

# Struct inheritance in C# with CoreCLR and Roslyn

📅 September 27, 2015   🔖 C#, Roslyn, CoreCLR, .Net   ✎ edit

Ever wanted to use struct inheritance in C#? In this post, we will go through a surprisingly simple upgrade of CoreCLR and Roslyn to add a *basic* support for struct inheritance in C#, one of the many little things that I would love to see unlocked in C#!

## ValueType vs ReferenceType (in .NET)

We should take the time here to revisit what exactly makes the difference between the two. Stating that the main difference is, for example, "*structs are allocated on the stack, and class on the heap*" is not only not enough, but partially wrong... (and I may have participated to simplify this when explaining the difference)

If we look at the "[Reference Types](#)" page on MSDN, it will start with the following definition which is actually quite relevant:

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data.

But this is continuing with something which could be a bit misleading:

With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other

Specially because it is ending by a little parenthesis that almost completely invalidates the above definition (!):

(except in the case of ref and out parameter variables, see ref (C# Reference) and out parameter modifier (C# Reference)).

The following code explains better why the definition above is confusing:

```
void MyFunction(ref MyValue a, ref MyValue b)
{
    // If MyValue was a struct, a and b could in fact point to the same "instance"!
}
```

So, let's just slightly reformulate the above:

- Variables of reference types store references to their data: Assignment to a reference type variable change the reference (the pointer to the object).
- Variables of value types directly contain their data: Assignment to a value type variable copies the values stored in the source valuetype to the assigned valuetype.

Notice from the above that **it is all about the interaction model of a variable**. It doesn't speculate about storage - heap or stack. Technically, on one side, a class/reference type could be *fat* instantiated on the stack or even embedded within another container ReferenceType/ValueType (two other things that I would like to see coming to C#), still the variable for the reference type would be seen *almost* as a reference (I'm saying almost because it is not that easy, but that will be part of another post!). On the other side, a struct can be passed by reference (by `ref` or `out` in C#), but whenever you interact with the variable that holds this reference, you cannot change the reference but only the values it points to.

Also, unlike a reference type, where the type of the object is defined by its instance, a valuetype does not have a hidden virtual method table (which is also usually used as a type descriptor) so the actual type of the valuetype as seen by the user of this variable is only defined by the type of the façade variable. It is statically known and there is no side effects (unless the valuetype would implement its own explicit method table management - by using a field that identifies the type of valuetype, in the end, it would look like a class). When

[Struct inheritance in C# with CoreCLR and ValueType vs ReferenceType \(in .NET\)](#)  
[Why struct inheritance?](#)  
[Bringing struct inheritance to CoreCLR](#)  
[Bringing struct inheritance to Roslyn](#)  
[The test case](#)  
[Next?](#)

we box a value type, it actually creates a small shell that transform it to a reference type by adding the method table in the header of the shell. Then this struct would only be seen through a reference type (an interface or [System.Object](#))

What should also be noted is that **the definition above doesn't constrain things like inheritance**. In other terms: *Value types are not doomed to be excluded from the inheritance folks*.

## Why struct inheritance?

---

There are some cases where inheritance would be quite handy. Simply relying on the benefit of `valuetype` (locality) while the inheritance would provide some abstraction/data/code reusability across a set of related structs. If I take the example of [SharpDX](#), all COM objects could be declared as inherited struct, where the root main struct would just be a pointer to the COM object (and providing the [IUnknown](#) interface) , and then each derived struct would expose the proper methods for each COM interface. This is typically where we don't need to have a managed object to represent a native object (and most of interop would not need to). It would make the library extremely lightweight in terms of GC pressure, as It would not create any managed object just to wrap a COM object.

Now, if we look around on stackoverflow about the subject, with the questions:

- [.net - Why don't structs support inheritance?](#)

The main answer with 79+ votes argues that "*The reason value types can't support inheritance is because of arrays.*". Unfortunately, this is completely invalid (as noted by some comments). An array of value type is simply not cast-able. You can't cast a `int[]` to `short[]` or vice et versa, and that would be the same for a `StructBase[]` vs `StructDerived[]`. The best you would get is an [InvalidCastException](#), not an undefined runtime behaviour as stated by the answer (and this is indeed what I could easily verify while adding support for struct inheritance)

- or [Why a C# struct cannot be inherited?](#)

The answer takes the example of "*Anything storing that struct type would take up a variable amount of memory based on which subtype it ended up containing*". Again, this is completely wrong. When you declare a struct (derived or not), **the size and layout will be statically known at compile/JIT time**. If a class would have to use a struct [StructDerived](#) derived from a [StructBase](#), the class would use the size of [StructDerived](#), nothing else, no magic or dynamic stuff involved.

