Stack vs heap

Memory locations

So, we know already **what** we need to store in memory - value and reference types instances.

Memory locations

So, we know already **what** we need to store in memory - value and reference types instances.

Now, we need to know **in what context** we can store it!

Memory locations

*"**I.12.1.6.1 Homes for values***

*The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home locations:*

- *An incoming **argument***
- *A **local variable** of a method*
- *An **instance field** of an object or value type*
- *A **static field** of a class, interface, or module*
- *An **array element**"*

Memory locations

*"**I.12.1.6.1 Homes for values***

*The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home locations:*

- *An incoming **argument***
- *A **local variable** of a method*
- *An **instance field** of an object or value type*
- *A **static field** of a class, interface, or module*
- *An **array element**"*

*"**I.12.3.2 Method state***

*The four areas of the method state — incoming arguments array, local variables array, local memory pool and evaluation stack — are specified as if logically distinct areas. A conforming implementation of the CLI can map these areas into **one contiguous array of memory**, held as a **conventional stack frame** on the underlying target architecture, or **use any other equivalent representation technique**."*

Additionaly there is *"**local memory pool** – (...) The memory allocated in the local memory pool is reclaimed upon method context termination."*

Memory locations

So, we have some memory locations to implement:

- method's local variable
- method's argument
- instance field of reference type - covers "array element" case
- instance field of value type
- static field
- local memory pool

Memory locations

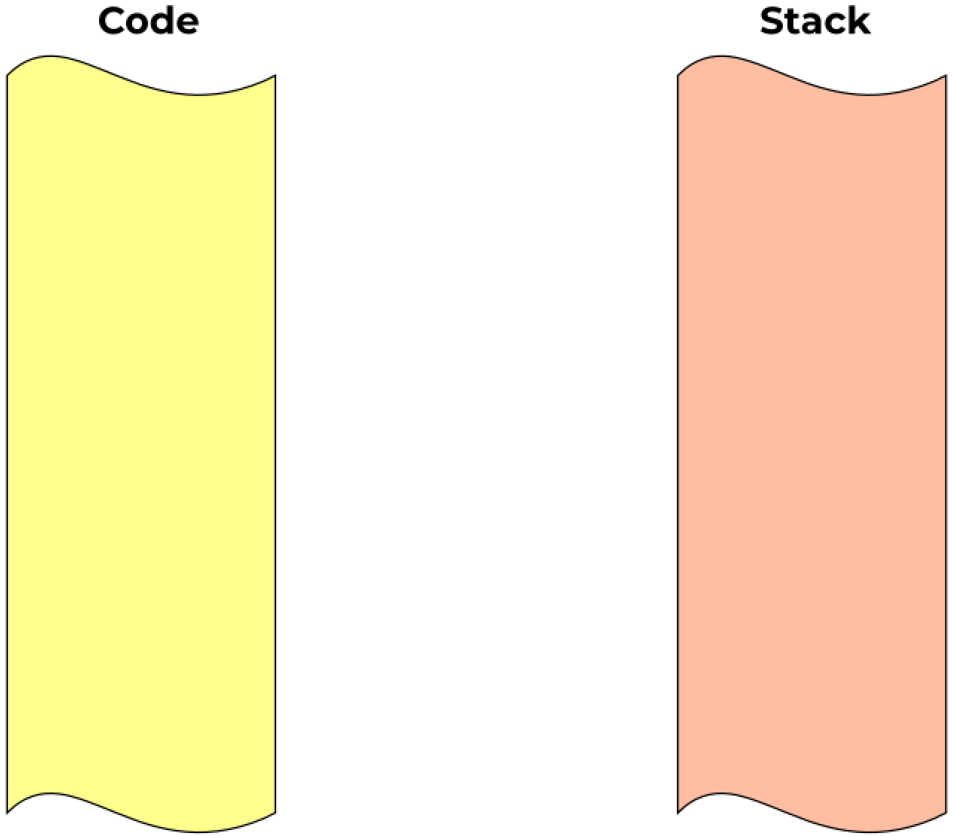
So, we have some memory locations to implement:

- method's local variable
- method's argument
- instance field of reference type - covers "array element" case
- instance field of value type
- static field
- local memory pool

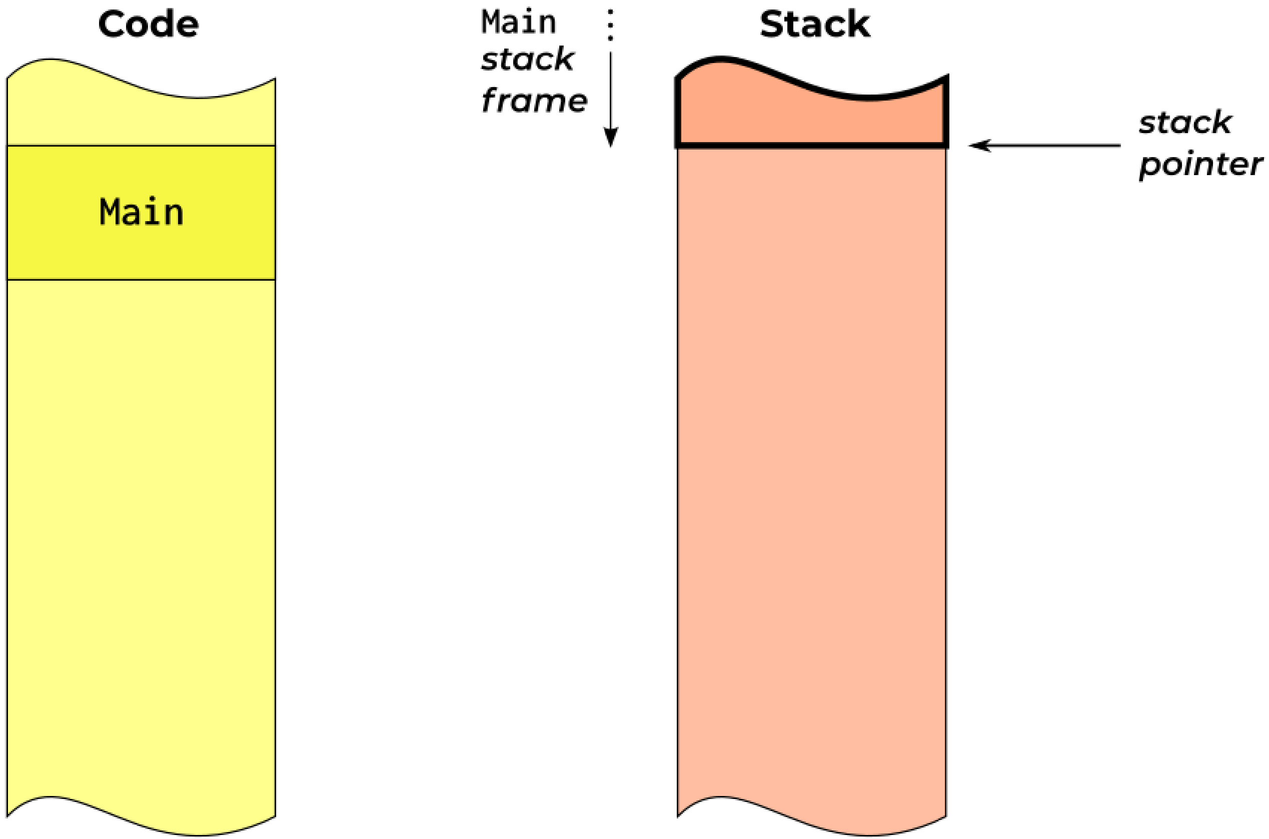
And we can use:

- method's stack frame - has a lifetime of a method
- managed heap - lifetime magically detected by the GC
- CPU registers - very volatile

