“Value types are on the stack, reference types are on the heap”

C# PROGRAMMER ANSWERING INTERVIEW QUESTION

World of Unsafe

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

About me

Experienced with backend, frontend, mobile, desktop, ML, databases.

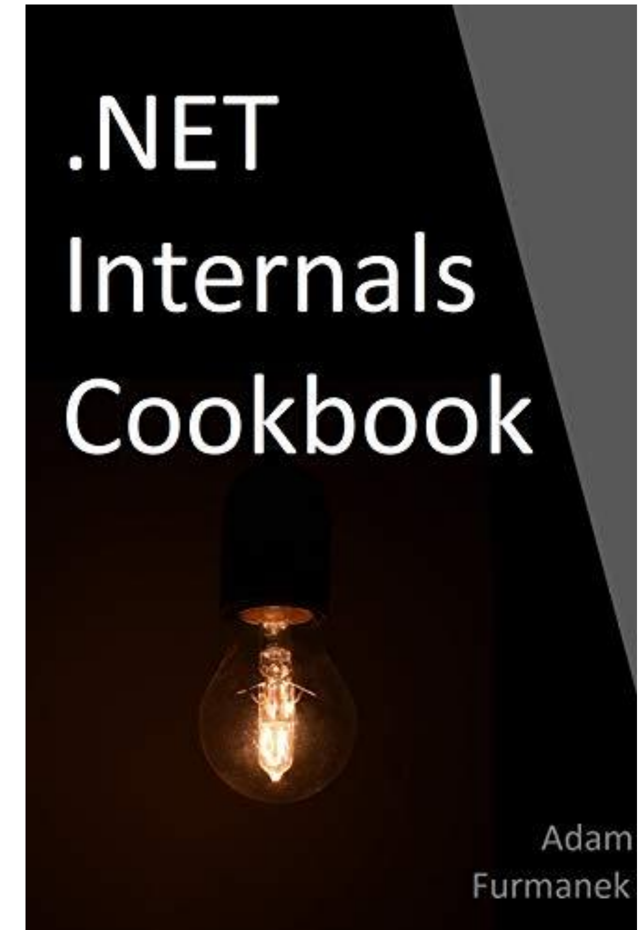
Blogger, public speaker.

Author of .NET Internals Cookbook.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Agenda

Object memory structure.

Reference, TypedReference, pointer.

Garbage Collector memory regions.

List<T> construction.

Allocation process.

new internals.

ECMA Standard ISO/IEC 23270

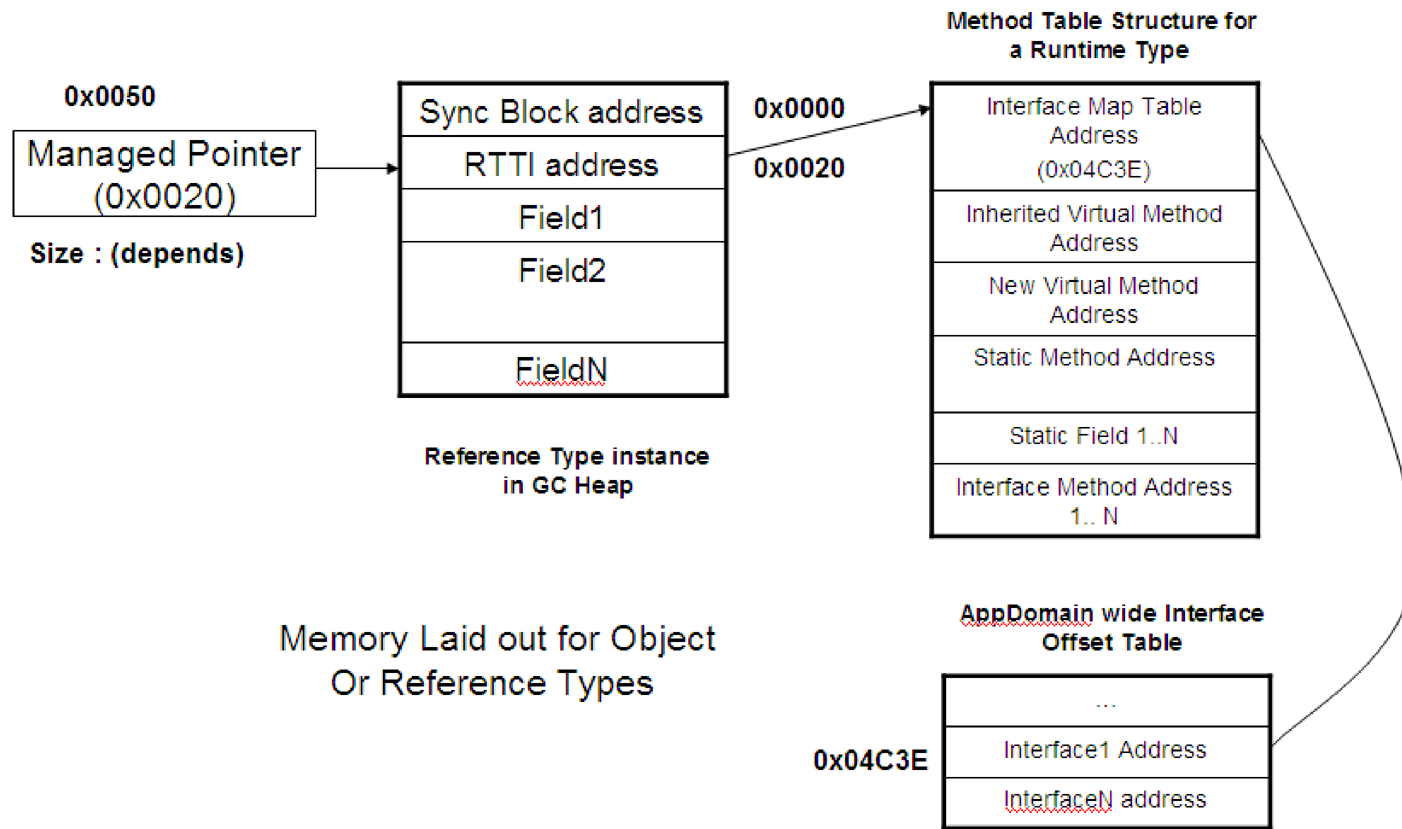
If M is an instance function member declared in a *reference-type (…)* For an instance constructor, this evaluation consists of allocating (**typically from a garbage-collected heap**) the storage for the new object.

