Marshaling

Occasionally you may need to access functionality that is exposed in a native DLL. This is not common, because the CLR is quite extensive, but it is a good idea to understand what happens behind the scenes. Also if you use .NET Reflector to browse the source code for the framework, you will find this crop up from time to time, and you need to know what is going on.

Marshaling can definitely have a memory impact. The MSDN documentation makes it clear when it states that the `Marshal` class "*provides a collection of methods for allocating unmanaged memory, copying unmanaged memory blocks, and converting managed to unmanaged types, as well as other miscellaneous methods used when interacting with unmanaged code.*" (see HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/SYSTEM.RUNTIME.INTEROPSERVICES.MARSHAL.ASPX).

Unmanaged memory means no GC which, in turn, means that anything that you allocate, you must also explicitly de-allocate. When you start working with unmanaged memory, you are explicitly going out on a limb and leaving the safety net that .NET provides. Naturally, this is not for the faint of heart, and any code that uses these techniques must be carefully monitored, not just for memory leaks but also for type safety.

Some data types are interoperable and will work the same in an unmanaged world as they do in the managed world, while others do not, and will need to be explicitly converted. The basic integral types are interoperable, and pointers are at least interoperable in unsafe mode. It may be surprising to find out that some other common types, such as `Char` , are not interoperable. In managed code, a char will be a 2-byte Unicode character, but in unmanaged code it will be a 1-byte ASCII character. `Bool` is another tricky type; in unmanaged code, a `bool` is typically an integral value with 0 meaning `false` and anything else being `true.` Any integer value can server in the place of a `bool` . For instance, the code in Listings 6.7 and 6.8 is valid in C++, but not in C#.

```
int x = list.Count;
if (x) // Invalid inC#
{
    Console.WriteLine("The list is not empty.");
}
```

Listing 6.7: Booleans are treated differently in C++.

```
int x = list.Count;
if (x > 0) // Valid in C#
{
    Console.WriteLine("The list is not empty.");
}
```

Listing 6.8: Integers are not booleans in C#.

When you call unmanaged methods that expect or return a `bool` , you should use `int` instead, and when you call unmanaged methods that expect or return a pointer (*), you should use `IntPtr` instead. This will limit the scope of "unsafe" operations.

Interoperability issues aside, there are potentially significant memory issues to pay attention to. It is common to have to allocate unmanaged memory to be referenced and manipulated by the target of your DLL call. Naturally, this unmanaged memory is outside the oversight of the GC, and so it is up to the humble developer to limit its scope and ensure that the memory is released appropriately.

This sounds like a great place for the disposable pattern that we have previously discussed but, without some clever reworking on your part, this will not work as expected. For starters, there is no single object that would need to implement the

IDisposable interface. We don't need to dispose of the Marshal class, but rather the memory allocated by this class. While you could create a memory wrapper and have it host the allocated memory and implement the IDisposable interface, you can also easily revert to a basic try-catch-finally block, as shown in Listing 6.9.

```
var size = 32*Marshal.SizeOf(new byte());
var allocatedMemory = IntPtr.Zero;
var stringPointer = IntPtr.Zero;
try
{
   allocatedMemory = Marshal.AllocHGlobal(size);
   stringPointer =
   Marshal.StringToHGlobalAnsi(stringValue);
   // Make your dll method calls
}
catch
{
   // log the exception as needed
   throw
}
finally
{
   if (allocatedMemory != IntPtr.Zero)
   Marshal.FreeHGlobal(allocatedMemory);
   if (stringPointer != IntPtr.Zero )
   Marshal.FreeHGlobal(stringPointer );
}
```

Listing 6.9: Explicitly free unmanaged memory that has been allocated.

In this example, you would effectively leak 32 bytes of data every time this method is called, in addition to however many bytes were needed to hold the stringValue. This could easily become a memory flood instead of just a leak.

As a rule, I'd recommend that you isolate code that allocates or uses marshaled memory to a central location. This type of logic should not be spread throughout your code base, but rather isolated to proxy classes providing access to the desired unmanaged functionality. Additionally, this code should be routinely monitored to ensure that all best practices are consistently followed, because these code constructs are easily abused.