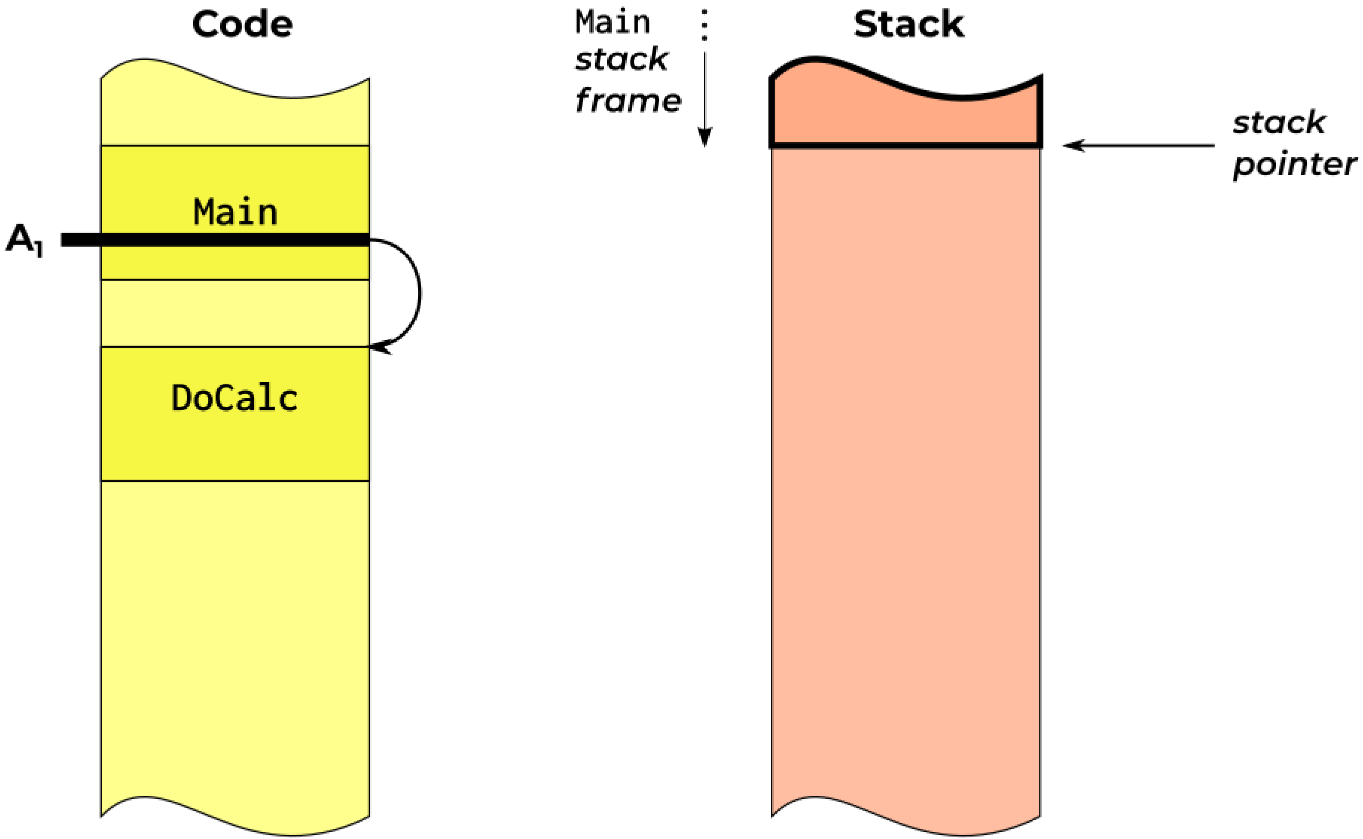
Memory locations - stack



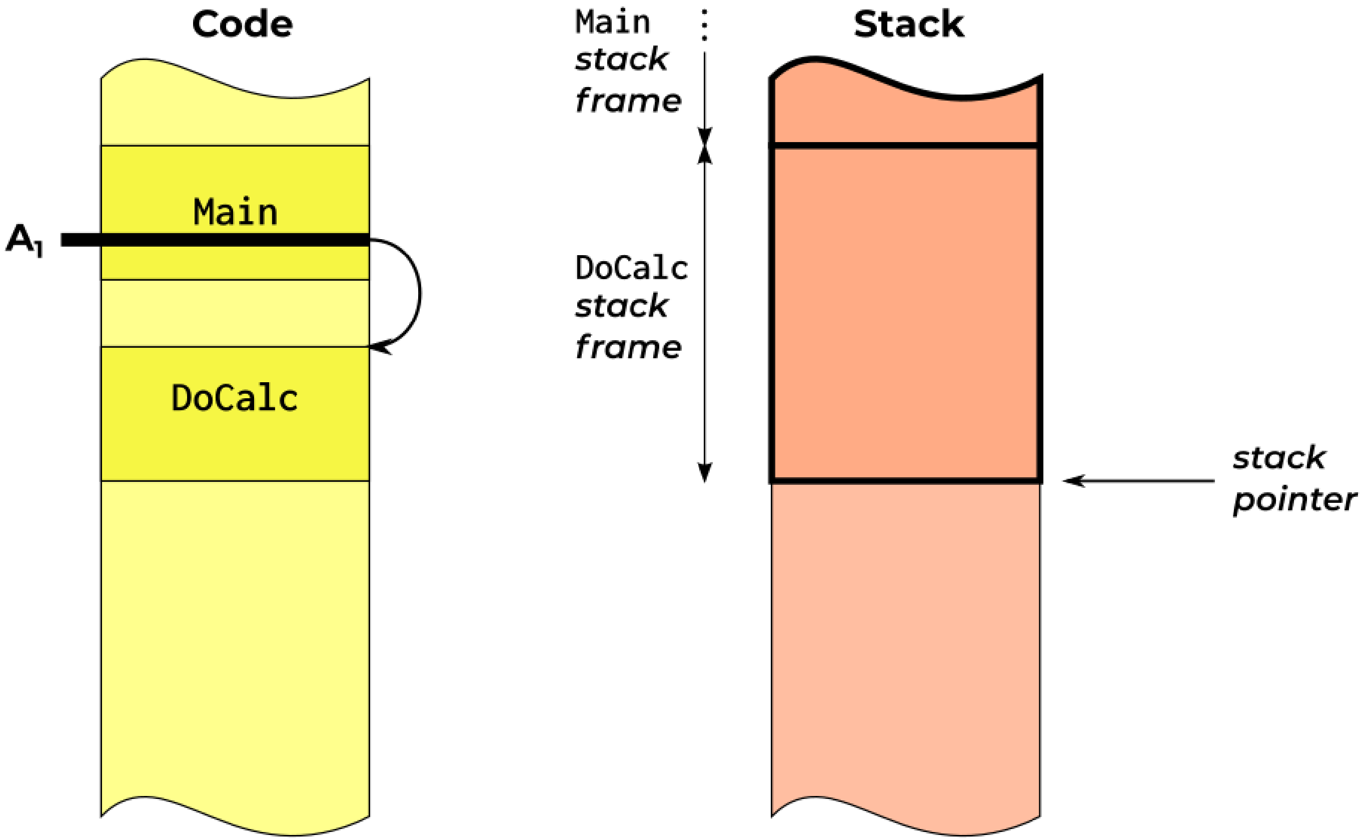
Memory locations - stack



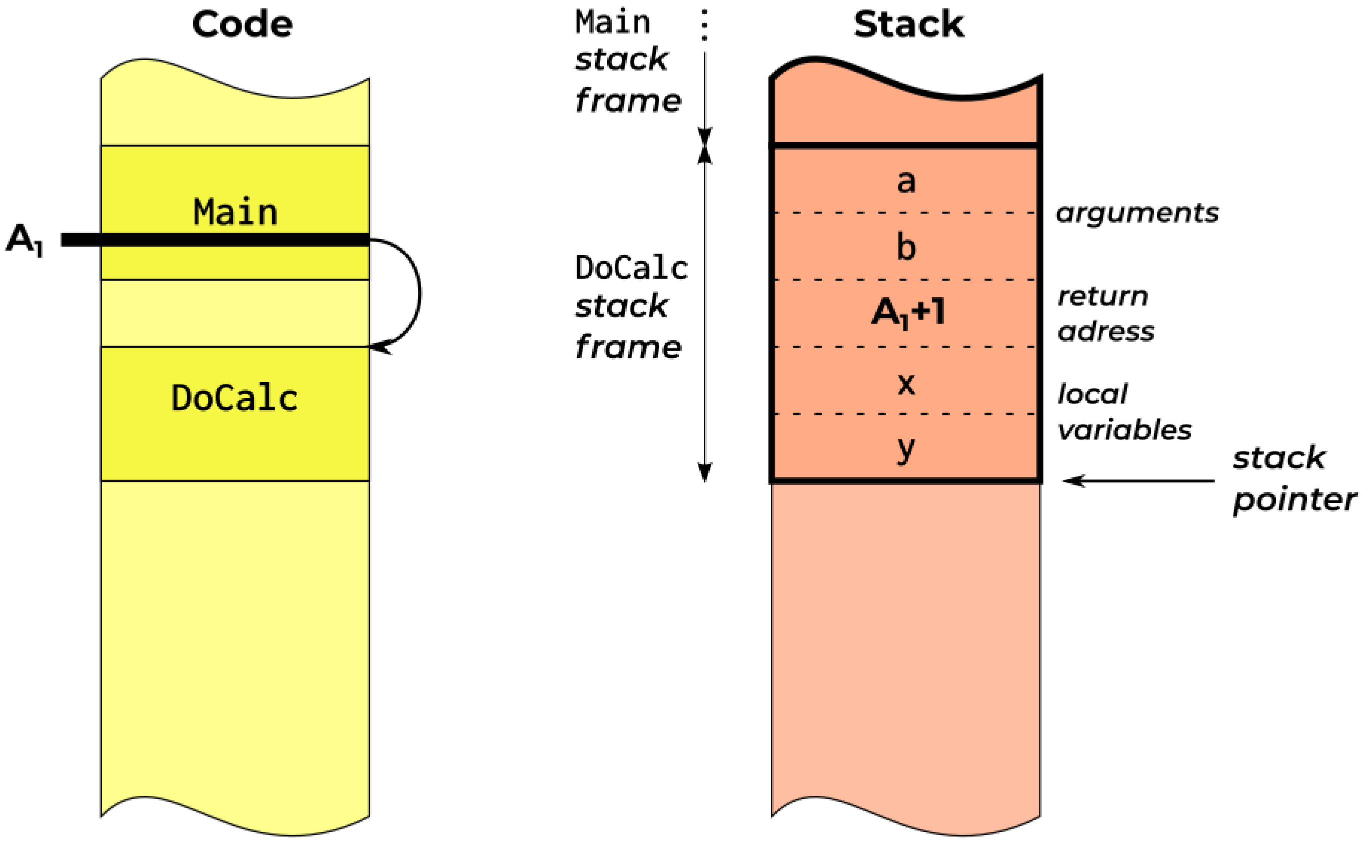
Memory locations - stack



