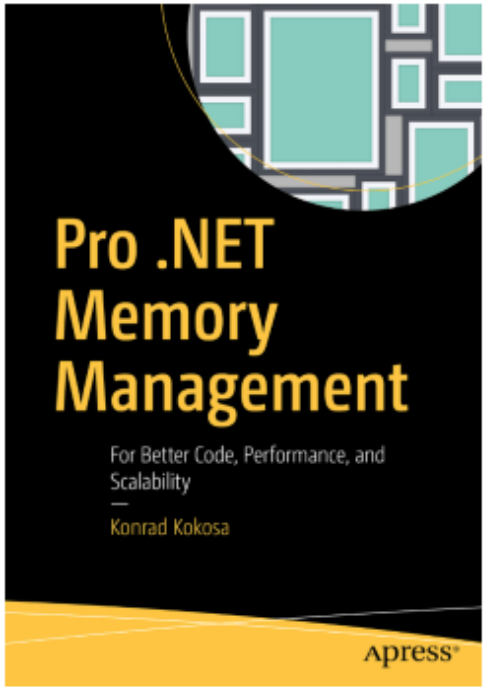
`ref struct`, `ref readonly struct`, `ref` returning, `ref` everything!

Konrad Kokosa

@konradkokosa

https://prodotnetmemory.com

@konradkokosa
TooSlowException.com
ProDotNetMemory.com



Let's start from the beginning...

Let's start from the beginning...

Class vs struct

```
class Program
{
    static int Main(string[] args)
    {
        static int Main(string[] args) => Experiment.Run();
    }
}

public class Experiment
{
    public static int Run()
    {
        SomeType data = new SomeType();
        data.F = 10;
        Helper(data);
        return data.F;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void Helper(SomeType data)
    {
        data.F = 12;
        //Console.WriteLine(data.F);
    }
}
```

```
public static int Run()
{
    SomeType data = new SomeType(); data.F = 10;
    Helper(data);
    return data.F;
}
```

```
public class SomeType
{
    public int F;
}
```

```
public struct SomeType
{
    public int F;
}
```

```
Run():
    mov     rcx,7FFC9AB5CC38h
    call    coreclr!alloc
    mov     rsi,rax
    mov     dword ptr [rsi+8],0Ah

    mov     rcx,rsi
    call    Helper(SomeType)

    mov     eax,dword ptr [rsi+8]
; Total bytes of code 36
```

```
Run():
    mov     ecx,0Ah
    call    Helper(SomeType)

    mov     eax,0Ah
; Total bytes of code 15
```

```
public static int Run()
{
    SomeType data = new SomeType(); data.F = 10;
    Helper(data);
    return data.F;
}
```

```
public class SomeType
{
    public int F; public int F2;
    public int F3; public int F4;
    public int F5;
}
```

```
Run():
    mov     rcx,7FFCC53BCC78h
    call    coreclr!alloc
    mov     rsi,rax
    mov     dword ptr [rsi+8],0Ah

    mov     rcx,rsi
    call    Helper(SomeType)

    mov     eax,dword ptr [rsi+8]
; Total bytes of code 36
```

```
public struct SomeType
{
    public int F; public int F2;
    public int F3; public int F4;
    public int F5;
}
```

```
Run():
    xor     ecx,ecx
    lea     rax,[rsp+38h]
    vxorps  xmm0,xmm0,xmm0
    vmovdqu xmmword ptr [rax],xmm0
    mov     dword ptr [rax+10h],ecx
    mov     dword ptr [rsp+38h],0Ah
    vmovdqu xmm0,xmmword ptr [rsp+38h]
    vmovdqu xmmword ptr [rsp+20h],xmm0
    mov     ecx,dword ptr [rsp+48h]
    mov     dword ptr [rsp+30h],ecx
    lea     rcx,[rsp+20h]
    call    Helper(SomeType)
    mov     eax,dword ptr [rsp+38h]
; Total bytes of code 60
```

```
public static int Run()
{
    SomeType data = new SomeType();
    data.F = 10;
    Helper(data);
    return data.F;
}

public struct SomeType
{
    public int F; public int F2;
    public int F3; public int F4;
    public int F5;
}
```

```
public static int Run()
{
    SomeType data = new SomeType();
    data.F = 10;
    Helper(ref data);
    return data.F;
}

public struct SomeType
{
    public int F; public int F2;
    public int F3; public int F4;
    public int F5;
}
```

```
Run():
    xor      ecx,ecx
    lea      rax,[rsp+38h]
    vxorps   xmm0,xmm0,xmm0
    vmovdqu  xmmword ptr [rax],xmm0
    mov      dword ptr [rax+10h],ecx
    mov      dword ptr [rsp+38h],0Ah
    vmovdqu  xmm0,xmmword ptr [rsp+38h]
    vmovdqu  xmmword ptr [rsp+20h],xmm0
    mov      ecx,dword ptr [rsp+48h]
    mov      dword ptr [rsp+30h],ecx
    lea      rcx,[rsp+20h]
    call     Helper(SomeType)
    mov      eax,dword ptr [rsp+38h]
; Total bytes of code 60
```

```
ConsoleApp.Experiment.Run():
    xor      ecx,ecx
    lea      rax,[rsp+28h]
    vxorps   xmm0,xmm0,xmm0
    vmovdqu  xmmword ptr [rax],xmm0
    mov      dword ptr [rax+10h],ecx
    mov      dword ptr [rsp+28h],0Ah
    lea      rcx,[rsp+28h]
    call     Helper(SomeType ByRef)
    mov      eax,dword ptr [rsp+28h]
; Total bytes of code 40
```

Manager pointer (byref)

Manager pointer

- strongly typed
  - i.e. `System.Int32&` or `SomeType&`
- it can point to:
  - local variable
  - method's argument
  - object's field
  - array's element
- it can live as:
  - local variable
  - method's argument
  - method's return value
- **CRITICAL** - it cannot land in the Managed Heap

  "they cannot be used for field signatures, as the element type of an array and boxing a value of managed pointer type is disallowed. Using a managed pointer type for the return type of methods is not verifiable" - ECMA-335

Manager pointer

- strongly typed
  - i.e. `System.Int32&` or `SomeType&`
- it can point to:
  - local variable
  - method's argument
  - object's field
  - array's element
- it can live as:
  - local variable
  - method's argument
  - method's return value
- **CRITICAL** - it cannot land in the Managed Heap

  "they cannot be used for field signatures, as the element type of an array and boxing a value of managed pointer type is disallowed. Using a managed pointer type for the return type of methods is not verifiable" - ECMA-335

  "Note. Recently, since C# 7.0, managed pointers usage has been widened in the form of ref locals and ref returns (collectively referred to as ref variables). Thus, the last sentence from the above ECMA citation about using a managed pointer type as the return type has been relaxed."

