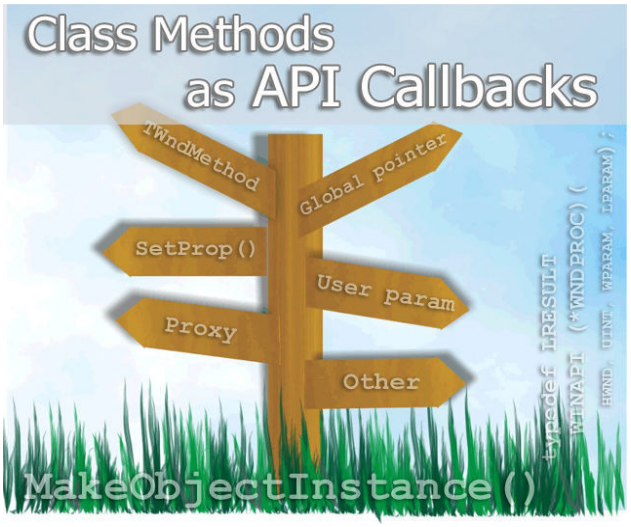
Using Class Methods as API Callbacks, Part I

By Remy Lebeau

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



Several past C++Builder Developer's Journal articles have touched on the general topics of what function pointers and closures are, how to implement events in classes, and how to use API callbacks in general [1]-[3].

Although those articles are good by themselves, they describe each of their topics separately and individually, but do not describe what to do in situations where they need to cross paths with each other. One question in particular that I have come across again and again over the years, in many different discussion forums, merges all three topics together very simply:

> *"How do I use a non-static method of a class as a callback for an API that expects a non-class function pointer instead?"*

That is a very good question, and one that does not always have an elegant answer, depending on the particular API being used.

In this three-part series, I will attempt to cover several different ways that this question can be answered, leading up to an introduction of something new I have developed to enhance a technique Borland has already pioneered in its VCL framework.

Accessing object members

In an object-oriented environment, such as C++, most developers want to write their API callbacks so that they can access data members and/or methods of one or more objects being using with the API. How that can be done, however, depends on the particular design of the API.

Using a user-defined parameter

A well-designed API should provide an option of passing a user-defined value, usually a `void*` pointer or a pointer-sized integer, as part of the callback. In that situation, the target object's `this` pointer can be passed as the value so the callback can access the object without any special tricks needed. Refer to **Listing A** for an example. This is the most desirable approach. It is clean, straightforward, and flexible. It can easily adapt on a per-call basis.

The VCL uses this approach internally when possible, as many Win32 API functions have `LPARAM` or `LPVOID` parameters that can be used. The `TThread` class, for instance, passes its `this` pointer as the lpParameter value of the `CreateThread()` function, allowing a private callback function to call `TThread`'s `Execute()` and `DoTerminate()` methods, similar to the example in **Listing A**.

Using a global pointer

Unfortunately, not all APIs provide such an option. So what can be done then? For most people, using a

Listing A: User-defined parameter

```
typedef void (*CallbackType)(void*, int);
void SomeAPIFunction(CallbackType, void*);

void MyCallback(void *UserData, int Value)
{
  TMyClass *myObject =
    (TMyClass*) UserData;
  myObject->DoSomethingWith(Value);
}

void TMyClass::SomeMethod()
{
  SomeAPIFunction(&MyCallback, this);
}
```

global pointer to the target object is usually the only way they know how to go. For example:

```
typedef void (*CallbackType)(int);
void SomeAPIFunction(CallbackType);

TMyClass *g_myObject;

void MyCallback(int Value)
{
  g_myObject->DoSomethingWith(Value);
}

void TMyClass::SomeMethod()
{
  g_myObject = this;
  SomeAPIFunction(&MyCallback);
}
```

Although this approach does work, one major limitation is that the callback can only operate on one object instance at a time, usually within the context of one thread at a time (if you want to use multiple threads, you may have to use TLS [thread local storage] to hold multiple global variables). You are responsible for making sure the global pointer remains valid while the callback is busy.

For projects with simple API usages, this is not so bad an approach (even though many developers tend to frown on the use of global variables if it can be avoided). However, this approach tends to fail miserably in more complex projects that use multiple objects and/or multiple threads with the same API. The more things that are involved, the messier the coding can become. I have hit this roadblock myself in more than one project.

Using a proxy

Recently, I started some work on a project that uses a particular third-party API (which I will not name) that I want to implement four different callbacks for. They access the same object instance, but at different times for different purposes. Due to time constraints, I used a global pointer, but it got me thinking about whether any alternative approach exists that I could use to accomplish the same results I want in a different manner later on when I have more time to fine-tune it. In fact, there is one option I want to discuss in more detail for the rest of this article series.

