<http://www.ondotnet.com/pub/a/dotnet/2003/03/03/mcppp2.html>

**Intro to Managed C++, Part 2: Mixing Managed and Unmanaged Code**

by [Sam Gentile](http://www.ondotnet.com/pub/au/1070)  
03/03/2003

Welcome back! Last time around, in the [first article of the series](http://www.ondotnet.com/pub/a/dotnet/2003/01/13/intromcpp.html), I focused on what Managed C++ was, some of its advantages and roles, as well as scenarios in which it excelled. One of those scenarios is the focus of the second article of this series: *the ability to mix managed and unmanaged code in the same module*. This is an ability that is unique to Managed C++; no other Common Runtime Library (CLR) language possesses this capability. In this article, I will explore why this is important to you as a working developer, and how to make use of this capability.

**Target Audience**

A quick note: This article, and this series, assume that the reader is familiar with the basics of the .NET Framework, including the CLR, and has worked with managed languages, such as C# and VB.NET. The reader may or may not be a C++ programmer. It helps, certainly, but adventuresome programmers are certainly welcome!

**Review: Managed vs. Unmanaged Code**

Every day, Google sends at least a half-dozen hits to my weblog based on the search criteria "managed vs. unmanaged code." It is clear that this is still a point of confusion for many programmers new to .NET programming. This is a vital concept to understand, particularly in terms of this article. *I can't put it any simpler than this: unmanaged code is everything you have been programming for years, before .NET.* Unmanaged, or "native" code, includes VB6, COM, Win32, native C++, and so forth. It is code that predated .NET and therefore, has absolutely no knowledge of .NET and cannot directly make use of any managed facilities. Good so far?

Well, then, managed code is code written for the .NET runtime or CLR. More specifically, managed code is "managed" because the code is under the control of the CLR. .NET or CLR compilers are required to emit Metadata and CIL (Common Intermediate Language; Microsoft's form is MSIL). CLR components use CIL for their representation. CIL can be thought of as a higher-level processor-neutral instruction set. It is higher-level because although it uses a stack-based virtual machine, the opcodes refer to metadata and higher-level OO-like operations. The metadata fully describes types according to the Common Type Specification (CTS). Because of the notion of strong types, the CLR can provide a "managed execution environment." What does that mean? It means that the CLR knows everything about a type and can provide services such as lifetime management (garbage collection), security, reflection, and more. These services can only be provided to code that is written to target the .NET Framework including the use of CTS types, and is compiled to managed code.

It is extremely important to note that the CLR never executes CIL directly. CIL is always translated into native machine code before it is executed. Usually, this is done through Just-In-Time (JIT) compilation, although it is possible to use a tool called NGEN to pre-compile CIL into the assembly.

There are many obvious advantages to using managed code, not the least of which is shifting the burden of explicit management of memory to the runtime.

So, if managed code is so awesome, why do we care about mixing managed and unmanaged code?

**Reasons for Mixing Managed and Unmanaged Code**

*Quite simply, managed code is not the right solution for every need, yet.* Unmanaged code is almost always faster than managed code, because it does not have any of the overhead associated with the CLR, such as garbage collection, run-time type checking, and reference checking. This is not important for many applications, such as GUI applications or n-tier applications where either the user or network latency dwarfs any other issues of performance. In addition, the JIT compilers in .NET are excellent and there are cases where JITed code approaches and sometimes exceeds the performance of native applications. But there are classes of applications such as system utilities, games, and so forth that require both the speed and determinism of native code.

In addition, although the benefits of the CLR are absolutely compelling, there are literally billions of lines of native C++ code that work perfectly well and will stay in operation for years to come. Indeed, even the Win32 API itself and the Shell functions are all unmanaged code, and will be for years to come. Every "port" involves work and risk; the move to managed code is no different. There are training issues, risks, and work involved. All of these issues have to be considered when deciding whether to rewrite a working piece of native code. These issues become significant when there is a large body of existing code and a programming staff that has knowledge and expertise in unmanaged languages such as C++. Make no mistake about it: programming for the CLR is vastly different than Win32 and COM programming. To a large extent, this is retraining to "let go and let the runtime," but that's not the only issue. To program even marginally well under the .NET Framework requires some knowledge of how the CLR works, in the sense of dealing with a non-determistic style of operation as well as learning how to use the vast Base Class Libraries (BCL). This does not happen overnight. The typical cycle I have seen is at least six months. For many shops facing shrinking IT staffs and budgets, as well as large amounts of native code, this issue is huge, and prohibits whole-scale "rewriting."

So, to summarize, the following situations would perhaps require a mixing of unmanaged and managed code:

* Allowing a developer to keep performance-critical portions of the code in native code.
* A need for deterministic and predictable code.
* Seamless access to all unmanaged libraries, such as DLLs, statically-linked libraries, COM objects, template libraries, and more.
* Leveraging existing investments in C++ programming skills and legacy C++ code.
* Porting unmanaged code to .NET: MC++ allows you to take existing unmanaged code and compile it to managed code (with the /clr compiler switch and IJW).
* The ability to port code at one's own rate rather than rewrite all at once.
* Provides the easiest way to add .NET support to your existing native C++ Windows applications, by allowing your to bridge the gap between the two environments with as little work on your behalf as possible, and with the lowest performance penalty.
* MC++ is the only language currently that allows some form of multi-paradigm design and development with full support for generic programming and templates. This can lead to more options and better designs and implementations.