Ref parameters

Managed pointers - aka *byref type*, because passing an argument "**by ref**erence" is nothing else than using them:

```
private static void Helper(ref SomeType data)
{
    data.Field = 11;
}
```

```
  Helper(SomeType ByRef):
    IL_0000: ldarg.0
    IL_0001: ldc.i4.s 12
    IL_0003: stfld System.Int32 ConsoleApp.SomeType::F
    mov      dword ptr [rcx],0Ch
    IL_0008: ret
    ret
; Total bytes of code 7
```

Ref locals (C# 7.0+)

Local variable storing *managed pointer*:

```
private static void Helper(ref SomeType data)
{
    ref SomeType refSomeType = ref data;
    ref int refData = ref data.F3;

    Console.WriteLine(refSomeType.F2);
    Console.WriteLine(refData);
}
```

Ref return (C# 7.0+)

Return type as *managed pointer* - obviously with limitations:

> *"The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method"* - MSDN

Ref return (C# 7.0+)

Return type as *managed pointer* - obviously with limitations:

> *"The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method"* - MSDN

Bad:

```
public static ref int ReturnByRefValueTypeInterior(int index)
{
   int localInt = 7;
   return ref localInt; // Compilation error: Cannot return local 'localInt' by
                        // reference because it is not a ref local
}
```

Ref return (C# 7.0+)

Return type as *managed pointer* - obviously with limitations:

> *"The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method"* - MSDN

Bad:

```
public static ref int ReturnByRefValueTypeInterior(int index)
{
   int localInt = 7;
   return ref localInt; // Compilation error: Cannot return local 'localInt' by
                        // reference because it is not a ref local
}
```

Good:

```
public static ref int GetArrayElementByRef(int[] array, int index)
{
   return ref array[index];
}
```

Ref return (cd.)

Bad or good?

```
public static ref int ReturnByRefReferenceTypeInterior(int index)
{
   int[] localArray = new[] { 1, 2, 3 };
   return ref localArray[index];
}
```

Ref return - an example

```
public class BookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new BookCollection();
        var book = collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Ref return - an example

```
public class BookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new BookCollection();
        var book = collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Question: what is the result?

Ref return - an example

```
public class BookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new BookCollection();
        var book = collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Question: what is the result?

```
Konrad Kokosa
```

Ref return - an example

```
public class RefBookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefBookCollection();
        ref var book = ref collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Ref return - an example

```
public class RefBookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefBookCollection();
        ref var book = ref collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Question: what is the result?

Ref return - an example

```
public class RefBookCollection
{
    private Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefBookCollection();
        ref var book = ref collection.GetBookByTitle("Tale of Two Cities");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(book.Author);
    }
}
```

Question: what is the result?

```
Konrad Kokosa
```

Ref return - an example

```
public class ValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new ValueBookCollection();
        var book = collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Ref return - an example

```
public class ValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new ValueBookCollection();
        var book = collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Question: what is the result?

Ref return - an example

```
public class ValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return books[ctr];
        return nobook;
    }

    public static void Run()
    {
        var collection = new ValueBookCollection();
        var book = collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Question: what is the result?

```
Jack London
```

Ref return - an example

```
public class RefValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ref ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefValueBookCollection();
        ref var book = ref collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Ref return - an example

```
public class RefValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ref ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefValueBookCollection();
        ref var book = ref collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Question: what is the result?

Ref return - an example

```
public class RefValueBookCollection
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild, The", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities, A", Author = "Charles Dickens
    private ValueBook nobook = default;
    public ref ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new RefValueBookCollection();
        ref var book = ref collection.GetBookByTitle("Call of the Wild, The");
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

Question: what is the result?

```
Konrad Kokosa
```

Ref returning collections

We can expect a new API appearing:

```
public class SomeStructRefList
{
    private SomeStruct[] items;
    public ref SomeStruct this[int index] => ref items[index];
}
```

As indexers are already taken (and we cannot break their back-compatibility) - `ItemRef` was choosen:

```
// ImmutableArray:
public ref readonly T ItemRef(int index)
{
    return ref this.array[index];
}
```

Currently:

- most of `System.Collections.Immutable`.
- problems with `List<T>` and `Dictionary<TKey, TValue>` - why?

ref ternary expression (C# 7.2)

Instead of a well-known `?:`:

```
<condition> ? <consequence> : <alternative>
```

we can `ref` it:

```
<condition> ? ref <consequence> : ref <alternative>
```

Examples:

```
public ref ValueBook RefTernary()
{
    // byref parameter
    Do(ref (books != null ? ref books[0] : ref nobook));
    // byref assignment
    (books != null ? ref books[0].Author : ref nobook.Author) = "Aaa";
    // byref method call
    (books != null ? ref books[0] : ref nobook).ModifyAuthor("AAA");
    // byref return
    return ref books?.Length > 0 ? ref books[0] : ref nobook;
}

public void Do(ref ValueBook book) { }
```

ref foreach (C# 7.3)

Thanks to a *ref reassignment* we can "iterate ref" if `Current` from an enumerator *ref returns*:

