- **TestArrayInStruct** exported from PinvokeLib.dll.

```
C++

void TestArrayInStruct(MYARRAYSTRUCT* pStruct);
```

PinvokeLib.dll is a custom unmanaged library that contains implementations for the previously listed functions and four structures: **MYPERSON**, **MYPERSON2**, **MYPERSON3**, and **MYARRAYSTRUCT**. These structures contain the following elements:

```
C++

typedef struct _MYPERSON
{
   char* first;
   char* last;
} MYPERSON, *LP_MYPERSON;

typedef struct _MYPERSON2
{
   MYPERSON* person;
   int age;
} MYPERSON2, *LP_MYPERSON2;

typedef struct _MYPERSON3
{
   MYPERSON person;
   int age;
} MYPERSON3;

typedef struct _MYARRAYSTRUCT
{
   bool flag;
   int vals[ 3 ];
} MYARRAYSTRUCT;
```

The managed `MyPerson`, `MyPerson2`, `MyPerson3`, and `MyArrayStruct` structures have the following characteristic:

- `MyPerson` contains only string members. The CharSet field sets the strings to ANSI format when passed to the unmanaged function.
- `MyPerson2` contains an **IntPtr** to the `MyPerson` structure. The **IntPtr** type replaces the original pointer to the unmanaged structure because .NET Framework applications do

original pointer to the unmanaged structure because .NET Framework applications do not use pointers unless the code is marked **unsafe**.

- `MyPerson3` contains `MyPerson` as an embedded structure. A structure embedded within another structure can be flattened by placing the elements of the embedded structure directly into the main structure, or it can be left as an embedded structure, as is done in this sample.

- `MyArrayStruct` contains an array of integers. The MarshalAsAttribute attribute sets the UnmanagedType enumeration value to **ByValArray**, which is used to indicate the number of elements in the array.

For all structures in this sample, the StructLayoutAttribute attribute is applied to ensure that the members are arranged in memory sequentially, in the order in which they appear.

The `NativeMethods` class contains managed prototypes for the `TestStructInStruct`, `TestStructInStruct3`, and `TestArrayInStruct` methods called by the `App` class. Each prototype declares a single parameter, as follows:

- `TestStructInStruct` declares a reference to type `MyPerson2` as its parameter.

- `TestStructInStruct3` declares type `MyPerson3` as its parameter and passes the parameter by value.

- `TestArrayInStruct` declares a reference to type `MyArrayStruct` as its parameter.

Structures as arguments to methods are passed by value unless the parameter contains the **ref** (**ByRef** in Visual Basic) keyword. For example, the `TestStructInStruct` method passes a reference (the value of an address) to an object of type `MyPerson2` to unmanaged code. To manipulate the structure that `MyPerson2` points to, the sample creates a buffer of a specified size and returns its address by combining the Marshal.AllocCoTaskMem and Marshal.SizeOf methods. Next, the sample copies the content of the managed structure to the unmanaged buffer. Finally, the sample uses the Marshal.PtrToStructure method to marshal data from the unmanaged buffer to a managed object and the Marshal.FreeCoTaskMem method to free the unmanaged block of memory.

Declaring Prototypes

C#

```
// Declares a managed structure for each unmanaged structure.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct MyPerson
{
    public string first;
    public string last;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyPerson2
{
    public IntPtr person;
    public int age;
}


[StructLayout(LayoutKind.Sequential)]
public struct MyPerson3
{
    public MyPerson person;
    public int age;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyArrayStruct
{
    public bool flag;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 3)]
    public int[] vals;
}

internal static class NativeMethods
{
    // Declares a managed prototype for unmanaged function.
    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestStructInStruct(ref MyPerson2 person2);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestStructInStruct3(MyPerson3 person3);

    [DllImport("..\\LIB\\PinvokeLib.dll", CallingConvention =
CallingConvention.Cdecl)]
    internal static extern int TestArrayInStruct(ref MyArrayStruct myStruct);
}
```