**It Just Works! (IJW)**

One of the most amazing engineering feats in the whole .NET effort, and one that doesn't get its due justice, is something known as It Just Works (IJW)! Unlike the C# team, which was starting from scratch, the Managed C++ team had an existing, standardized language to deal with: C++. They needed to find some mechanism to take that existing unmanaged C++ code, with all of its features, and make it compile and run on the CLR, as CIL. This involved quite a bit of work. I will dive into the details of IJW in my next article, but for the purposes of this article, imagine the following typical scenario. Take any existing C++ source code, sprinkle in some old-style printf() functions, add some MFC, add some STL, and recompile using some sort of managed "switch" in the C++ compiler. Would you expect such code to compile and work under .NET? Well, it largely does (there are a very small set of exceptions to the rule)!

The secret sauce, if you will, is IJW. The compiler "switch" to make this happen is the /clr switch. The documentation has some vague notion that this switch is "to enable Managed Extensions for C++." While this is true, it does not begin to even describe what is going on with the /clr switch and IJW. This switch allows you to take your native C++ code and (mostly) "make it" managed. The output of code compiled with the /clr switch is MSIL. The amazing part of this is that the native C++ code has no clue about the CLR, doesn't have metadata, doesn't have any managed types, but yet, you are able to recompile to managed code and run, without rewriting or "porting" a single line of code! This is all being done by the compiler; It Just Works!

The key idea here, and one we will explore at the IL level in the next article, is that although native classes are compiled to MSIL, they are not managed. They are compiled as \_\_nogcclasses, signifying non-managed classes. Why? Well, there are several reasons. The first of these is that the C++ object model is totally different than that of the CLR. This means that not every native C++ class can become a managed one. Remember that the CLS and CLR do not support the notion of multiple inheritance, for instance. The second reason has to do with memory allocation. In the CLR, some managed types can only be created on the GC Heap (Reference Types), and some only on the stack, and with some restrictions, on the C++ or global heap (value types). Creating a managed type on both the stack or heap, prohibits making a GC class or value type, without causing significant limitations in functionality.

But what IJW does do, for the majority of C++ code that can be compiled with /clr, is provide a "transition thunk" from unmanaged to managed code. Again, we will explore the details of what this looks like in the next article. For the purposes of this article, it is sufficient to know that the /clr flag and IJW allow you to recompile native C++ code to MSIL.

IJW does allow the ability of these \_\_nogc types to call into managed code and use managed types. It is also possible to embed a pointer to an unmanaged class in a managed class, most of the time. Before we look at these situations, we need to first look at the new pragma directives: managed and unmanaged.

**Mixing Managed and Unmanaged C++ via Pragma Directives**

One of the biggest advantages of MC++ is the ability to mix managed and unmanaged C++ code in the same executable in the same source file. This is used together with the /clrcompiler option I just discussed. As I just mentioned in the previous section, IJW will allow the code to compile and run under a managed environment. This approach allows you to incrementally port your code to managed at your own pace. The other piece required is some way to mark which parts of a source file are managed and which are unmanaged. That's what the #pragma managed and #pragma unmanaged directives are for. Let's look at an example.

#using <mscorlib.dll>

using namespace System;

#include "stdio.h"

void ManagedFunction()

{

printf("Hello, I'm managed in this section\n");

}

#pragma unmanaged

UnmanagedFunction()

{

printf("Hello, I am unmanaged through the wonder of IJW!\n");

ManagedFunction();

}

#pragma managed

int main()

{

UnmanagedFunction();

return 0;

}

So what's going on in this program? This is a Managed C++ program that is compiled with /clr. The pragma managed directive tells the compiler to generate managed code, andpragma unmanaged tells it to generate unmanaged native code. When compiled with /clr, the absence of any pragma defaults to managed code. Thus, the function ManagedFunction()gets compiled as a \_\_nogc class, and the call to printf happens via IJW. The pragma unmanaged directive tells the compiler to compile UnmanagedFunction() as unmanaged native code. Then, pragma managed switches things back to managed compilation again. So, we have a transition in this program from managed to unmanaged to managed.

**OK, So How Is This Useful?**

Well, other than being interesting, how is this useful? Well, just think of the billions of lines of existing unmanaged C++ code. Do you really think that it will get thrown away, and rewritten using C#, overnight? Not in any companies that I know of. What Managed C++ allows you and your company to do is selectively mix managed and unmanaged code together in the same module. Want to change one function to managed at a time? You can do it. Want to keep a time-critical piece of code in native code? You can do it. This approach allows maximum flexibility in your choices. You don't have to stop everything for six months while you rewrite everything in C#. You have the ability, using your existing skill set, to move your code base over to managed code, at your speed.

|  |  |
| --- | --- |
|  |  |

**Making Your Classes Managed**

As I have stated previously, when native C++ code is recompiled with /clr, the classes don't automatically become managed and are marked \_\_nogc for the reasons I cited. If your class does meet the requirements of the CLR, however, you can make your class managed by marking it with the \_\_gc modifier to indicate that it is a garbage-collected class, or the\_\_value modifier to indicate that it is a CTS value type.

