

## Re: Calling a managed function from native code

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.vc/2008-02/msg00166.html>

---

- *From:* Tamas Demjen <[tdemjen@xxxxxxxxxx](mailto:tdemjen@xxxxxxxxxx)>
  - *Date:* Fri, 22 Feb 2008 15:59:30 -0800
- 

Bob Altman wrote:

Now you've introduced C++ code that may be running on top of the CLR (meaning that it must be JIT compiled the first time it's accessed) but which is isn't garbage collected (as you point out, things that behave like value classes).

Yes. I'm not sure if any other .NET language is capable of compiling such code, though. Certainly not C# and VB.NET. It's a C++/CLI thing.

1. If I compile with `/clr` and don't include any `#pragma` directives then the code compiles to IL which is JIT compiled and runs on top of the CLR.

Correct. With `#pragma managed` (default with `/clr`), even `std::vector`, `std::string`, `boost::shared_ptr` compile into unmanaged data, but JIT-ed code.

2. I export routines from my DLL using a `.def` file. If I include the name of a function that was compiled with `/clr` in my `.def` file then C++ creates what appears to the outside world to be a native function. But native code that calls into that function goes through a relatively expensive process called "thunking" to run the managed code.

Correct. A mixed mode DLL can export native C-style functions. This is very useful when you have to call fully managed .NET libraries from a native C++ application. Thunking is not nearly as expensive as a web service. Just make sure you don't switch context in the middle of a loop that runs a million iterations. I'm sure WinForms itself thunks into Win32 at a lower level.

3. (Now I start getting to the crux of my original question...) If I compile a function with `/clr`, and precede the function with `#pragma unmanaged`, then the code compiles to native code. No IL, no CLR, no JIT compiler, just good old fashioned machine code.

Correct. `#pragma unmanaged` is really native, no GC, not JIT.

## Re: Calling a managed function from native code

4. But, native code compiled with `/clr` can do things that (equally native) code compiled without `/clr` can not.

Yes, it can thunk into managed code.

It can statically allocate new managed class instances on the native stack.

Only if it's a C++ class. Not full CLR classes:

```
#pragma managed

class MC { };
ref class RC { };
value class VC { };

#pragma unmanaged

void test()
{
MC mc; // OK
RC rc; // error
VC vc; // error
}
```

The error is: "managed type of function cannot be used in an unmanaged function."

```
#pragma unmanaged
void test() {
// Not sure what I can do with a managed string,
// but I can allocate one...
System::String s;
}
```

No, not even in `#pragma managed`. You can't do that, `String` is not disposable, it doesn't have a destructor, it doesn't have a `Dispose` method. `String` is a special case that can't be used with stack syntax, it must be `gcnew`'ed. That's because `String` is immutable — once an instance is created, you can't modify it. Let's say, when you append to a `String`, a whole new object is created. `Strings` must be garbage collected, it's not a resource.

5. Also, native code compiled with `/clr` (and `#pragma unmanaged`) knows about managed namespaces. This is necessary when specifying a managed type as a template parameter (for `gcroot<>`, for example). This also lets you call managed, static functions directly from the unmanaged code.

## Re: Calling a managed function from native code

I doubt that static functions make a difference. Full CLR methods are not available from `#pragma unmanaged`, regardless if they're static or not:

```
#pragma managed

ref struct SC
{
    static void Do() { }
};

#pragma unmanaged

void test()
{
    SC::Do(); // error
}
```

6. I can call the static `GCHandle.Alloc` method from unmanaged code to instantiate a managed class on the managed heap and to get back a `GCHandle` object that I can treat like a `void*`. That is, I can make calls on the managed object by casting the `GCHandle` to a pointer to the managed class...

```
#pragma unmanaged
void test()
{
    System::Runtime::InteropServices::GCHandle::Alloc(nullptr);
}
```

I think the `gcroot` template is defined inside `#pragma managed`, that's why it can do such tricks.

7. Or, I can use `gcroot`, which is sexy because it has a template parameter that lets it behave like a strongly typed reference to the managed object.

You can use `gcroot` from `#pragma unmanaged` for the same reason you can use any other class:

```
#pragma managed
class C { };

#pragma unmanaged

void test()
{
    C c;
}
```

Re: Calling a managed function from native code

Tom

.