.NET 2.0 Interoperability Recipes

A Problem-Solution Approach

Bruce Bukovics

Apress®

.NET 2.0 Interoperability Recipes: A Problem-Solution Approach

ISBN-13: 978-1-59059-669-2

ISBN-10: 1-59059-669-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Development Editor: Ralph Davis
Technical Reviewers: Christophe Nasarre, Nicholas Paldino
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Elizabeth Berry
Indexer: John Collin
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 2

C-Style APIs: Structures, Classes, and Arrays

Classes and structures allow you to create your own custom data types. Whether they are C++ or C# structs and classes, or Visual Basic .NET structures and classes, you can define complex types and implement business logic that would otherwise be either impossible or cumbersome. Custom data types free you from the restrictions of simple integers, floats, and doubles. Without the ability to define custom types, a developer's life would indeed be dull.

Likewise, arrays are an important tool for any developer. They allow you to operate on sets of like data rather than individual variables.

It is fitting then that this chapter is devoted to the handling of classes, structures, and arrays in Windows Interop. As it does for built-in data types, .NET provides the facilities you need to pass classes, structures, and arrays between managed and unmanaged code.

Structures used by both managed and unmanaged code are defined twice, once for each environment. In order to allow structures to pass successfully between the two environments, the layout of the structures must agree. This doesn't mean that all available fields must be defined in both places, since there are ways to define partial structures.

The first few recipes in this chapter show you how to use structures (C++/C# `structs`, Visual Basic .NET `Structures`) with PInvoke. Although structures are more limited than classes, and therefore not as widely used, they clearly illustrate all the necessary techniques for passing programmer-defined types. To complete the discussion, passing classes instead of structures is demonstrated in another recipe.

This chapter includes a number of recipes that demonstrate different ways to define and pass structures. Fields in a structure can use a sequential layout, or the position of each field can be individually controlled. Structures can be allocated in managed code and passed to unmanaged code, or vice versa.

One recipe demonstrates how to specify the way each field in a structure should be marshaled, while another discusses how to allocate and free memory for individual fields in a structure.

The chapter ends with recipes that focus on passing arrays of data between managed and unmanaged code. Simple arrays using built-in types are discussed, followed by arrays of strings and finally arrays of structures. Throughout the chapter, the use of the directional attributes (`In` and `Out`) is demonstrated.

2-1. Passing Structures

Problem

The unmanaged function you need to call requires a structure rather than individual fields to be passed. You need to know how to define and pass structures between managed and unmanaged code.

Solution

An unmanaged function that requires a structure has its own definition of the structure that it will use. Managed code that calls the function will need to supply an equivalent definition in managed code, allowing any managed code to reference the individual fields in the structure. The key to marshaling structures is properly aligning the structure definitions in managed and unmanaged code.

For example, consider this simple unmanaged function:

```
//function for structure passing
void ProcessStruct1(UnmanagedStruct1* aStruct)
{
    if (aStruct != NULL)
    {
        aStruct->UmCount    = 1;
        aStruct->UmDelta    = 2;
        aStruct->UmPercent  = 1.4567;
    }
}
```

The structure `UnmanagedStruct1` is defined like this in unmanaged code:

```
struct UnmanagedStruct1
{
    int    UmCount;
    char   UmTypeIndicator;
    int    UmDelta;
    double UmPercent;
};
```

In order to call this function from managed code, you need to define a managed structure that has the same memory layout as this structure. The implementation in C# might look like this:

```
//struct is properly aligned with unmanaged struct
public struct ManagedStruct1
{
    public int      maCount;
    byte            maUnused;
    public int      maDelta;
    public double   maPercent;
}
```

While the names of the individual fields in the structure may differ, the type and size of each field is the same. The sequence of fields is also the same, allowing you to use the default marshaling provided for structures.

The C# code that calls this unmanaged function is implemented in this way:

```
class StructurePassingTest
{
    [DllImport("FlatAPIStructLib.DLL")]
    public static extern void ProcessStruct1(
        ref ManagedStruct1 aStruct);

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        //create an instance of the struct
        ManagedStruct1 struct1  = new ManagedStruct1();
        struct1.maCount         = 12345;
        struct1.maDelta         = 45678;
        struct1.maPercent       = 5.4321;
        //call the unmanaged function
        ProcessStruct1(ref struct1);
        //show the results
        Console.WriteLine("ProcessStruct1 results: {0}, {1}, {2}",
            struct1.maCount, struct1.maDelta, struct1.maPercent);

        //wait for input
        Console.WriteLine("Press any key to exit");
        Console.Read();
    }
}
```

When we run the code, we receive these results:

```
ProcessStruct1 results: 1, 2, 1.4567
Press any key to exit
```

This works because the structure definitions in managed and unmanaged code are essentially the same. Specifically, the size, data type, and sequence of the individual fields in the structure are the same.

■ **Note** Notice that the unmanaged function expects to be passed a pointer to a structure that has been allocated by the caller of the function. It then uses the pointer to reference the individual fields of the structure.

