Size of an Object

Object sizes are determined by their field values, as laid out in the *Simple Value Types* section earlier, but are often padded for performance reasons. Value types are in-lined with the object, and reference types have a `TypedReference` value allocated which will likely point to the heap. Both reference types and value types provide control over their layout in memory and the size of the padding between fields.

We may need to control the layout of an object when interacting with unmanaged code. Since reference types are not very compatible with unmanaged code, you will normally need to achieve this with structs. There may be cases where you wish to change a class structure as well, and this information applies to both.

The `System.Runtime.InteropServices.StructLayoutAttribute` class controls how fields are laid out in memory. Despite its name, this attribute can be used to influence the structure of both classes and structs. Its constructor takes the `LayoutKind` enum. By default in C# and VB.NET, structs are marked as `LayoutKind.Sequential` , meaning that fields are defined in memory in the order defined in code.

The `Size` field dictates the absolute size of the object; memory beyond the defined fields is typically accessed with unmanaged code. The `CharSet` field determines the string marshaling: whether strings will be converted to `LPWSTR` or `LPSTR` (Long Pointer to Wide String or Long Pointer to String). `LPWSTR` should be used for UNICODE strings. `LPSTR` is used for a regular array of 8-bit characters. By default, strings in the Framework are stored as UTF-16, and those in Windows are Unicode or ANSI. Setting `CharSet.Auto` chooses the appropriate encoding based upon the platform. This will often be needed to facilitate COM interop.

The `Pack` field sets the memory boundaries for `LayoutKind.Sequential.` Setting `Pack` with a value of `0` will choose the default for the platform, while setting it to `1` will ensure there are no boundaries between each field in memory. Pack settings `2, 4, 8, 16, 32, 64` , and `128` lay out the fields on memory offsets relative to the address of the first field. `Pack` sizes larger than that processor's boundary are substituted for the processor boundary size, which provides a performance penalty for the processor. For reference, an Intel processor's boundary sizes are 4 bytes. This may be relevant when exporting a structure (sending over the network, writing to disk, using `p/invoke` and `interop` , etc.), to ensure that the layout is consistent.

You might think you could use this internally to reduce memory usage (by reducing the pack size). In fact, it might do that, but there could also be a significant performance hit on some hardware, while on other hardware it might not even work, due to unaligned access to the data (though I don't believe this applies to anything .NET runs on). This is *not* a recommended strategy for reducing memory usage, and should only be used to solve a specific problem with exporting an object.

Consider the code in Listing 4.35. Even though the objects of this type may take up less memory than they would without the `StructLayout` attribute, code accessing this struct will suffer a performance penalty. How much of a performance penalty depends on your hardware and the underlying data types.

```
[StructLayout(LayoutKind.Sequential, Pack = 0)]
public structSequential
{
 private byte b;
 private char c;
 private float f;
 private int i;

 public byteByte
 {
  get{ return b; }
  set{ b = value; }
 }

 public char Char
 {
  get { return c; }
  set { c = value; }
 }

 public float Float
 {
  get { return f; }
  set { f = value; }
 }

 public int Integer
 {
  get { return i; }
  set { i = value; }
 }

}
```

Listing 4.35: Struct layout with no packing. May slow down memory retrieval.

The `LayoutKind.Explicit` value gives you full control over your data. However, you should have a good reason for doing this, such as calling unmanaged code that explicitly needs data laid out in a specific manner, as this process is error prone and could lead to data corruption. Listing 4.36 gives an example.

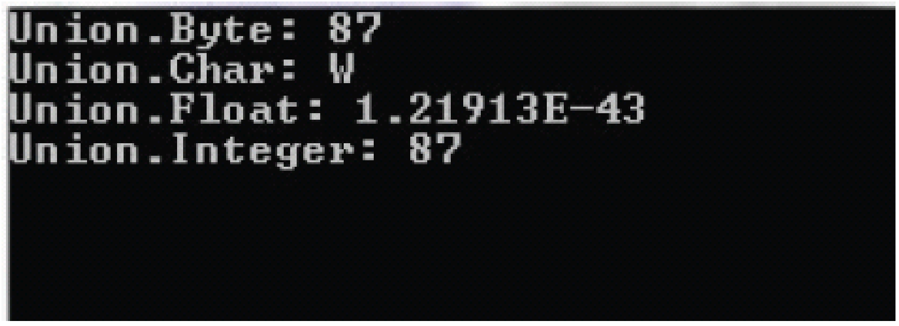
```
[StructLayout(LayoutKind.Explicit)]
public struct Union
{
 [FieldOffset(0)]
 private byte b;

 [FieldOffset(0)]
 private char c;

 [FieldOffset(0)]
 private float f;

 [FieldOffset(0)]
 private int i;

 public byte Byte
 {
  get { return b; }
  set { b = value; }
 }

 public char Char
 {
  get { return c; }
  set { c = value; }
 }

 public float Float
 {
  get { return f; }
  set { f = value; }
 }

 public int Integer
 {
  get { return i; }
  set { i = value; }
 }
}
```

Listing 4.36: A struct explicitly setting all fields to point to the same memory location.