Calling Functions

C#

```
public class App
{
    public static void Main()
    {
        // Structure with a pointer to another structure.
        MyPerson personName;
        personName.first = "Mark";
        personName.last = "Lee";

        MyPerson2 personAll;
        personAll.age = 30;

        IntPtr buffer = Marshal.AllocCoTaskMem(Marshal.SizeOf(personName));
        Marshal.StructureToPtr(personName, buffer, false);

        personAll.person = buffer;

        Console.WriteLine("\nPerson before call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personName.first, personName.last, personAll.age);

        int res = NativeMethods.TestStructInStruct(ref personAll);

        MyPerson personRes =
            (MyPerson)Marshal.PtrToStructure(personAll.person,
            typeof(MyPerson));

        Marshal.FreeCoTaskMem(buffer);

        Console.WriteLine("Person after call:");
        Console.WriteLine("first = {0}, last = {1}, age = {2}",
            personRes.first, personRes.last, personAll.age);

        // Structure with an embedded structure.
        MyPerson3 person3 = new MyPerson3();
        person3.person.first = "John";
        person3.person.last = "Evans";
        person3.age = 27;
        NativeMethods.TestStructInStruct3(person3);

        // Structure with an embedded array.
        MyArrayStruct myStruct = new MyArrayStruct();

        myStruct.flag = false;
        myStruct.vals = new int[3];
        myStruct.vals[0] = 1;
        myStruct.vals[1] = 4;
        myStruct.vals[2] = 9;
```

```
        Console.WriteLine("\nStructure with array before call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);

        NativeMethods.TestArrayInStruct(ref myStruct);
        Console.WriteLine("\nStructure with array after call:");
        Console.WriteLine(myStruct.flag);
        Console.WriteLine("{0} {1} {2}", myStruct.vals[0],
            myStruct.vals[1], myStruct.vals[2]);
    }
}
```

FindFile sample

This sample demonstrates how to pass a structure that contains a second, embedded structure to an unmanaged function. It also demonstrates how to use the MarshalAsAttribute attribute to declare a fixed-length array within the structure. In this sample, the embedded structure elements are added to the parent structure. For a sample of an embedded structure that is not flattened, see Structures Sample.

The FindFile sample uses the following unmanaged function, shown with its original function declaration:

- **FindFirstFile** exported from Kernel32.dll.

  C++

  ```
  HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA
  lpFindFileData);
  ```

The original structure passed to the function contains the following elements:

C++

```
typedef struct _WIN32_FIND_DATA
{
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
```

```
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[ MAX_PATH ];
  TCHAR    cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
```

In this sample, the `FindData` class contains a corresponding data member for each element of the original structure and the embedded structure. In place of two original character buffers, the class substitutes strings. **MarshalAsAttribute** sets the UnmanagedType enumeration to **ByValTStr**, which is used to identify the inline, fixed-length character arrays that appear within the unmanaged structures.

The `NativeMethods` class contains a managed prototype of the `FindFirstFile` method, which passes the `FindData` class as a parameter. The parameter must be declared with the InAttribute and OutAttribute attributes because classes, which are reference types, are passed as In parameters by default.

Declaring Prototypes

C#

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
public class FindData
{
    public int fileAttributes = 0;
    // creationTime was an embedded FILETIME structure.
    public int creationTime_lowDateTime = 0;
    public int creationTime_highDateTime = 0;
    // lastAccessTime was an embedded FILETIME structure.
    public int lastAccessTime_lowDateTime = 0;
    public int lastAccessTime_highDateTime = 0;
    // lastWriteTime was an embedded FILETIME structure.
    public int lastWriteTime_lowDateTime = 0;
    public int lastWriteTime_highDateTime = 0;
    public int nFileSizeHigh = 0;
    public int nFileSizeLow = 0;
    public int dwReserved0 = 0;
    public int dwReserved1 = 0;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
    public string fileName = null;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
    public string alternateFileName = null;
}
```