Since the managed code is the caller of the function, it must create the structure (allocating the memory for it in managed code) and pass the structure by reference to the function. The unmanaged code will receive this as a pointer to the memory that was allocated.

The equivalent Visual Basic .NET code to run this same test could be implemented like this:

```
Imports System.Runtime.InteropServices  'needed for DllImport
Module StructurePassingVBClient

    Public Structure ManagedStruct1
        Public maCount As Integer
        Public maUnused As Byte
        Public maDelta As Integer
        Public maPercent As Double
    End Structure

    <DllImport("FlatAPIStructLib.DLL")> _
    Public Sub ProcessStruct1(ByRef aStruct As ManagedStruct1)
    End Sub

    Sub Main()
        'allocate the structure
        Dim struct1 As ManagedStruct1 = New ManagedStruct1
        struct1.maCount = 12345
        struct1.maDelta = 45678
        struct1.maPercent = 5.4321
        'call the unmanaged function that fills the struct
        ProcessStruct1(struct1)
        'show the results
        Console.WriteLine("ProcessStruct1 results: {0}, {1}, {2}", _
            struct1.maCount, struct1.maDelta, struct1.maPercent)
        'wait for input
        Console.WriteLine("Press any key to exit")
        Console.Read()
    End Sub

End Module
```

How It Works

The `StructLayout` attribute is provided to control the way structures are marshaled. The attribute is optional and is not necessary as long as the managed and unmanaged structure definitions agree.

`StructLayout` provides a number of fields that you can use to modify structure marshaling. Among them are the `LayoutKind` and `Pack` fields.

`LayoutKind` is an enumeration that determines the method used to control layout of the individual fields within the structure. The default value for a structure when the `StructLayout` attribute is omitted is `LayoutKind.Sequential`. This instructs the marshaler to assume that the individual fields of the structure should be laid out in the sequence in which they are defined. This default value works for the preceding example since the structures are defined with the same field sequence and each field is the same size in memory.

The managed structure used previously could have been defined this way with the same result:

```
[StructLayout(LayoutKind.Sequential)]
public struct ManagedStruct1
{
    public int      maCount;
    byte            maUnused;
    public int      maDelta;
    public double   maPercent;
}
```

The other possible value for `LayoutKind` is `Explicit`. When this value is used, you take complete control of the structure, specifying the exact offset of each field in memory. Use of this option is discussed in recipe 2-3 (Specifying the Exact Layout of a Structure).

The `Pack` field of the `StructLayout` attribute determines the alignment of the individual fields in the structure and is used only when `LayoutKind.Sequential` is specified. Depending on the pack size specified, individual fields may be internally aligned in a way that you would not expect.

Fields within a structure are always aligned on a boundary. That boundary is the smaller of a multiple of the pack size, or a multiple of the field size. If the pack size is not specified, the default is 8. This corresponds to the C++ default of 8 used within Visual C++ for unmanaged code.

Using the example structure, a default pack size of 8 would result in an internal representation like this:

```
struct UnmanagedStruct1
{
    int    UmCount;            <- Offset 0
    char   UmTypeIndicator;    <- Offset 4
    int    UmDelta;            <- Offset 8  (not 5 as expected)
    double UmPercent;          <- Offset 16 (not 12 as expected)
};
```

Since the default pack size is 8 for managed and unmanaged code, the marshaler took care of this little alignment detail. However, if the unmanaged code used a different pack size, we would need to adjust the managed definition of the structure to match. As an example, this unmanaged struct explicitly sets the pack size to 1:

```
#pragma pack(1)
//struct aligned on 1 byte boundary
struct UnmanagedStruct2
{
    int    UmCount;
    char   UmTypeIndicator;
    int    UmDelta;
    double UmPercent;
};
#pragma pack() //reset pack size to default
```

If we need to pass a managed version of this structure to a function, we would set the `Pack` field of the `StructLayout` attribute. A working managed definition of this structure looks like this:

```
//struct is aligned on 1 byte packing boundary
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct ManagedStruct2
{
    public int      maCount;
    byte            maUnused;
    public int      maDelta;
    public double   maPercent;
}
```

Failure to correctly match the pack size between managed and unmanaged structs will result in incorrect (or at best unpredictable) results.

Related Information

See recipe 2-3 (Specifying the Exact Layout of a Structure).

2-2. Returning a Structure from Unmanaged Code

Problem

An unmanaged function returns a structure rather than receiving one from the caller. You need to know how to receive the structure in managed code and to free the memory that was allocated.

Solution

When an unmanaged function expects to receive a pointer to a structure, the managed code is in charge. You allocate a managed structure in managed code and then pass it to the function for its own use.

However, structures that are allocated within unmanaged code and returned to the caller must be handled differently. Since unmanaged memory is allocated within the function, it must be freed by the caller (the managed code) when it is no longer needed.

The solution is to return the pointer to the structure as an `IntPtr` and marshal this to the structure within managed code. You can then pass the returned `IntPtr` back to another unmanaged function to free the memory.

