MethodPtr;
   void *ObjectPtr;
};
#pragma pack(pop)
```

If you recall from Part II of this article series, Borland's proxy stub uses the x86 Assembly `CALL`, `POP ECX`, and `JMP` instructions to assign the target method pointer to the `ECX` register before jumping into a proxy wrapper function.

As you can see from this code, I use a single `MOV E?X` instruction instead of Borland's `CALL/POP` pair (where "?" can be `A`, `B`, `C`, or `D`, depending on the value of the `CPURegister` parameter of `MakeObjectProxy()`). The effect is the same—upon entering a wrapper function, the `E?X` register points to the stub's `MethodPtr` member, just like Borland's proxy does—but the logic is more direct, requires fewer instruction bytes, and does not involve pushing/popping any values to/from the stack. I have also restructured the proxy stub to place all data members at the end of the stub, rather than in the middle as Borland does.

Table 1: *Prewritten proxy wrappers.*

Passthrough proxies	Conversion proxies
`__cdecl to __cdecl`	`__cdecl to __stdcall`
`__stdcall to __stdcall`	`__cdecl to __fastcall`
`__fastcall to __fastcall`	`__cdecl to __msfastcall`
`__msfastcall to __msfastcall`	`__stdcall to __cdecl`
	`__stdcall to __fastcall`
	`__stdcall to __msfastcall`
	`__fastcall to __cdecl`
	`__fastcall to __stdcall`
	`__fastcall to __msfastcall`
	`__msfastcall to __cdecl`
	`__msfastcall to __stdcall`
	`__msfastcall to __fastcall`

A note about __cdecl proxies

The `__stdcall` and `__fastcall` calling conventions both require the called method to pop parameters from the stack when exiting. However, the `__cdecl` calling convention requires the caller to do the popping instead. This causes a small problem for my `__cdecl` conversion proxies. The caller's return address cannot remain on the stack while the proxy calls the class method, otherwise any parameter(s) passed in will be at the wrong offset(s). However, the return address, and sometimes the CPU's ESP register, must be remembered so the proxy knows where to return to when it is finished with its work, and to account for differences in how different calling conventions manage the stack. At the time of this writing, these values are stored in a temporary per-thread memory block that is allocated and de-allocated for each proxy call. So these proxies have slightly higher runtime overhead. In the future, I may re-write the code to reduce that overhead by caching the memory blocks.

By using temporary memory blocks, this presents another problem. Since a new memory block is being allocated, it has to be freed after the called method exits. However, the stack is not available for storing extra data, so normal `__try/__finally` semantics cannot be used to ensure cleanup is always performed. So please make sure that any class methods you implement for use with effected proxy types *do not* throw exceptions back into the proxy code! Otherwise, memory will be leaked, and stack corruption may occur. In the next part of this series, after the code has been solidified, I will list all of the specific proxy types that are affected.

A note about __fastcall proxies

If you look closer at the `OBJPROXY_CONVENTION` enum, you will notice special handling for `__fastcall` and `__msfastcall` proxies. Both conventions use the stack and CPU registers for passing parameters, depending on their byte sizes and relative positions to each other. This is different from `__cdecl` and `__stdcall`, which use the stack exclusively instead. As such, this requires some extra handling in `__fastcall` and `__msfastcall` proxies, and thus has some limitations. The suffix for those `enum` values denotes the number of CPU registers, not the number of parameters, which are available for a proxy wrapper function to use for receiving 32-bit parameters.

Even though `__fastcall` can use up to three CPU registers to pass parameters around—`EAX`, `EDX`, and `ECX`—only two of them can be used at most, for the following reasons:

- For all `__fastcall` passthrough proxies, and for conversion proxies that specify `__fastcall` as the destination calling convention, the EAX register is used for the object's `this` pointer, so only the EDX and ECX registers are available for passing parameters to the class method. The caller could use the EAX register when passing in three parameters, but the proxy would overwrite it.
- For conversion proxies that specify `__fastcall` as the source calling convention, there is no way for my wrapper functions to know whether any stack-based parameters are being used, and in what order they appear in relation to the register-based parameters. Thus, those proxies are limited to 32-bit parameters so that only CPU registers are used.

The `__msfastcall` convention, on the other hand, uses only two CPU registers for passing 32-bit parameters, instead of three. So `__msfastcall` conversion proxies are that much more limited.

The example code I will provide in the next part of this series will show how to use custom proxy wrapper functions to get around these limitations when needed.

Conclusions

Now that the underlying proxy logic has been abstracted out, you can write your own custom proxies to suit your particular needs. Of course, you need to have an understanding of x86 Assembly and how compilers use it to implement calling conventions.

As I continue to develop this new proxy system for my own use, I am hopeful that I will eventually be able to further adapt it to other compilers and platforms. Right now, it is primarily geared towards Borland compilers, with minimal support for Microsoft compilers.

Contact Remy at remy@lebeausoftware.org.