Reducing Memory Footprint and Object Instance Sizes: StructLayoutAttribute Is Only the Beginning

by R. Stacy Smyth

One way to reduce the memory used by an application is to modify the definitions of the application's classes and structures so that instances of those types become smaller in memory. For types with tens of thousands of instances, this can result in substantial savings.

Often, though, taking a common-sense approach to reducing the size of objects yields smaller than expected benefits, and sometimes yields no improvement at all. Since some of these optimizations can take considerable time to implement, making accurate predictions about which changes are worthwhile can be important.

This article explains how to accurately calculate the improvements in memory footprint that can be expected from particular optimizations, so that you can select in advance which optimizations will be worthwhile.

When Should you Bother?

Before I introduce the math behind reducing the size of objects, I should start with the question "Should you even bother?", because the techniques in this article, while occasionally essential, frequently are *not* the way to solve your memory footprint problems. Specifically:

- The gains to be made from reducing the size of object instances are generally measured in bytes per instance — as in, generally fewer than 50 bytes per instance. So unless you know that you have *lots* of instances of a particular type in memory, it generally isn't worth bothering with reducing the size of the instances.

- Even if your memory profiler (or your own analysis of the code) has confirmed that you have tens of thousands of instances of a type in memory, rather than starting with the question "How can I make each instance smaller?", you should probably start with "Do I need all of these instances in memory?". Is there a way you could make some of them available for garbage collection sooner? Or could you use a flyweight design pattern to reduce the number of instances required? And so on.

If the bottom line is that yes, you have an awful lot of them in memory, and no, there isn't any way to get rid of lots of instances entirely, then the information in this article might be helpful.

Sources of Information / Limits of the Article

Most of this article is based directly on experimental evidence and the rules I've derived from it. So even though this article explains all of the data I've acquired, it's probably missing various refinements. I welcome improvements, of course.

Calculating the Size of a Class or Structure — Overview

Calculating the size of an instance of a class or structure (the "target type") is an iterative process, starting with system.object, and working down the inheritance chain to the target type. (All structs are derived directly from system.object.) For each link of the inheritance chain below system.object, here are the broad outlines of the procedure you'll follow -- I'll get to the details shortly. The math is simple, but can be time-consuming for long inheritance chains and/or for types with many fields.

1. Determine the size of an instance of the base class.
2. Determine the amount of "free space" available inside the end of instances of the base class.
3. Determine how many instance (i.e. non-static) fields of the current type will fit into the free space inside the end of the base class.
4. Determine the size of the memory block required to hold the instance fields of the current type that don't fit into the free space in the end of the base class.
5. Round the number from Step 4 up to be a multiple of 4 bytes.
6. Calculate the two results you need:
    - The number you rounded up *by* (i.e. Step 5 minus Step 4) is the amount of free space available in the end of the current type. For example, if you rounded up from 17 to 20 bytes, there are 3 bytes available.
    - The number you rounded up *to*, plus the size of the base class (i.e., Step 5 + Step 1) is the size of each instance of the current type.

Calculating the Size of a Class or Structure — Details

Steps 1 & 2 — Characteristics of the Base Class

Here's the starting place, which is always the same:

- Size of system.object: 12 bytes
- Available free space inside the end of each system.object: 4 bytes

For every class other than system.object, the available free space varies from 0-3 bytes.

The idea behind "the free space inside the end of the base class" is that the size associated with an instance of a class or struct will be a multiple of 4 bytes, but the fields inside that class or struct may not need all of that space. If this unused space falls at the end of the memory used by the type, that space can be used by the fields of derived classes, thereby decreasing the space that would otherwise be required for instances of the derived type. This optimization

happens automatically, all the time, and the only time you need to be aware of it is when you're doing these sorts of calculations.

Step 3 — Fields that Move into the Free Space of the Base Class

(Important note on automatically implemented fields: all of this discussion involves calculating the size and arrangement of fields in memory. It's important to remember that even if a property has no explicit backing field — that is, it has "set" and "get" accessors but no defined accessor bodies — it still has an automatically implemented, invisible field in which the data associated with the property is stored. When you're counting fields and field sizes, you need to count these fields too.)