```
public int Hyperlinq_Array()
{
    var count = 0;
    foreach(ref readonly var item in array.Where(_ => true))
        count++;
    return count;
}
```

ref foreach (C# 7.3)

Thanks to a *ref reassignment* we can "iterate ref" if `Current` from an enumerator *ref returns*:

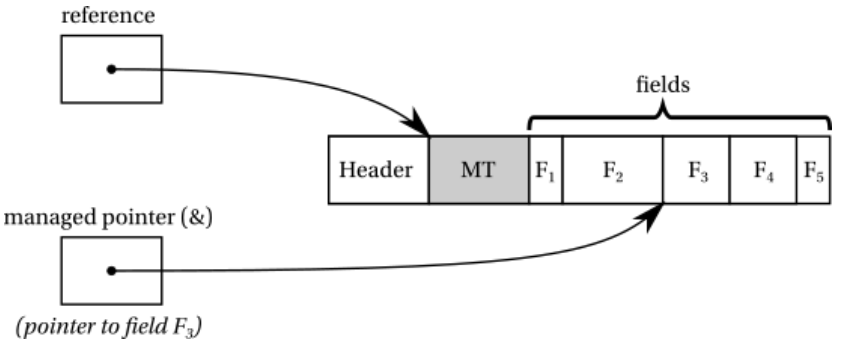
```
public int Hyperlinq_Array()
{
    var count = 0;
    foreach(ref readonly var item in array.Where(_ => true))
        count++;
    return count;
}
```

Currently there are no APIs supporting it, but who knows?!

Byref internals

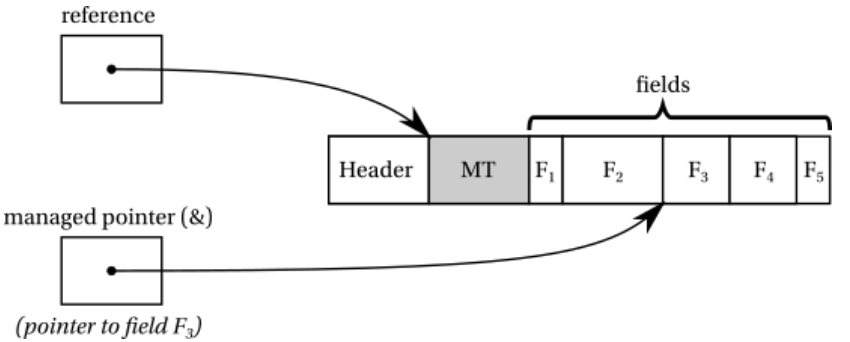
Byref internals

```
public static ref int ReturnByRefReferenceTypeInterior(int index)
{
   int[] localArray = new[] { 1, 2, 3 };
   return ref localArray[index];
}

ref int local = ref ReturnByRefReferenceTypeInterior(2);
```



Byref internals

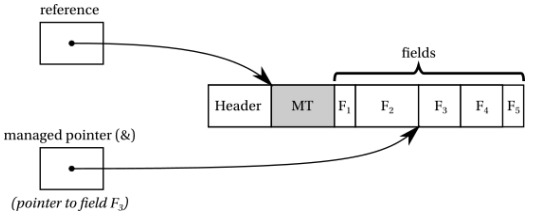
```
public static ref int ReturnByRefReferenceTypeInterior(int index)
{
   int[] localArray = new[] { 1, 2, 3 };
   return ref localArray[index];
}

ref int local = ref ReturnByRefReferenceTypeInterior(2);
```



- `local` is seen as so-called *interior pointer* - a pointer **into** an object
- it is just an ordinary pointer...
  - unless GC happens - it now must be interpreted as the containing object root

Byref internals (cd.)

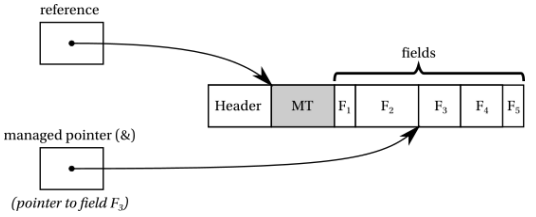
```
ref int local = ref ReturnByRef(2);
```



Byref internals (cd.)

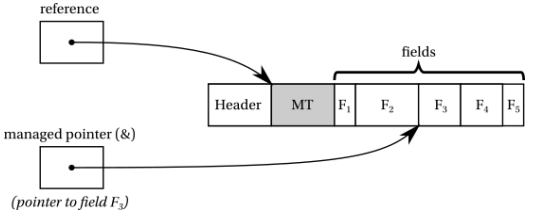
```
ref int local = ref ReturnByRef(2);
```

Interpretation:

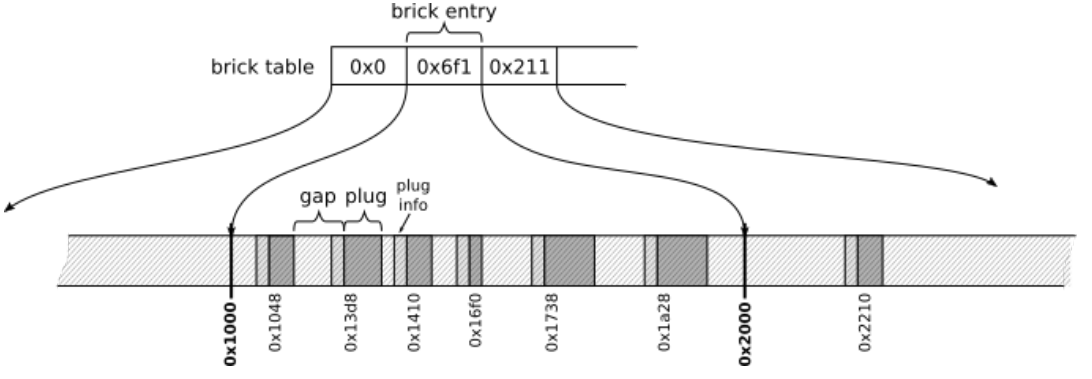


Byref internals (cd.)