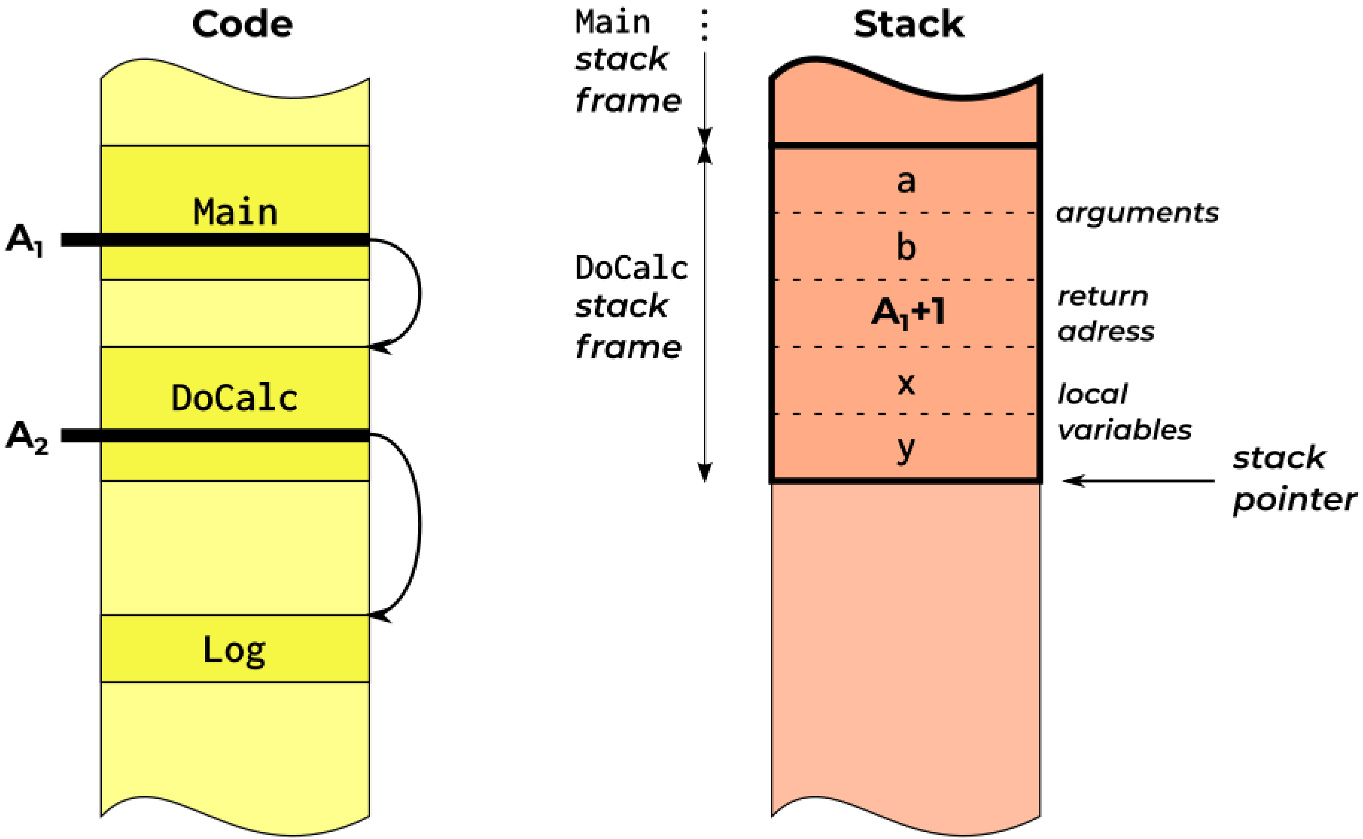
Memory locations - stack



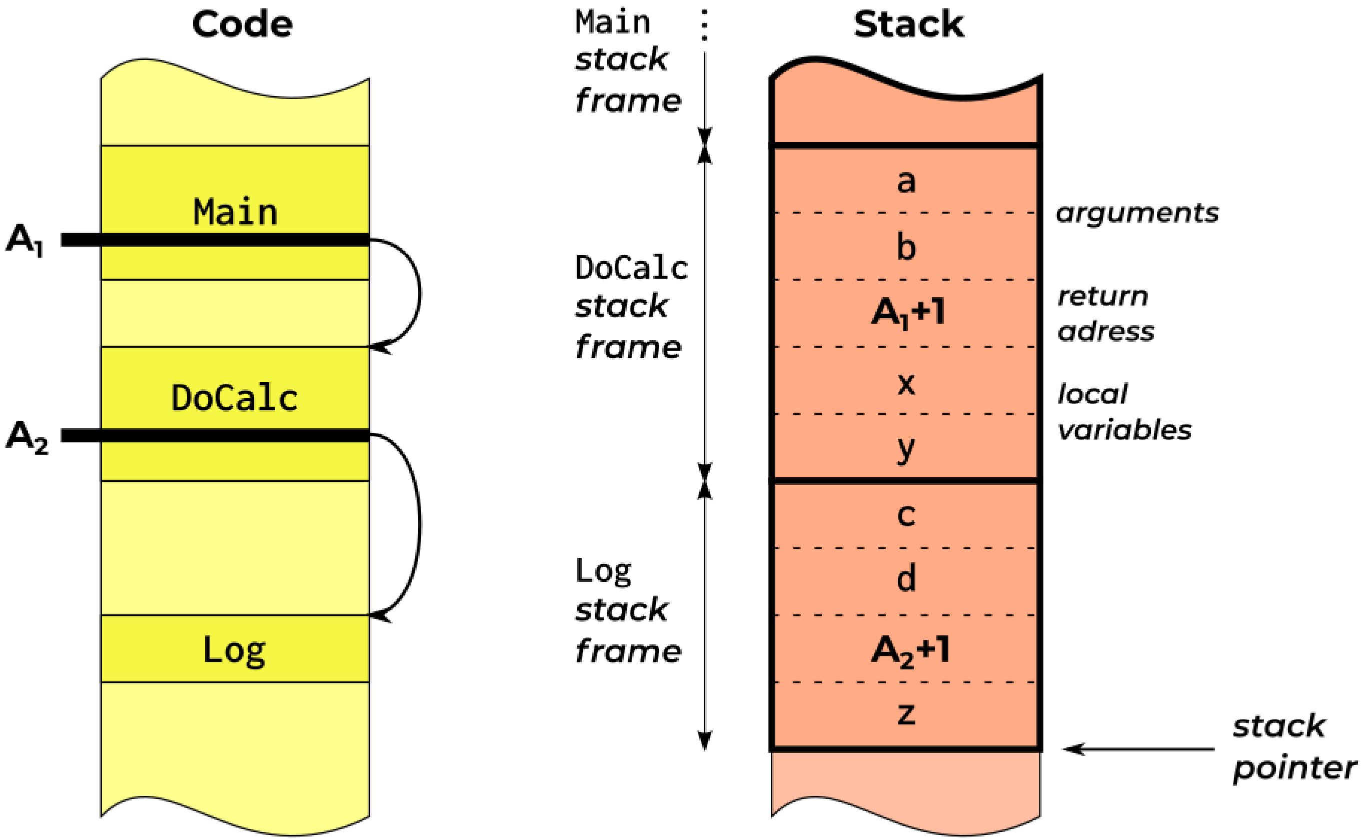
Memory locations - stack



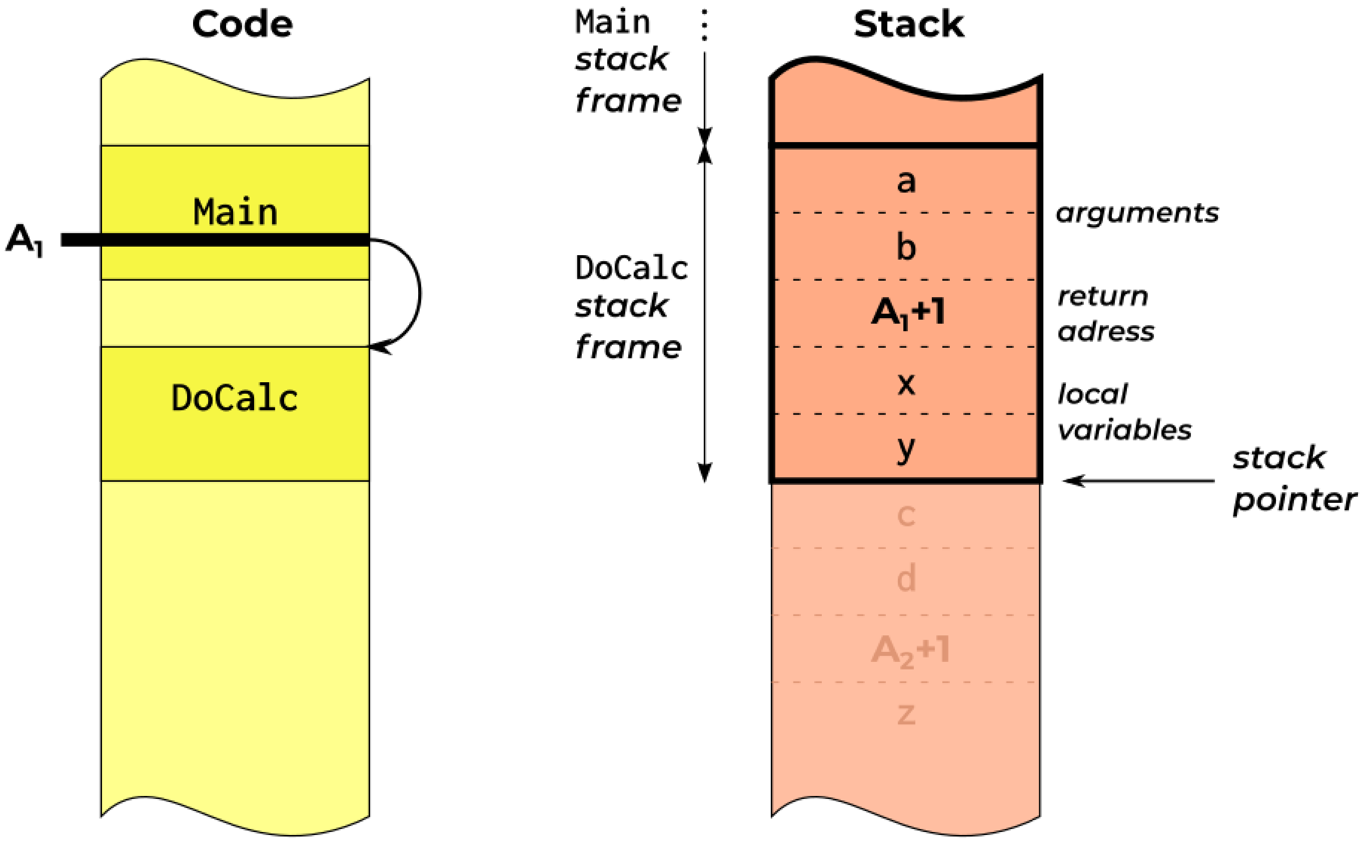