**Embedding Unmanaged Types in Managed Classes**

What if your class cannot be made managed by adding the \_\_gc or \_\_value modifiers? You may have code that uses templates or multiple inheritance. You may have code that uses inline assembly to reuse the functionality from something you would usually inherit from that class. For obvious reasons outlined earlier, you cannot inherit a managed class from an unmanaged one, and vice versa. So what do you do if you want to reuse the functionality? For problems like this, the general solution is to either "aggregate" or "embed a pointer." Without going into a lot of low-level details, aggregating an unmanaged class within a managed one causes a lot of problems, and the compiler cannot convert an object from the fcheap to a non-GC reference. The way to do this is to embed a pointer to an unmanaged type within the managed class. It looks something like this (a contrived example):

#using <mscorlib.dll>

using namespace System;

#include <string>

\_\_nogc class Container

{

int value\_;

public:

Container() : value\_(0) {}

void SetValue(int \*val) { value\_ = \*val;}

const int& GetValue() { return value\_; }

};

\_\_gc class ManagedContainer

{

Container\* pContainer;

public:

ManagedContainer()

{

pContainer = new Container();

}

void SetValue(int val)

{

int someValue = val;

pContainer->SetValue(&someValue);

}

~ManagedContainer()

{

delete pContainer;

}

};

void main()

{

ManagedContainer \*mc = new ManagedContainer();

int someValue = 42;

mc->SetValue(someValue);

System::Console::WriteLine("The value is ",

someValue.ToString());

}

In this solution, I create an embedded pointer to the the unmanaged class type Container inside ManagedContainer and control it explicitly. Will this code work? Perhaps sometimes, but there is a big problem with the code as it stands.

**Pinning Pointers**

The problem is that the CLR, in its GC, moves object references around. This is not a problem until such a reference is passed to an unmanaged function or used in an unmanaged object. The CLR has no way to keep track of the reference once it transitions to unmanaged code. So in order to prevent the value from getting corrupted, we need to "pin" the pointer using the \_\_pin keyword:

#using <mscorlib.dll>

using namespace System;

#include <string>

\_\_nogc class Container

{

int value\_;

public:

Container() : value\_(0) {}

void SetValue(int \*val) { value\_ = \*val;}

const int& GetValue() { return value\_; }

};

\_\_gc class ManagedContainer

{

Container\* pContainer;

public:

ManagedContainer()

{

pContainer = new Container();

}

void SetValue(int val)

{

int someValue = val;

int \_\_pin\* pinnedInt = &someValue;

pContainer->SetValue(pinnedInt);

}

~ManagedContainer()

{

delete pContainer;

}

};

void main()

{

ManagedContainer \*mc = new ManagedContainer();

int someValue = 42;

mc->SetValue(someValue);

System::Console::WriteLine("The value is ",

someValue.ToString());

}

This type of approach leads to the ability to wrap your C++ code with managed wrappers, enabling your C++ code to be used from other managed languages like C# and VB.NET. I will explore more of this in detail, in the next article of this series.

Now, what if we want to go the other way? That is, use managed types from unmanaged code?

**Using Managed Types from Unmanaged Code**

Managed types cannot directly be used from unmanaged types. This is again related to the fact that the CLR must keep track of object references to implement garbage collection. What if we did want to use an object reference in an unmanaged type? The CLR provides a type, System::Runtime::InteropServices::GCHandle, that treats object references as integers from unmanaged code. The technique for using this function is quite simple: call GCHandle::Alloc() to generate a handle and GCHandle::Free() to free it.

However, this pattern can get quite messy, so there is a better way. In the file *gcroot.h*, the VC++ team has provided a smart pointer, called gcroot<>, to simplify the use of GCHandlein unmanaged types. With this template, we are able to use a System::String in an unmanaged class, like so:

#using <mscorlib.dll>

#include <vcclr.h>

using namespace System;

class CppClass {

public:

gcroot<String\*> str; // can use str as if it were String\*

CppClass() {}

};

int main() {

CppClass c;

c.str = new String("hello");

Console::WriteLine( c.str ); // no cast required

}

**What's Next?**

In this article, I have only scratched the surface of what's possible. The next step is to further look at what IJW accomplishes and where it falls short, and how to make functions managed, how to make your data managed, and how to write managed wrappers around unmanaged functions.

[*Sam Gentile*](http://www.ondotnet.com/pub/au/1070)*is a well-known .NET consultant and is currently working with a large firm, using Visual C++ .NET 2003 to develop both unmanaged and managed C++ applications.*