```
internal static class NativeMethods
{
    // Declares a managed prototype for the unmanaged function.
    [DllImport("Kernel32.dll", CharSet = CharSet.Auto)]
    internal static extern IntPtr FindFirstFile(
        string fileName, [In, Out] FindData findFileData);
}
```

Calling Functions

C#

```
public class App
{
    public static void Main()
    {
        FindData fd = new FindData();
        IntPtr handle = NativeMethods.FindFirstFile("C:\\*.*", fd);
        Console.WriteLine($"The first file: {fd.fileName}");
    }
}
```

Unions sample

This sample demonstrates how to pass structures containing only value types, and structures containing a value type and a string as parameters to an unmanaged function expecting a union. A union represents a memory location that can be shared by two or more variables.

The Unions sample uses the following unmanaged function, shown with its original function declaration:

- **TestUnion** exported from PinvokeLib.dll.

  C++

  ```
  void TestUnion(MYUNION u, int type);
  ```

PinvokeLib.dll is a custom unmanaged library that contains an implementation for the previously listed function and two unions, **MYUNION** and **MYUNION2**. The unions contain

the following elements:

C++

```
union MYUNION
{
    int number;
    double d;
}

union MYUNION2
{
    int i;
    char str[128];
};
```

In managed code, unions are defined as structures. The `MyUnion` structure contains two value types as its members: an integer and a double. The StructLayoutAttribute attribute is set to control the precise position of each data member. The FieldOffsetAttribute attribute provides the physical position of fields within the unmanaged representation of a union. Notice that both members have the same offset values, so the members can define the same piece of memory.

`MyUnion2_1` and `MyUnion2_2` contain a value type (integer) and a string, respectively. In managed code, value types and reference types are not permitted to overlap. This sample uses method overloading to enable the caller to use both types when calling the same unmanaged function. The layout of `MyUnion2_1` is explicit and has a precise offset value. In contrast, `MyUnion2_2` has a sequential layout, because explicit layouts are not permitted with reference types. The MarshalAsAttribute attribute sets the UnmanagedType enumeration to **ByValTStr**, which is used to identify the inline, fixed-length character arrays that appear within the unmanaged representation of the union.

The `NativeMethods` class contains the prototypes for the `TestUnion` and `TestUnion2` methods. `TestUnion2` is overloaded to declare `MyUnion2_1` or `MyUnion2_2` as parameters.

Declaring Prototypes

C#

```
// Declares managed structures instead of unions.
[StructLayout(LayoutKind.Explicit)]
public struct MyUnion
{
```

```
{
    [FieldOffset(0)]
    public int i;
    [FieldOffset(0)]
    public double d;
}

[StructLayout(LayoutKind.Explicit, Size = 128)]
public struct MyUnion2_1
{
    [FieldOffset(0)]
    public int i;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyUnion2_2

{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string str;
}

internal static class NativeMethods
{
    // Declares managed prototypes for unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestUnion(MyUnion u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestUnion2(MyUnion2_1 u, int type);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestUnion2(MyUnion2_2 u, int type);
}
```

Calling Functions

C#

```
public class App
{
    public static void Main()
    {
        MyUnion mu = new MyUnion();
        mu.i = 99;
        NativeMethods.TestUnion(mu, 1);

        mu.d = 99.99;
        NativeMethods.TestUnion(mu, 2);
```

```
        MyUnion2_1 mu2_1 = new MyUnion2_1();
        mu2_1.i = 99;
        NativeMethods.TestUnion2(mu2_1, 1);

        MyUnion2_2 mu2_2 = new MyUnion2_2();
        mu2_2.str = "*** string ***";
        NativeMethods.TestUnion2(mu2_2, 2);
    }
}
```

Platform sample

In some scenarios, `struct` and `union` layouts can differ depending on the targeted platform. For example, consider the STRRET type when defined in a COM scenario:

c++

