How to use C# Structs to create a Union type (Similar to C Unions)

C-Style Unions in C#

Union types are used in several languages, like C-language, to contain several different types which can "overlap". In other words, they might contain different fields all of which start at the same memory offset, even when they might have different lengths and types. This has the benefit of both saving memory, and doing automatic conversion. Think of an IP address, as an example. Internally, an IP address is represented as an integer, but sometimes we want to access the different Byte component, as in Byte1.Byte2.Byte3.Byte4. This works for any value types, be it primitives like Int32 or long, or for other structs that you define yourself.

We can achieve the same effect in C# by using Explicit Layout Structs.

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

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
```

```
    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Having defined out Struct in this way, we can use it as we would use a Union in C. For example, let's create an IP address as a Random Integer and then modify the first token in the address to '100', by changing it from 'A.B.C.D' to '100.B.C.D':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Output:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

View Demo

Union Types in C# can also contain Struct fields

Apart from primitives, the Explicit Layout structs (Unions) in C#, can also contain other Structs. As long as a field is a Value type and not a Reference, it can be contained in a Union:

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

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
```

```
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http, Https, Ftp, Sftp, Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port  = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}
```

We can now verify that the whole Service Union fits into the size of a long (8 bytes).

```
var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;

Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

View Demo

Remarks

Union types are used in several languages, notably C-language, to contain several different types which can "overlap" in the same memory space. In other words, they might contain different fields all of which start at the same memory offset, even when they might have different lengths and types. This has the benefit of both saving memory, and doing automatic conversion.

Please, note the comments in the constructor of the Struct. The order in which the fields are initialized is extremely important. You want to first initialize all of the other fields and then set the value that you intend to change as the last statement. Because the fields overlap, the last value setup is the one that counts.