```
ref int local = ref ReturnByRef(2);
```



Interpretation:

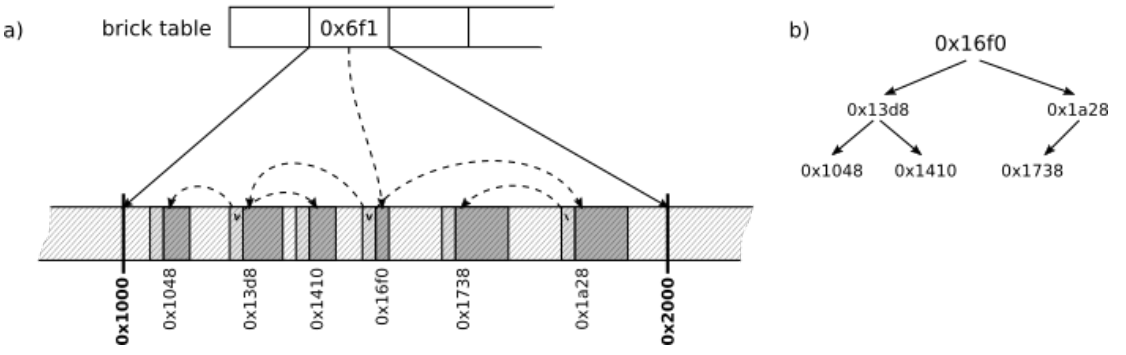


brick - 4kB memory region:

- during *Mark* phase - we serch inside **the whole region** to find a corresponding object for an *interior pointer*

Byref internals (cd.)

Interpretation:



brick - 4kB memory region:

- during *Compact* phase - we just search inside a specific *plug* found through *plug tree*

Byref internals (cd.)

```
[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref int data)
{
  data = 11; // it now has to be a root!
}
```

Byref internals (cd.)

```
[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref int data)
{
  data = 11; // it now has to be a root!
}
```

```
PassingByref.Test(Int32 ByRef)
L0000: mov dword [rcx], 0xb
L0006: ret
```

Byref internals (cd.)

```
[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref int data)
{
  data = 11; // it now has to be a root!
}
```

```
PassingByref.Test(Int32 ByRef)
L0000: mov dword [rcx], 0xb
L0006: ret
```

```
PassingByref.Test(Int32 ByRef)
push rdi
push rsi
sub rsp,28h
mov rsi,rcx
00000009 interruptible
00000009 +rsi(interior)
...
0000003a not interruptible
0000003a -rsi(interior)
add rsp,28h
pop rsi
pop rdi
ret
```

Byref internals (cd.)

```
[MethodImpl(MethodImplOptions.NoInlining)]
public static void Test(ref int data)
{
  data = 11; // it now has to be a root!
}
```

```
PassingByref.Test(Int32 ByRef)
L0000: mov dword [rcx], 0xb
L0006: ret
```

```
PassingByref.Test(Int32 ByRef)
push rdi
push rsi
sub rsp,28h
mov rsi,rcx
00000009 interruptible
00000009 +rsi(interior)
...
0000003a not interruptible
0000003a -rsi(interior)
add rsp,28h
pop rsi
pop rdi
ret
```

A method may be:

- *fully interruptible* - it can be suspended at "every line"
- *partially interruptible* - it can be suspended only at "safe-point" (mainly method calls)
- not interruptible at all

For the first two - there is a risk of the GC suspension overhead.

So below code just works!

```
public class Helpers {
   public static ref T MakeInterior<T>(T obj) => ref (new T[] { obj })[0];
}
```

readonly semantics

Readonly ref variables (C# 7.2)

- it disallows changing a value stored in the *managed pointer*:
  - for reference-type - it is a reference
  - for value-type - it is the value itself
- two forms in C#:
  - `ref readonly` - for return values and local variables
  - `in` - for method parameters

ref readonly - reference-typed

```
public class BookCollectionByReadonlyRef
{
    public Book[] books = {
        new Book { Title = "Call of the Wild", Author = "Jack London" },
        new Book { Title = "Tale of Two Cities", Author = "Charles Dickens" } };
    private Book nobook = default;
    public ref readonly Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new BookCollectionByReadonlyRef();
        ref readonly var book = ref collection.GetBookByTitle("Call of the Wild");
        //book = new Book();
        book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

ref readonly - value-typed

```
public class ValueBookCollectionByReadonlyRef
{
    public ValueBook[] books = {
        new ValueBook { Title = "Call of the Wild", Author = "Jack London" },
        new ValueBook { Title = "Tale of Two Cities", Author = "Charles Dickens" }
    private ValueBook nobook = default;
    public ref readonly ValueBook GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
            if (title == books[ctr].Title)
                return ref books[ctr];
        return ref nobook;
    }

    public static void Run()
    {
        var collection = new ValueBookCollectionByReadonlyRef();
        ref readonly var book = ref collection.GetBookByTitle("Call of the Wild");
        //book = new Book();
        //book.Author = "Konrad Kokosa";
        Console.WriteLine(collection.books[0].Author);
    }
}
```

ref readonly - an example

```
struct Point3D
{
   private static Point3D origin = new Point3D();
   public static ref readonly Point3D Origin => ref origin;
   ...
}
```

in parameter

```
public class BookCollection
{
   ...
   public void CheckBook(in ValueBook book)
   {
      book.Title = "XXX"; // Cannot assign to a member of variable 'in ValueBook'
                          // because it is a readonly variable.
   }
}
```

```
collection.CheckBook(in someBook);
```

Remember - it disallows value change:

- for reference-type - the reference itself (as the reference is the "value")
- for value-type - the value itself

in parameter

Method arguments, lambdas/delegates, local functions, indexers, extension methods...

```
public class InParameters
{
    private delegate string BookRefAction(in ValueBook t);

    void Method(in ValueBook book)
    {
        string LocalFunction(in ValueBook b)
        {
            return b.Title.ToLowerInvariant();
        }

        BookRefAction action = (in ValueBook x) => x.Author + x.Title;
    }

    public string this[in Book book] => "something";
}
```

```
public static class GuidExtensions
{
    public static string Transform(in this Guid guid)
    {
        return guid.GetHashCode().ToString();
    }
}
```

But...

```
public struct ValueBook
{
    public string Title;
    public string Author;
    public void ModifyAuthor(string author)
    {
        this.Author = author;
    }
}
```

```
public static void Run()
{
    var collection = new ValueBookCollectionByReadonlyRef();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild");
    //book = new Book();
    //book.Author = "Konrad Kokosa";
    book.ModifyAuthor("Konrad Kokosa");
    Console.WriteLine(collection.books[0].Author);
}
```

But...

```
public struct ValueBook
{
    public string Title;
    public string Author;
    public void ModifyAuthor(string author)
    {
        this.Author = author;
    }
}
```

```
public static void Run()
{
    var collection = new ValueBookCollectionByReadonlyRef();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild");
    //book = new Book();
    //book.Author = "Konrad Kokosa";
    book.ModifyAuthor("Konrad Kokosa");
    Console.WriteLine(collection.books[0].Author);
}
```

Question: What will happen? What is the result?