If M is an instance function member declared in a *value-type*: (…) For an instance constructor, this evaluation consists of allocating the storage (**typically from an execution stack**) for the new object.

”Erm, what?”

C# PROGRAMMER ANSWERING PRECISE INTERVIEW QUESTION

Object Structure
var o = new object();



https://www.codeproject.com/Articles/20481/NET-Type-Internals-From-a-Microsoft-CLR-Perspecti#11

Reference

Just a pointer to the data on the heap.

4 bytes on x86, 8 bytes on x64.

Can be considered unsigned integer or unsigned long.

null represented as value 0. Technically anything pointing to *Null Partition* is considered *null* on the CPU level.

__*makeref*

The **mkrefany** instruction supports the passing of dynamically typed references. The pointer must be of type **&**, *****, or **native int**, and hold the valid address of a piece of data.

Class is the class token describing the type of the data referenced by the pointer.

Mkrefany pushes a typed reference on the stack, providing an opaque descriptor of the pointer and the type *class*.

__*refvalue*

The **refanyval** instruction retrieves the address embedded in the a typed reference.

The type embedded in the typed reference supplied on the stack must match the type specified by *type* (a metadata token, either a **typedef** or a **typeref**).

Typed Reference

Special type pointing to the data (reference for reference types, value for value types) and remembering the underlying type.

Used with __*makeref* keyword compilet to *mkrefany* instruction, which pushes a typed reference on the stack, providing an opaque descriptor of the pointer and the type *class*.

Used before generics.

Can be turned into ordinary reference with __*refvalue*.

```
object o = new object();
TypedReference typedReference = __makeref(o);
object o2 = __refvalue(typedReference);
```

Pointers

We can create pointers to data in unsafe context. We cannot create pointers to reference types.

Often represented as *IntPtr* in *Marshal* context.

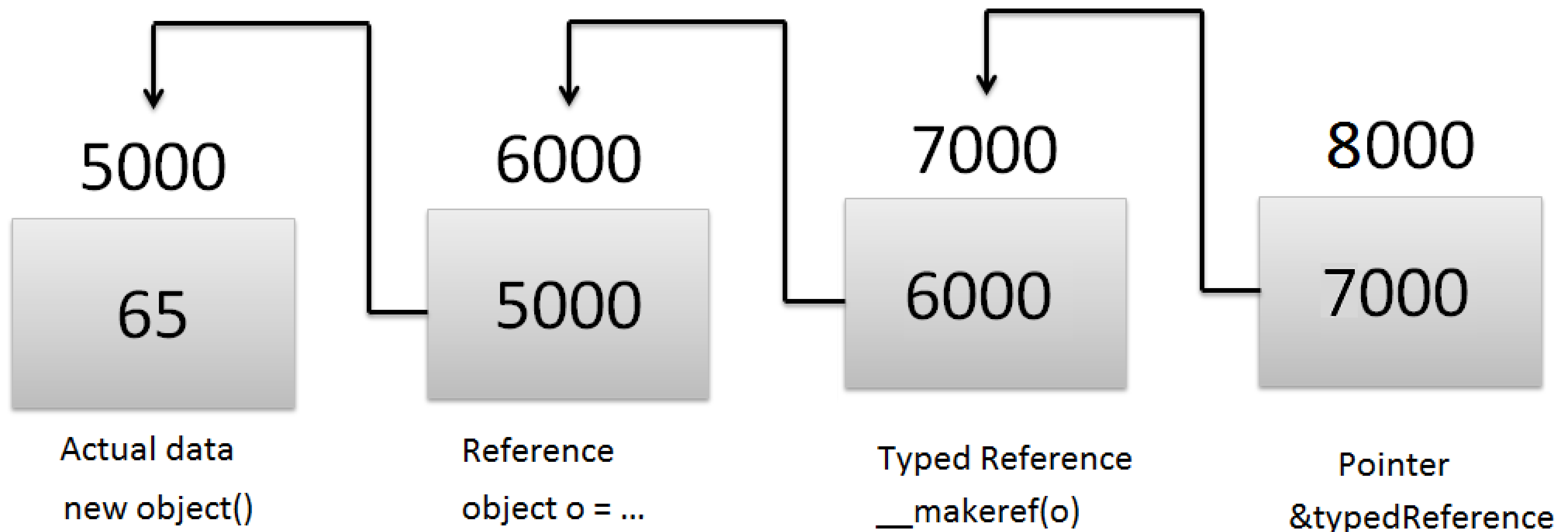
We can get pointer to the reference types if we pin it. It must be blittable and cannot be moved by GC then.

We can get pointer to the *Typed Reference*.

