<http://www.csharphelp.com/archives/archive63.html>

**Call Unmanaged Code Part 2 - Marshal Class**  
By [Vyacheslav Biktagirov](http://www.csharphelp.com/bio/vbiktagirov.html)

|  |
| --- |
| In previous part we seen, how we can interop with unmanaged code and some simple examples. We observed MarshalAs attribute, that helps us to work with a lot of different unmanaged types,like LPStruct,LPArray,LPStr,LPWStr and others.  But, you can ask,- there is unlimited(!) number of possible unmanaged types, while number of different MarshalAs variables is limited, of course. So, how we can marshal any, any type? How we can manage non-standart application-specific types like, for instance, char\*\*\*\*(pointer to pointer to pointer to string)? What about more complicated cases?  I can see 3 possible ways of doing such a thing in C#:  1. MarshalAs Attribute Combining different special-defined managed user types with complex metadata attributes with metadata in external function declaration. In most cases it will work. This way is nice & clean, but will not work for some cases and pretty complex.  2. Unsafe code Write unsafe external function declarations with same types as in unmanaged library and than call there functions from unsafe code. Will work for any case, almost simple to implement. Anyway, avoid this way, becouse "unsafe" code is unsafe - with all what it means.  3. Marshal class There is a special class that can do all possible interop with unmanaged code. In some cases it can be not so simple to use, but when you have no choice - it's a good choice. It can take any unmanaged object, allocate/deallocate it, and move data between managed and unmanaged objects. In cases of complex interop, use of Marshal class is more simple, that first possibility - MarshalAs attribute.  Conclusion:  When you have some functions to call in unmanaged code, try first using MarshalAs attribute. For simple interop it's simple, nice and work without surprices, becouse it's standart mechanism. For cases of complex interop, try first also possibility 1. When it's complex or takes a lot of time to find proper combinetion of attributes, try possibiltity 3 - it's more simple and gives more control, but in this case programmer forced to allocate/deallocate/copy unmanaged memory, that probably unsafe. Of course, possibility 2 is opened always, but be careful using it.  Uff.. After a bit of theory, let's see some examples.  Example 1 : Print char\*\*\*\*  Imagine that I wrote some unmanaged DLL in C++,for example, contains function: |

extern "C" \_\_declspec(dllexport) void GetUserMessage(char\*\*\*\* Message)

{

strcpy(\*\*\*Message,"Hello unmanaged");

}

|  |
| --- |
| Imagine also, that we want call this fuction from C# and present a message box with this beautiful string. Point that before we can call this function, we must appropriately allocate the memory and, after we call it, it will be just nice to free it. Lets's try all 3 approaches one after one.  Example 1 : Approach 1 - MarshalAs attribute |

Hello, I'm glad to say you, that I see no any "legal" way to marshal "char\*\*\*\*". Do you?

Some facts:

\* In Beta1 was such way(define multi-nested class), but not in Beta2

\* There is a way to marshal char\*\*\* only.

Ok, we need to difine some helper method, that will cast down one level:

extern "C" \_\_declspec(dllexport) void GetUserMessageHelper(char\*\*\*\* Message)

{

GetUserMessage(&Message);

}

Now we can write .NET declarstions:

[StructLayout(LayoutKind.Sequential)]

public class StringP

{

[MarshalAs(UnmanagedType.LPStr)]

public string str;

}

|  |
| --- |
| This declaration will add one level of indirectance. Put attention, that we defined class, and not struct, becouse struct we can marshal by-value only, ans class also by reference. Class to be mashalled must expose layout information. In our case it does not make sence, becouse class has one member only, but it's MarshalAsAttribute class restriction. And, of course, we must say, how we want to treat a string in unmanaged code - as LPStr.  Now we can declare the function: |

[DllImport("TestDll.dll",EntryPoint="GetUserMessageHelper",CharSet=CharSet.Ansi)]

static extern void GetUserMessageHelper(

[MarshalAs(UnmanagedType.LPStruct)]

ref StringP str);

|  |
| --- |
| EntryPoint and CharSet is not so necessary in out case, becouse we explicitely sayd, that we want 'LPStr' and defined our function as 'extern "C"'. Impotrant is define parameter as 'ref', says input and output(it add last,third level of indirectance) and say that we want marshal our class by reference - LPStruct and not by value. |

The work is done, just call it:

private void button1\_Click(object sender, System.EventArgs e)

{

StringP strP=new StringP();

strP.str=new string('\*',20);

GetUserMessageHelper(ref strP);

MessageBox.Show(strP.str);

}

Not so simple, but works fine, not?