Before we can talk about which fields can be moved into the free space of the base class, we need to understand, in general, how the fields of a type are arranged in memory.

There are three basic ways that the fields in a class or struct can be arranged in memory, as determined by the class attribute StructLayoutAttribute:

1. **LayoutKind.Sequential**: By default, this is the value of StructLayoutAttribute used by structs. This layout indicates that all of the fields in the type are laid out in memory in the same order that they appear in the type definition, but spaced such that each field starts on a byte offset (within the type) that is divisible by the size of the field. For example, ints and object references are placed on 4-byte divisible offsets (0, 4, 8, etc. — also known as "4-byte boundaries"), shorts and chars are placed on 2-byte boundaries, and bytes and bools can be placed at any location ("1-byte boundaries").

2. **LayoutKind.Auto:** By default, this is the value of StructLayoutAttribute used by classes. This layout indicates that the fields in the type can be reordered by the compiler in whatever way the compiler likes. For the Microsoft C# compiler, this means that the fields are spaced so that they fall on byte boundaries divisible by the field sizes (as with LayoutKind.Sequential), *and* they are re-ordered so that this spacing results in zero "dead space" between fields. In effect, this is done by placing all of the 8-byte fields at the beginning of the class, followed by the 4-byte fields, followed by the 2-byte fields, followed by 1-byte fields. There's an exception to this ordering for filling the free space in the end of the base class, but we'll get to that shortly.

3. **LayoutKind.Explicit:** This layout isn't the default for anything. To use an explicit layout, you need to specify the StructLayoutAttribute explicitly, and then specify a FieldOffset attribute for each individual field. Not surprisingly, all of the fields will be placed at exactly the offsets you specify. You can use this layout to place multiple fields at exactly the same offset as each other and thereby create the equivalent of the old C concept of unions, but aside from that obscure capability, I haven't found any way to use this layout to improve memory footprint beyond what you can accomplish with the other two layouts. (It has other uses related to data marshalling, but that's not the topic of this article.) As such, I won't discuss this layout further in this article.

For the purpose of determining which fields will move into the free space at the end of a base class, there is a big difference between LayoutKind.Sequential and LayoutKind.Auto.

Suppose you have 1 byte of free space available at the end of the base class. If your derived class is using LayoutKind.Auto, and if you have a 1-byte field in the derived class, anywhere in the sequence of fields, the compiler is free to move that 1-byte field into the free space. If your derived class is using LayoutKind.Sequential, the only way the compiler can take advantage of the 1-byte free space is if the *first* field in the derived class is a 1-byte field.

Likewise, if the base class has two bytes of free space, a derived class using LayoutKind.Auto can take advantage of both free bytes if it has either a 2-byte field, or two 1-byte fields anywhere in its sequence of fields. If the class is LayoutKind.Sequential, the only way it can use both free bytes is if either the *first* field is a 2-byte field, or the *first two* fields are both 1-byte fields.

Because the compiler aligns fields on byte boundaries that are multiples of the field sizes, the requirements for completely using 3 bytes of free space are even more stringent for a class using LayoutKind.Sequential. In this case, there are only two ways to use all 3 bytes of free space:

- if the first three fields are all 1 byte fields
- if the first field is a 1-byte field and the second field is a 2-byte field.

The other way around (a 2-byte field followed by a 1-byte field) won't fill all of the free space because the first (2-byte) field will be moved to the available 2-byte boundary, skipping the first byte of free space, and then there will be no room left in the free space for the second (1-byte) field. For a derived class using LayoutKind.Auto, all that is required to use all of the free space is any three 1-byte fields, or any 1-byte field plus any 2-byte field.

Step 4 — The Size of the Block Required to Hold the Remaining Fields

Now that we've determined which fields (if any) the compiler will move into the free space at the end of the base class, what's next is determining the size of the block of memory required to hold the remaining fields. (OK, I just fudged, but only a little bit: in the case of LayoutKind.Auto, we don't really know which exact field(s) the compiler will move, but it also doesn't matter. For example, when filling 2 bytes of free space, it doesn't matter whether the compiler moves a particular 2-byte field, a different 2-byte field, or any two 1-byte fields — the math is going to work out the same in the end.)