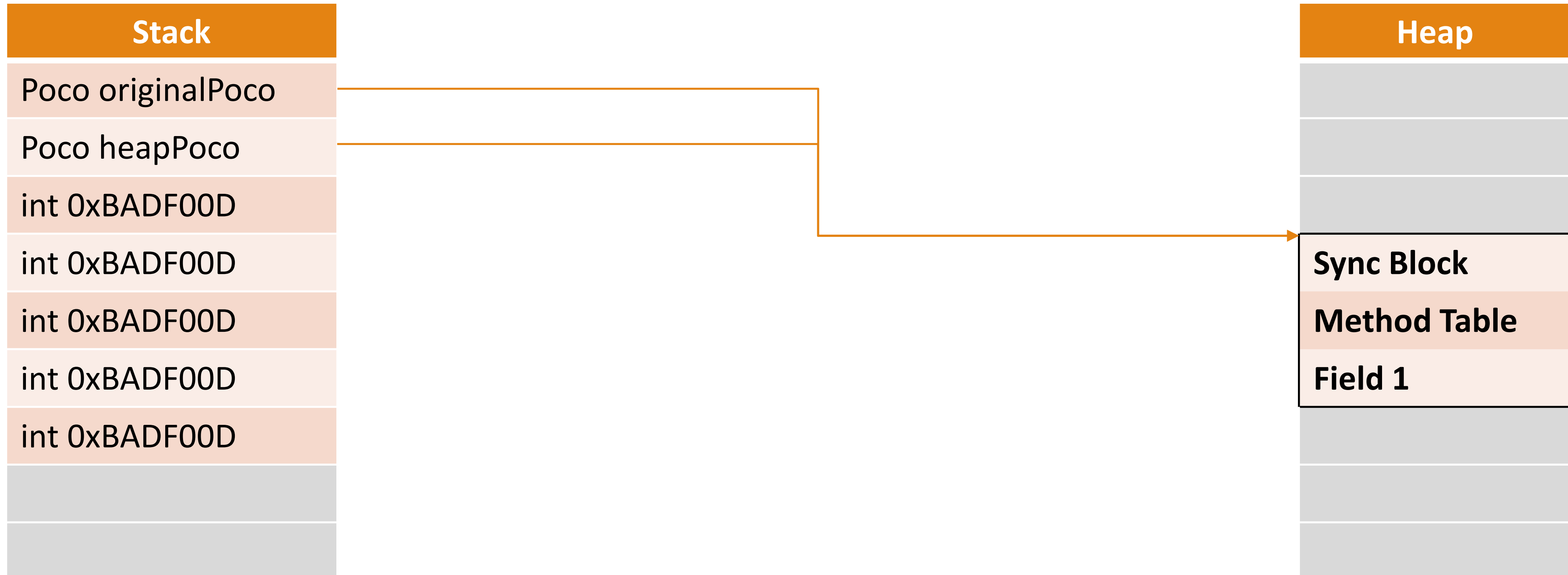
We can cast pointers between each other, so we can cast *TypedReference** to *int**.

Adding 1 to pointer increases its value by pointer size (for *int** it is 4 bytes).