Memory locations - stack



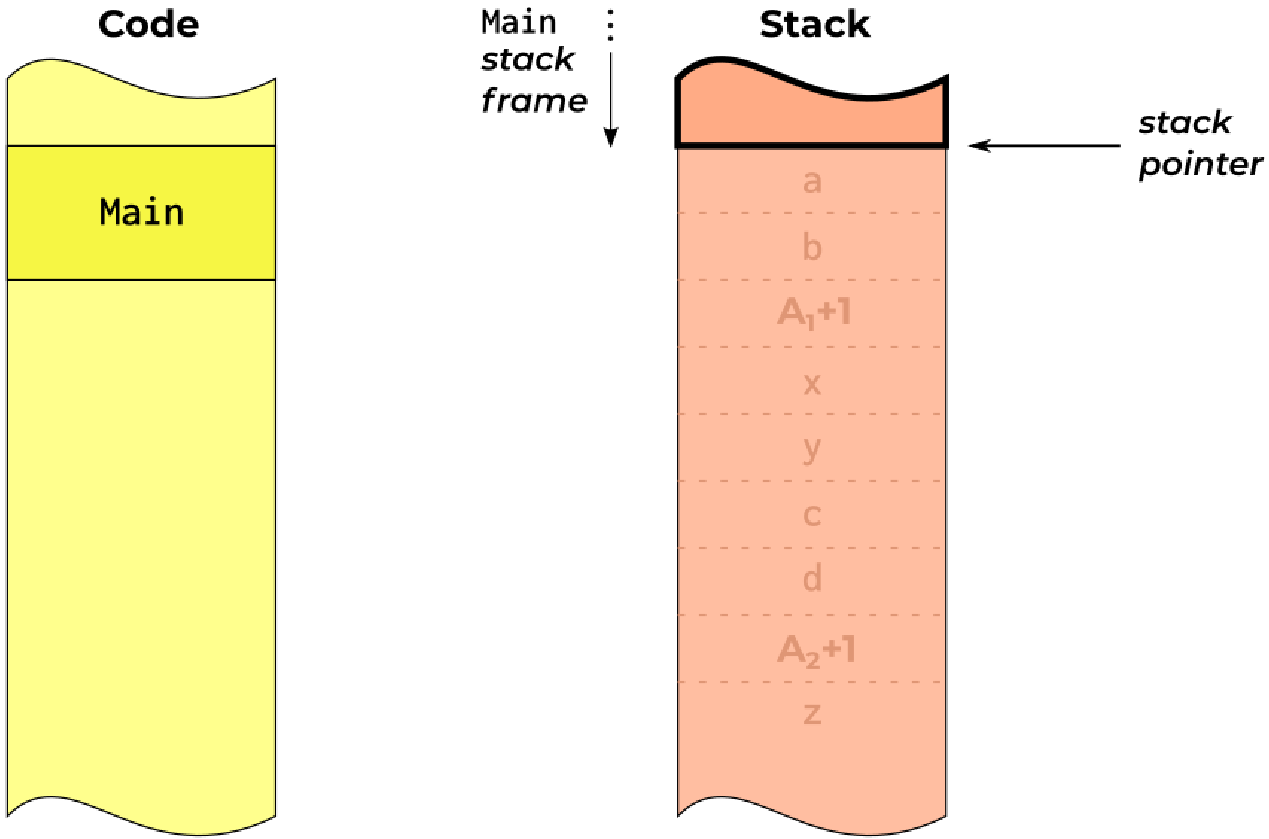
Memory locations - stack



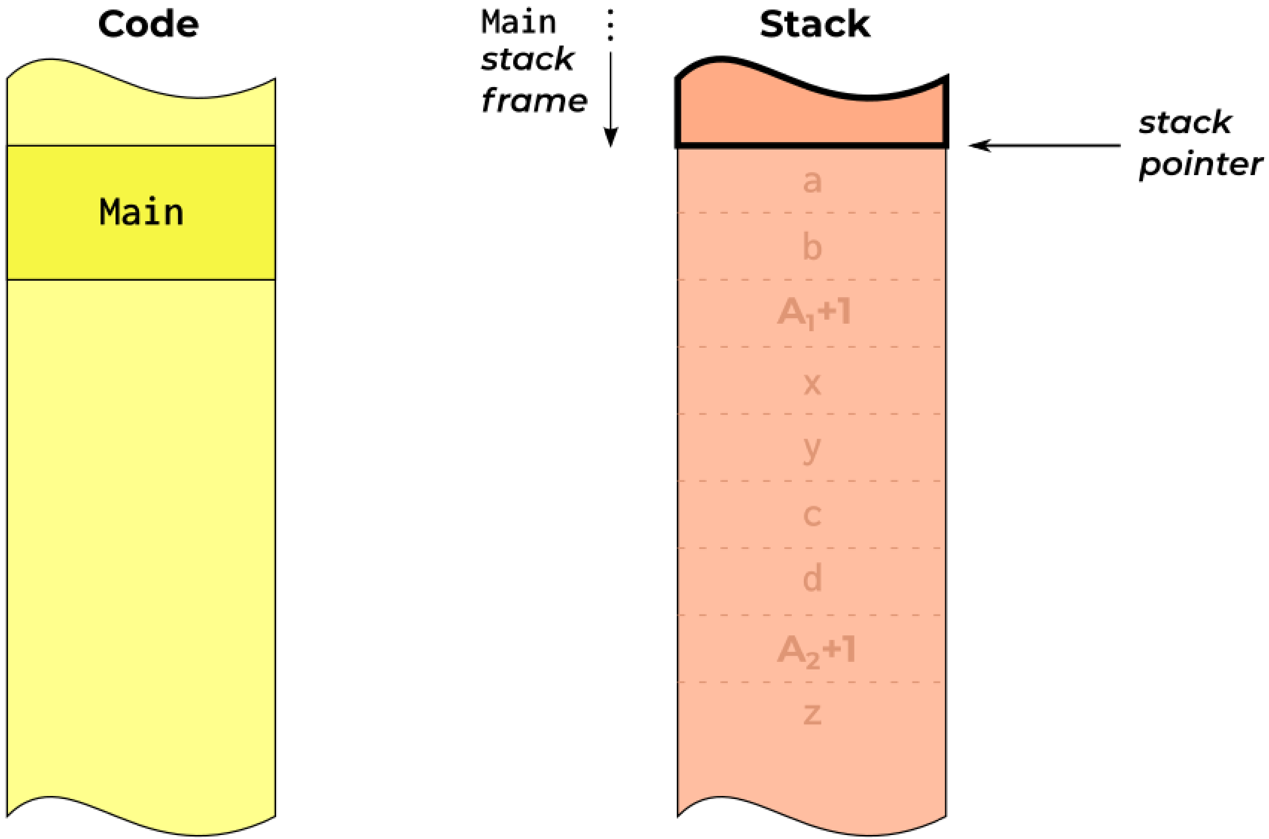
Memory locations - stack



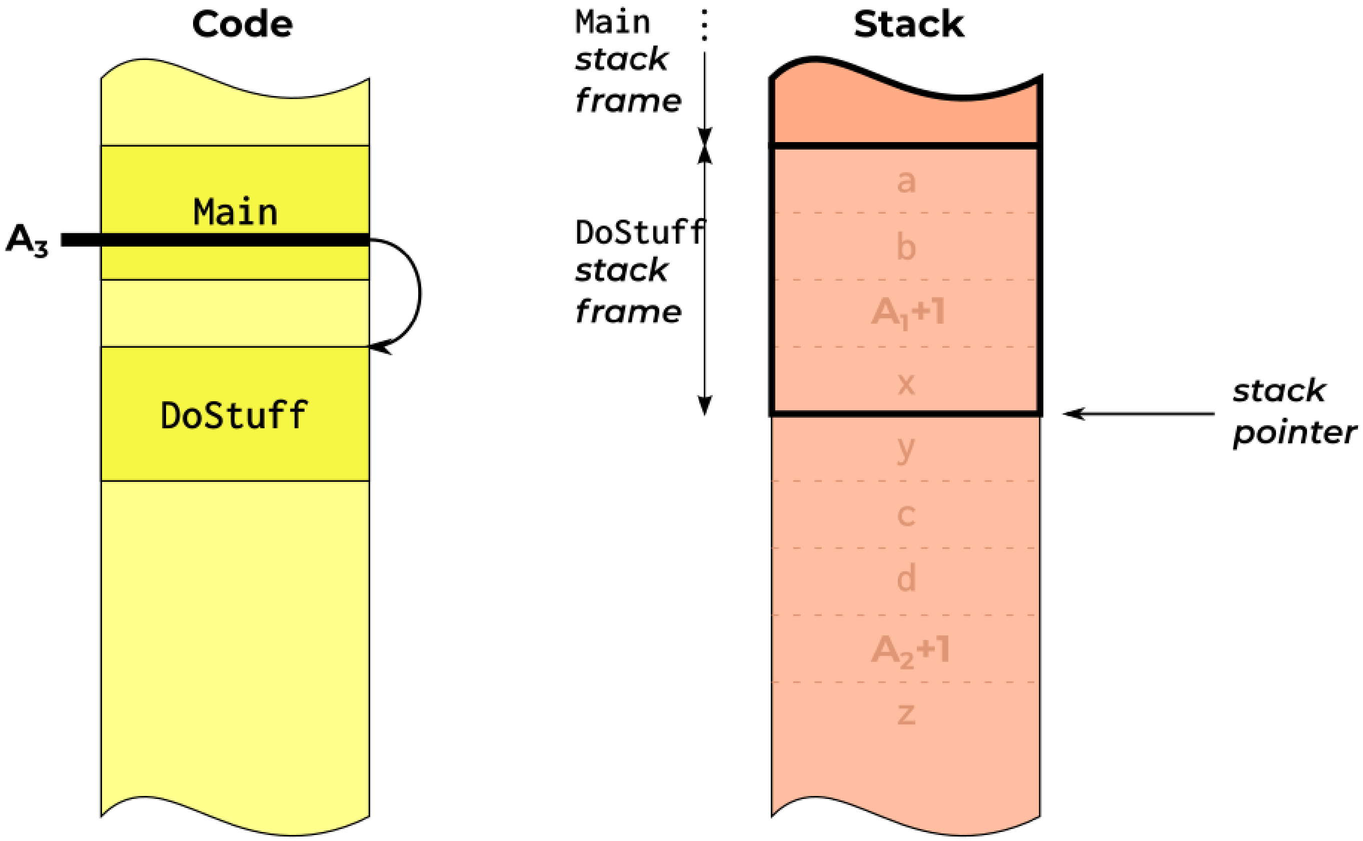