For LayoutKind.Auto, determining the size of the required memory block is simple: the compiler is going to arrange the remaining fields in memory so that there is no dead space between them. *This means that the size of the required memory block is simply the total of the sizes of the fields.*

For LayoutKind.Sequential, computing the size of the required block is more involved, and can

be rather tiresome. You start with the size of the first field that wasn't moved into the free space, then look at the size of the next field. Use the size of the next field to determine how many bytes of dead space will need to be added at the end of the first field to allow the second field to be placed on a byte boundary that is a multiple of the size of the second field. Add the dead space to the total, and add the size of the second field to the total. Then repeat: look at the size of the third field, and determine how many bytes of dead space will need to be added after the second field so that the third field can start at an offset that is a multiple of its size. Add the size of the second field's dead space to the total, then add the size of the third field to the total. And so on, through all the fields in order.

This brings us to the first big, easy way to reduce the size of instances.

TIP: *If a **struct** is using LayoutKind.Sequential (the default), organize the field declarations so that 4-byte fields come first, then 2-byte fields, then 1-byte fields. If a **class** is using LayoutKind.Sequential, calculate the free space available in the base class and place fields that can fill the free space first. Then organize the remaining fields in the same way you would for structs: 4-byte fields, then 2-byte fields, then 1-byte fields.*

Following these recommendations will guarantee that all free space at the end of the base class can be taken advantage of, and no dead space will be required to pad between the fields. The reason for the different recommendations between structs and classes is that structs are always derived from system.object. We know that 4 bytes of free space are always available at the end of system.object, and no special care is required to take advantage of 4 bytes of free space, unlike the case with 1, 2, or 3 bytes.

Steps 5 & 6 — Rounding up and Calculating

These two steps are self-explanatory.

Example Size Calculations

A few examples are in order before I introduce more concepts.

1. A struct has a single field of type int. This field has a size of 4 bytes, which will fit into the free space of the base class, system.object, which has 4 bytes of free space available. The struct will therefore require *no* additional space beyond what is required for system.object, namely 12 bytes per instance.

2. A struct has two fields, each of which is an int. The first field is placed by the compiler in the free space of the base class, system.object, as in example 1 above. The second field is placed at offset 0 within the space for the derived type (every struct is a derived type — they're just derived from system.object) , and occupies 4 bytes. 4 bytes is already a multiple of 4, so it does not need to be rounded up. The total size of an instance of this type is:

size of system.object	12 bytes

+ additional field in space of derived type	4 bytes
=	16 bytes

If we change either of the fields to a byte, the space required for an instance of this type will remain unchanged: if we change the first field to a byte, it will still be placed in the free space of the base class, and the second field will still be placed at byte 0 within the space for the derived type (since it won't fit into the remaining free space of 3 bytes). If we change the second field to a byte, the first field will still be placed in the free space, and the second field will still be placed at offset 0 in the space for the derived type. True, the second field will only occupy 1 byte instead of 4, but the space required by the type will remain unchanged, since we round the required space up to a multiple of 4 bytes.

If we change both fields to bytes, however, the situation changes: now, *both* fields will fit inside the free space at the end of system.object, and the additional space required for the derived type falls to 0 — meaning the size of instances of this type is merely the size of instances of system.object, 12 bytes.

3. A struct has 3 fields — a byte, an int, and a byte. The first field starts at byte 0 in the free space of the base class, the second field starts at byte 0 in the derived class (since it won't fit in the remaining 3 bytes of free space), and the third field starts at byte 4. The total size of the struct is:

size of system.object	12 bytes
+ additional fields (5 bytes), rounded up to a multiple of 4	8 bytes
=	20 bytes

If we change all three fields to byte fields, the required size collapses to 12 bytes: all of the fields will fit in the free space in system.object, which requires 12 bytes per instance.

The StructLayoutAttribute and Class Hierarchies

This isn't directly related to calculating the size of a type, but if you're going to be experimenting with the StructLayoutAttribute as you optimize your memory footprint, you still need to know it: as you descend through a class hierarchy, you cannot increase the precision with which you control your field layouts.