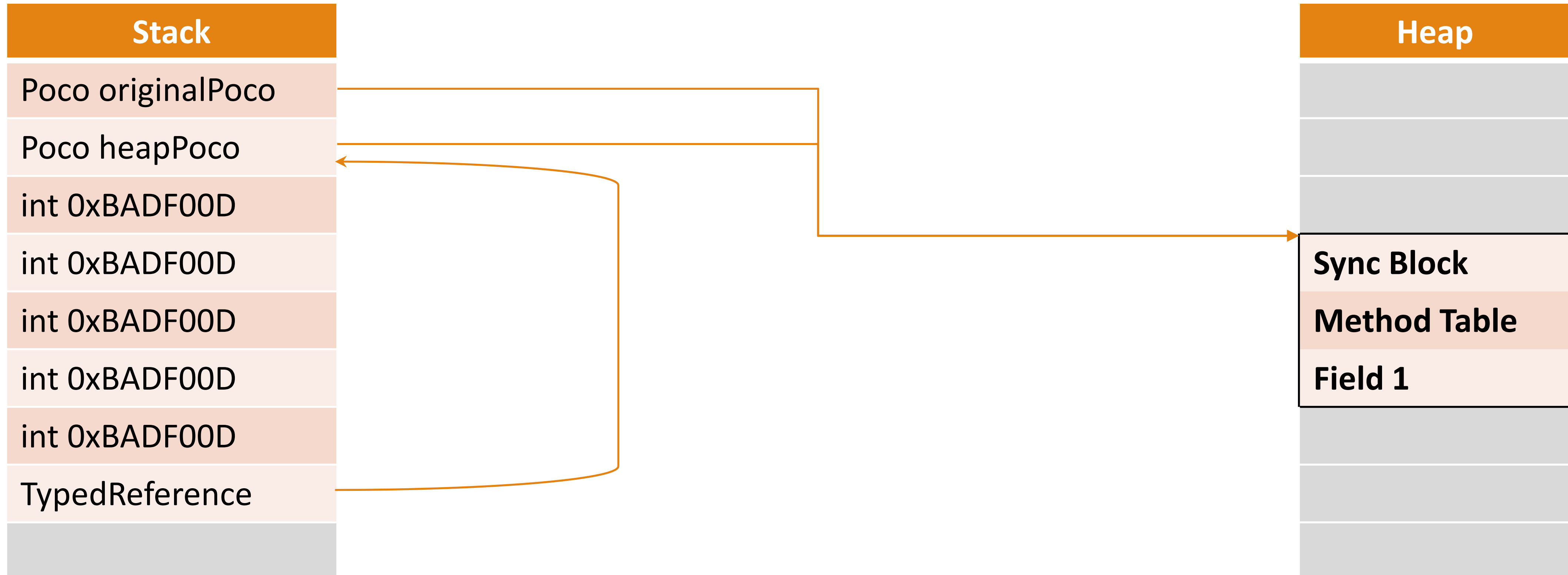
Indirection



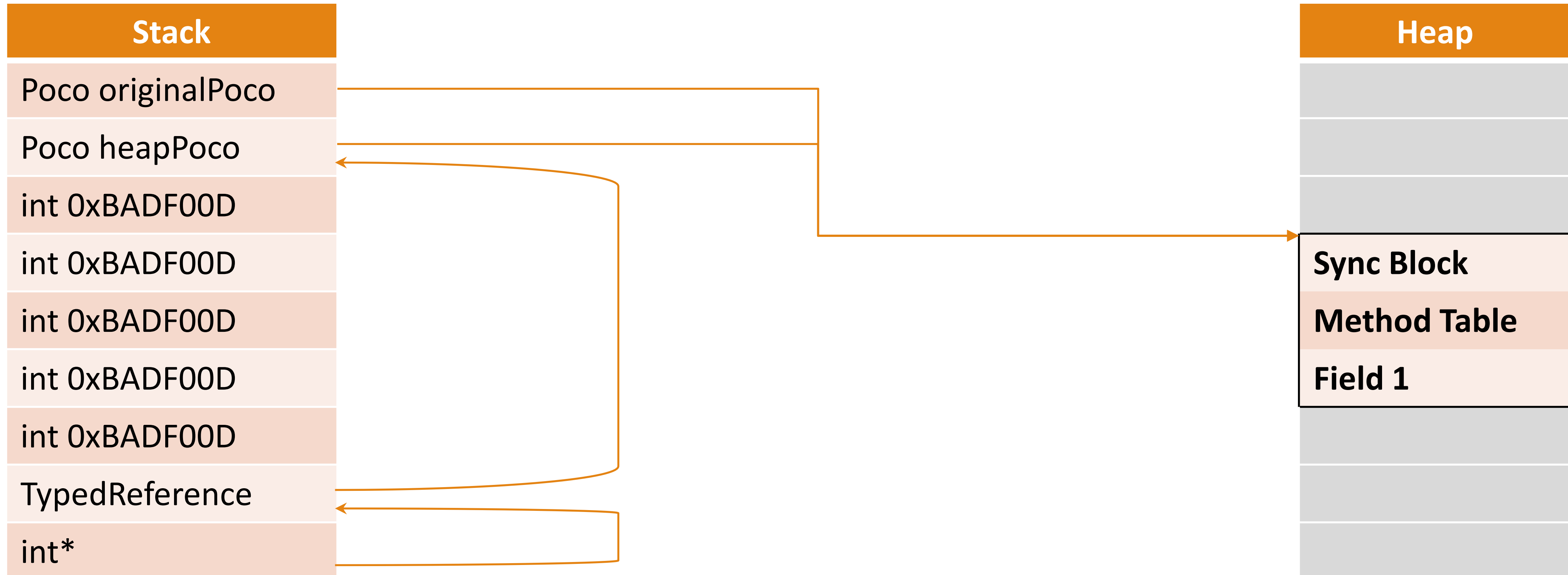
Copying object - Begin



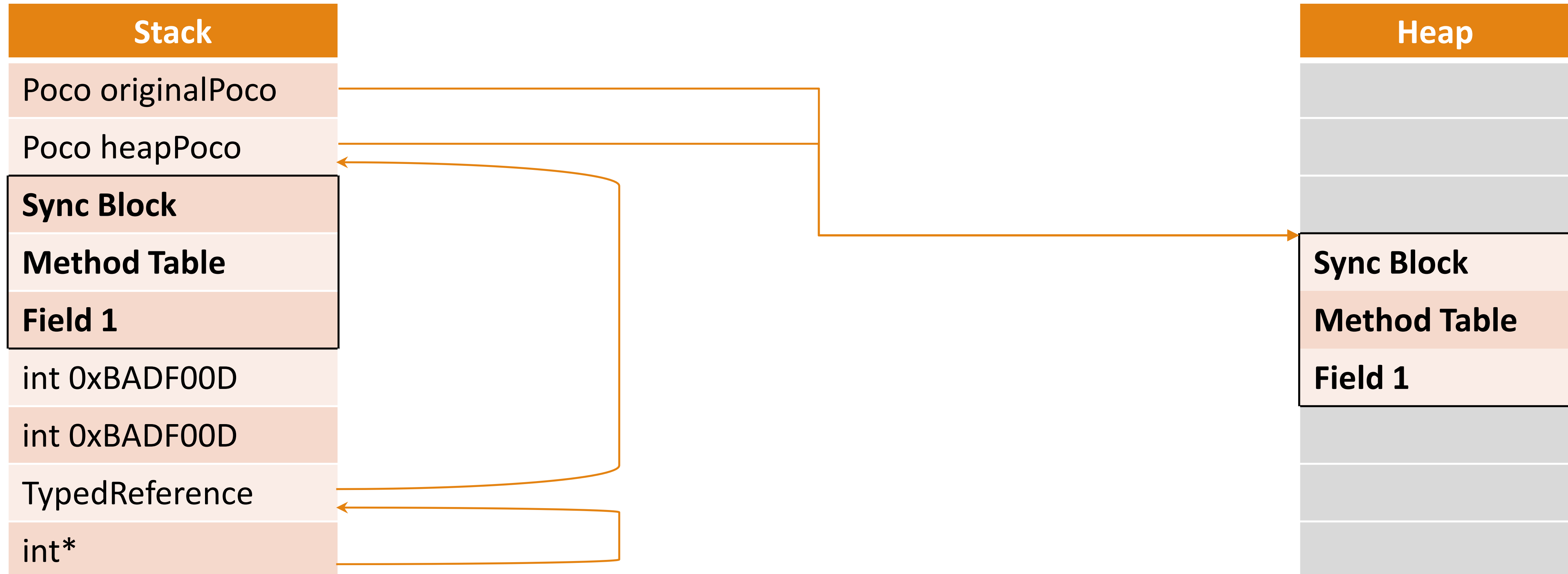
Copying object – Step 1



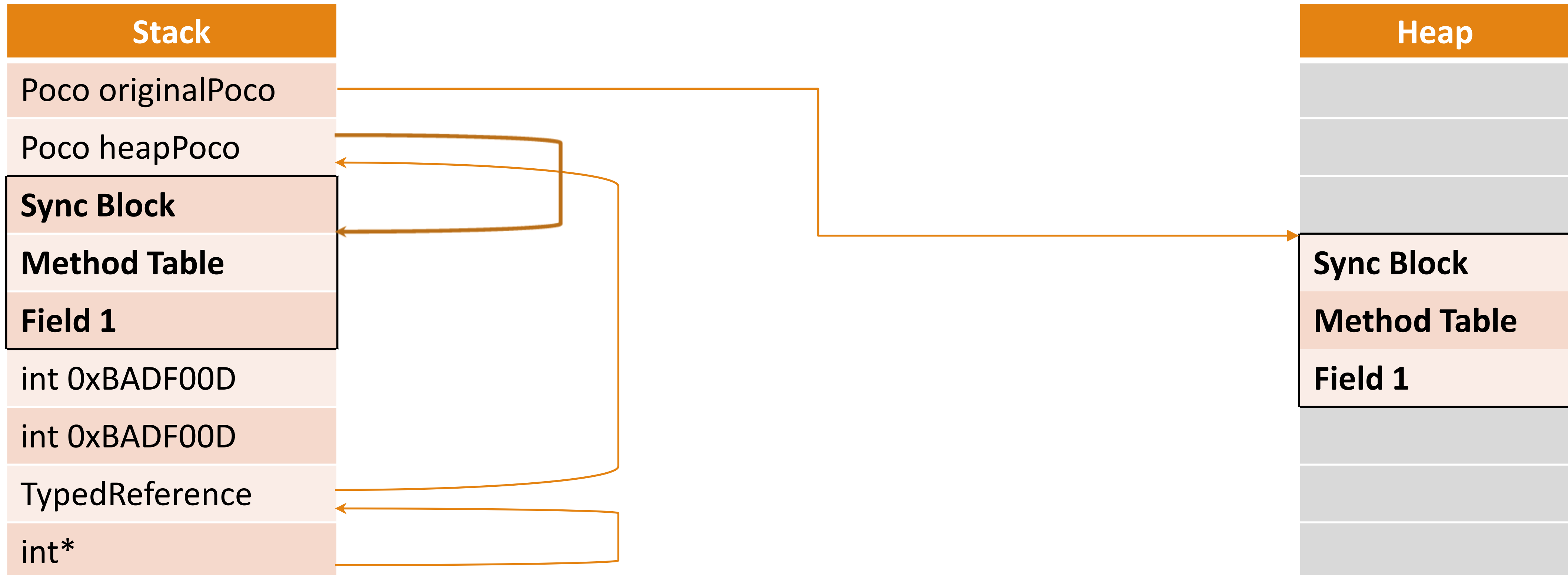
Copying object – Step 2



Copying object – Step 3



Copying object – Step 4



GC “Facts”

Facts:

- There are three generations: 0, 1, and 2
- Large object heap contains objects having at least 85000 bytes (**only?**)
- Large object heap is in generation **2**

Questions:

- Which generation holds a stack?