- or [Why are .NET value types sealed?](#)

The answer focus on the fact that inheritance with `valuetypes` without virtual methods are useless. Well, sorry, but there are cases where It makes sense: Just for the sake of sharing some data and process in a base struct and having derived structs using base data + exposing additional data+process+behaviours (like the native COM interfaces through I cited above), while still getting benefits from `valuetype` storage, this is relatively something that I encountered many times, and not only in interop scenarios.

If you also think about generics, having a `struct StructBase`, and a `class StructProcessor<T>` where `T : StructBase` the code could access the base field of [StructBase](#) even if we use it with a `struct StructDerived`. Currently, this kind of behaviour can only be done through interfaces on structs, but with direct access to field, it opens lots of opportunities (like when you have a [Matrix](#) 4x4 floats, 64 bytes field, you don't want your matrix to be accessible through a property but only through a field to allow tight access to it (pinning... etc.)

Though this is valid objection that [object slicing](#) can be harmful. On a similar subject, I recently made a mistake of moving a method from a struct to an extension method applied, but of course, the `this` parameter was not making the parameter an `implicit ref` but was actually copying the struct, which was absolutely not what I wanted to do! Some inattention and struct implicit copy by value can introduce severe bugs... plain structs harmful?

Still, something potentially harmful should not always be banned from a usage. In .NET, most developers barely develop structs, so it would not hurt this majority to introduce a feature that they would not really use...

Anyway, let's dive into how to implement this with CoreCLR and Roslyn. **At least, this is an interesting and small journey, even if we don't necessarily intend to bring back our discovery to home!**

## Bringing struct inheritance to CoreCLR

---

Before going the deep route of modifying Roslyn, I wanted to quickly check how much this would require to upgrade the CLR to support struct inheritance. Now that [CoreCLR](#) is fully available and easily compilable, it gives amazing opportunities to try out ideas directly on the CLR!

In order to experiment this, I first implemented a [HelloWorld.cs](#) that would assume inheritance from [ValueDerived](#) from [ValueBase](#):

```
using System;

// Base struct
public struct ValueBase
{
    public int A;

    public int B;
}

// We would like ValueDerived to inherit from ValueBase
public struct ValueDerived
{
    public int C;
}

public class Program
{
    public unsafe static void Main(string[] args)
    {
        var valueDerived = new ValueDerived();
        // valueDerived.A = 1; // We still can't access A and B without modifying the comp
        // valueDerived.B = 3;
        valueDerived.C = 5;

        // Check memory layout.
        var ptr = (int*)&valueDerived;
        Console.WriteLine("valueDerived.A = {0}", ptr[0]); // Should be A = 0
        Console.WriteLine("valueDerived.B = {0}", ptr[1]); // Should be B = 0
        Console.WriteLine("valueDerived.C = {0}", ptr[2]); // Should be C = 5
    }
}
```

Just compile the code above with a regular csc command and then after, apply the following program using [Cecil](#) to generate an assembly with a struct inheritance:

```
using System.IO;
using Mono.Cecil;

namespace TestStructInheritance
{
    class Program
    {
        static void Main(string[] args)
        {
            // Load the assembly
            var assemblyPath = args[0];
            var assembly = AssemblyDefinition.ReadAssembly(assemblyPath);

            // Remove sealed from ValueBase
            var valueBase = assembly.MainModule.GetType("ValueBase");
            valueBase.Attributes &= ~TypeAttributes.Sealed;

            // Remove sealed from ValueDerived and add ValueBase as base.
            var valueDerived = assembly.MainModule.GetType("ValueDerived");
            valueDerived.Attributes &= ~TypeAttributes.Sealed;
            valueDerived.BaseType = valueBase;

            // Save!
            assembly.Write(Path.ChangeExtension(assemblyPath, ".new.exe"));
        }
    }
}
```

and run [CoreRun HelloWorld.exe](#) to check what kind of errors it generates:

```
Unhandled Exception: System.TypeLoadException: Could not load type 'ValueDerived' from
assembly 'HelloWorld, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null' because the
parent type is sealed.
at Program.Main(String[] args)
```