```
#include <pshpack8.h> /* Defines the packing of the struct */
typedef struct _STRRET
    {
    UINT uType;
    /* [switch_is][switch_type] */ union
        {
        /* [case()][string] */ LPWSTR pOleStr;
        /* [case()] */ UINT uOffset;
        /* [case()] */ char cStr[ 260 ];
        }  DUMMYUNIONNAME;
    }  STRRET;
#include <poppack.h>
```

The above `struct` is declared with Windows' headers that influence the memory layout of the type. When defined in a managed environment, these layout details are needed to properly interoperate with native code.

The correct managed definition of this type in a 32-bit process is:

CSharp

```
[StructLayout(LayoutKind.Explicit, Size = 264)]
public struct STRRET_32
{
    [FieldOffset(0)]
    public uint uType;
```

```
    [FieldOffset(4)]
    public IntPtr pOleStr;

    [FieldOffset(4)]
    public uint uOffset;

    [FieldOffset(4)]
    public IntPtr cStr;
}
```

On a 64-bit process, the size *and* field offsets are different. The correct layout is:

CSharp

```
[StructLayout(LayoutKind.Explicit, Size = 272)]
public struct STRRET_64
{
    [FieldOffset(0)]
    public uint uType;

    [FieldOffset(8)]
    public IntPtr pOleStr;

    [FieldOffset(8)]
    public uint uOffset;

    [FieldOffset(8)]
    public IntPtr cStr;
}
```

Failure to properly consider the native layout in an interop scenario can result in random crashes or worse, incorrect computations.

By default, .NET assemblies can run in both a 32-bit and 64-bit version of the .NET runtime. The app must wait until run time to decide which of the previous definitions to use.

The following code snippet shows an example of how to choose between the 32-bit and 64-bit definition at run time.

C#

```
if (IntPtr.Size == 8)
{
    // Use the STRRET_64 definition
}
```

```
else
{
    Debug.Assert(IntPtr.Size == 4);
    // Use the STRRET_32 definition
}
```

SysTime sample

This sample demonstrates how to pass a pointer to a class to an unmanaged function that expects a pointer to a structure.

The SysTime sample uses the following unmanaged function, shown with its original function declaration:

- **GetSystemTime** exported from Kernel32.dll.

```
C++

VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

The original structure passed to the function contains the following elements:

```
C++

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

In this sample, the `SystemTime` class contains the elements of the original structure represented as class members. The StructLayoutAttribute attribute is set to ensure that the members are arranged in memory sequentially, in the order in which they appear.

The `NativeMethods` class contains a managed prototype of the `GetSystemTime` method, which passes the `SystemTime` class as an In/Out parameter by default. The parameter must be declared with the InAttribute and OutAttribute attributes because classes, which are reference types, are passed as In parameters by default. For the caller to receive the results, these directional attributes must be applied explicitly. The A... class creates a new instance

these directional attributes must be applied explicitly. The `App` class creates a new instance of the `SystemTime` class and accesses its data fields.

Code Samples

C#

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

[StructLayout(LayoutKind.Sequential)]
public class SystemTime
{
    public ushort year;
    public ushort month;
    public ushort weekday;
    public ushort day;
    public ushort hour;
    public ushort minute;
    public ushort second;
    public ushort millisecond;
}

internal static class NativeMethods
{
    // Declares a managed prototype for the unmanaged function using Platform
Invoke.
    [DllImport("Kernel32.dll")]
    internal static extern void GetSystemTime([In, Out] SystemTime st);
}

public class App
{
    public static void Main()
    {
        Console.WriteLine("C# SysTime Sample using Platform Invoke");
        SystemTime st = new SystemTime();
        NativeMethods.GetSystemTime(st);
        Console.Write("The Date is: ");
        Console.Write($"{st.month} {st.day} {st.year}");
    }
}

// The program produces output similar to the following:
//
// C# SysTime Sample using Platform Invoke
// The Date is: 3 21 2010
```

OutArrayOfStructs sample

This sample shows how to pass an array of structures that contains integers and strings as Out parameters to an unmanaged function.

This sample demonstrates how to call a native function by using the Marshal class and by using unsafe code.