Example 1 : Approach 2 - unsafe code  
Let's define our extern function as unsafe:

[DllImport("TestDll.dll",EntryPoint="GetUserMessage")]

unsafe static extern void GetUserMessage2(byte\*\*\*\* str);

[DllImport("user32.dll")]

unsafe static extern int MessageBoxA(int hwnd,byte\* Text,byte\* Caption,int Type);

I defined parameter like 'byte\*\*\*\*' and not 'char\*\*\*\*', becouse managed

char is 2-byte long..

I also explicitely sayd that my function is unsafe, else I can't use pointer

semantic.

I imported also MessageBox function, becouse managed MessageBox class don't

know to operate with byte pointer, but only with managed string class. Now use it:

unsafe private void button2\_Click(object sender, System.EventArgs e)

{

byte\* str=(byte\*)Marshal.AllocCoTaskMem(20);

byte\*\* strP=&str;

byte\*\*\* strPP=&strP;

GetUserMessage2(&strPP);

MessageBoxA(0,str,(byte\*)0,0);

Marshal.FreeCoTaskMem((IntPtr)str);

}

|  |
| --- |
| I used Marshal class for memory allocation. For good example I must import "CoTaskMemAlloc" and "CoTaskMemFree" from "ole32.dll" and use them. But me is lazy, so please excuse me for just using Marshal class. So what happen here? I allocated 20 bytes from unmanaged heap, that called GetUserMessage with 'pointer to pointer to pinter', use the result and freed the memory. Very,very simple.But, like all unmanaged operations, it waits from developer to be ver careful..  Example 1 : Approach 3 - Marshal class |

[DllImport("TestDll.dll",EntryPoint="GetUserMessage")]

static extern void GetUserMessage3(ref IntPtr str);

IntPtr is just system-deined integer(like C++ int). For Win32, it's 4-byte length.

Let's use Marshal class:

private void button3\_Click(object sender, System.EventArgs e)

{

// Ok, of course firstly allocate memory

IntPtr str=Marshal.AllocCoTaskMem(20);

// add level of indirection

IntPtr strP=Marshal.AllocCoTaskMem(4);

Marshal.StructureToPtr(str,strP,false);

// add one more level of indirection

IntPtr strPP=Marshal.AllocCoTaskMem(4);

Marshal.StructureToPtr(strP,strPP,false);

// call it

GetUserMessage3(ref strPP);

// remove 2 levels of indirection

strP=Marshal.ReadIntPtr(strPP);

str=Marshal.ReadIntPtr(strP);

// get string class from pointer

MessageBox.Show(Marshal.PtrToStringAnsi(str));

// Free used memory

Marshal.FreeCoTaskMem(str);

Marshal.FreeCoTaskMem(strP);

Marshal.FreeCoTaskMem(strPP);

}

|  |
| --- |
| Ok, only point of this code it that is no 'address of' operation like C++ '&'. So we use Marshal class to get this functionality. For adding indirection level, we must allocate memory for new pointer and that ask Marhal class to put the pointer to given location.  Such method is more safe that using unsafe code.  Here is full program code that explains all 3 methods of marshalling 'char\*\*\*\*': |

using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.[Windows[http://images.intellitxt.com/ast/adTypes/mag-glass_10x10.gif](http://www.csharphelp.com/archives/archive63.html)](http://www.csharphelp.com/archives/archive63.html).Forms;

using System.Data;

using System.Runtime.InteropServices;

using System.Text;

namespace WindowsApplication8

{

///

/// Summary description for Form1.

///

public class Form1 : System.Windows.Forms.Form

{

private System.Windows.Forms.Button button1;

private System.Windows.Forms.Button button2;

private System.Windows.Forms.Button button3;

///

/// Required designer variable.

///

private System.ComponentModel.Container components = null;

public Form1()

{

//

// Required for Windows Form Designer support

//

InitializeComponent();

//

// TODO: Add any constructor code after InitializeComponent call

//

}

///

/// Clean up any resources being used.