As I didn't know where to start from, getting the error was already a first step to identify where to fix stuff to go further, so looking into the code, the error was thrown from [methodtablebuilder.cpp at the line 13100](#).

```

// Since methods on System.Array assume the layout of arrays, we can not allow
// subclassing of arrays, it is sealed from the users point of view.
// Value types and enums should be sealed - disable inheritting from them (we cannot re
// flag because of AppCompat)
if (pParentMethodTable->IsSealed() ||
    (pParentMethodTable == g_pArrayClass) ||
    pParentMethodTable->IsValueType())
{
    pAssembly->ThrowTypeLoadException(pInternalImport, cl, IDS_CLASSLOAD_SEALEDPARENT)
}

```

So removing the test `pParentMethodTable->IsValueType()` was just fine to let the runtime to advance a bit more. Then, I had to step-by-step into the CLR to check through what kind of checks it verifies a valuetype. While I was not particularly aware about how the CLR was precisely working on this matter, It didn't take much to try a couple of changes and to finally end-up with the simple fix (see the [commit on github](#)):

In [methodtablebuilder.cpp:1502-1512](#) :

```

// Check to see if the class is a valuetype; but we don't want to mark System.Enum
// as a ValueType. To accomplish this, the check takes advantage of the fact
// that System.ValueType and System.Enum are loaded one immediately after the
// other in that order, and so if the parent MethodTable is System.ValueType and
// the System.Enum MethodTable is unset, then we must be building System.Enum and
// so we don't mark it as a ValueType.
if (HasParent() &&
    ((g_pEnumClass != NULL && GetParentMethodTable() == g_pValueTypeClass) ||
     GetParentMethodTable() == g_pEnumClass))
{
    bmtProp->fIsValueClass = true;
}

```

I only had to change the condition to take into account the inheritance:

```

if (HasParent() &&
    ((g_pEnumClass != NULL && (GetParentMethodTable() == g_pValueTypeClass || GetParentMethodTable() == g_pEnumClass))
     GetParentMethodTable() == g_pEnumClass))
{
    bmtProp->fIsValueClass = true;
}

```

As the `bmtProp->fIsValueClass = true;` is indirectly used all around to check whether a class is a value type.

And the `HelloWorld.cs` above was able to output:

```

valueDerived.A = 0
valueDerived.B = 0
valueDerived.C = 5

```

**Bingo!** Of course, I was not sure that this code change would be enough (and I'm still not, as I have only scratched the surface and haven't checked any regression made by this code change) but It was already good enough to check how much it would require to upgrade Roslyn to support this (instead of relying on Cecil to modify the code, which is quite limited).

It turns out that modifications for Roslyn, while still minor, where a bit more involving (as I didn't have much experience with the source, so it didn't help...)

## Bringing struct inheritance to Roslyn

I didn't want to go too much into the details of a full/clean implem, but I still wanted to support the following scenarios:

- **Allow to access base field from derived struct:**

```

var valueDerived = new ValueDerived();
// Object initializer is not supported in this example, so we initialize fields directly
valueDerived.A = 1;
valueDerived.B = 3;
valueDerived.C = 5;

```

- **Allow to explicitly cast from a derived struct to a base struct** (like the compiler requires a cast from `int` to `short`), and correctly generate the object slicing code (copying only the base value from the derived struct):

```

var valueBaseFromDerived = (ValueBase)valueDerived

```

- **Allow to invoke a method with a by ref on base** with passing a derived value struct by ref without casting (as this is valid):

```
public static int StaticCalculateByRefBase(ref ValueBase value)
{
    return value.A + value.B;
}

[...]

// Call a static method with by ref to base type
var result1 = StaticCalculateByRefBase(ref valueDerived);
```

- Allow to invoke a base method from a derived method in a struct. This is not about supporting `virtual/override` which should be in fact forbidden for structs, but to allow to simply call to base method (note the usage of `new` on the method instead of `virtual/override`):

```
public struct ValueBase
{
    public int A;















    public int B;

    public int Calculate()
    {
        return this.A + this.B;
    }
}

public struct ValueDerived : ValueBase
{
    public int C;

    public new int Calculate()
    {
        return base.Calculate() + C; // call base method Calculate()
    }
}
```

And here goes the [list of commits](#) required to support these scenarios:

Commits on Sep 27, 2015		
 <b>Generates valid IL code for the downcasting of a struct</b> xoofox authored 3 minutes ago	 8e2a127	
 <b>Allow usage of base.MethodBase() in struct methods</b> xoofox authored 3 minutes ago	 0671169	
 <b>Allow downcasting from derived to base for valuetypes</b> xoofox authored 4 minutes ago	 421c0be	
 <b>Allow to pass a derived struct instance to a method with a parameter ...</b> <span>...</span> xoofox authored 5 minutes ago	 cef6e29	
 <b>Allow struct inheritance to pass</b> xoofox authored 7 minutes ago	 3c656f9	
 <b>Remove sealed from struct</b> xoofox authored 7 minutes ago	 4121345	

The only changes related to the IL codegen part was for the downcast from a derived type to a base struct type (commit [8e2a127](#)), which basically loads the address of the operand onto the stack (`Ldloca_S` and friends) and then use this address to load the data using the base struct (`ldobj class`):

```
case ConversionKind.ExplicitReference:
    // Handle valuetype cast here
    if (conversion.Type.IsValueType && conversion.Operand.Type.IsValueType)
    {
        EmitAddress(conversion.Operand, AddressKind.ReadOnly);
        _builder.EmitOpCode(LOpCode.Ldobj);
        EmitSymbolToken(conversion.Type, conversion.Syntax);
        return;
    }
    break;
```

and handling of the base method call (commit [0671169](#)):

```
case BoundKind.BaseReference:
    Debug.Assert(expression.Type.IsValueType, "only value types may need a ref to base");
    _builder.EmitOpCode(LOpCode.Ldarg_0);
    break;
```

As you can see, the changes are relatively small. Again, I didn't test a lot all the regressions (and invalid code) that could be introduced. Take this whole thing as just a proof of concept, nothing more!

## The test case

Here is the sourcecode for the testcase I used to test this struct inheritance CoreCLR/Roslyn patches (or grab it on [gist](#))

```

using System;

public struct ValueBase
{
    public int A;

    public int B;

    public int Calculate()
    {
        return this.A + this.B;
    }
}

public struct ValueDerived : ValueBase
{
    public int C;

    public new int Calculate()
    {
        return base.Calculate() + C;
    }
}

public class Program
{
    public static int StaticCalculateByRefBase(ref ValueBase value)
    {
        return value.A + value.B;
    }

    public static int StaticCalculateOnBase(ValueBase value)
    {
        return value.A + value.B;
    }

    static void Assert(string context, int value, int expected)
    {
        Console.WriteLine(context + " = {0}, expect = {1}, {2}", value, expected, value ==
    }

    public unsafe static void Main(string[] args)
    {
        var valueDerived = new ValueDerived();
        // Object initializer is not supported in this example, so we initialize fields directly
        valueDerived.A = 1;
        valueDerived.B = 3;
        valueDerived.C = 5;

        // Check memory layout.
        var ptr = (int*)&valueDerived;
        Assert("valueDerived.A", valueDerived.A, ptr[0]);
        Assert("valueDerived.B", valueDerived.B, ptr[1]);
        Assert("valueDerived.C", valueDerived.C, ptr[2]);

        // Call the Calculate method on derived struct
        Assert("valueDerived.Calculate()", valueDerived.Calculate(), 1 + 3 + 5);

        // Call the Calculate method on base struct
        var valueBase = (ValueBase)valueDerived;
        Assert("valueBase.Calculate()", valueBase.Calculate(), 1 + 3);

        // Call a static method with by ref to base type
        var result1 = StaticCalculateByRefBase(ref valueDerived);
        Assert("StaticCalculateByRefBase(ref valueDerived)", result1, 1 + 3);

        // Call a static method with cast to base type (no cast will generate an error)
        var result2 = StaticCalculateOnBase((ValueBase)valueDerived);
        Assert("StaticCalculateOnBase((ValueBase)valueDerived)", result2, 1 + 3);
    }
}

```

## Next?

I'm curious to here your thoughts. Am-I alone for wanting this struct inheritance feature to get real in C#? :D Let me know what do you think about the concept!

While struct inheritance is a little thing, there are more interesting things like interface implementation (Allow to add implementation to interfaces, this could even lower a bit the need for struct inheritance) and still bigger things related to improve data locality in C#, which is really **THE next big thing** after **LLILC**, if C# wants to tackle some of the last bits of performance:

- Allow to allocate class instance on the stack

- Allow to allocate class instance embedded into another class

Not sure If I will have the courage, the time, and the skills, honestly, they should be a bit more involving in terms of modifications... but even the idea of being able to work on this is quite exciting!

Anyway, there is a new toy in town, happy coding with Struct Inheritance now ;)