With the `Union` struct, you can modify the value in one field and affect the value of another field. This is because the `FieldOffset` attributes point all of the fields to the same location.

```
static void Main(string[] args)
 {
  var union = new Union();
  union.Char = ‘W’;
  Console.WriteLine(“Union.Byte: ” + union.Byte);
  Console.WriteLine(“Union.Char: ” + union.Char);
  Console.WriteLine(“Union.Float: ” + union.Float);
  Console.WriteLine(“Union.Integer: ” + union.Integer);
 }
```

Listing 4.37: We explicitly set the value of `Char` but all fields now have a value.



```
Union.Byte: 87
Union.Char: W
Union.Float: 1.21913E-43
Union.Integer: 87
```

Figure 4.5: The bits at this memory location are interpreted according to the rules for the various types.

We assigned a valid value for one of the fields, and the other fields interpret the bits in the shared location as a valid value in their respective datatypes. `W` has an ASCII value of 87, which has a valid value as a byte and an integer. On the other hand, the interpreted value for the `Float` is truly bizarre.

If that wasn't wild enough, you can use this to point reference types at the same memory space in managed code. This is not something I recommend, but just be aware that it can be done (this is almost guaranteed to create a logic error and at the very least result in some very confusing code).

```
public class Person
```

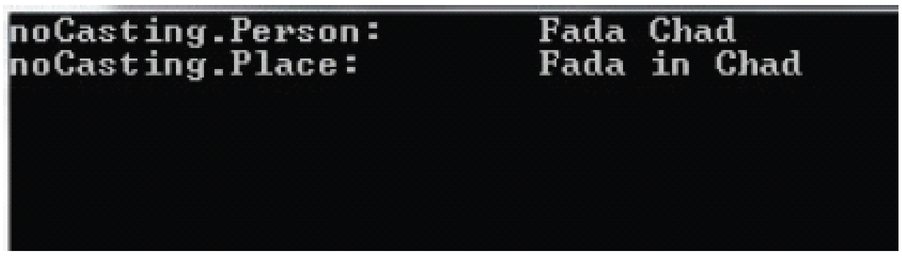
```
{
 public string FirstName { get; set; }
 public string LastName { get; set; }
}

public class Place
{
 public string City { get; set; }
 public string Country { get; set; }
}

[StructLayout(LayoutKind.Explicit)]
public struct NoCasting
{
 [FieldOffset(0)]
 private Person person;

 [FieldOffset(0)]
 private Place place;

 public Person Person
 {
  get { return person; }
  set { person = value; }
 }

 public Place Place
 {
  get { return place; }
  set { place = value; }
 }
}
```

Listing 4.38: The `Person` and `Place` properties will refer to the same set of bits.

In the example in Listing 4.38, both `Person` and `Place` will refer back to the same set of bits allocated to the `NoCasting` object.

```
NoCasting noCasting = new NoCasting
{
 Person = new Person { FirstName = "Fada", LastName = "Chad" }
};
Console.WriteLine("noCasting.Person:\t"
       + noCasting.Person.FirstName + " " + noCasting.Person.LastName);
Console.WriteLine("noCasting.Place:\t"
       + noCasting.Place.City + " in " + noCasting.Place.Country);
```

Listing 4.39: Updating one property will result in both properties being updated.



```
noCasting.Person:        Fada Chad
noCasting.Place:         Fada in Chad
```

Figure 4.6: This is almost guaranteed not to be the expected result.

You can assign the `Person` or `Place` property on a `NoCasting` instance at any time, and it will point the other property at the same member space. They don't inherit from each other, and you can't cast one to the other. The only reason this makes any sense is because the field layout is the same. If you were to rely on programming paradigms such as this, it would only take a field reordering to break your code.

This is fascinating and definitely produces some bizarre behavior. It also has no place in code that you plan to run in production. It is fun for a demonstration or to confuse the unsuspecting, but it will eventually create a logic error that will be difficult to track down.

Unless you have very specific marshaling requirements and you know exactly what you are doing, leave the `StructureLayout` alone. This is not a good way to squeeze out better memory management, and will almost always lead to performance problems or logic errors.