But...

```
public struct ValueBook
{
    public string Title;
    public string Author;
    public void ModifyAuthor(string author)
    {
        this.Author = author;
    }
}
```

```
public static void Run()
{
    var collection = new ValueBookCollectionByReadonlyRef();
    ref readonly var book = ref collection.GetBookByTitle("Call of the Wild");
    //book = new Book();
    //book.Author = "Konrad Kokosa";
    book.ModifyAuthor("Konrad Kokosa");
    Console.WriteLine(collection.books[0].Author);
}
```

Question: What will happen? What is the result?

```
Jack London
```

Defensive copy

```
ref readonly var book = ref collection.GetBookByTitle("Tale of Two Cities, A");
book.ModifyAuthor("Konrad Kokosa");
Console.WriteLine(book.Author);
```

Defensive copy

```
ref readonly var book = ref collection.GetBookByTitle("Tale of Two Cities, A");
book.ModifyAuthor("Konrad Kokosa");
Console.WriteLine(book.Author);
```

```
.method public hidebysig static
    void Main () cil managed
{
    .locals init (
        [0] valuetype Book
    )
    IL_0000: newobj instance void BookCollection::.ctor()
    IL_0005: ldstr "Tale of Two Cities, A"
    IL_000a: callvirt instance valuetype Book& BookCollection::GetBookByTitle(stri
    IL_000f: ldobj Book
    IL_0014: stloc.0
    IL_0015: ldloca.s 0
    IL_0017: ldstr "Konrad Kokosa"
    IL_001c: call instance void Book::ModifyAuthor(string)
    IL_0021: ret
}
```

"ldobj - Copy the value stored at address src to the stack"

Defensive copy (cd.)

```
public class BookCollection
{
   ...
   public void CheckBook(in Book book)
   {
      book.ModifyAuthor(); // Calling on the defensive copy, original not touched
   }
}
```

Question: how to avoid that? We can, stay tuned!

Sidenote: *Defensive copy* (cd.) - not only for *byref*

```
class Program
{
    private static readonly TestStruct _Account = new TestStruct();
    static void Main(string[] args)
    {
        Console.WriteLine(_Account.MyMoney);
        _Account.UpdateValue(100);
        Console.WriteLine(_Account.MyMoney);
    }

    public struct TestStruct
    {
        private int _money;
        public int MyMoney => _money;
        public void UpdateValue(int moneyAmount)
        {
            _money += moneyAmount;
        }
    }
}
```

Result is still `0`!

Sidenote: *Defensive copy* (cd.) - not only for *byref*

We can be surprised sometimes - [Readonly Structs vs Classes have dangerous inconsistency - failed spin lock]():

```
class Program
{
    private static readonly SpinLock sl = new SpinLock();
    static void Main(string[] args)
    {
        // good luck with sl.Enter and sl.Exit
    }
}
```

Sidenote: *Defensive copy* (cd.) - not only for *byref*

We can be surprised sometimes - [Readonly Structs vs Classes have dangerous inconsistency - failed spin lock]:

```
class Program
{
    private static readonly SpinLock sl = new SpinLock();
    static void Main(string[] args)
    {
        // good luck with sl.Enter and sl.Exit
    }
}
```

Fresh stuff: [Readonly structs should warn when fields are implicitly copied for member invocation #33968]

Readonly struct

Readonly struct

- Non-mutable struct - we cannot modify it after creation
- Limitations:
  - all fields must be `readonly`
  - initializing constructor is required

```
public readonly struct ReadonlyValueBook
{
    public readonly string Title;
    public readonly string Author;

    public ReadonlyValueBook(string title, string author)
    {
        this.Title = title;
        this.Author = author;
    }

    public void ModifyAuthor()
    {
        //this.Author = "XXX"; // A readonly field cannot be assigned to (except
                               // in a constructor or a variable initializer)
        Console.WriteLine(this.Author);
    }
}
```

Readonly struct

- Non-mutable struct - we cannot modify it after creation
- Limitations:
  - all fields must be `readonly`
  - initializing constructor is required

```
public readonly struct ReadonlyValueBook
{
    public readonly string Title;
    public readonly string Author;

    public ReadonlyValueBook(string title, string author)
    {
        this.Title = title;
        this.Author = author;
    }

    public void ModifyAuthor()
    {
        //this.Author = "XXX"; // A readonly field cannot be assigned to (except
                               // in a constructor or a variable initializer)
        Console.WriteLine(this.Author);
    }
}
```

Compiler/JIT can treat *readonly refs* much better!

```
public class ReadOnlyBookCollection
{
   private ReadonlyValueBook[] books = {
      new ReadonlyValueBook("Call of the Wild", "Jack London" ),
      new ReadonlyValueBook("Tale of Two Cities", "Charles Dickens") };
   private ReadonlyValueBook nobook = default;

   public ref readonly ReadonlyValueBook GetBookByTitle(string title)
   {
      for (int ctr = 0; ctr < books.Length; ctr++)
         if (title == books[ctr].Title)
            return ref books[ctr];
      return ref nobook;
   }

   public void CheckBook(in ReadonlyBook book)
   {
      //book.Title = "XXX"; // Would generate compiler error.
      book.ModifyAuthor(); // It is guaranteed that ModifyAuthor does not modify
                           // book's fields. No defensive copy created.
   }
}

public static void Run()
{
   var coll = new ReadOnlyBookCollection();
   ref readonly var book = ref coll.GetBookByTitle("Call of the Wild");
   //book.Author = "XXX"; // A readonly field cannot be assigned to (except in a
                          // constructor or a variable initializer)
}
```