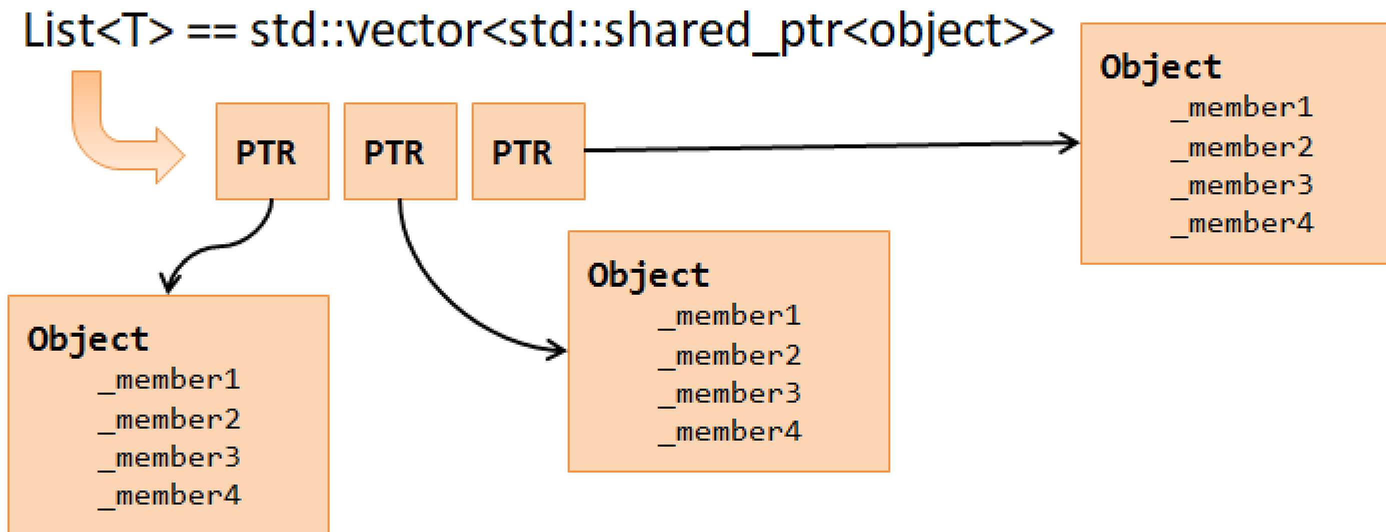
Managed object on a stack

DEMO 1

Unsafe list

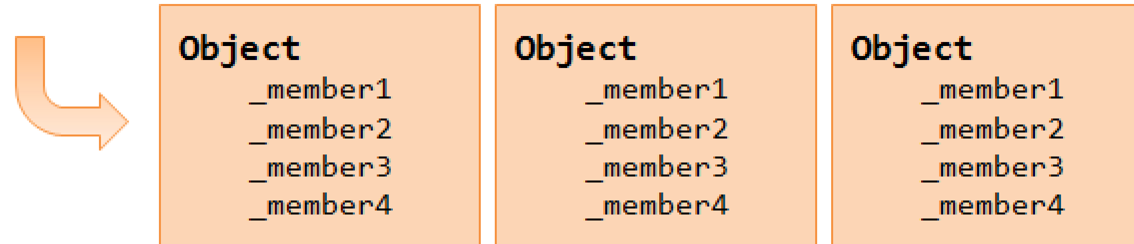
DEMO 2

Ordinary list



Unsafe list

UnsafeList == std::vector<object>



“Expected” results

```
C:\WINDOWS\system32\cmd.exe
-----------------------
Array
Insertion time: 3460
Sum: -1023623168
Calculation time: 138
Array
Insertion time: 3654
Sum: -1023623168
Calculation time: 131
Array
Insertion time: 3715
Sum: -1023623168
Calculation time: 131
-----------------------
List
Insertion time: 3072
Sum: -1023623168
Calculation time: 180
List
Insertion time: 3699
Sum: -1023623168
Calculation time: 146
List
Insertion time: 3481
Sum: -1023623168
Calculation time: 143
-----------------------
UnsafeList
Insertion time: 1396
Sum: -1023623168
Calculation time: 130
UnsafeList
Insertion time: 1568
Sum: -1023623168
Calculation time: 135
UnsafeList
Insertion time: 1546
Sum: -1023623168
Calculation time: 136
Press any key to continue . . .
```

Native Code – Insert

```
00eb09b0 push    ebp
00eb09b1 mov     ebp,esp
00eb09b3 sub     esp,8
00eb09b6 mov     dword ptr [ebp-8],ecx
00eb09b9 mov     dword ptr [ebp-4],edx
00eb09bc cmp     dword ptr ds:[0E64268h],0
00eb09c3 je      00eb09ca
00eb09c5 call    clr!JIT_DbgIsJustMyCode (72e17925)
00eb09ca nop
00eb09cb push    dword ptr [ebp+8]
00eb09ce mov     ecx,dword ptr [ebp-8]
00eb09d1 mov     edx,dword ptr [ebp-4]
00eb09d4 call    clr!JIT_Stelem_Ref (72af8c70)
00eb09d9 nop
00eb09da mov     esp,ebp
00eb09dc pop     ebp
00eb09dd ret     4
```

Native Code – stelem.ref (excerpt)

```
72af8c70 mov   eax,dword ptr [esp+4]
72af8c74 test  ecx,ecx
72af8c76 je    clr!JIT_Stelem_Ref+0x6c (72e13e0b)
72af8c7c cmp   edx,dword ptr [ecx+4]
72af8c7f jae   clr!JIT_Stelem_Ref+0x73 (72e13e1a)
72af8c85 test  eax,eax
72af8c87 je    clr!JIT_Stelem_Ref+0x28 (72af8cac)
72af8c89 push  edx
72af8c8a mov   edx,dword ptr [ecx]
72af8c8c mov   edx,dword ptr [edx+20h]
72af8c8f cmp   edx,dword ptr [eax]
72af8c91 je    clr!JIT_Stelem_Ref+0x1b (72af8c9f)
72af8c93 cmp   edx,dword ptr [clr!g_pObjectClass (73135364)]
72af8c99 jne   clr!JIT_Stelem_Ref+0x47 (72b01672)
72af8c9f pop   edx
72af8ca0 lea   edx,[ecx+edx*4+8]
72af8ca4 call  clr!JIT_WriteBarrierEAX (72af22f0)
72af8ca9 ret   4
72af8cac mov   dword ptr [ecx+edx*4+8],eax
72af8cb0 ret   4
```

IL Allocator

DEMO 3

Custom new operator

DEMO 4

Allocation - newobj

The **newobj** instruction:

- allocates a new instance of the class associated with *ctor*
- initializes all the fields in the new instance to 0 (of the proper type) or null references as appropriate
- calls the constructor *ctor* with the given arguments along with the newly created instance
- after the constructor has been called, the now initialized object reference (type **O**) is pushed on the stack.