Consider this unmanaged function:

```
ReturnedUnmanagedStruct* ReturnAStruct(void)
{
    //allocate the struct using C++ new keyword
    ReturnedUnmanagedStruct* pResult = new ReturnedUnmanagedStruct();
    pResult->Hours          = 1;
    pResult->Minutes        = 59;
    pResult->Seconds        = 11;
    return pResult;
}
```

The function allocates memory for a structure using the C++ `new` keyword, populates the structure, and returns it to the caller as a pointer. The structure is defined like this:

```
struct ReturnedUnmanagedStruct
{
    int    Hours;
    int    Minutes;
    int    Seconds;
};
```

We also need an unmanaged function that we can call to free the memory that was previously allocated. This simple function is implemented like this:

```
void FreeAStruct(ReturnedUnmanagedStruct* pStruct)
{
    if (pStruct != NULL)
    {
        delete pStruct;
    }
}
```

We can successfully call this function from managed C# code if we handle the returned pointer as an `IntPtr` like this:

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

/// <summary>
/// Returning of structures from unmanaged code
/// </summary>
class StructureReturningTest
{
```

```
    [DllImport("FlatAPIStructLib.DLL")]
    public static extern IntPtr ReturnAStruct();

    [DllImport("FlatAPIStructLib.DLL")]
    public static extern void FreeAStruct(IntPtr structPtr);

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        //call the unmanaged function returning a struct
        IntPtr structPtr = ReturnAStruct();
        //marshal the returned pointer to a struct
        ReturnedManagedStruct aStruct
            = (ReturnedManagedStruct)Marshal.PtrToStructure(
                structPtr, typeof(ReturnedManagedStruct));
        //free the memory for the unmanaged struct
        FreeAStruct(structPtr);
        //show the results
        Console.WriteLine("ReturnAStruct results: {0}, {1}, {2}",
            aStruct.Hours, aStruct.Minutes, aStruct.Seconds);

        //wait for input
        Console.WriteLine("Press any key to exit");
        Console.Read();
    }
}
```

We define the managed version of the structure this way, matching the layout defined in unmanaged code:

```
public struct ReturnedManagedStruct
{
    public int      Hours;
    public int      Minutes;
    public int      Seconds;
}
```

Looking through the code, we see that the unmanaged function is declared as returning an `IntPtr`. When we make the call to the function, we are able to use the `Marshal.PtrToStructure` static method to marshal the `IntPtr` to the managed version of the structure. This copies (and converts if necessary) the data that the `IntPtr` points to. Using the `IntPtr`, we then call the unmanaged `FreeAStruct` method that frees the memory, avoiding a memory leak.

A Visual Basic .NET version of this code looks like this:

```
Imports System
Imports System.Runtime.InteropServices
```

```
Module StructureReturningTest

    Public Structure ReturnedManagedStruct
        Public Hours As Integer
        Public Minutes As Integer
        Public Seconds As Integer
    End Structure

    <DllImport("FlatAPIStructLib.DLL")> _
    Public Function ReturnAStruct() As IntPtr
    End Function

    <DllImport("FlatAPIStructLib.DLL")> _
    Public Sub FreeAStruct(ByVal structPtr As IntPtr)
    End Sub

    Sub Main()
        'call the unmanaged function returning a struct
        Dim structPtr As IntPtr = ReturnAStruct()
        'marshal the returned pointer to a struct
        Dim aStruct As ReturnedManagedStruct _
            = Marshal.PtrToStructure( _
                structPtr, GetType(ReturnedManagedStruct))
        'free the memory for the unmanaged struct
        FreeAStruct(structPtr)
        'show the results
        Console.WriteLine( _
            "ReturnAStruct results: {0}, {1}, {2}", _
            aStruct.Hours, aStruct.Minutes, aStruct.Seconds)

        'wait for input
        Console.WriteLine("Press any key to exit")
        Console.Read()

    End Sub

End Module
```

When this code is executed, the results are exactly the same as those for the C# version.

How It Works

As an alternative to calling a second unmanaged function to free memory, we can modify the original function to use `CoTaskMemAlloc` instead of `new`. Once we do this, we are able to free the memory from managed code using the `Marshal` class.

For example, the function discussed previously could be rewritten this way:

```
ReturnedUnmanagedStruct* ReturnCoMemStruct(void)
{
    //allocate the struct using CoTaskMemAlloc
```

```
    ReturnedUnmanagedStruct* pResult =
       (ReturnedUnmanagedStruct*)CoTaskMemAlloc(
            sizeof(ReturnedUnmanagedStruct));
    pResult->Hours          = 1;
    pResult->Minutes        = 59;
    pResult->Seconds        = 11;
    return pResult;
}
```

The only substantial difference between this and the original version is the use of `CoTaskMemAlloc` to allocate the structure. `CoTaskMemAlloc` allocates memory that we can free directly from managed code.

The managed code to call this revised function looks like this in C#:

```
[DllImport("FlatAPIStructLib.DLL")]
public static extern IntPtr ReturnCoMemStruct();