I spend a lot of time studying and browsing the VCL source code during the course of my daily activities. A little-known jewel that immediately came to mind is the VCL's `MakeObjectInstance()` and its associates:

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

void* __fastcall MakeObjectInstance(
  TWndMethod Method);

void __fastcall FreeObjectInstance(
  void* ObjectInstance);
```

In a nutshell, `MakeObjectInstance()` dynamically allocates a special proxy that allows a non-static class method with a signature matching `TWndMethod` to be used as a Win32 API WNDPROC window procedure callback with the following signature:

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

The proxy can be passed to the Win32 API `CreateWindow/Ex()` and `SetWindowLong()` functions to assign it as the window procedure for any `HWND`. When the OS then sends window messages to the `HWND`, the OS thinks it is calling a standalone non-class function, but the proxy stub is translating the calls into suitable invocations of the target class method – complete with passing the correct object instance as the necessary `this` pointer. As far as the OS is concerned, it does not have a clue about any object method being called, and the class method thinks it is being called normally like any normal call.

Every VCL control that is based on the `TWinControl` class uses `MakeObjectInstance()` internally to receive its window messages from the OS. So does any class that creates private `HWND`s using the VCL's `AllocateHWnd()` function (`TTimer`, for instance).

However, there are two downsides to `MakeObjectInstance()`. First, it is not thread-safe. It makes use of global resources that are not protected from concurrent access by multiple threads. As such, it can only be used safely in the context of the main thread. Second, it only works for `WNDPROC` callbacks specifically and no others.

WNDPROC's alternative to proxies

`WNDPROC` callbacks have an `HWND` parameter available, so there is an alternative approach that can be used without using a proxy. The object's `this` pointer can

be stored in the HWND directly, using either the SetProp() or SetWindowLong() function. TWinControl, for instance, uses SetProp() to store its this pointer into any HWNDs that it creates. When the window procedure is invoked, GetProp() is used to retrieve the object's this pointer so its members can be accessed. For example:

```
LRESULT WINAPI MyWndProc(HWND hWnd,
  UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  TMyClass *myObject = (TMyClass*)
    ::GetProp(hWnd, TEXT("MyObjectProp"));
  return myObject->HandleMsg(
    uMsg, wParam, lParam);
}

TMyClass::TMyClass()
{
  m_hWnd = ::CreateWindowEx(…);
  ::SetProp(m_hWnd, TEXT("MyObjectProp"),
    (HANDLE) this);
  m_DefProc = (WNDPROC)::SetWindowLong(
    m_hWnd, GWL_WNDPROC, (LONG)&MyWndProc);
}
```

This approach is similar to the example in **Listing A**, just with an extra level of indirection, as the callback is receiving a container (the HWND) that contains the actual object pointer.

Conclusions

The callbacks I want to implement in my project are not WNDPROC callbacks. In fact, the API I am using represents a worst-case scenario. Each callback has a completely different signature from the others, requiring me to write a different wrapper function for each class method that I want to call. I cannot use a common wrapper for all of them (well, not easily anyway, which I will discuss in Part 3 of this article). But more than that, none of the callbacks provide any kind of user-defined values, so all of the wrapper functions have to share a common global object pointer.

So how can MakeObjectInstance() help me in this situation? Well, by itself, it actually does not help me at all. To explain what I have come up with, you have to first understand how the proxy generated by MakeObjectInstance() actually does its work. I will explain that in Part II of this article. In Part III, I will then show how I have taken that knowledge to develop a new general-purpose proxy system that can be used in all kinds of different situations.

Before moving on, let me warn you that from this point on, a basic understanding of 32-bit x86 Assembly code is needed (I am not going to cover 64-bit at this time). Specifically, you should be familiar with the use of CPU registers and the stack. Make sure you read up on those topics if you need to. If you know how Borland compilers use x86 Assembly for class method calls, that would also be helpful, though I will give a brief overview of that as well.

Contact Remy at **remy@lebeausoftware.org**.

References

1. K. Reisdorph, "Using callbacks in DLLs," *C++Builder Dev. Journal*, **3** (2), 1999. http://www.bcbjournal.com/articles/vol3/9902/Using_callbacks_in_DLLs.htm
2. B. Knigge, "Understanding function pointers," *C++Builder Dev. Journal*, **4** (7), 2000. http://www.bcbjournal.com/articles/vol4/0007/Understanding_function_pointers.htm
3. D. Bridges, "Events and callback functions," *C++Builder Dev. Journal*, **5** (10), 2001. http://www.bcbjournal.com/articles/vol4/0007/Understanding_function_pointers.htm