Native code – calling helper method for **new** keyword

```
006c0486 b9a84d5e00      mov     ecx,5E4DA8h (MT: GenericUnsafeAlloc.GenericMemoryAllocator)

006c048b e8382cf1ff      call    005d30c8 (JitHelp: CORINFO_HELP_NEWSFAST)

006c0490 8945d0          mov     dword ptr [ebp-30h],eax

006c0493 8b4dd0          mov     ecx,dword ptr [ebp-30h]

006c0496 ff15e44d5e00    call    dword ptr ds:[5E4DE4h] (GenericUnsafeAlloc.GenericMemoryAllocator..ctor(), mdToken: 06000001)

006c049c 8b45d0          mov     eax,dword ptr [ebp-30h]

006c049f 8945e0          mov     dword ptr [ebp-20h],eax
```

Native code – helper method for allocating objects

```
004b30c8 8b4104          mov  eax,dword ptr [ecx+4]          ; extract object size
004b30cb 648b15340e0000  mov  edx,dword ptr fs:[0E34h]       ; extract chunks descriptor
004b30d2 034240          add  eax,dword ptr [edx+40h]        ; add chunk offset to object size
004b30d5 3b4244          cmp  eax,dword ptr [edx+44h]        ; check if new object fits
004b30d8 7709            ja   004b30e3                       ; it didn't so we jump to call ordinary allocate method
004b30da 894240          mov  dword ptr [edx+40h],eax        ; store new offset
004b30dd 2b4104          sub  eax,dword ptr [ecx+4]          ; calculate new object begin address
004b30e0 8908            mov  dword ptr [eax],ecx            ; set object method table
004b30e2 c3              ret                                 ; we are done
004b30e3 e91f683270      jmp  clr!JIT_New (707d9907)         ; we call ordinary JIT_New
```

Native code – calling custom method

```
0048074b 8b4de8        mov     ecx,dword ptr [ebp-18h]

0048074e ff15c44d2e00  call    dword ptr ds:[2E4DC4h]
(GenericUnsafeAlloc.GenericMemoryAllocator.RawAllocate(IntPtr), mdToken: 06000004)

00480754 8945d0        mov     dword ptr [ebp-30h],eax

00480757 90            nop
```

Hiding objects from GC

DEMO 5

Improving GC performance

Allocating objects in custom pool doesn’t change GC logic.

As soon as there is a reference pointing to them, GC will traverse the object graph.

Can we do something about it?

Regular object

Sync Block
Method Table: BigGraph
BigGraph left
BigGraph right
long value

Sync Block
Method Table: typeof(BigGraph)
BigGraph left
BigGraph right
long value

Sync Block
Method Table: typeof(BigGraph)
BigGraph left
BigGraph right
long value

Hidden object

Sync Block
Method Table: BigGraph
IntPtr left
IntPtr right
long value

Sync Block
Method Table: typeof(BigGraph)
IntPtr left
IntPtr right
long value

Sync Block
Method Table: typeof(BigGraph)
IntPtr left
IntPtr right
long value

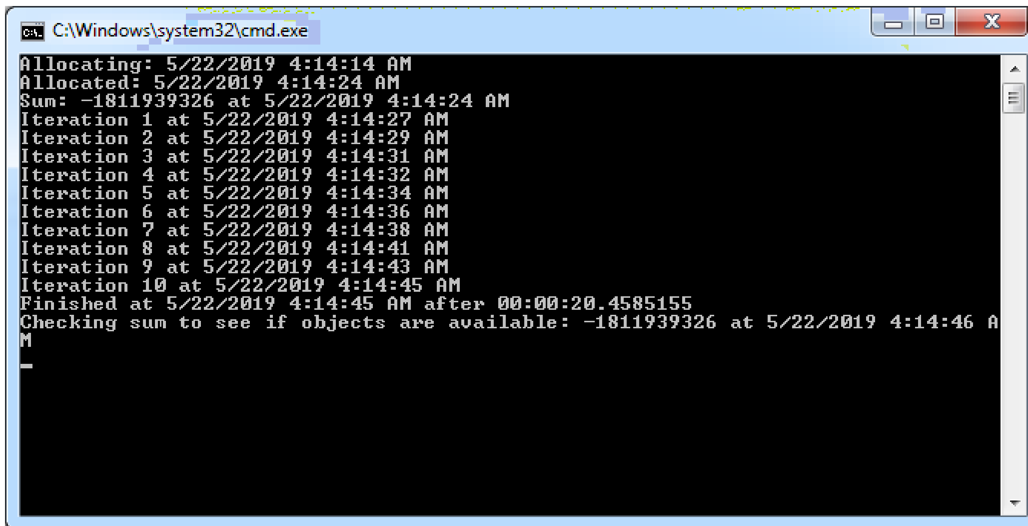
Hiding objects from GC

```
class Program
{
    public static int depth = 25;
    public static int iterations = 10;

    static void Main(string[] args)
    {
        Console.WriteLine($"Allocating: {DateTime.Now}");
        var graph = new BigGraphNoGcRef(1, depth);
        Console.WriteLine($"Allocated: {DateTime.Now}");
        Console.WriteLine($"Sum: {graph.GetSum()} at {DateTime.Now}");

        var stopwatch = new Stopwatch();
        stopwatch.Start();
        for(int i = 0; i < iterations; ++i)
        {
            GC.Collect(2);
            GC.WaitForFullGCComplete();
            GC.WaitForPendingFinalizers();
            Console.WriteLine($"Iteration {i + 1} at {DateTime.Now}");
        }
        stopwatch.Stop();

        Console.WriteLine($"Finished at {DateTime.Now} after {stopwatch.Elapsed}");
        Console.WriteLine($"Checking sum to see if objects are available: {graph.GetSum()} at {DateTime.Now}");

        Console.ReadLine();
    }
}
```