Memory locations - stack



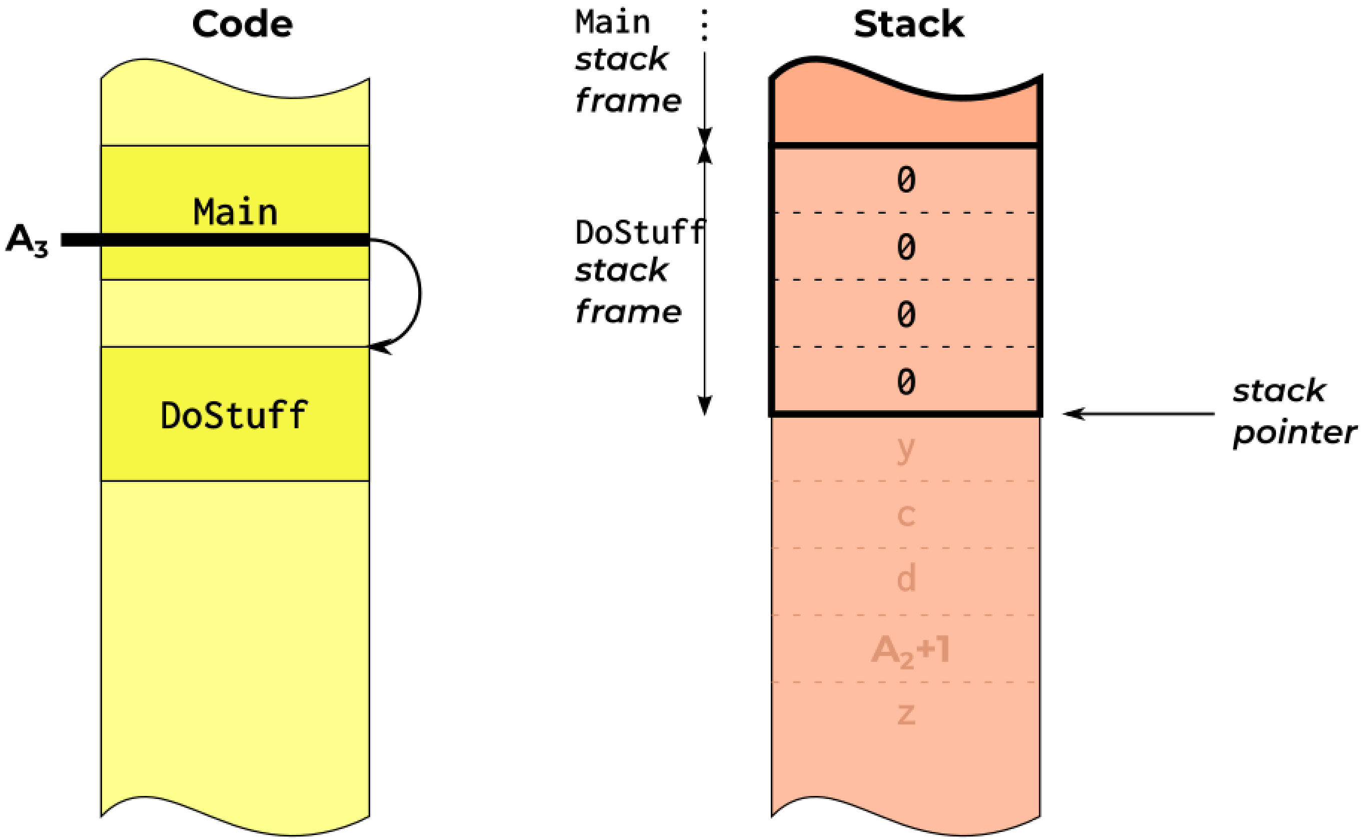
Memory locations - stack zeroing



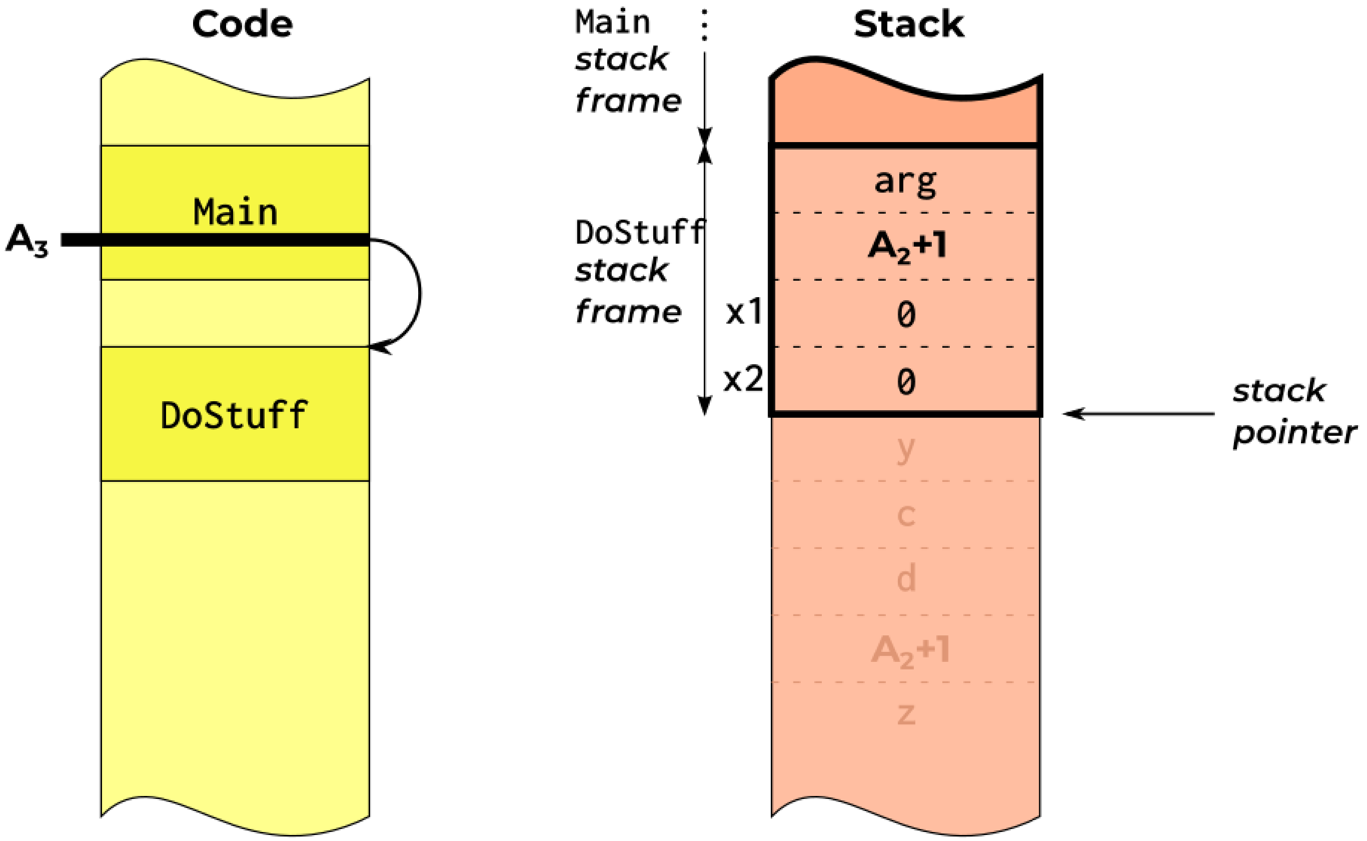
Memory locations - stack zeroing



Memory locations - stack zeroing

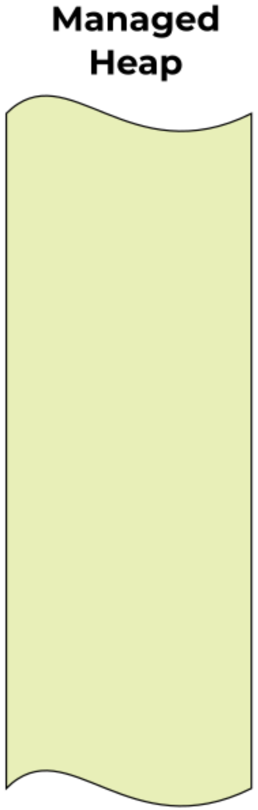


Memory locations - stack zeroing

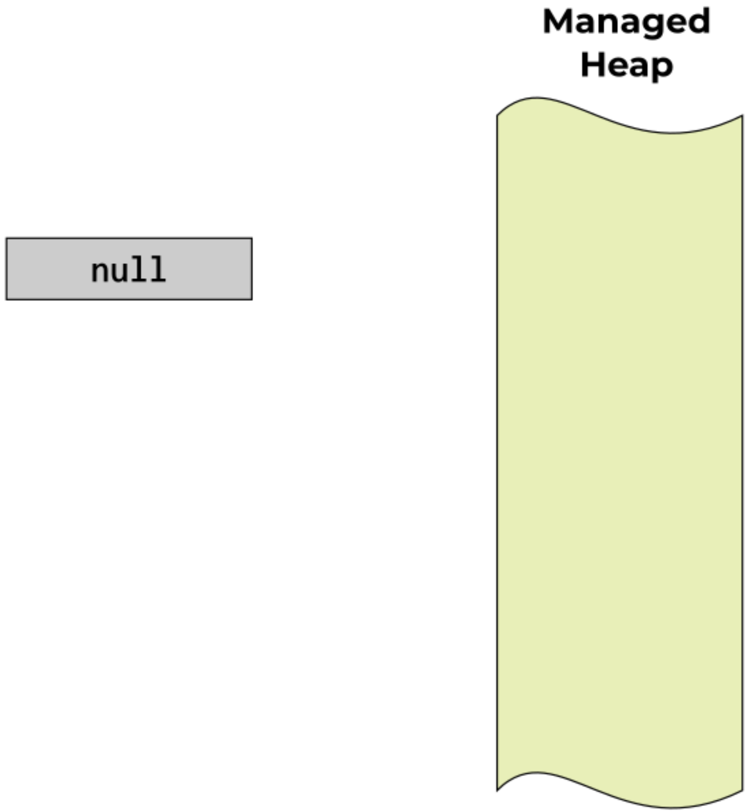


Sidenote: Every thread has its own stack!