That is:

- If a class has LayoutKind.Explicit, classes immediately derived from it can have any of the three kinds of layouts.
- If a class has LayoutKind.Sequential, classes immediately derived from it can have LayoutKind.Sequential or LayoutKind.Auto, but not LayoutKind.Explicit.
- If a class has LayoutKind.Auto, classes derived from it must have LayoutKind.Auto as

well.

Making Unclaimed Free Space Available to Descendant Classes

And now for a really unexpected way to make instances of derived classes smaller in memory.

Suppose we have this class hierarchy:

```
public class A
{
      public short Field1;
}

public class B : A
{
      public int Field2;
}

public class C : B
{
    public short Field3;
}
```

Assuming that the field sizes can't be made any smaller (i.e. we really do require a short, an int, and a short for the three fields), is there any way we can shrink the memory required by instances of C?

At first glance, the answer looks like "No": with only one field per class, there's no useful way to re-organize the fields, either manually or with the help of layout-related attributes.

To see how we can improve the memory footprint of C, let's start by looking at how instances of the classes will use memory by default, before we fiddle with them:

- In class A, Field1 will fit entirely within the free space of the base class, system.object, so the size of class A is 12 bytes, with 2 bytes of unused free space remaining inside the end of each instance. (Field1 uses only 2 of the 4 available bytes of free space.)

- In class B, Field2 requires 4 bytes so it will not fit inside the available free space at the end of class A (2 bytes). That means Field2 will start at byte 0 of the space for the derived class. This means that each instance of class B will occupy

	size of base class (class A)	12 bytes
+	additional field (4 bytes), rounded up to a multiple of 4	4 bytes
=		16 bytes

  Since the space required by the fields of class B is an even multiple of 4 bytes, there is no free space available within the end of instances of class B.

- In class C, Field3 requires 2 bytes. There's no free space at the end of class B, so Field3 has to start at byte 0 of class C. The fields of class C only require 2 bytes (there's only the 1 field), but this gets rounded up to a multiple of 4 for determining the size of class C. So each instance of C requires:

size of base class (class B)	16 bytes
+ additional field (2 bytes), rounded up to a multiple of 4	4 bytes
=	20 bytes

Is there any way to improve on this?

The answer is yes, because even though the compiler can't *automatically* make use of the free space at the end of class A, we *can use it ourselves.* Like this:

```
public class A
{
    public short Field1;
    protected short _freeSpace;
}

public class B : A
{
    public int Field2;
}

public class C : B
{
    public short Field3
    {
        get
        {
            return _freeSpace;
        }
        set
        {
            _freeSpace = value;
        }
    }
}
```

After we make this change, here's how the memory usage works out:

Instances of A are the same size as before, because the new field ("_freeSpace") fits entirely inside the free space which was already available at the end of instances of class A. So instances of A still require only 12 bytes.

Instances of B are the same size as before, because A hasn't changed sizes, and Field2 of B wasn't using the free space of A anyway. So instances of B still require 16 bytes.

Instances of C *no longer require any additional memory at all*, beyond what is required for the

base class B. So instances of C have dropped from requiring 20 bytes per instance to only requiring 16 bytes.

Pretty cool, huh?

Yes, but this technique comes with a cautionary note. For the class hierarchy above, and where it is important to reduce the size of C, the approach I've described does nothing but good.

But just imagine that A has an additional derived class, D, that you hadn't considered while optimizing class C:

```
public class D : A
{
    public short Field4;
}
```

Without the "_freeSpace" change, Field4 would have fit into the free space of A and instances of D would have required only 12 bytes. With the "_freeSpace" change, instances of D now require 16 bytes. If there are more instances of D than there are of C, we just moved our memory footprint in the wrong direction!

Of course, you can fix it like this:

```
public class D : A
{
    public short Field4
    {
        get
        {
            return _freeSpace;
        }
        set
        {
            _freeSpace = value;
        }
    }
}
```

This way, we can get the benefits for C while keeping the size of D unchanged, but we have to optimize D ourselves (as we've just done) instead of having the compiler take care of it for us. In other words: if you take control of the free space in a base class, the only way to get optimal results is to take responsibility for allocating that free space in *all* of the child classes. If we're not going to let the compiler do its job in allocating the free space, we're going to need to do it ourselves!