///

protected override void Dispose( bool disposing )

{

if( disposing )

{

if (components != null)

{

components.Dispose();

}

}

base.Dispose( disposing );

}

#region Windows Form Designer generated code

///

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

///

private void InitializeComponent()

{

this.button1 = new System.Windows.Forms.Button();

this.button2 = new System.Windows.Forms.Button();

this.button3 = new System.Windows.Forms.Button();

this.SuspendLayout();

//

// button1

//

this.button1.Location = new System.Drawing.Point(64, 40);

this.button1.Name = "button1";

this.button1.Size = new System.Drawing.Size(120, 56);

this.button1.TabIndex = 0;

this.button1.Text = "MarshalAs attribute";

this.button1.Click += new System.EventHandler(this.button1\_Click);

//

// button2

//

this.button2.Location = new System.Drawing.Point(64, 112);

this.button2.Name = "button2";

this.button2.Size = new System.Drawing.Size(120, 56);

this.button2.TabIndex = 0;

this.button2.Text = "Unsafe code";

this.button2.Click += new System.EventHandler(this.button2\_Click);

//

// button3

//

this.button3.Location = new System.Drawing.Point(64, 184);

this.button3.Name = "button3";

this.button3.Size = new System.Drawing.Size(120, 56);

this.button3.TabIndex = 0;

this.button3.Text = "Marshal class";

this.button3.Click += new System.EventHandler(this.button3\_Click);

//

// Form1

//

this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);

this.ClientSize = new System.Drawing.Size(240, 282);

this.Controls.AddRange(new System.Windows.Forms.Control[] {

this.button3,

this.button2,

this.button1});

this.Name = "Form1";

this.Text = "Form1";

this.ResumeLayout(false);

}

#endregion

///

/// The main entry point for the application.

///

[STAThread]

static void Main()

{

Application.Run(new Form1());

}

[StructLayout(LayoutKind.Sequential)]

public class StringP

{

[MarshalAs(UnmanagedType.LPStr)]

public string str;

}

[DllImport("TestDll.dll",EntryPoint="GetUserMessageHelper",CharSet=CharSet.Ansi)]

static extern void GetUserMessageHelper(

[MarshalAs(UnmanagedType.LPStruct)]

ref StringP str);

private void button1\_Click(object sender, System.EventArgs e)

{

StringP strP=new StringP();

strP.str=new string('\*',20);

GetUserMessageHelper(ref strP);

MessageBox.Show(strP.str);

}

[DllImport("TestDll.dll",EntryPoint="GetUserMessage")]

unsafe static extern void GetUserMessage2(byte\*\*\*\* str);

[DllImport("user32.dll")]

unsafe static extern int MessageBoxA(int hwnd,byte\* Text,byte\* Caption,int Type);

unsafe private void button2\_Click(object sender, System.EventArgs e)

{

byte\* str=(byte\*)Marshal.AllocCoTaskMem(20);

byte\*\* strP=&str;

byte\*\*\* strPP=&strP;

GetUserMessage2(&strPP);

MessageBoxA(0,str,(byte\*)0,0);

Marshal.FreeCoTaskMem((IntPtr)str);

}

[DllImport("TestDll.dll",EntryPoint="GetUserMessage")]

static extern void GetUserMessage3(ref IntPtr str);

private void button3\_Click(object sender, System.EventArgs e)

{

IntPtr str=Marshal.AllocCoTaskMem(20);

IntPtr strP=Marshal.AllocCoTaskMem(4);

Marshal.StructureToPtr(str,strP,false);

IntPtr strPP=Marshal.AllocCoTaskMem(4);

Marshal.StructureToPtr(strP,strPP,false);

GetUserMessage3(ref strPP);

strP=Marshal.ReadIntPtr(strPP);

str=Marshal.ReadIntPtr(strP);

MessageBox.Show(Marshal.PtrToStringAnsi(str));

Marshal.FreeCoTaskMem(str);

Marshal.FreeCoTaskMem(strP);

Marshal.FreeCoTaskMem(strPP);

}

}

}

|  |
| --- |
| Example 2: GetUserName()  But not every day you want marshal 'char\*\*\*\*'. Let's try to marshal something more meaningful.Try to imagine that your strongest wish is to know what's username of currently logged user. So far, I don't know managed class that can answer this question. So we must call Windows API. As I can see from MSND, there is function that helps: |

BOOL GetUserName(

LPTSTR lpBuffer, // name buffer

LPDWORD nSize // size of name buffer

);

|  |
| --- |
| This function expects me, a caller, to provide buffer for ANSI string will contain currently loged user name. Length of name will be returned in second argument, given by pointer. Of course, I can't know how many space I need to allocate before I call the function. Due to it, I must call GetUserName() twice:  1. GetUserName(NULL,&Size); // Size is DWORD type  After this call, Size will contain length of buffer I need to allocate. After I allocate the buffer, I can call secondly to the function:  2. GetUserName(pointer\_to\_buffer,&Size);  At this point buffer contains needed username. Now we can see, how we can implement this API call in 3 methods. And,so long:  Example 2: Approach 1 - MarshalAs attribute.  For start, you must understand, that we can't just define |

[DllImport("Advapi32.dll"]

static extern bool GetUserName2Length(

[MarshalAs(UnmanagedType.LPStr)] string lpBuffer,

[MarshalAs(UnmanagedType.LPArray)] int& nSize );

for following reasons:

\* we have no pointers - int& will not ever compile

\* we can't marshal lpBuffer as string, becouse marshaller concludes,

that it's by value parameter,so it will only marshal string as "in"

parameter, so after call to function lpBuffer will remain unchanged.