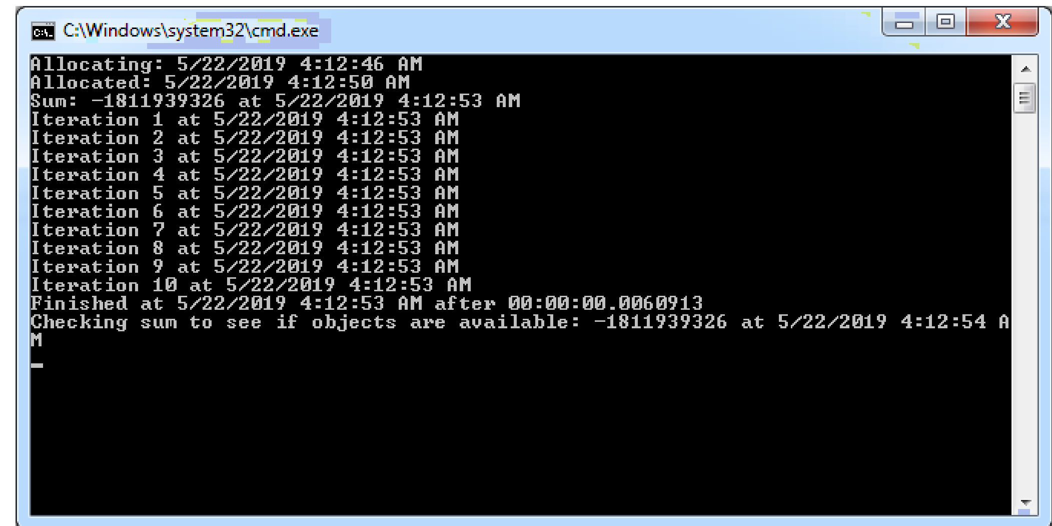
```
class BigGraph
{
    public int value;
    public BigGraph left;
    public BigGraph right;

    public BigGraph(int value, int level)
    {
        this.value = value;
        if (level == 0) return;
        this.left = new BigGraph(value + 1, level - 1);
        this.right = new BigGraph(value + 2, level - 1);
    }

    public int GetSum()
    {
        return value + (right != null ? left.GetSum() + right.GetSum() : 0);
    }
}

class BigGraphNoGcRef
{
    public static UnsafeList.UnsafeList<BigGraphNoGcRef> list = new UnsafeList.UnsafeList<BigGraphNoGcRef>((int)Math.Pow(2, Program.depth + 1), 6);

    public int value;
    public int left;
    public int right;

    public BigGraphNoGcRef(int value, int level)
    {
        this.value = value;
        if (level == 0) return;
        this.left = list.Add(new BigGraphNoGcRef(value + 1, level - 1));
        this.right = list.Add(new BigGraphNoGcRef(value + 2, level - 1));
    }

    public int GetSum()
    {
        return value + (right != 0 ? list.Get(left).GetSum() + list.Get(right).GetSum() : 0);
    }
}
```

Results

NORMAL OBJECTS



NO REFERENCES



Q&A

References

Jeffrey Richter - „CLR via C#”

Jeffrey Richter, Christophe Nasarre - „Windows via C/C++”

Mark Russinovich, David A. Solomon, Alex Ionescu - „Windows Internals”

Penny Orwick – „Developing drivers with the Microsoft Windows Driver Foundation”

Mario Hewardt, Daniel Pravat - „Advanced Windows Debugging”

Mario Hewardt - „Advanced .NET Debugging”

Steven Pratschner - „Customizing the Microsoft .NET Framework Common Language Runtime”

Serge Lidin - „Expert .NET 2.0 IL Assembler”

Joel Pobar, Ted Neward — „Shared Source CLI 2.0 Internals”

Adam Furmanek – „.NET Internals Cookbook”

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md — „Book of the Runtime”

https://blogs.msdn.microsoft.com/oldnewthing/ — Raymond Chen „The Old New Thing”

References

https://blog.adamfurmanek.pl/2016/04/23/custom-memory-allocation-in-c-part-1/ — allocating object on a stack

https://blog.adamfurmanek.pl/2016/05/07/custom-memory-allocation-in-c-part-3/ — hijacking new

https://blog.adamfurmanek.pl/2017/02/11/how-to-override-sealed-function-in-c/ — hijacking any function in .NET

https://blog.adamfurmanek.pl/2017/06/03/capturing-thread-creation-to-catch-exceptions/ — hijacking Thread constructor to handle exceptions

https://blog.adamfurmanek.pl/2018/03/24/generating-func-from-bunch-of-bytes-in-c/ — running any machine code

References

https://www.codeproject.com/Articles/20481/NET-Type-Internals-From-a-Microsoft-CLR-Perspecti#11 — .NET memory structure

https://blogs.msdn.microsoft.com/tess/2008/02/04/net-debugging-demos-information-and-setup-instructions/ — .NET Debugging demos by Tess Ferrandez

https://blog.bramp.net/post/2015/08/24/unsafe-part-1-sun.misc.unsafe-helper-classes/ — playing with Unsafe in JVM

http://benbowen.blog/post/fun_with_makeref/ — __makeref, __reftype, __refvalue, __arglist

http://xoofx.com/blog/2015/10/08/stackalloc-for-class-with-roslyn-and-coreclr/ — stackalloc in Roslyn

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM