Chapter 24. Native and COM Interoperability

This chapter describes how to integrate with native (unmanaged) Dynamic-Link Libraries (DLLs) and Component Object Model (COM) components. Unless otherwise stated, the types mentioned in this chapter exist in either the `System` or the `System.Runtime.InteropServices` namespace.

Calling into Native DLLs

P/Invoke, short for *Platform Invocation Services*, allows you to access functions, structs, and callbacks in unmanaged DLLs (*shared libraries* on Unix).

For example, consider the `MessageBox` function, defined in the Windows DLL *user32.dll*, as follows:

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

You can call this function directly by declaring a static method of the same name, applying the `extern` keyword, and adding the `DllImport` attribute:

```
using System;
using System.Runtime.InteropServices;

MessageBox (IntPtr.Zero,
            "Please do not press this again.", "Attention", 0);

[DllImport("user32.dll")]
static extern int MessageBox (IntPtr hWnd, string text, string caption,
                              int type);
```

The `MessageBox` classes in the `System.Windows` and `System.Windows.Forms` namespaces themselves call similar unmanaged methods.

Here's a `DllImport` example for Ubuntu Linux:

```
Console.WriteLine ($"User ID: {getuid()}");

[DllImport("libc")]
static extern uint getuid();
```

The CLR includes a marshaler that knows how to convert parameters and return values between .NET types and unmanaged types. In the Windows example, the `int` parameters translate directly to four-byte integers that the function expects, and the string parameters are converted into null-terminated arrays of Unicode characters (encoded in UTF-16). `IntPtr` is a struct designed to encapsulate an unmanaged handle; it's 32 bits wide on 32-bit platforms and 64 bits wide on 64-bit platforms. A similar translation happens on Unix. (From C# 9, you can also use the `nint` type, which maps to `IntPtr`.)

Type and Parameter Marshaling

Marshaling Common Types

On the unmanaged side, there can be more than one way to represent a given data type. A string, for instance, can contain single-byte ANSI characters or UTF-16 Unicode characters, and can be length prefixed, null terminated, or of fixed length. With the `MarshalAs` attribute, you can specify to the CLR marshaler the variation in use, so it can provide the correct translation. Here's an example:

```
[DllImport("...")]
static extern int Foo ( [MarshalAs (UnmanagedType.LPStr)] string s );
```

The `UnmanagedType` enumeration includes all the Win32 and COM types that the marshaler understands. In this case, the marshaler was told to translate to `LPStr`, which is a null-terminated single-byte ANSI string.

On the .NET side, you also have some choice as to what data type to use. Unmanaged handles, for instance, can map to `IntPtr`, `int`, `uint`, `long`, or `ulong`.

> **NOTE**
>
> Most unmanaged handles encapsulate an address or pointer and so must be mapped to `IntPtr` for compatibility with both 32- and 64-bit operating systems. A typical example is HWND.

Quite often with Win32 and POSIX functions, you come across an integer parameter that accepts a set of constants, defined in a C++ header file such as *WinUser.h*. Rather than defining these as simple C# constants, you can define them within an enum instead. Using an enum can make for tidier code as well as increase static type safety. We provide an example in "Shared Memory".

> **NOTE**
>
> When installing Microsoft Visual Studio, be sure to install the C++ header files—even if you choose nothing else in the C++ category. This is where all the native Win32 constants are defined. You can then locate all header files by searching for **.h* in the Visual Studio program directory.
>
> On Unix, the POSIX standard defines names of constants, but individual implementations of POSIX-compliant Unix systems may assign different numeric values to these constants. You must use the correct numeric value for your operating system of choice. Similarly, POSIX defines a standard for structs used in interop calls. The ordering of fields in the struct is not fixed by the standard, and a Unix implementation might add additional fields. C++ header files defining functions and types are often installed in */usr/include* or */usr/local/include*.

Receiving strings from unmanaged code back to .NET requires that some memory management take place. The marshaler automatically performs this

work if you declare the external method with a `StringBuilder` rather than a `string`, as follows:

```
StringBuilder s = new StringBuilder (256);
GetWindowsDirectory (s, 256);
Console.WriteLine (s);

[DllImport("kernel32.dll")]
static extern int GetWindowsDirectory (StringBuilder sb, int maxChars);
```

On Unix, it works similarly. The following calls `getcwd` to return the current directory:

```
var sb = new StringBuilder (256);
Console.WriteLine (getcwd (sb, sb.Capacity));

[DllImport("libc")]
static extern string getcwd (StringBuilder buf, int size);
```

Although `StringBuilder` is convenient to use, it's somewhat inefficient in that the CLR must perform additional memory allocations and copying. In performance hotspots, you can avoid this overhead by using `char[]` instead:

```
[DllImport ("kernel32.dll", CharSet = CharSet.Unicode)]
static extern int GetWindowsDirectory (char[] buffer, int maxChars);
```

Notice that you must specify a `CharSet` in the `DllImport` attribute. You must also trim the output string to length after calling the function. You can achieve this while minimizing memory allocations with the use of array pooling (see "Array Pooling"), as follows:

```
string GetWindowsDirectory()
{
  var array = ArrayPool<char>.Shared.Rent (256);
  try
  {
    int length = GetWindowsDirectory (array, 256);
    return new string (array, 0, length).ToString();
```

```
  }
  finally { ArrayPool<char>.Shared.Return (array); }
}
```

(Of course, this example is contrived in that you can obtain the Windows directory via the built-in `Environment.GetFolderPath` method.)

> **NOTE**
>
> If you are unsure how to call a particular Win32 or Unix method, you will usually find an example on the internet if you search for the method name and *DllImport*. For Windows, the site *http://www.pinvoke.net* is a wiki that aims to document all Win32 signatures.

Marshaling Classes and Structs

Sometimes, you need to pass a struct to an unmanaged method. For example, `GetSystemTime` in the Win32 API is defined as follows:

```
void GetSystemTime (LPSYSTEMTIME lpSystemTime);
```

`LPSYSTEMTIME` conforms to this C struct:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

To call `GetSystemTime`, we must define a .NET class or struct that matches this C struct:

```
using System;
using System.Runtime.InteropServices;
```

```
[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
   public ushort Year;
   public ushort Month;
   public ushort DayOfWeek;
   public ushort Day;
   public ushort Hour;
   public ushort Minute;
   public ushort Second;
   public ushort Milliseconds;
}
```

The `StructLayout` attribute instructs the marshaler how to map each field to its unmanaged counterpart. `LayoutKind.Sequential` means that we want the fields aligned sequentially on *pack-size* boundaries (you'll see what this means shortly), just as they would be in a C struct. The field names here are irrelevant; it's the ordering of fields that's important.

Now we can call `GetSystemTime`:

```
SystemTime t = new SystemTime();
GetSystemTime (t);
Console.WriteLine (t.Year);

[DllImport("kernel32.dll")]
static extern void GetSystemTime (SystemTime t);
```

Similarly, on Unix:

```
Console.WriteLine (GetSystemTime());

static DateTime GetSystemTime()
{
  DateTime startOfUnixTime =
    new DateTime(1970, 1, 1, 0, 0, 0, 0, System.DateTimeKind.Utc);

  Timespec tp = new Timespec();
  int success = clock_gettime (0, ref tp);
  if (success != 0) throw new Exception ("Error checking the time.");
  return startOfUnixTime.AddSeconds (tp.tv_sec).ToLocalTime();
}
```

```
[DllImport("libc")]
static extern int clock_gettime (int clk_id, ref Timespec tp);

[StructLayout(LayoutKind.Sequential)]
struct Timespec
{
  public long tv_sec;   /* seconds */
  public long tv_nsec;  /* nanoseconds */
}
```

In both C and C#, fields in an object are located at *n* number of bytes from the address of that object. The difference is that in a C# program, the CLR finds this offset by looking it up using the field token; C field names are compiled directly into offsets. For instance, in C, `wDay` is just a token to represent whatever is at the address of a `SystemTime` instance plus 24 bytes.

For access speed, each field is placed at an offset that is a multiple of the field's size. That multiplier, however, is restricted to a maximum of *x* bytes, where *x* is the *pack size*. In the current implementation, the default pack size is 8 bytes, so a struct comprising an `sbyte` followed by an (8-byte) `long` occupies 16 bytes, and the 7 bytes following the `sbyte` are wasted. You can lessen or eliminate this wastage by specifying a *pack size* via the `Pack` property of the `StructLayout` attribute: this makes the fields align to offsets that are multiples of the specified pack size. So, with a pack size of 1, the struct just described would occupy just 9 bytes. You can specify pack sizes of 1, 2, 4, 8, or 16 bytes.

The `StructLayout` attribute also lets you specify explicit field offsets (see "Simulating a C Union").

In and Out Marshaling

In the previous example, we implemented `SystemTime` as a class. We could have instead chosen a struct—provided that `GetSystemTime` was declared with a `ref` or `out` parameter:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (out SystemTime t);
```

In most cases, C#'s directional parameter semantics work the same with external methods. Pass-by-value parameters are copied in, C# ref parameters are copied in/out, and C# out parameters are copied out. However, there are some exceptions for types that have special conversions. For instance, array classes and the `StringBuilder` class require copying when coming out of a function, so they are in/out. It is occasionally useful to override this behavior, with the `In` and `Out` attributes. For example, if an array should be read-only, the `in` modifier indicates to copy only the array going into the function, not coming out of it:

```
static extern void Foo ( [In] int[] array);
```

Calling Conventions

Unmanaged methods receive arguments and return values via the stack and (optionally) CPU registers. Because there's more than one way to accomplish this, a number of different protocols have emerged. These protocols are known as *calling conventions*.

The CLR currently supports three calling conventions: StdCall, Cdecl, and ThisCall.

By default, the CLR uses the *platform default* calling convention (the standard convention for that platform). On Windows, it's StdCall, and on Linux x86, it's Cdecl.

Should an unmanaged method not follow this default, you can explicitly state its calling convention as follows:

```
[DllImport ("MyLib.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void SomeFunc (...)
```

The somewhat misleadingly named `CallingConvention.WinApi` refers to the platform default.

Callbacks from Unmanaged Code

C# also allows external functions to call C# code, via callbacks. There are two ways to accomplish callbacks:

- Via function pointers
- Via delegates

To illustrate, we will call the following Windows function in *User32.dll*, which enumerates all top-level window handles:

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

`WNDENUMPROC` is a callback that is fired with the handle of each window in sequence (or until the callback returns `false`). Here is its definition:

```
BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);
```

Callbacks with Function Pointers

From C# 9, the simplest and most performant option—when your callback is a static method—is to use a *function pointer*. In the case of the `WNDENUMPROC` callback, we can use the following function pointer:

```
delegate*<IntPtr, IntPtr, bool>
```

This denotes a function that accepts two `IntPtr` arguments and returns a `bool`. You can then use the `&` operator to feed it a static method:

```
using System;
using System.Runtime.InteropServices;

unsafe
{
  EnumWindows (&PrintWindow, IntPtr.Zero);

  [DllImport ("user32.dll")]
  static extern int EnumWindows (
    delegate*<IntPtr, IntPtr, bool> hWnd, IntPtr lParam);
```

```
  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }
}
```

With function pointers, the callback must be a static method (or a static local function, as in this example).

UnmanagedCallersOnly

You can improve performance by applying the `unmanaged` keyword to the function pointer declaration, and the `[UnmanagedCallersOnly]` attribute to the callback method:

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

unsafe
{
  EnumWindows (&PrintWindow, IntPtr.Zero);

  [DllImport ("user32.dll")]
  static extern int EnumWindows (
    delegate* unmanaged <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);

  [UnmanagedCallersOnly]
  static byte PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return 1;
  }
}
```

This attribute flags the `PrintWindow` method such that it can be called *only* from unmanaged code, allowing the runtime to take shortcuts. Notice that we've also changed the method's return type from `bool` to `byte`: this is because methods to which you apply `[UnmanagedCallersOnly]` can use only *blittable* value types in the signature. Blittable types are those that don't require any special marshalling logic because they're represented

identically in the managed and unmanaged worlds. These include the primitive integral types, `float`, `double`, and structs that contain only blittable types. The `char` type is also blittable, if part of a struct with a `StructLayout` attribute specifying `CharSet.Unicode`:

```
[StructLayout (LayoutKind.Sequential, CharSet=CharSet.Unicode)]
```

Nondefault calling conventions

By default, the compiler assumes that the unmanaged callback follows the platform-default calling convention. Should this not be so, you can explicitly state its calling convention via the `CallConvs` parameter of the `[UnmanagedCallersOnly]` attribute:

```
[UnmanagedCallersOnly (CallConvs = new[] { typeof (CallConvStdcall) })]
static byte PrintWindow (IntPtr hWnd, IntPtr lParam) ...
```

You must also update the function pointer type by inserting a special modifier after the `unmanaged` keyword:

```
delegate* unmanaged[Stdcall] <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);
```

> **NOTE**
>
> The compiler lets you put any identifier (such as `XYZ`) inside the square brackets, as long as there's a .NET type called `CallConvXYZ` (that's understood by the runtime and matches what you specified when applying the `[UnmanagedCallersOnly]` attribute). This makes it easier for Microsoft to add new calling conventions in the future.

In this case, we specified StdCall, which is the platform default for Windows (Cdecl is the default for Linux x86). Here are all the options that are currently supported:

Name	unmanaged modifier	Supporting type
Stdcall	`unmanaged[Stdcall]`	`CallConvStdcall`
Cdecl	`unmanaged[Cdecl]`	`CallConvCdecl`
ThisCall	`unmanaged[Thiscall]`	`CallConvThiscall`

Callbacks with Delegates

Unmanaged callbacks can also be accomplished with delegates. This approach works in all versions of C#, and allows for callbacks that reference instance methods.

To proceed, first declare a delegate type with a signature that matches the callback. Then you can pass a delegate instance to the external method:

```
class CallbackFun
{
  delegate bool EnumWindowsCallback (IntPtr hWnd, IntPtr lParam);

  [DllImport("user32.dll")]
  static extern int EnumWindows (EnumWindowsCallback hWnd, IntPtr lParam);

  static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
  {
    Console.WriteLine (hWnd.ToInt64());
    return true;
  }
  static readonly EnumWindowsCallback printWindowFunc = PrintWindow;

  static void Main() => EnumWindows (printWindowFunc, IntPtr.Zero);
}
```

Using delegates for unmanaged callbacks is ironically unsafe, because it's easy to fall into the trap of allowing a callback to occur after the delegate instance falls out of scope (at which point the delegate becomes eligible for garbage collection). This can result in the worst kind of runtime exception —one with no useful stack trace. In the case of static method callbacks, you

can avoid this by assigning the delegate instance to a read-only static field (as we did in this example). With instance method callbacks, this pattern won't help, so you must code carefully to ensure that you maintain at least one reference to the delegate instance for the duration of any potential callback. Even then, if there's a bug on the unmanaged side—whereby it invokes a callback after you've told it not to—you may still have to deal with an untraceable exception. A workaround is to define a unique delegate type per unmanaged function: this helps diagnostically because the delegate type is reported in the exception.

You can change the callback's calling convention from the platform default by applying the `[UnmanagedFunctionPointer]` attribute to the delegate:

```
[UnmanagedFunctionPointer (CallingConvention.Cdecl)]
delegate void MyCallback (int foo, short bar);
```

Simulating a C Union

Each field in a `struct` is given enough room to store its data. Consider a `struct` containing one `int` and one `char`. The `int` is likely to start at an offset of `0` and is guaranteed at least four bytes. So, the `char` would start at an offset of at least `4`. If, for some reason, the `char` started at an offset of `2`, you'd change the value of the `int` if you assigned a value to the `char`. Sounds like mayhem, doesn't it? Strangely enough, the C language supports a variation on a struct called a *union* that does exactly this. You can simulate this in C# by using `LayoutKind.Explicit` and the `FieldOffset` attribute.