//call the unmanaged function returning a CoTaskMemAlloc struct
IntPtr structPtr = ReturnCoMemStruct();
//marshal the returned pointer to a struct
ReturnedManagedStruct bStruct
    = (ReturnedManagedStruct)Marshal.PtrToStructure(
        structPtr, typeof(ReturnedManagedStruct));
//free the CoTaskMemAlloc memory
Marshal.FreeCoTaskMem(structPtr);
//show the results
Console.WriteLine("ReturnCoMemStruct results: {0}, {1}, {2}",
    bStruct.Hours, bStruct.Minutes, bStruct.Seconds);
```

It is implemented like this in Visual Basic .NET:

```
<DllImport("FlatAPIStructLib.DLL")> _
Public Function ReturnCoMemStruct() As IntPtr
End Function

'call the unmanaged function
'returning a CoTaskMemAlloc struct
structPtr = ReturnCoMemStruct()
'marshal the returned pointer to a struct
Dim bStruct As ReturnedManagedStruct _
    = Marshal.PtrToStructure( _
        structPtr, GetType(ReturnedManagedStruct))
'free the CoTaskMemAlloc memory
Marshal.FreeCoTaskMem(structPtr)
'show the results
Console.WriteLine( _
    "ReturnCoMemStruct results: {0}, {1}, {2}", _
    bStruct.Hours, bStruct.Minutes, bStruct.Seconds)
```

Using `CoTaskMemAlloc` doesn't eliminate our responsibility to free the memory. However, we can easily take care of that duty using the static `FreeCoTaskMem` method of the `Marshal` class.

The output from the tests shows that regardless of the memory allocation method, we get the same results:

```
ReturnAStruct results: 1, 59, 11
ReturnCoMemStruct results: 1, 59, 11
Press any key to exit
```

Related Information

See recipes 2-1 (Passing Structures) and 1-11 (Freeing Unmanaged Memory).

2-3. Specifying the Exact Layout of a Structure

Problem

An unmanaged function requires a large structure as an input parameter; however, you are concerned with only one or two fields in the structure. You need a way to pass a partially defined structure to the function.

Solution

The `StructLayout` attribute can be applied to the managed version of a structure to control the overall approach used to marshal the structure. By specifying `LayoutKind.Explicit` in the attribute constructor, you take complete control of how the structure is marshaled to unmanaged code. Use of the `FieldOffset` attribute on each field in the structure is required to indicate the exact position within the structure.

For example, consider this unmanaged structure that defines a number of fields:

```
#pragma pack(1)
//structure for account info retrieval
struct UnmanagedAccountStruct
{
    int    AccountId;           //4 bytes
    int    AccountStatus;       //4 bytes
    short  AccountAgingMethod;  //2 bytes
    short  AccountType;         //2 bytes
    int    RegionId;            //4 bytes
    double CurrentBalance;      //8 bytes
    double PastDueBalance;      //8 bytes
    int    SalesRepId;          //4 bytes
    char*  AccountName;         //4 bytes
    char*  Address1;            //4 bytes
    char*  Address2;            //4 bytes
    char*  City;                //4 bytes
    char*  State;               //4 bytes
    int    PostalCode;          //4 bytes
    double LastPurchaseAmt;     //8 bytes
};
#pragma pack() //reset pack size to default
```

The total size of this unmanaged structure is 68 bytes.

We may not need to use all of these fields in our managed code. Alternatively, multiple unmanaged functions may each populate only a subset of the fields.

■ **Note** In this example, the pack boundary size is set to 1. The pack size determines the layout of individual fields, placing fields internally on a multiple of the specified boundary or a multiple of the field size.

For this example, we choose a pack size of 1 simply to make the calculation of field offsets more obvious. In the real world, other pack sizes are likely, and the calculation of the actual field offset may become much more difficult.

Using the `StructLayout` and `FieldOffset` attributes, we can define a managed version of this structure in C# that contains only the fields that concern us:

```
//partial structure definition
[StructLayout(LayoutKind.Explicit)]
public struct AccountBalanceStruct
{
    [FieldOffset(0)]    public int      AccountId;
    [FieldOffset(16)]   public double   CurrentBalance;
    [FieldOffset(24)]   public double   PastDueBalance;
    [FieldOffset(60)]   public double   LastPurchaseAmt;
}
```

Here is the same structure in Visual Basic .NET:

```
'partial structure definition
<StructLayout(LayoutKind.Explicit)> _
Public Structure AccountBalanceStruct
    <FieldOffset(0)> Public AccountId As Integer
    <FieldOffset(16)> Public CurrentBalance As Double
    <FieldOffset(24)> Public PastDueBalance As Double
    <FieldOffset(60)> Public LastPurchaseAmt As Double
End Structure
```

The use of `LayoutKind.Explicit` informs the marshaler that we will be specifying `FieldOffset` attributes for each field in the structure.

The `FieldOffset` attribute specifies the offset of the field from the beginning of the unmanaged structure, not the managed structure. If we calculated the offsets of each field correctly, the fields should correspond to the unmanaged structure.

How It Works

To complete the example started previously, we can implement an unmanaged function that uses a portion of this structure. The C++ code looks like this:

```
void RetrieveAccountBalances(int accountId,
    UnmanagedAccountStruct* pAccount)
```

```
{
    //return selected account fields
    if (pAccount != NULL)
    {
        pAccount->AccountId         = accountId;
        pAccount->AccountType       = sizeof(char*);
        pAccount->CurrentBalance    = 500.00;
        pAccount->PastDueBalance    = 350.00;
        pAccount->LastPurchaseAmt   = 10.95;
    }
}
```

The function expects an integer to identify the account ID and a pointer to the unmanaged struct defined previously. Presumably, the function would access a database or other data store to retrieve account information (those minor details don't affect this example). A subset of fields within the structure is then populated, making the values available to the calling code.

The complete C# implementation of code that accesses this function looks like this:

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

namespace StructureExactLayoutTest
{
    //partial structure definition
    [StructLayout(LayoutKind.Explicit)]
    public struct AccountBalanceStruct
    {
        [FieldOffset(0)]    public int      AccountId;
        [FieldOffset(16)]   public double   CurrentBalance;
        [FieldOffset(24)]   public double   PastDueBalance;
        [FieldOffset(60)]   public double   LastPurchaseAmt;
    }

    /// <summary>
    /// Passing of structures between managed and unmanaged code
    /// </summary>
    class StructureExactLayoutTest
    {
        [DllImport("FlatAPIStructLib.DLL")]
        public static extern void RetrieveAccountBalances(
            int accountId, ref AccountBalanceStruct account);

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
```

```
        {
            //uses a partially defined managed structure
            AccountBalanceStruct account = new AccountBalanceStruct();
            //make the unmanaged function call
            RetrieveAccountBalances(1001, ref account);
            //show the results
            Console.WriteLine(
                "RetrieveAccountBalances results: {0}, {1}, {2}, {3}",
                    account.AccountId, account.CurrentBalance,
                    account.PastDueBalance, account.LastPurchaseAmt);

               //wait for input
            Console.WriteLine("Press any key to exit");
            Console.Read();
        }
    }
}
```

The Visual Basic .NET implementation looks like this:

```
Imports System
Imports System.Runtime.InteropServices

''' <summary>
''' Passing of structures between managed and unmanaged code
''' </summary>
''' <remarks></remarks>
Module StructureExactLayoutTest

    'partial structure definition
    <StructLayout(LayoutKind.Explicit)> _
    Public Structure AccountBalanceStruct
        <FieldOffset(0)> Public AccountId As Integer
        <FieldOffset(16)> Public CurrentBalance As Double
        <FieldOffset(24)> Public PastDueBalance As Double
        <FieldOffset(60)> Public LastPurchaseAmt As Double
    End Structure

    <DllImport("FlatAPIStructLib.DLL")> _
    Public Sub RetrieveAccountBalances( _
        ByVal accountId As Integer, _
        ByRef account As AccountBalanceStruct)
    End Sub

    Sub Main()
        'uses a partially defined managed structure
        Dim account As AccountBalanceStruct _
            = New AccountBalanceStruct()
        'make the unmanaged function call
```

```
        RetrieveAccountBalances(1001, account)
        'show the results
        Console.WriteLine( _
            "RetrieveAccountBalances results: {0}, {1}, {2}, {3}", _
            account.AccountId, account.CurrentBalance, _
            account.PastDueBalance, account.LastPurchaseAmt)

        'wait for input
        Console.WriteLine("Press any key to exit")
        Console.Read()
    End Sub

End Module
```

Regardless of the language used, we see these results when the code is executed:

```
RetrieveAccountBalances results: 1001, 500, 350, 10.95
Press any key to exit
```

In this example, the managed version of the structure (`AccountBalanceStruct`) does not define all of the fields in the unmanaged version. However, notice that it does define the last field named `LastPurchaseAmt` at field offset 60. Since this final field has a size of 8 bytes, the overall size of the managed structure is the same as the unmanaged version: 68 bytes. The field with the largest `FieldOffset` determines the overall size of the structure.

The unmanaged code will thus receive a structure that has been initialized to the full length of 68 bytes. All memory in the unmanaged structure is first cleared. Fields that are not defined in the managed structure are seen as zeros in the unmanaged code. The values of all fields that are defined in the managed structure are marshaled to the unmanaged version.

We must be careful when defining the managed version of a structure. If we omit the last field in a structure, the unmanaged function will receive a structure that has not been completely initialized.

We can illustrate this behavior using this unmanaged function:

```
double RevisePurchaseAmt(UnmanagedAccountStruct* pAccount,
    double purchaseAmt)
{
    double lastPurchaseAmt;
    //revise the LastPurchaseAmt
    if (pAccount != NULL)
    {
        //save the LastPurchaseAmt
        lastPurchaseAmt = pAccount->LastPurchaseAmt;
        //update with the new amount
        pAccount->LastPurchaseAmt = purchaseAmt;
    }
    return lastPurchaseAmt;
}
```

This example uses the same `UnmanagedAccountStruct` used in the last example. The function is passed a pointer to this structure along with a double. The `LastPurchaseAmt` field in the structure is updated with the value that is passed to the function. The original value of the `LastPurchaseAmt` field is the return value of the function.

In the C# code, we define a version of the structure that deliberately omits the final `LastPurchaseAmt` field:

```
[StructLayout(LayoutKind.Explicit)]
public struct AccountBalanceStructShort
{
    [FieldOffset(0)]     public int       AccountId;
    [FieldOffset(16)]    public double    CurrentBalance;
    [FieldOffset(24)]    public double    PastDueBalance;

    //LastPurchaseAmt field is omitted
}
```

The C# declaration of the function that uses this managed structure looks like this:

```
[DllImport("FlatAPIStructLib.DLL")]
public static extern double RevisePurchaseAmt(
    ref AccountBalanceStructShort account,
    double purchaseAmt);
```

The code to test this function looks like this:

```
AccountBalanceStructShort accountShort
    = new AccountBalanceStructShort();
double lastPurchaseAmt
    = RevisePurchaseAmt(ref accountShort, 249.95);
Console.WriteLine(
    "RevisePurchaseAmt results: {0}",
    lastPurchaseAmt);
```

Since we create a new instance of the structure before we pass it to the function, we would expect to receive a return value of zero. The `LastPurchaseAmt` field that is returned from the function isn't defined in the managed version of the structure, so it should have a value of zero, right? Wrong. When we execute this code using the debugger, we see something similar to these results (the exact results shown on your machine may be different):

```
RevisePurchaseAmt results: 5.34315632523118E-315
```

Because the managed structure defined is shorter than the expected length of the unmanaged structure, the unmanaged code received memory that wasn't completely initialized. In this case, the `LastPurchaseAmt` field at the very end of the structure contains garbage.

■ **Note** It's interesting to note that we see these results only when we run the code under the Visual Studio .NET debugger. When we execute the same code outside of the debugger, the fields appear to be initialized.

In this function we are simply returning the value of this field. Really bad things would happen if the function actually used this field for something useful—perhaps for some type of calculation.

The `StructLayout` attribute has an optional `Size` field that corrects this problem. If `Size` is specified, it extends the memory that is passed to unmanaged code beyond the actual size of the defined fields. The `Size` field can only increase the total size of the structure; it cannot be used to make it smaller. Therefore, `Size` must be equal to or greater than the total size of the structure based on the defined fields.

To see the use of the `Size` field, we can modify the C# structure to look like this:

```
//partial structure definition
[StructLayout(LayoutKind.Explicit, Size = 68)]
public struct AccountBalanceStructSize
{
    [FieldOffset(0)]     public int       AccountId;
    [FieldOffset(16)]    public double    CurrentBalance;
    [FieldOffset(24)]    public double    PastDueBalance;

    //LastPurchaseAmt field is omitted
}
```

This structure uses the `Size` field to indicate that a total of 68 bytes of memory should be initialized and passed to unmanaged code. Notice that the structure is renamed to avoid confusion with the last example.

We now declare the unmanaged function again, this time using the revised structure:

```
[DllImport("FlatAPIStructLib.DLL",
    EntryPoint = "RevisePurchaseAmt")]
public static extern double RevisePurchaseAmtFull(
    ref AccountBalanceStructSize account,
    double purchaseAmt);
```

The revised C# code to use this version of the structure looks like this:

```
AccountBalanceStructSize accountWithSize
    = new AccountBalanceStructSize();
double lastPurchaseAmt
    = RevisePurchaseAmtFull(ref accountWithSize, 249.95);
Console.WriteLine(
    "RevisePurchaseAmtFull results: {0}",
    lastPurchaseAmt);
```

This time when we execute the code, we see the expected results:

```
RevisePurchaseAmtFull results: 0
```

Because of this behavior, it's actually a good idea to always include the `Size` field when you use `LayoutKind.Explicit`, especially for a structure that is only partially defined. The `Size` field isn't required, but including it will help to avoid potential problems like this.

Related Information

See recipe 2-1 (Passing Structures).

2-4. Controlling Field-Level Marshaling Within Structures

Problem

You need to pass a structure to unmanaged code that includes strings of multiple types. Some of the strings are defined as ANSI strings while others are Unicode. The `CharSet` field of the `DllImport` attribute won't do the trick, since it affects all strings in the function call, not individual fields within a structure.

Solution

When using structures, you can apply the `MarshalAs` attribute to control how individual fields in the structure are marshaled. The `MarshalAs` attribute is applied to those fields that require nondefault marshaling behavior.

The `MarshalAs` attribute allows you to specify the unmanaged type that is mapped to the managed field during marshaling. While you are free to apply this attribute to every field of every structure, in practice you only need it in cases where the data type conversion is ambiguous.

For example, a managed `System.String` can be marshaled in a number of ways, either as an ANSI string, a Unicode string, or a COM BSTR. Likewise, a `System.Boolean` could be marshaled as a C++-style `bool` (1 byte) or a Win32 `BOOL` (a 4-byte integer). In cases such as this, you may need to apply the `MarshalAs` attribute.

For example, this unmanaged structure contains a number of fields that demonstrate this problem:

```
struct UnmanagedAmbiguousStruct
{
    char*    AnsiString;
    wchar_t* WideString;
    BOOL     Win32Boolean;      //4 bytes
    bool     CStyleBoolean;     //1 byte
    unsigned short ShortInteger;
};
```

We have deliberately created a problem for ourselves by including both types of string and Boolean fields.

A managed version of this structure could be implemented like this in C#:

```
[StructLayout(LayoutKind.Sequential)]
struct ManagedAmbiguousStruct
{
    [MarshalAs(UnmanagedType.LPStr)]  public string AnsiString;
    [MarshalAs(UnmanagedType.LPWStr)] public string WideString;
    [MarshalAs(UnmanagedType.Bool)]   public bool   Win32Boolean;
```

```
    [MarshalAs(UnmanagedType.I1)]     public bool   CStyleBoolean;
    public ushort   ShortInteger;
};
```

Here is the same structure in Visual Basic .NET:

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure ManagedAmbiguousStruct
    <MarshalAs(UnmanagedType.LPStr)> _
    Public AnsiString As String
    <MarshalAs(UnmanagedType.LPWStr)> _
    Public WideString As String
    <MarshalAs(UnmanagedType.Bool)> _
    Public Win32Boolean As Boolean
    <MarshalAs(UnmanagedType.I1)> _
    Public CStyleBoolean As Boolean
    Public ShortInteger As UShort
End Structure
```

By applying the `MarshalAs` attribute to each field, we remove the ambiguity. The marshaler now has explicit instructions from us as to how we want these fields converted to unmanaged values.

■ **Note** The `MarshalAs` attribute supports a large number of unmanaged data types. This example illustrates the use of attribute values that modify string and Boolean fields. Please consult the Microsoft documentation for a complete list of available unmanaged types.

How It Works

As demonstrated in this example structure, use of the `MarshalAs` attribute is not an all-or-nothing matter. You are free to mix and match the use of the attribute as needed, applying it only in cases where it is truly required. In the example, the `ShortInteger` field does not use the `MarshalAs` attribute; instead, it uses the default marshaling.

Furthermore, we could simplify the example structure slightly if we want to totally rely upon default marshaling behavior. The default marshaling for a string is to an ANSI string, and the default for a Boolean is to a 4-byte integer (Win32 `BOOL`). Therefore, we can omit the `MarshalAs` attributes from those fields without any adverse affect. The revised structure could look like this:

```
[StructLayout(LayoutKind.Sequential)]
struct ManagedAmbiguousStruct
{
    public string   AnsiString;
    [MarshalAs(UnmanagedType.LPWStr)] public string   WideString;
    public bool     Win32Boolean;
    [MarshalAs(UnmanagedType.I1)]     public bool     CStyleBoolean;
    public ushort   ShortInteger;
};
```

However, there is a benefit to being explicit, and in this case the inclusion of the `MarshalAs` attribute on the fields makes it clearer how things will be marshaled.

To illustrate the use of this structure, we can implement an unmanaged function that uses it:

```
int UseAmbiguousStruct(UnmanagedAmbiguousStruct aStruct)
{
    int result = 0;
    result += (int)strlen(aStruct.AnsiString);
    result += (int)wcslen(aStruct.WideString);
    if (aStruct.CStyleBoolean)
    {
        result++;
    }
    if (aStruct.Win32Boolean)
    {
        result++;
    }
    result += aStruct.ShortInteger;

    return result;
}
```

For testing purposes only, the function retrieves the length of each string and checks the value of each Boolean field. An integer with the results is returned.

The complete C# code to execute this function is implemented in this way:

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

namespace StructureMarshalAsTest
{
    [StructLayout(LayoutKind.Sequential)]
    struct ManagedAmbiguousStruct
    {
        [MarshalAs(UnmanagedType.LPStr)]
        public string   AnsiString;
        [MarshalAs(UnmanagedType.LPWStr)]
        public string   WideString;
        [MarshalAs(UnmanagedType.Bool)]
        public bool     Win32Boolean;
        [MarshalAs(UnmanagedType.I1)]
        public bool     CStyleBoolean;
        public ushort   ShortInteger;
    };
```

```
    /// <summary>
    /// Passing of structures between managed and unmanaged code
    /// </summary>
    class StructureMarshalAsTest
    {
        [DllImport("FlatAPIStructLib.DLL")]
        public static extern int UseAmbiguousStruct(
            ManagedAmbiguousStruct aStruct);

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            ManagedAmbiguousStruct mStruct
                = new ManagedAmbiguousStruct();
            mStruct.AnsiString          = "ansistring";
            mStruct.WideString          = "widestring";
            mStruct.CStyleBoolean       = true;
            mStruct.Win32Boolean        = true;
            mStruct.ShortInteger        = 5;

            int result = UseAmbiguousStruct(mStruct);

            //show the results
            Console.WriteLine("UseAmbiguousStruct results: {0}",
                result);

            //wait for input
            Console.WriteLine("Press any key to exit");
            Console.Read();
        }
    }
}
```

The Visual Basic .NET implementation looks like this:

```
Imports System
Imports System.Runtime.InteropServices

Module StructureMarshalAsTest

    <StructLayout(LayoutKind.Sequential)> _
    Public Structure ManagedAmbiguousStruct
```

```
        <MarshalAs(UnmanagedType.LPStr)> _
        Public AnsiString As String
        <MarshalAs(UnmanagedType.LPWStr)> _
        Public WideString As String
        <MarshalAs(UnmanagedType.Bool)> _
        Public Win32Boolean As Boolean
        <MarshalAs(UnmanagedType.I1)> _
        Public CStyleBoolean As Boolean
        Public ShortInteger As UShort
    End Structure

    <DllImport("FlatAPIStructLib.DLL")> _
    Public Function UseAmbiguousStruct( _
        ByVal aStruct As ManagedAmbiguousStruct) _
        As Integer
    End Function

    Sub Main()
        Dim mStruct As ManagedAmbiguousStruct _
            = New ManagedAmbiguousStruct()
        mStruct.AnsiString = "ansistring"
        mStruct.WideString = "widestring"
        mStruct.CStyleBoolean = True
        mStruct.Win32Boolean = True
        mStruct.ShortInteger = 5

        Dim result As Integer = UseAmbiguousStruct(mStruct)

        'show the results
        Console.WriteLine( _
            "UseAmbiguousStruct results: {0}", result)

        'wait for input
        Console.WriteLine("Press any key to exit")
        Console.Read()
    End Sub

End Module
```

It should be noted that the `StructLayout` attribute and the `DllImport` attribute both contain a `CharSet` field. The purpose of this field is to specify the way strings are marshaled to unmanaged code. The `StructLayout.CharSet` field controls string marshaling for the structure, while `DllImport.CharSet` affects all strings in the function call.

However, the `CharSet` field won't solve the problem created here. If we were to rely upon the `CharSet` field, it would affect all strings in the structure. The problem at hand is the use of both types of strings within the same structure. The `CharSet` field is not the correct solution to this problem, but it might be useful in other scenarios.

DEBUGGING ENTRYPOINTNOTFOUNDEXCEPTION

When first working with structures, it is not uncommon to receive an `EntryPointNotFoundException` during testing. While this exception may appear to indicate an incorrect DLL or function name, the root cause is more typically an incorrectly defined structure.

If a structure is one of the input or output arguments for a function, depending on the calling convention, it forms part of the function signature. For example, if the `UseAmbiguousStruct` method uses the `__stdcall` calling convention, it is exported from the unmanaged DLL with a decorated entry point of `_UseAmbiguousStruct@16`. The `16` in this case represents the expected number of bytes in the structure passed as an argument. The number of bytes in the structure is affected by the packing size and the order of fields in the structure. PInvoke uses the size of the managed structure to determine the decorated entry point name for the call. If the managed and unmanaged versions of the structure do not match, PInvoke won't necessarily tell you that the structures don't match. Instead, PInvoke will inform you that no entry point using the parameters that you have defined (the incorrect structure) can be found.

If you have decorated individual fields in the structure with the `MarshalAs` attribute, the attribute settings should be reviewed along with the data types of the structure fields. The `MarshalAs` attribute does affect the PInvoke view of the function signature, and an incorrect `MarshalAs` attribute may be the root cause of the problem.

For example, we might incorrectly define the structure used in the example code like this:

```
[StructLayout(LayoutKind.Sequential)]
struct ManagedAmbiguousStruct
{
    public ushort ShortInteger;
    [MarshalAs(UnmanagedType.LPStr)]
    public string AnsiString;
    [MarshalAs(UnmanagedType.LPWStr)]
    public string WideString;
    [MarshalAs(UnmanagedType.Bool)]
    public bool Win32Boolean;
    [MarshalAs(UnmanagedType.I1)]
    public bool CStyleBoolean;

};
```

This incorrectly places the `ShortInteger` field at the beginning of the structure. If we make this change and run the sample code again, an `EntryPointNotFoundException` will be thrown with this message: "Unable to find an entry point named 'UseAmbiguousStruct' in DLL 'FlatAPIStructLib.DLL".

After comparing the two versions of the structure and correcting any errors, the exception should be eliminated. If the function uses `__cdecl` instead of `__stdcall`, we would receive a different result. The entry point would be found and the call would occur. However, since the managed and unmanaged structures do not agree, the unmanaged function would receive and act upon incorrect data.

Related Information

See recipe 1-6 (Changing the Character Set Used for Strings).