For getting string back we must declare lpBuffer as 'out' or 'ref',

but in this case we will get pointer to pointer to string, while

GetUserName() expects pointer.

\* In first call we must marshal NULL,or 0,and this is no any way cast 0

to string, unlike C++, where (char\*)0 is good.

So, what we can do? We will write 2 declarations:

[DllImport("Advapi32.dll", EntryPoint="GetUserName",

ExactSpelling=false, CharSet=CharSet.Ansi, SetLastError=true)]

static extern bool GetUserName2(

[MarshalAs(UnmanagedType.LPArray)] byte[] lpBuffer,

[MarshalAs(UnmanagedType.LPArray)] Int32[] nSize );

[DllImport("Advapi32.dll", EntryPoint="GetUserName",

ExactSpelling=false, CharSet=CharSet.Ansi,SetLastError=true)]

static extern bool GetUserName2Length(

[MarshalAs(UnmanagedType.I4)] int lpBuffer,

[MarshalAs(UnmanagedType.LPArray)] Int32[] nSize );

First parameter will be marshalled as 'int' for first call

and byte array for second one. Second parameter is just

int array(in our case, it will contain 1 member only). After

this, calling is simple:

private void button2\_Click(object sender, System.EventArgs e)

{

// allocate 1-member array for length

int[] len=new int[1];

len[0]=0;

// get length

GetUserName2Length(0,len);

// allocate byte array for ANSI string

byte[] str=new byte[len[0]];

// Get username to preallocated buffer

GetUserName2(str,len);

// Use text decoder to get string from byte array of ASCII symbols

MessageBox.Show(System.Text.Encoding.ASCII.GetString(str));

}

Example 2: Approach 1 - unsafe code.

In this approach, declaration is more naturally:

[DllImport("Advapi32.dll", EntryPoint="GetUserName",

ExactSpelling=false, CharSet=CharSet.Ansi,SetLastError=true)]

unsafe static extern bool GetUserName3(byte\* lpBuffer,int\* nSize );

Also we need MessageBox() declaration to represent it:

unsafe static extern int MessageBoxA(int hwnd,byte\* Text,byte\* Caption,int Type);

Calling it also very C++ like:

unsafe private void button3\_Click(object sender, System.EventArgs e)

{

int Size=0;

GetUserName3((byte\*)0,&Size);

byte\* buffer=(byte\*)Marshal.AllocCoTaskMem(Size);

GetUserName3(buffer,&Size);

MessageBoxA(0,buffer,(byte\*)0,0);

Marshal.FreeCoTaskMem((IntPtr)buffer);

}

I use Marshal class for allocations the same reason like in char\*\*\*\*

example: I have no power to declare external allocators and use them.

Example 2: Approach 1 - Marshal class.

While working with Marshall class, all problematic arguments must be

defined as IntPtr.

Recall, that IntPtr is not pointer class, it just platform-dependent

integer, just like C++ int.

[DllImport("Advapi32.dll", EntryPoint="GetUserName", ExactSpelling=false,

CharSet=CharSet.Ansi,SetLastError=true)]

static extern bool GetUserName1(

IntPtr lpBuffer,

[MarshalAs(UnmanagedType.LPArray)] Int32[] nSize );

private void button1\_Click(object sender, System.EventArgs e)

{

string str;

Int32[] len=new Int32[1];

len[0]=0;

GetUserName1((IntPtr)0,len);

IntPtr ptr=Marshal.AllocCoTaskMem(len[0]);

GetUserName1(ptr,len);

str=Marshal.PtrToStringAnsi(ptr);

Marshal.FreeCoTaskMem(ptr);

MessageBox.Show(str);

}

As you can see, I casted 0 to IntPtr - no problem, value type.

After I know length, I can allocate needed memory, than call

GetUSerName(),ask Marshal get string from pointer to ASCII string,

free the memory, and I have the string. Using of Marshal for

such problems can be very elegant.

In this 2 parts we covered calling DLL libraries from .NET code.

What about:

\* Call COM servers

\* Call managed components from unmanaged code.

I'll be writing about these topics soon!