It might be challenging to think of a case in which this would be useful. However, suppose that you want to play a note on an external synthesizer. The Windows Multimedia API provides a function for doing just this via the MIDI protocol:

```
[DllImport ("winmm.dll")]
public static extern uint midiOutShortMsg (IntPtr handle, uint message);
```

The second argument, `message`, describes what note to play. The problem is in constructing this 32-bit unsigned integer: it's divided internally into bytes, representing a MIDI channel, note, and velocity at which to strike. One solution is to shift and mask via the bitwise `<<`, `>>`, `&`, and `|` operators to convert these bytes to and from the 32-bit "packed" message. Far simpler, though, is to define a struct with explicit layout:

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
  [FieldOffset(0)] public uint PackedMsg;    // 4 bytes long

  [FieldOffset(0)] public byte Channel;      // FieldOffset also at 0
  [FieldOffset(1)] public byte Note;
  [FieldOffset(2)] public byte Velocity;
}
```

The `Channel`, `Note`, and `Velocity` fields deliberately overlap with the 32-bit packed message. This allows you to read and write using either. No calculations are required to keep other fields in sync:

```
NoteMessage n = new NoteMessage();
Console.WriteLine (n.PackedMsg);    // 0

n.Channel = 10;
n.Note = 100;
n.Velocity = 50;
Console.WriteLine (n.PackedMsg);    // 3302410

n.PackedMsg = 3328010;
Console.WriteLine (n.Note);         // 200
```

Shared Memory

Memory-mapped files, or *shared memory*, is a feature in Windows that allows multiple processes on the same computer to share data. Shared memory is extremely fast and, unlike pipes, offers *random* access to the shared data. We saw in Chapter 15 how you can use the `MemoryMappedFile`

class to access memory-mapped files; bypassing this and calling the Win32 methods directly is a good way to demonstrate P/Invoke.

The Win32 `CreateFileMapping` function allocates shared memory. You tell it how many bytes you need and the name with which to identify the share. Another application can then subscribe to this memory by calling `OpenFileMapping` with the same name. Both methods return a *handle*, which you can convert to a pointer by calling `MapViewOfFile`.

Here's a class that encapsulates access to shared memory:

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;

public sealed class SharedMem : IDisposable
{
  // Here we're using enums because they're safer than constants

  enum FileProtection : uint      // constants from winnt.h
  {
    ReadOnly = 2,
    ReadWrite = 4
  }

  enum FileRights : uint          // constants from WinBASE.h
  {
    Read = 4,
    Write = 2,
    ReadWrite = Read + Write
  }

  static readonly IntPtr NoFileHandle = new IntPtr (-1);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr CreateFileMapping (IntPtr hFile,
                                          int lpAttributes,
                                          FileProtection flProtect,
                                          uint dwMaximumSizeHigh,
                                          uint dwMaximumSizeLow,
                                          string lpName);

  [DllImport ("kernel32.dll", SetLastError=true)]
  static extern IntPtr OpenFileMapping (FileRights dwDesiredAccess,
                                        bool bInheritHandle,
```

```
                                        string lpName);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern IntPtr MapViewOfFile (IntPtr hFileMappingObject,
                                      FileRights dwDesiredAccess,
                                      uint dwFileOffsetHigh,
                                      uint dwFileOffsetLow,
                                      uint dwNumberOfBytesToMap);

  [DllImport ("Kernel32.dll", SetLastError = true)]
  static extern bool UnmapViewOfFile (IntPtr map);

  [DllImport ("kernel32.dll", SetLastError = true)]
  static extern int CloseHandle (IntPtr hObject);

  IntPtr fileHandle, fileMap;

  public IntPtr Root => fileMap;

  public SharedMem (string name, bool existing, uint sizeInBytes)
  {
    if (existing)
      fileHandle = OpenFileMapping (FileRights.ReadWrite, false, name);
    else
      fileHandle = CreateFileMapping (NoFileHandle, 0,
                                      FileProtection.ReadWrite,
                                      0, sizeInBytes, name);
    if (fileHandle == IntPtr.Zero)
      throw new Win32Exception();

    // Obtain a read/write map for the entire file
    fileMap = MapViewOfFile (fileHandle, FileRights.ReadWrite, 0, 0, 0);

    if (fileMap == IntPtr.Zero)
      throw new Win32Exception();
  }

  public void Dispose()
  {
    if (fileMap != IntPtr.Zero) UnmapViewOfFile (fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle (fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
  }
}
```

In this example, we set `SetLastError=true` on the `DllImport` methods that use the `SetLastError` protocol for emitting error codes. This ensures that the `Win32Exception` is populated with details of the error when that exception is thrown. (It also allows you to query the error explicitly by calling `Marshal.GetLastWin32Error`.)

To demonstrate this class, we need to run two applications. The first one creates the shared memory, as follows:

```
using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
{
  IntPtr root = sm.Root;
  // I have shared memory!

  Console.ReadLine();         // Here's where we start a second app...
}
```

The second application subscribes to the shared memory by constructing a `SharedMem` object of the same name, with the `existing` argument `true`:

```
using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
{
  IntPtr root = sm.Root;
  // I have the same shared memory!
  // ...
}
```

The net result is that each program has an `IntPtr`—a pointer to the same unmanaged memory. The two applications now need somehow to read and write to memory via this common pointer. One approach is to write a class that encapsulates all the shared data and then serialize (and deserialize) the data to the unmanaged memory using an `UnmanagedMemoryStream`. This is inefficient, however, if there's a lot of data. Imagine if the shared memory class had a megabyte of data, and just one integer needed to be updated. A better approach is to define the shared data construct as a struct and then map it directly into shared memory. We discuss this in the following section.

Mapping a Struct to Unmanaged Memory

You can directly map a struct with a `StructLayout` of `Sequential` or `Explicit` into unmanaged memory. Consider the following struct:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  public int Value;
  public char Letter;
  public fixed float Numbers [50];
}
```

The `fixed` directive allows us to define fixed-length value-type arrays inline, and it is what takes us into the `unsafe` realm. Space in this struct is allocated inline for 50 floating-point numbers. Unlike with standard C# arrays, `Numbers` is not a *reference* to an array—it *is* the array. If we run the following:

```
static unsafe void Main() => Console.WriteLine (sizeof (MySharedData));
```

the result is 208: 50 four-byte floats, plus the four bytes for the `Value` integer, plus two bytes for the `Letter` character. The total, 206, is rounded to 208 due to the `floats` being aligned on four-byte boundaries (four bytes being the size of a `float`).

We can demonstrate `MySharedData` in an `unsafe` context, most simply, with stack-allocated memory:

```
MySharedData d;
MySharedData* data = &d;       // Get the address of d

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;

or:

// Allocate the array on the stack:
```

```
MySharedData* data = stackalloc MySharedData[1];

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Of course, we're not demonstrating anything that couldn't otherwise be achieved in a managed context. Suppose, however, that we want to store an instance of `MySharedData` on the *unmanaged heap*, outside the realm of the CLR's garbage collector. This is where pointers become really useful:

```
MySharedData* data = (MySharedData*)
  Marshal.AllocHGlobal (sizeof (MySharedData)).ToPointer();

data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

`Marshal.AllocHGlobal` allocates memory on the unmanaged heap. Here's how to later free the same memory:

```
Marshal.FreeHGlobal (new IntPtr (data));
```

(The result of forgetting to free the memory is a good old-fashioned memory leak.)

> **NOTE**
>
> From .NET 6, you can instead use the `NativeMemory` class for allocating and freeing unmanaged memory. `NativeMemory` uses a newer (and better) underlying API than `AllocHGlobal` and also includes methods for performing aligned allocations.

In keeping with its name, here we use `MySharedData` in conjunction with the `SharedMem` class we wrote in the preceding section. The following program allocates a block of shared memory, and then maps the `MySharedData` struct into that memory:

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", false,
                          (uint) sizeof (MySharedData)))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    data->Value = 123;
    data->Letter = 'X';
    data->Numbers[10] = 1.45f;
    Console.WriteLine ("Written to shared memory");

    Console.ReadLine();

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);
    Console.ReadLine();
  }
}
```

NOTE

You can use the built-in `MemoryMappedFile` class instead of `SharedMem`, as follows:

```
using (MemoryMappedFile mmFile =
       MemoryMappedFile.CreateNew ("MyShare", 1000))
using (MemoryMappedViewAccessor accessor =
       mmFile.CreateViewAccessor())
{
  byte* pointer = null;
  accessor.SafeMemoryMappedViewHandle.AcquirePointer
   (ref pointer);
  void* root = pointer;
  ...
}
```

Here's a second program that attaches to the same shared memory, reading the values written by the first program (it must be run while the first

program is waiting on the `ReadLine` statement because the shared memory object is disposed upon leaving its `using` statement):

```
static unsafe void Main()
{
  using (SharedMem sm = new SharedMem ("MyShare", true,
                          (uint) sizeof (MySharedData)))
  {
    void* root = sm.Root.ToPointer();
    MySharedData* data = (MySharedData*) root;

    Console.WriteLine ("Value is " + data->Value);
    Console.WriteLine ("Letter is " + data->Letter);
    Console.WriteLine ("11th Number is " + data->Numbers[10]);

    // Our turn to update values in shared memory!
    data->Value++;
    data->Letter = '!';
    data->Numbers[10] = 987.5f;
    Console.WriteLine ("Updated shared memory");
    Console.ReadLine();
  }
}
```

The output from each of these programs is as follows:

```
// First program:

Written to shared memory
Value is 124
Letter is !
11th Number is 987.5

// Second program:

Value is 123
Letter is X
11th Number is 1.45
Updated shared memory
```

Don’t be put off by the pointers: C++ programmers use them throughout whole applications and are able to get everything working. At least most of the time! This sort of usage is fairly simple by comparison.

As it happens, our example is unsafe—quite literally—for another reason. We've not considered the thread-safety (or more precisely, process-safety) issues that arise with two programs accessing the same memory at once. To use this in a production application, we'd need to add the `volatile` keyword to the `Value` and `Letter` fields in the `MySharedData` struct to prevent fields from being cached by the Just-in-Time (JIT) compiler (or by the hardware in CPU registers). Furthermore, as our interaction with the fields grew beyond the trivial, we would most likely need to protect their access via a cross-process `Mutex`, just as we would use `lock` statements to protect access to fields in a multithreaded program. We discussed thread safety in detail in Chapter 21.

fixed and fixed {...}

One limitation of mapping structs directly into memory is that the struct can contain only unmanaged types. If you need to share string data, for instance, you must use a fixed-character array instead. This means manual conversion to and from the `string` type. Here's how to do it:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
  ...
  // Allocate space for 200 chars (i.e., 400 bytes).
  const int MessageSize = 200;
  fixed char message [MessageSize];

  // One would most likely put this code into a helper class:
  public string Message
  {
    get { fixed (char* cp = message) return new string (cp); }
    set
    {
      fixed (char* cp = message)
      {
        int i = 0;
        for (; i < value.Length && i < MessageSize - 1; i++)
          cp [i] = value [i];

        // Add the null terminator
```

```
            cp [i] = '\0';
          }
        }
      }
    }
```

> **NOTE**
>
> There's no such thing as a reference to a fixed array; instead, you get a pointer. When you index into a fixed array, you're actually performing pointer arithmetic!

With the first use of the `fixed` keyword, we allocate space, inline, for 200 characters in the struct. The same keyword (somewhat confusingly) has a different meaning when used later in the property definition. It instructs the CLR to *pin* an object so that should it decide to perform a garbage collection inside the `fixed` block, it will not move the underlying struct about on the memory heap (because its contents are being iterated via direct memory pointers). Looking at our program, you might wonder how `MySharedData` could ever shift in memory, given that it resides not on the heap but in the unmanaged world, where the garbage collector has no jurisdiction. The compiler doesn't know this, however, and is concerned that we *might* use `MySharedData` in a managed context, so it insists that we add the `fixed` keyword to make our `unsafe` code safe in managed contexts. And the compiler does have a point—here's all it would take to put `MySharedData` on the heap:

```
object obj = new MySharedData();
```

This results in a boxed `MySharedData`—on the heap and eligible for transit during garbage collection.

This example illustrates how a string can be represented in a struct mapped to unmanaged memory. For more complex types, you also have the option of using existing serialization code. The one proviso is that the serialized

data must never exceed, in length, its allocation of space in the struct; otherwise, the result is an unintended union with subsequent fields.

COM Interoperability

The .NET runtime provides special support for COM, enabling COM objects to be used from .NET, and vice versa. COM is available only on Windows.

The Purpose of COM

COM is an acronym for Component Object Model, a binary standard for interfacing with libraries, released by Microsoft in 1993. The motivation for inventing COM was to enable components to communicate with each other in a language-independent and version-tolerant manner. Before COM, the approach in Windows was to publish DLLs that declared structures and functions using the C programming language. Not only is this approach language specific, but it's also brittle. The specification of a type in such a library is inseparable from its implementation: even updating a structure with a new field means breaking its specification.

The beauty of COM was to separate the specification of a type from its underlying implementation through a construct known as a *COM interface*. COM also allowed for the calling of methods on stateful *objects*—rather than being limited to simple procedure calls.

> **NOTE**
>
> In a way, the .NET programming model is an evolution of the principles of COM programming: the .NET platform also facilitates cross-language development and allows binary components to evolve without breaking applications that depend on them.

The Basics of the COM Type System

The COM type system revolves around interfaces. A COM interface is rather like a .NET interface, but it's more prevalent because a COM type exposes its functionality *only* through an interface. In the .NET world, for instance, we could declare a type simply, as follows:

```
public class Foo
{
  public string Test() => "Hello, world";
}
```

Consumers of that type can use `Foo` directly. And if we later changed the *implementation* of `Test()`, calling assemblies would not require recompilation. In this respect, .NET separates interface from implementation—without requiring interfaces. We could even add an overload without breaking callers:

```
  public string Test (string s) => $"Hello, world {s}";
```

In the COM world, `Foo` exposes its functionality through an interface to achieve this same decoupling. So, in `Foo`'s type library, an interface such as this would exist:

```
public interface IFoo { string Test(); }
```

(We've illustrated this by showing a C# interface—not a COM interface. The principle, however, is the same—although the plumbing is different.)

Callers would then interact with `IFoo` rather than `Foo`.

When it comes to adding the overloaded version of `Test`, life is more complicated with COM than with .NET. First, we would avoid modifying the `IFoo` interface because this would break binary compatibility with the previous version (one of the principles of COM is that interfaces, once published, are *immutable*). Second, COM doesn't allow method overloading. The solution is to instead have `Foo` implement a *second interface*:

```
public interface IFoo2 { string Test (string s); }
```

(Again, we've transliterated this into a .NET interface for familiarity.)

Supporting multiple interfaces is of key importance in making COM libraries versionable.

IUnknown and IDispatch

All COM interfaces are identified with a Globally Unique Identifier (GUID).

The root interface in COM is `IUnknown`—all COM objects must implement it. This interface has three methods:

- `AddRef`
- `Release`
- `QueryInterface`

`AddRef` and `Release` are for lifetime management given that COM uses reference counting rather than automatic garbage collection (COM was designed to work with unmanaged code, where automatic garbage collection isn't feasible). The `QueryInterface` method returns an object reference that supports that interface, if it can do so.

To enable dynamic programming (e.g., scripting and automation), a COM object can also implement `IDispatch`. This enables dynamic languages to call COM objects in a late-bound manner—rather like `dynamic` in C# (although only for simple invocations).

Calling a COM Component from C#

The CLR's built-in support for COM means that you don't work directly with `IUnknown` and `IDispatch`. Instead, you work with CLR objects, and the runtime marshals your calls to the COM world via Runtime-Callable Wrappers (RCWs). The runtime also handles lifetime management by

calling `AddRef` and `Release` (when the .NET object is finalized) and takes care of the primitive type conversions between the two worlds. Type conversion ensures that each side sees, for example, the integer and string types in their familiar forms.

Additionally, there needs to be some way to access RCWs in a statically typed fashion. This is the job of *COM interop types*. COM interop types are automatically generated proxy types that expose a .NET member for each COM member. The type library importer tool (*tlbimp.exe*) generates COM interop types from the command line, based on a COM library that you choose, and compiles them into a *COM interop assembly*.

> **NOTE**
>
> If a COM component implements multiple interfaces, the *tlbimp.exe* tool generates a single type that contains a union of members from all interfaces.

You can create a COM interop assembly in Visual Studio by going to the Add Reference dialog box and choosing a library from the COM tab. For example, if you have Microsoft Excel installed, adding a reference to the Microsoft Excel Object Library allows you to interoperate with Excel's COM classes. Here's the C# code to create and show a workbook, and then populate a cell in that workbook:

```
using System;
using Excel = Microsoft.Office.Interop.Excel;

var excel = new Excel.Application();
excel.Visible = true;
excel.WindowState = Excel.XlWindowState.xlMaximized;
Excel.Workbook workBook = excel.Workbooks.Add();
((Excel.Range)excel.Cells[1, 1]).Font.FontStyle = "Bold";
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello World";
workBook.SaveAs (@"d:\temp.xlsx");
```

NOTE

It is currently necessary to embed interop types in your application (otherwise, the runtime won't locate them at runtime). Either click the COM reference in Visual Studio's Solution Explorer and set the Embed Interop Types property to true in the Properties window, or open your *.csproj* file and add the following line (in boldface):

```
<ItemGroup>
  <COMReference Include="Microsoft.Office.Excel.dll">
    ...
    <EmbedInteropTypes>true</EmbedInteropTypes>
  </COMReference>
</ItemGroup>
```

The `Excel.Application` class is a COM interop type whose runtime type is an RCW. When we access the `Workbooks` and `Cells` properties, we get back more interop types.

Optional Parameters and Named Arguments

Because COM APIs don't support function overloading, it's very common to have functions with numerous parameters, many of which are optional. For instance, here's how you might call an Excel workbook's `Save` method:

```
var missing = System.Reflection.Missing.Value;

workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
  missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
  missing, missing, missing);
```

The good news is that the C#'s support for optional parameters is COM-aware, so we can just do this:

```
workBook.SaveAs (@"d:\temp.xlsx");
```

(As we stated in Chapter 3, optional parameters are "expanded" by the compiler into the full verbose form.)

Named arguments allow you to specify additional arguments, regardless of their position:

```
workBook.SaveAs (@"d:\test.xlsx", Password:"foo");
```

Implicit ref Parameters

Some COM APIs (Microsoft Word, in particular) expose functions that declare *every* parameter as pass-by-reference—whether or not the function modifies the parameter value. This is because of the perceived performance gain from not copying argument values (the *real* performance gain is negligible).

Historically, calling such methods from C# has been clumsy because you must specify the `ref` keyword with every argument, and this prevents the use of optional parameters. For instance, to open a Word document, we used to have to do this:

```
object filename = "foo.doc";
object notUsed1 = Missing.Value;
object notUsed2 = Missing.Value;
object notUsed3 = Missing.Value;
...
Open (ref filename, ref notUsed1, ref notUsed2, ref notUsed3, ...);
```

Thanks to implicit ref parameters, you can omit the `ref` modifier on COM function calls, allowing the use of optional parameters:

```
word.Open ("foo.doc");
```

The caveat is that you will get neither a compile-time nor a runtime error if the COM method you're calling actually does mutate an argument value.

Indexers

The ability to omit the `ref` modifier has another benefit: it makes COM indexers with `ref` parameters accessible via ordinary C# indexer syntax.

This would otherwise be forbidden because `ref`/`out` parameters are not supported with C# indexers.

You can also call COM properties that accept arguments. In the following example, `Foo` is a property that accepts an integer argument:

```
myComObject.Foo [123] = "Hello";
```

Writing such properties yourself in C# is still prohibited: a type can expose an indexer only on itself (the "default" indexer). Therefore, if you wanted to write code in C# that would make the preceding statement legal, `Foo` would need to return another type that exposed a (default) indexer.

Dynamic Binding

There are two ways that dynamic binding can help when calling COM components.

The first way is in allowing access to a COM component without a COM interop type. To do this, call `Type.GetTypeFromProgID` with the COM component name to obtain a COM instance, and then use dynamic binding to call members from then on. Of course, there's no IntelliSense, and compile-time checks are impossible:

```
Type excelAppType = Type.GetTypeFromProgID ("Excel.Application", true);
dynamic excel = Activator.CreateInstance (excelAppType);
excel.Visible = true;
dynamic wb = excel.Workbooks.Add();
excel.Cells [1, 1].Value2 = "foo";
```

(The same thing can be achieved, much more clumsily, with reflection instead of dynamic binding.)

> **NOTE**
>
> A variation of this theme is calling a COM component that supports *only* `IDispatch`. Such components are quite rare, however.

Dynamic binding can also be useful (to a lesser extent) in dealing with the COM `variant` type. For reasons due more to poor design than necessity, COM API functions are often peppered with this type, which is roughly equivalent to `object` in .NET. If you enable "Embed Interop Types" in your project (more on this soon), the runtime will map `variant` to `dynamic`, instead of mapping `variant` to `object`, avoiding the need for casts. For instance, you could legally do

```
excel.Cells [1, 1].Font.FontStyle = "Bold";
```

instead of:

```
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

The disadvantage of working in this way is that you lose autocompletion, so you must know that a property called `Font` happens to exist. For this reason, it's usually easier to *dynamically* assign the result to its known interop type:

```
Excel.Range range = excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

As you can see, this saves only five characters over the old-fashioned approach!

The mapping of `variant` to `dynamic` is the default, and is a function of enabling Embed Interop Types on a reference.

Embedding Interop Types

We said previously that C# ordinarily calls COM components via interop types that are generated by calling the *tlbimp.exe* tool (directly or via Visual Studio).

Historically, your only option was to *reference* interop assemblies just as you would with any other assembly. This could be troublesome because interop assemblies can get quite large with complex COM components. A tiny add-in for Microsoft Word, for instance, requires an interop assembly that is orders of magnitude larger than itself.

Rather than *referencing* an interop assembly, you have the option of embedding the portions that you use. The compiler analyzes the assembly to work out precisely the types and members that your application requires, and embeds definitions for (just) those types and members directly in your application. This avoids bloat as well as the need to ship an additional file.

To enable this feature, either select the COM reference in Visual Studio's Solution Explorer and then set Embed Interop Types to true in the Properties window, or edit your *.csproj* file as we described earlier (see "Calling a COM Component from C#").

Type Equivalence

The CLR supports *type equivalence* for linked interop types. This means that if two assemblies each link to an interop type, those types will be considered equivalent if they wrap the same COM type. This holds true even if the interop assemblies to which they linked were generated independently.

> **NOTE**
>
> Type equivalence relies on the `TypeIdentifierAttribute` attribute in the `System.Runtime.InteropServices` namespace. The compiler automatically applies this attribute when you link to interop assemblies. COM types are then considered equivalent if they have the same GUID.

Exposing C# Objects to COM

It's also possible to write classes in C# that can be consumed in the COM world. The CLR makes this possible through a proxy called a *COM-Callable Wrapper* (CCW). A CCW marshals types between the two worlds (as with an RCW) and implements `IUnknown` (and optionally `IDispatch`) as required by the COM protocol. A CCW is lifetime-controlled from the COM side via reference counting (rather than through the CLR's garbage collector).

You can expose any public class to COM (as an "in-proc" server). To do so, first create an interface, assign it a unique GUID (in Visual Studio, you can use Tools > Create GUID), declare it visible to COM, and then set the interface type:

```
namespace MyCom
{
  [ComVisible(true)]
  [Guid ("226E5561-C68E-4B2B-BD28-25103ABCA3B1")]  // Change this GUID
  [InterfaceType (ComInterfaceType.InterfaceIsIUnknown)]
  public interface IServer
  {
    int Fibonacci();
  }
}
```

Next, provide an implementation of your interface, assigning a unique GUID to that implementation:

```
namespace MyCom
{
  [ComVisible(true)]
  [Guid ("09E01FCD-9970-4DB3-B537-0EC555967DD9")]  // Change this GUID
  public class Server
  {
    public ulong Fibonacci (ulong whichTerm)
    {
      if (whichTerm < 1) throw new ArgumentException ("...");
      ulong a = 0;
      ulong b = 1;
      for (ulong i = 0; i < whichTerm; i++)
```

```
        {
          ulong tmp = a;
          a = b;
          b = tmp + b;
        }
        return a;
      }
    }
}
```

Edit your *.csproj* file, adding the following line:

```
<PropertyGroup>
  <EnableComHosting>true</EnableComHosting>
</PropertyGroup>
```

Now, when you build your project, an additional file is generated, *MyCom .comhost.dll*, which can be registered for COM interop. (Keep in mind that the file will always be 32 bit or 64 bit depending on your project configuration: there's no such thing as "Any CPU" in this scenario.) From an *elevated* command prompt, switch to the directory holding your DLL and run *regsvr32 MyCom.comhost.dll*.

You can then consume your COM component from most COM-capable languages. For example, you can create this Visual Basic Script in a text editor and run it by double-clicking the file in Windows Explorer, or by starting it from a command prompt as you would a program:

```
REM Save file as ComClient.vbs
Dim obj
Set obj = CreateObject("MyCom.Server")

result = obj.Fibonacci(12)
Wscript.Echo result
```

Note that .NET Framework cannot be loaded into the same process as .NET 5+ or .NET Core. Therefore, a .NET 5+ COM server cannot be loaded into a .NET Framework COM client process, or vice versa.

Enabling Registry-Free COM

Traditionally, COM adds type information to the registry. Registry-free COM uses a manifest file instead of the registry to control object activation. To enable this feature, add the following line (in boldface) to your *.csproj* file:

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <EnableComHosting>true</EnableComHosting>
  <EnableRegFreeCom>true</EnableRegFreeCom>
</PropertyGroup>
```

Your build will then generate *MyCom.X.manifest*.

> **NOTE**
>
> There is no support in .NET 5+ for generating a COM type library (*.tlb). You can manually write an IDL (Interface Definition Language) file or C++ header for the native declarations in your interface.