This sample uses a wrapper functions and platform invokes defined in PinvokeLib.dll, also provided in the source files. It uses the `TestOutArrayOfStructs` function and the `MYSTRSTRUCT2` structure. The structure contains the following elements:

C++

```
typedef struct _MYSTRSTRUCT2
{
   char* buffer;
   UINT size;
} MYSTRSTRUCT2;
```

The `MyStruct` class contains a string object of ANSI characters. The CharSet field specifies ANSI format. `MyUnsafeStruct`, is a structure containing an IntPtr type instead of a string.

The `NativeMethods` class contains the overloaded `TestOutArrayOfStructs` prototype method. If a method declares a pointer as a parameter, the class should be marked with the `unsafe` keyword. Because Visual Basic cannot use unsafe code, the overloaded method, unsafe modifier, and the `MyUnsafeStruct` structure are unnecessary.

The `App` class implements the `UsingMarshaling` method, which performs all the tasks necessary to pass the array. The array is marked with the `out` (`ByRef` in Visual Basic) keyword to indicate that data passes from callee to caller. The implementation uses the following Marshal class methods:

- PtrToStructure to marshal data from the unmanaged buffer to a managed object.
- DestroyStructure to release the memory reserved for strings in the structure.
- FreeCoTaskMem to release the memory reserved for the array.

As previously mentioned, C# allows unsafe code and Visual Basic does not. In the C# sample, `UsingUnsafePointer` is an alternative method implementation that uses pointers

instead of the Marshal class to pass back the array containing the `MyUnsafeStruct` structure.

Declaring Prototypes

C#

```
// Declares a class member for each structure element.
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public class MyStruct
{
    public string buffer;
    public int size;

}

// Declares a structure with a pointer.
[StructLayout(LayoutKind.Sequential)]
public struct MyUnsafeStruct
{
    public IntPtr buffer;
    public int size;
}

internal static unsafe class NativeMethods
{
    // Declares managed prototypes for the unmanaged function.
    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestOutArrayOfStructs(
        out int size, out IntPtr outArray);

    [DllImport("..\\LIB\\PInvokeLib.dll")]
    internal static extern void TestOutArrayOfStructs(
        out int size, MyUnsafeStruct** outArray);
}
```

Calling Functions

C#

```
public class App
{
    public static void Main()
    {
        Console.WriteLine("\nUsing marshal class\n");
        UsingMarshaling();
        Console.WriteLine("\nUsing unsafe code\n");
```

```
        Console.WriteLine("\nUsing unsafe code\n");
        UsingUnsafePointer();
    }

    public static void UsingMarshaling()
    {
        int size;
        IntPtr outArray;

        NativeMethods.TestOutArrayOfStructs(out size, out outArray);
        MyStruct[] manArray = new MyStruct[size];
        IntPtr current = outArray;
        for (int i = 0; i < size; i++)
        {
            manArray[i] = new MyStruct();
            Marshal.PtrToStructure(current, manArray[i]);

            //Marshal.FreeCoTaskMem((IntPtr)Marshal.ReadInt32(current));
            Marshal.DestroyStructure(current, typeof(MyStruct));
            current = (IntPtr)((long)current + Marshal.SizeOf(manArray[i]));

            Console.WriteLine("Element {0}: {1} {2}", i, manArray[i].buffer,
                manArray[i].size);
        }

        Marshal.FreeCoTaskMem(outArray);
    }

    public static unsafe void UsingUnsafePointer()
    {
        int size;
        MyUnsafeStruct* pResult;

        NativeMethods.TestOutArrayOfStructs(out size, &pResult);
        MyUnsafeStruct* pCurrent = pResult;
        for (int i = 0; i < size; i++, pCurrent++)
        {
            Console.WriteLine("Element {0}: {1} {2}", i,
                Marshal.PtrToStringAnsi(pCurrent->buffer), pCurrent->size);
            Marshal.FreeCoTaskMem(pCurrent->buffer);
        }

        Marshal.FreeCoTaskMem((IntPtr)pResult);
    }
}
```

See also

- Marshalling Data with Platform Invoke

- Marshalling Strings
- Marshalling Different Types of Arrays