Type marshalling

Article • 05/10/2024

Marshalling is the process of transforming types when they need to cross between managed and native code.

Marshalling is needed because the types in the managed and unmanaged code are different. In managed code, for instance, you have a `string`, while unmanaged strings can be .NET `string` encoding (UTF-16), ANSI Code Page encoding, UTF-8, null-terminated, ASCII, etc. By default, the P/Invoke subsystem tries to do the right thing based on the default behavior, described in this article. However, for those situations where you need extra control, you can employ the MarshalAs attribute to specify what is the expected type on the unmanaged side. For instance, if you want the string to be sent as a null-terminated UTF-8 string, you could do it like this:

C#

```
[LibraryImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string pa-
rameter);

// or

[LibraryImport("somenativelibrary.dll", StringMarshalling =
StringMarshalling.Utf8)]
static extern int MethodB(string parameter);
```

If you apply the System.Runtime.CompilerServices.DisableRuntimeMarshallingAttribute attribute to the assembly, the rules in the following section don't apply. For information on how .NET values are exposed to native code when this attribute is applied, see disabled runtime marshalling.

Default rules for marshalling common types

Generally, the runtime tries to do the "right thing" when marshalling to require the least amount of work from you. The following tables describe how each type is marshalled by default when used in a parameter or field. The C99/C++11 fixed-width integer and character types are used to ensure that the following table is correct for all platforms. You can use any native type that has the same alignment and size requirements as these types.

This first table describes the mappings for various types for whom the marshalling is the same for both P/Invoke and field marshalling.

Expand table

C# keyword	.NET Type	Native Type
`byte`	`System.Byte`	`uint8_t`
`sbyte`	`System.SByte`	`int8_t`
`short`	`System.Int16`	`int16_t`
`ushort`	`System.UInt16`	`uint16_t`
`int`	`System.Int32`	`int32_t`
`uint`	`System.UInt32`	`uint32_t`
`long`	`System.Int64`	`int64_t`
`ulong`	`System.UInt64`	`uint64_t`
`char`	`System.Char`	Either `char` or `char16_t` depending on the encoding of the P/Invoke or structure. See the charset documentation.
	`System.Char`	Either `char*` or `char16_t*` depending on the encoding of the P/Invoke or structure. See the charset documentation.
`nint`	`System.IntPtr`	`intptr_t`
`nuint`	`System.UIntPtr`	`uintptr_t`
	.NET Pointer types (ex. `void*`)	`void*`
	Type derived from `System.Runtime.InteropServices.SafeHandle`	`void*`
	Type derived from `System.Runtime.InteropServices.CriticalHandle`	`void*`
`bool`	`System.Boolean`	Win32 `BOOL` type
`decimal`	`System.Decimal`	COM `DECIMAL` struct

C# keyword	.NET Type	Native Type
	.NET Delegate	Native function pointer
	`System.DateTime`	Win32 `DATE` type
	`System.Guid`	Win32 `GUID` type

A few categories of marshalling have different defaults if you're marshalling as a parameter or structure.

Expand table

.NET Type	Native Type (Parameter)	Native Type (Field)
.NET array	A pointer to the start of an array of native representations of the array elements.	Not allowed without a `[MarshalAs]` attribute
A class with a `LayoutKind` of `Sequential` or `Explicit`	A pointer to the native representation of the class	The native representation of the class

The following table includes the default marshalling rules that are Windows-only. On non-Windows platforms, you cannot marshal these types.

Expand table

.NET Type	Native Type (Parameter)	Native Type (Field)
`System.Object`	`VARIANT`	`IUnknown*`
`System.Array`	COM interface	Not allowed without a `[MarshalAs]` attribute
`System.ArgIterator`	`va_list`	Not allowed
`System.Collections.IEnumerator`	`IEnumVARIANT*`	Not allowed
`System.Collections.IEnumerable`	`IDispatch*`	Not allowed
`System.DateTimeOffset`	`int64_t` representing the number of ticks since midnight on January 1, 1601	`int64_t` representing the number of ticks since midnight on January 1, 1601

Some types can only be marshalled as parameters and not as fields. These types are listed in the following table:

Expand table

.NET Type	Native Type (Parameter Only)
`System.Text.StringBuilder`	Either `char*` or `char16_t*` depending on the `CharSet` of the P/Invoke. See the charset documentation.
`System.ArgIterator`	`va_list` (on Windows x86/x64/arm64 only)
`System.Runtime.InteropServices.ArrayWithOffset`	`void*`
`System.Runtime.InteropServices.HandleRef`	`void*`

If these defaults don't do exactly what you want, you can customize how parameters are marshalled. The parameter marshalling article walks you through how to customize how different parameter types are marshalled.

Default marshalling in COM scenarios

When you are calling methods on COM objects in .NET, the .NET runtime changes the default marshalling rules to match common COM semantics. The following table lists the rules that .NET runtimes uses in COM scenarios:

Expand table

.NET Type	Native Type (COM method calls)
`System.Boolean`	`VARIANT_BOOL`
`StringBuilder`	`LPWSTR`
`System.String`	`BSTR`
Delegate types	`_Delegate*` in .NET Framework. Disallowed in .NET Core and .NET 5+.
`System.Drawing.Color`	`OLECOLOR`
.NET array	`SAFEARRAY`
`System.String[]`	`SAFEARRAY` of `BSTR`s

Marshalling classes and structs

Another aspect of type marshalling is how to pass in a struct to an unmanaged method. For instance, some of the unmanaged methods require a struct as a parameter. In these cases, you need to create a corresponding struct or a class in managed part of the world to use it as a parameter. However, just defining the class isn't enough, you also need to instruct the marshaller how to map fields in the class to the unmanaged struct. Here the `StructLayout` attribute becomes useful.

C#

```
[LibraryImport("kernel32.dll")]
static partial void GetSystemTime(out SystemTime systemTime);

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

public static void Main(string[] args)
{
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

The previous code shows a simple example of calling into `GetSystemTime()` function. The interesting bit is on line 4. The attribute specifies that the fields of the class should be mapped sequentially to the struct on the other (unmanaged) side. This means that the naming of the fields isn't important, only their order is important, as it needs to correspond to the unmanaged struct, shown in the following example:

C

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
```

```
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Sometimes the default marshalling for your structure doesn't do what you need. The Customizing structure marshalling article teaches you how to customize how your structure is marshalled.