Readonly struct vs defensive copy

```
public void CheckBook(in ReadonlyBook book)
{
  book.ModifyAuthor();
}
```

readonly struct:

```
.method public hidebysig
instance void CheckBook (
    [in] valuetype ReadonlyBook& book
) cil managed
{
    ldarg.1
    call instance void ModifyAuthor()
    ret
}
```

normal struct:

```
.method public hidebysig
instance void CheckBook (
   [in] valuetype ReadonlyBook& book
) cil managed
{
    .locals init (
        [0] valuetype ReadonlyBook
    )
    ldarg.1
    ldobj ReadonlyBook
    stloc.0
    ldloca.s 0
    call instance void ModifyAuthor()
    ret
}
```

Ref struct

Ref struct (byref-like type)

- we would like to have a type that can contains *managed pointers* as fields ("*byrefs*")
  - it should have similar limitations as *byrefs*
  - hence they would become... *byref-like type*
- `ref struct` introduced in C# is such a *byref-like* type:

```
public ref struct RefBook
{
    public string Title;
    public string Author;
}
```

- a bunch of limitations:
  - it cannot be a field of a class or struct (boxing!)
  - it cannot be a static field
  - it cannot be an array type element (boxing!)
  - it cannot be *boxed* explicitly
  - it cannot implement an interface
  - it cannot be a generic type argument
  - it cannot be a local variable in `async` method
  - it cannot be a part of *closure*
- everything **to not occur on the Managed Heap** so it **can contain *managed pointer***

Ref struct (byref-like type)

Examples of limitations:

```
public class RefBookTest
{
  private RefBook book; // Field or auto-implemented property cannot be of
                        // type 'RefBook' unless it is an instance member
                        // of a ref struct
  private static RefBook book; // j.w.

  public void Test()
  {
    RefBook localBook = new RefBook();
    object box = (object) localBook;  // Cannot convert type 'RefBook' to 'object'
    RefBook[] array = new RefBook[4]; // Array elements cannot be of type 'RefBook
    GenericTest<RefBook>(localBook);  // The type 'RefBook' may not be used as
                                      // a type argument
  }

  public async Task TestAsync()
  {
    RefBook localBook = new RefBook(); // Parameters or locals of type 'RefBook'
                                       // cannot be declared in async methods or
                                       // lambda expressions
    ...
  }

  public void GenericTest<T>(T arg) { ... }
}
```

Ref struct (byref-like type)

It can be a field of other *ref struct*:

```
public ref struct RefBook
{
   public string Title;
   public string Author;
   public RefPublisher Publisher;
}

public ref struct RefPublisher
{
   public string Name;
}
```

It can be a local variable and method's argument (also *byref*):

```
public void Test(RefBook refBook)
{
    RefBook localBook = new RefBook();
}
```

```
public void Test(ref RefBook refBook)
{
    ref RefPublisher refPublisher = ref refBook.Publisher;
}
```

Ref struct (byref-like type)

So what does it give us?

- they are never *heap-allocated*
  - fast allocation/deallocation guaranteed (stack/CPU)
  - guaranteed no GC overhead
    - never leaks through boxing
  - can contain *managed pointer* (*)
- guaranteed safe thread access
  - only single thread can use it
  - no need for any synchronization

Ref struct (byref-like type)

So what does it give us?

- they are never *heap-allocated*
  - fast allocation/deallocation guaranteed (stack/CPU)
  - guaranteed no GC overhead
    - never leaks through boxing
  - can contain *managed pointer* (*)
- guaranteed safe thread access
  - only single thread can use it
  - no need for any synchronization

(*) currently not exposed directly in C#/CIL

Readonly ref struct

```
public readonly ref struct RefBook
{
    public readonly string Title;
    public readonly string Author;
}
```

Disposable ref struct

Questions: what is `IDisposable`?

Disposable ref struct

Questions: what is `IDisposable`?

Answer: Widely established contract that an object has `Dispose` method, that should be called (because an object holds some resources that wants to be cleaned up).

- it does not have anything with the GC itself
- it can be easily checked through static code analysis
- there is a helper in the form of the *using statement*

Disposable ref struct

Questions: what is `IDisposable`?

Answer: Widely established contract that an object has `Dispose` method, that should be called (because an object holds some resources that wants to be cleaned up).

- it does not have anything with the GC itself
- it can be easily checked through static code analysis
- there is a helper in the form of the *using statement*

Hence:

- *explicit cleanup* is a preferred way
- *implicit cleanup* is not possible (`ref struct` cannot have a finalizer)

Disposable ref struct

Question: What's wrong with this code?

```
class Program
{
   static void Main(string[] args)
   {
      using (var book = new RefBook())
      {
         Console.WriteLine(book.Author);
      }
   }
}

ref struct RefBook : IDisposable
{
   public void Dispose()
   {
        // ...
   }
   public string Author;
}
```

Disposable ref struct

Question: What's wrong with this code?

```
class Program
{
   static void Main(string[] args)
   {
      using (var book = new RefBook())
      {
         Console.WriteLine(book.Author);
      }
   }
}

ref struct RefBook : IDisposable
{
   public void Dispose()
   {
        // ...
   }
   public string Author;
}
```

An answer:

```
Error CS8343 'RefBook': ref structs cannot implement interfaces
```

Disposable ref struct

C# 8.0 introduces such possibility:

```
class Program
{
   static void Main(string[] args)
   {
      using (var book = new RefBook()) // Duck-typing
      {
         Console.WriteLine(book.Author);
      }
   }
}

ref struct RefBook
{
   public void Dispose()
   {
        // ...
   }
   public string Author;
}
```

```
public unsafe ref struct UnmanagedArray<T> where T : unmanaged
{
    private readonly T* data;
    public UnmanagedArray(int length)
    {
        data = // get memory from some pool
    }

    public ref T this[int index] => ref data[index];

    public void Dispose()
    {
        // return memory to the pool
    }
}
```

```
static void Main(string[] args)
{
   using (var array = new UnmanagedArray<int>(10))
   {
      Console.WriteLine(array[0]);
   }
}
```

```
static void Main(string[] args)
{
   using var array = new UnmanagedArray<int>(10); // C# 8.0
   Console.WriteLine(array[0]);
}
```

Disposable ref struct - an example

```
internal ref struct ValueUtf8Converter
{
   private byte[] _arrayToReturnToPool;
   ...

   public ValueUtf8Converter(Span<byte> initialBuffer)
   {
      _arrayToReturnToPool = null;
   }

   public Span<byte> ConvertAndTerminateString(ReadOnlySpan<char> value)
   {
      ...
   }

   public void Dispose()
   {
      byte[] toReturn = _arrayToReturnToPool;
      if (toReturn != null)
      {
         _arrayToReturnToPool = null;
         ArrayPool<byte>.Shared.Return(toReturn);
      }
   }
}
```

byref-like instance field

byref-like instance field

So we have our beloved `Span<T>`:

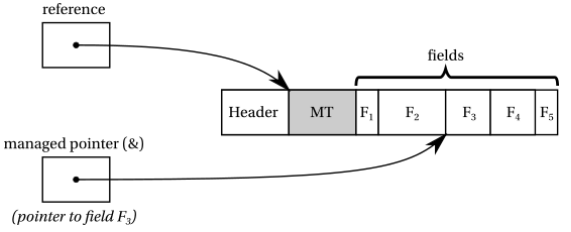
```
var array = new int[64];
Span<int> span1 = new Span<int>(array);
Span<int> span2 = new Span<int>(array, start: 8, length: 4);
```

byref-like instance field

So we have our beloved `Span<T>`:

```
var array = new int[64];
Span<int> span1 = new Span<int>(array);
Span<int> span2 = new Span<int>(array, start: 8, length: 4);
```

We would love to point inside arrays with it:

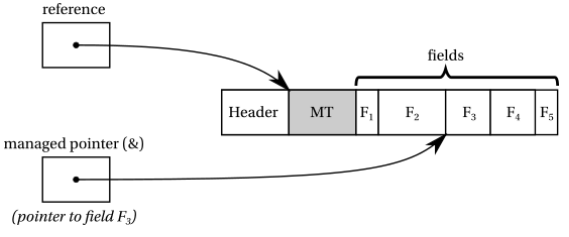


byref-like instance field

So we have our beloved `Span<T>`:

```
var array = new int[64];
Span<int> span1 = new Span<int>(array);
Span<int> span2 = new Span<int>(array, start: 8, length: 4);
```

We would love to point inside arrays with it:



Ideally it could contain a *managed pointer* and a length:

```
public struct Span<T>
{
   internal ref T _pointer;
   private int _length;
   ...
}
```

byref-like instance field

So thanks to the `ref struct`, we have it!

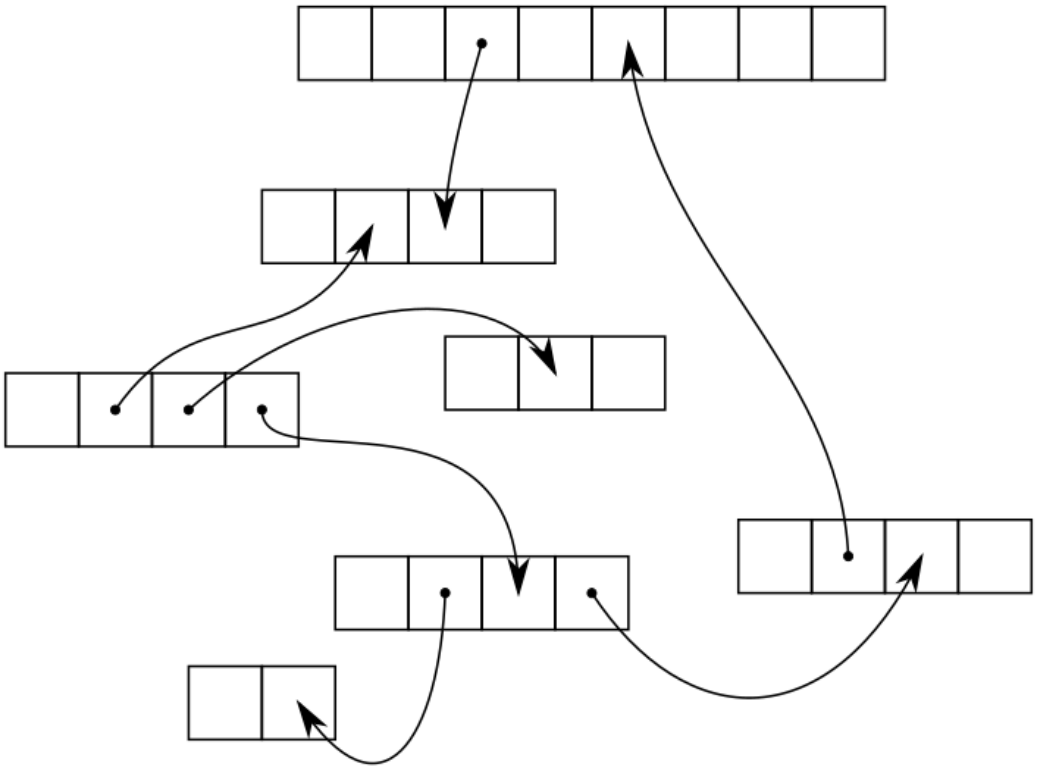
```
public readonly ref struct Span<T>
{
   internal readonly ref T _pointer;
   private readonly int _length;
   ...
}
```

- but `ref` field is not handled by C# nor even CIL
- it is represented by special, intrinsic type `ByReference<T>`
  - `ref T` == `ByReference<T>`

`ByReference` (byref-like instance field)

```
// ByReference<T> is meant to be used to represent "ref T" fields. It is
// working around lack of first class support for byref fields in C# and IL.
// The JIT and type loader has special handling for it that turns it
// into a thin wrapper around ref T.
[NonVersionable]
internal ref struct ByReference<T>
{
   private IntPtr _value;
   ...
}

public readonly ref partial struct Span<T>
{
   /// <summary>A byref or a native ptr.</summary>
   internal readonly ByReference<T> _pointer;
   /// <summary>The number of elements this Span contains.</summary>
   private readonly int _length;
   ...
}
```

General byref-like fields?



Rather - no-no.

Unsafe

Unsafe

```
public static partial class Unsafe
{
    // Casting/reinterpretation
    public unsafe static void* AsPointer<T>(ref T value)
    public unsafe static ref T AsRef<T>(void* source)
    public static ref TTo As<TFrom, TTo>(ref TFrom source)
    // Pointer arithmetic
    public static ref T Add<T>(ref T source, int elementOffset)
    public static ref T Subtract<T>(ref T source, int elementOffset)
    // Informative methods
    public static int SizeOf<T>()
    public static System.IntPtr ByteOffset<T>(ref T origin, ref T target)
    public static bool IsAddressGreaterThan<T>(ref T left, ref T right)
    public static bool IsAddressLessThan<T>(ref T left, ref T right)
    public static bool AreSame<T>(ref T left, ref T right)
    // Memory access methods
    public unsafe static T Read<T>(void* source)
    public unsafe static void Write<T>(void* destination, T value)
    public unsafe static void Copy<T>(void* destination, ref T source)
    // Block-based memory access
    public static void CopyBlock(ref byte dest, ref byte source, uint byteCount)
    public unsafe static void InitBlock(void* address, byte value, uint byteCount)
}
```

Unsafe - examples

```
public class SomeClass
{
    public int Field1; public int Field2;
}
public class SomeOtherClass
{
    public long Field;
}
public void DangerousPlays(SomeClass obj)
{
    ref SomeOtherClass target = ref Unsafe.As<SomeClass, SomeOtherClass>(ref obj);
    Console.WriteLine(target.Field);
}
```

```
// Bit converter
public static byte[] GetBytes(double value)
{
    byte[] bytes = new byte[sizeof(double)];
    Unsafe.As<byte, double>(ref bytes[0]) = value;
    return bytes;
}
```

```
// jemalloc.NET
public unsafe ref C Read<C>(int index) where C : struct
{
    return ref Unsafe.AsRef<C>(PtrTo(index));
}
```

Unsafe - examples

```
public ref struct FixedRefStruct<T> where T : unmanaged
{
    private T field;
    public void Use()
    {
        T* p = &field; // You can only take the address of an unfixed expression
                       // inside of a fixed statement initializer

        unsafe
        {
            fixed (T* ptr = &field)
            {

            }

            T* p2 = (T*) Unsafe.AsPointer(ref field);
        }
    }
}
```

Summary

- *managed pointers* (aka *byref*)
  - *ref parameters* (also *in parameter*)
  - *ref locals*
  - *ref returns* - and *ref returning collections*
  - *ref ternary expression*
- *readonly ref*
  - including *in parameter*
  - *defensive copy*
  - *readonly struct*
- *ref structs* (aka *byref-like type*)
  - including *readonly ref struct*
  - including *disposable ref structs*
- *byref-like instance field*
  - currently only `Span<T>`

But... where?!

- cloud/datacenter scenarios where computation is billed for and responsiveness is a competitive advantage.
- Games/VR/AR with soft-realtime requirements on latencies

```
public ref struct ValueStringBuilder
{
    private char[] _arrayToReturnToPool;
    private Span<char> _chars;
    private int _pos;

    public ValueStringBuilder(Span<char> initialBuffer)
    {
        _arrayToReturnToPool = null;
        _chars = initialBuffer;
        _pos = 0;
    }

    public ref char this[int index] => ref _chars[index];

    public void Append(char c)
    {
        int pos = _pos;
        if (pos < _chars.Length)
        {
            _chars[pos] = c;
            _pos = pos + 1;
        }
        else
            GrowAndAppend(c);
    }

    private void GrowAndAppend(char c)
    {
        Grow(1);
        Append(c);
    }
    ...
```

```
    ...
    private void Grow(int requiredAdditionalCapacity)
    {
        Debug.Assert(requiredAdditionalCapacity > 0);
        char[] poolArray = ArrayPool<char>.Shared.Rent(Math.Max(_pos +
                           requiredAdditionalCapacity, _chars.Length * 2));
        _chars.CopyTo(poolArray);
        char[] toReturn = _arrayToReturnToPool;
        _chars = _arrayToReturnToPool = poolArray;
        if (toReturn != null)
        {
            ArrayPool<char>.Shared.Return(toReturn);
        }
    }

    public void Dispose()
    {
        char[] toReturn = _arrayToReturnToPool;
        if (toReturn != null)
        {
            ArrayPool<char>.Shared.Return(toReturn);
        }
    }

    public void Append(string str)
    {
        foreach (char c in str)
            Append(c);
    }

    public override string ToString() => new string(_chars.Slice(0, _pos));

    ...
}
```

Example usage:

```
static string UseValueStringBuilder(string data)
{
    Span<char> initialBuffer = stackalloc char[32];
    using var builder = new ValueStringBuilder(initialBuffer);
    builder.Append("<tag>");
    builder.Append(data);
    builder.Append("</tag>");
    return builder.ToString();
}
```

Example usage:

```
static string UseValueStringBuilder(string data)
{
    Span<char> initialBuffer = stackalloc char[32];
    using var builder = new ValueStringBuilder(initialBuffer);
    builder.Append("<tag>");
    builder.Append(data);
    builder.Append("</tag>");
    return builder.ToString();
}
```

Benchmarks:

```
|                Method |     Mean | Gen 0/1k Op | Gen 1 | Gen 2 | Allocated/Op |
|---------------------- |---------:|------------:|------:|------:|-------------:|
| UseValueStringBuilder | 44.23 ns |      0.0134 |     - |     - |         56 B |
|      UseStringBuilder | 51.53 ns |      0.0459 |     - |     - |        192 B |
```

```
public ref struct ValueStringBuilder
{
    ...
    public Enumerator GetEnumerator() => new Enumerator(ref this);

    public ref struct Enumerator
    {
        private readonly Span<char> source;
        private readonly int count;
        private int index;

        public Enumerator(ref ValueStringBuilder valueStringBuilder)
        {
            source = valueStringBuilder._chars;
            count = valueStringBuilder._pos;
            index = -1;
        }

        public ref char Current => ref source[index];

        public bool MoveNext()
        {
            if (index < count)
            {
                index++;
                return (index < count);
            }
            return false;
        }
    }
}
```

Example usage:

```
static string UseValueStringBuilder()
{
    Span<char> initialBuffer = stackalloc char[10];
    using var builder = new ValueStringBuilder(initialBuffer);
    builder.Append("Konrad");
    foreach (ref var @char in builder)
    {
        @char++;
    }
    return builder.ToString();
}
```

That's all! Thank you! Any questions?!