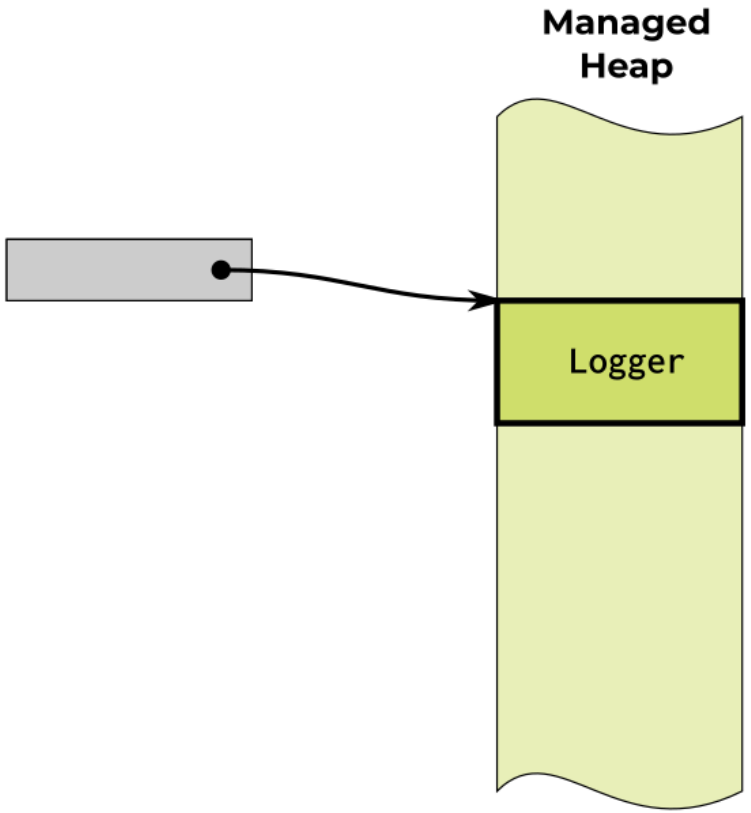
Memory locations - heap



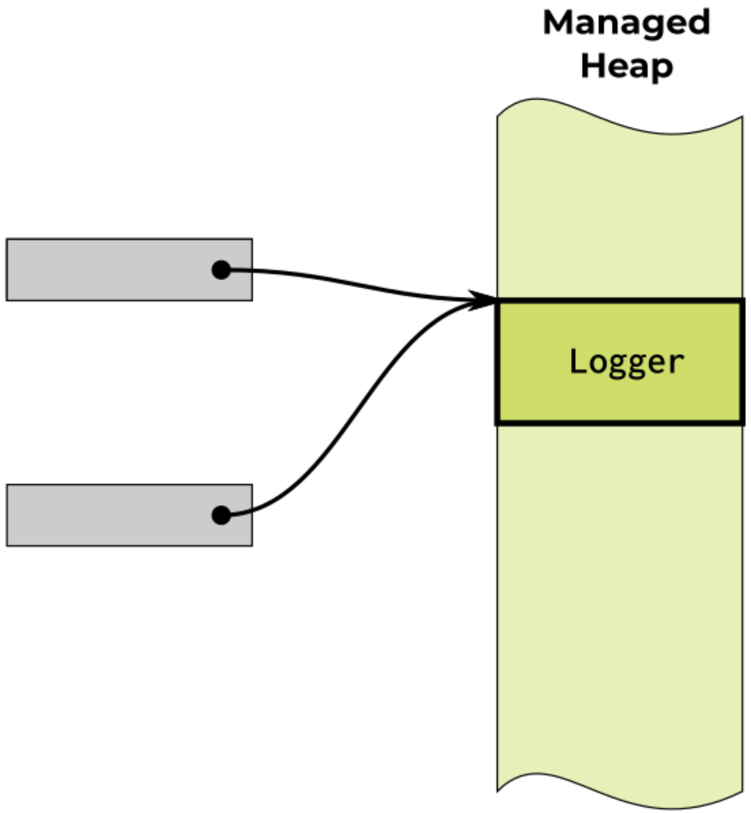
Memory locations - heap



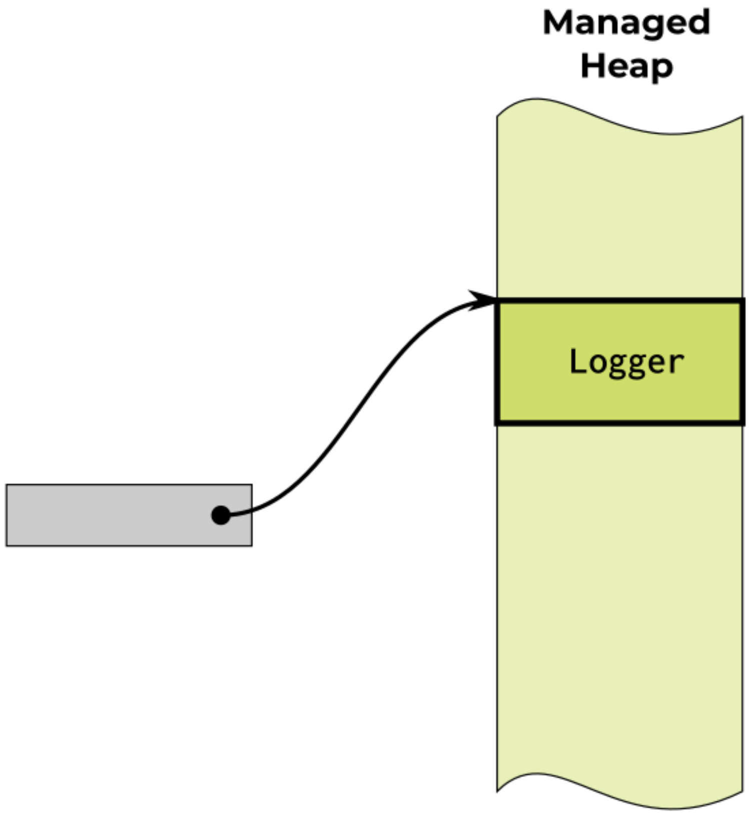
Memory locations - heap



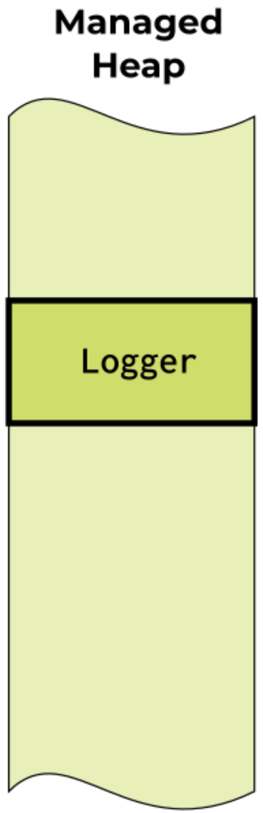
Memory locations - heap



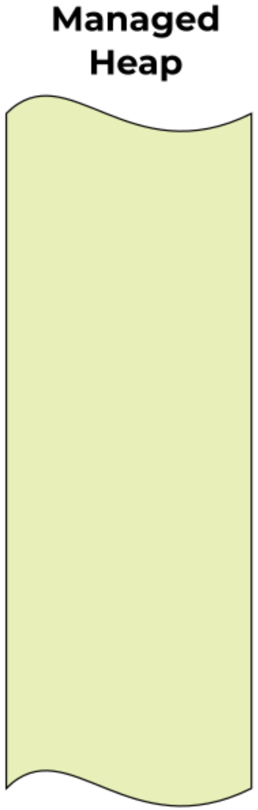
Memory locations - heap



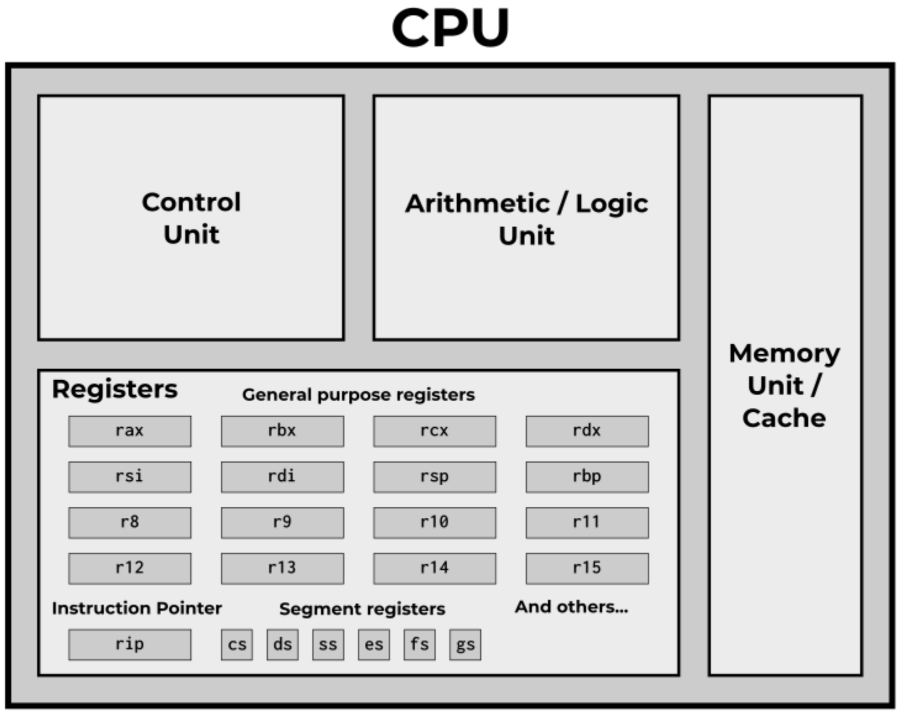
Memory locations - heap



Memory locations - heap



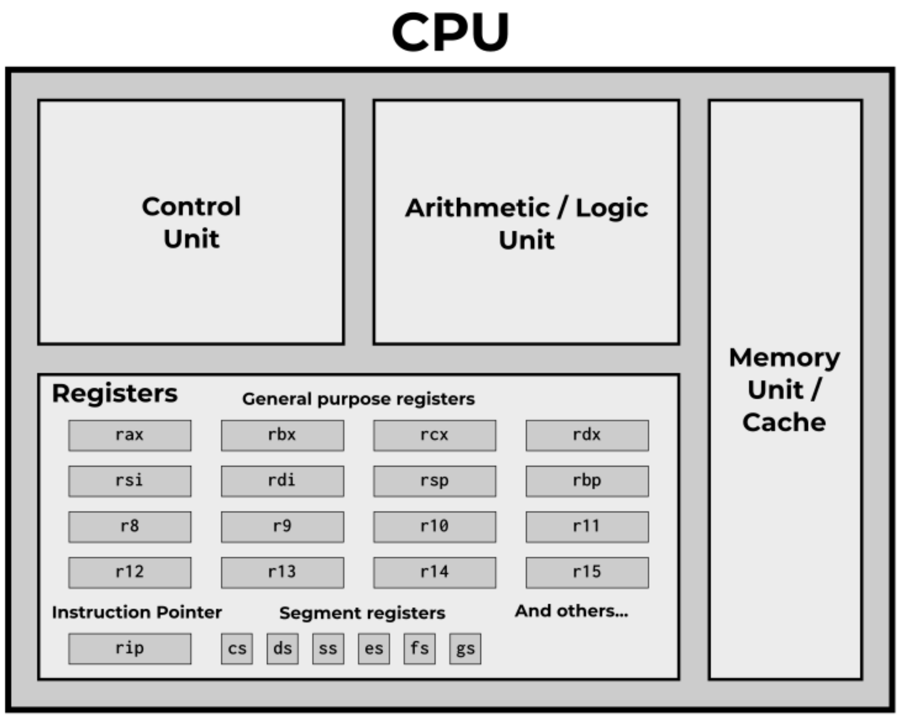
Memory locations - registers



- 16 general purpose registers may be used to store 64-bit data, but:
    - **rsp** - is *stack pointer*
    - **rbp** - is *base pointer*
- for example:

```
mov      rax, dword ptr [0x000000ffffee0000]
mov      edx, dword ptr [0x000000ffffee0100]
add      eax, edx
```

Memory locations - registers



Calling conventions:

- Microsoft x64 (Windows):
  - first four arguments: **`rcx`**, **`rdx`**, **`r8`**, **`r9`**
  - next arguments: stack
  - return value: **`rax`**
- AMD64 (Linux, macOS):
  - first six arguments: **`rdi`**, **`rsi`**, **`rdx`**, **`rcx`**, **`r8`**, **`r9`**
  - next arguments: stack
  - return value: **`rax`**
- floating point arguments and return value use special **`xmm`** registers

Memory locations

	Value type	Reference type
method's local variable	method call lifetime ☞**Stack/CPU**	reference - method's lifetime, referenced data - probably outlives method ☞**Heap***
method's argument	*as above*	*as above (without EA)*
instance field of reference type **	same as the lifetime of the containing reference type value ☞**Heap***	at least the lifetime of the containing reference type value ☞**Heap**
instance field of value type	same as the value type instance ☞**Stack/CPU** or ☞**Heap**	reference - value's lifetime, referenced data - unknown lifetime ☞**Heap**
static field	(long) module lifetime ☞Module-related **blob** or **Heap**	(long) module lifetime ☞**Heap**
local memory pool	method call lifetime ☞**Stack/CPU**	...

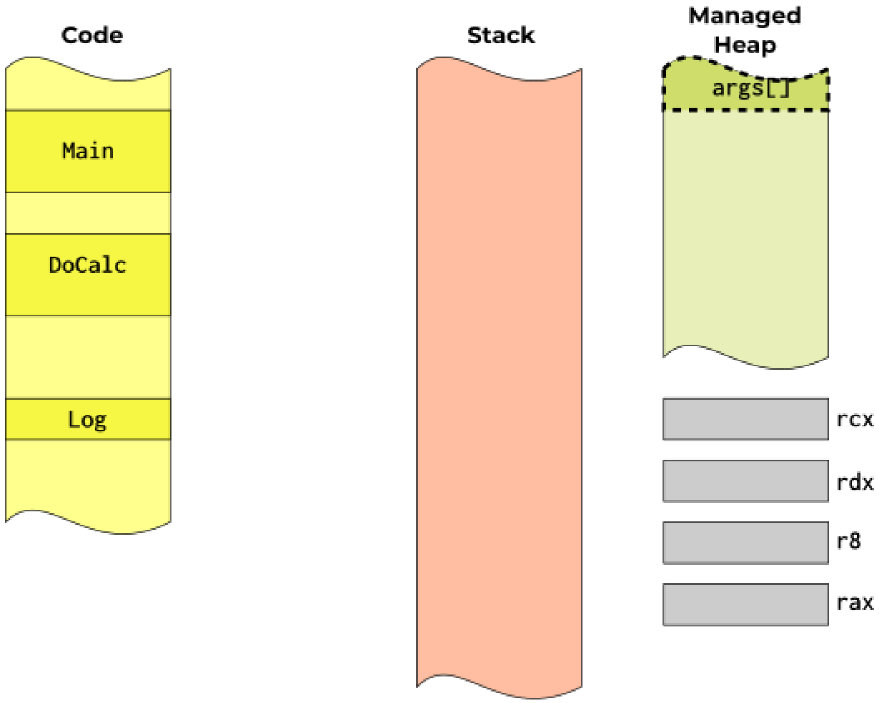
* unless **Escape analysis**/JIT detects it is method-limited (does not "escape") and **Stack/CPU** is enough, ** including arrays

Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

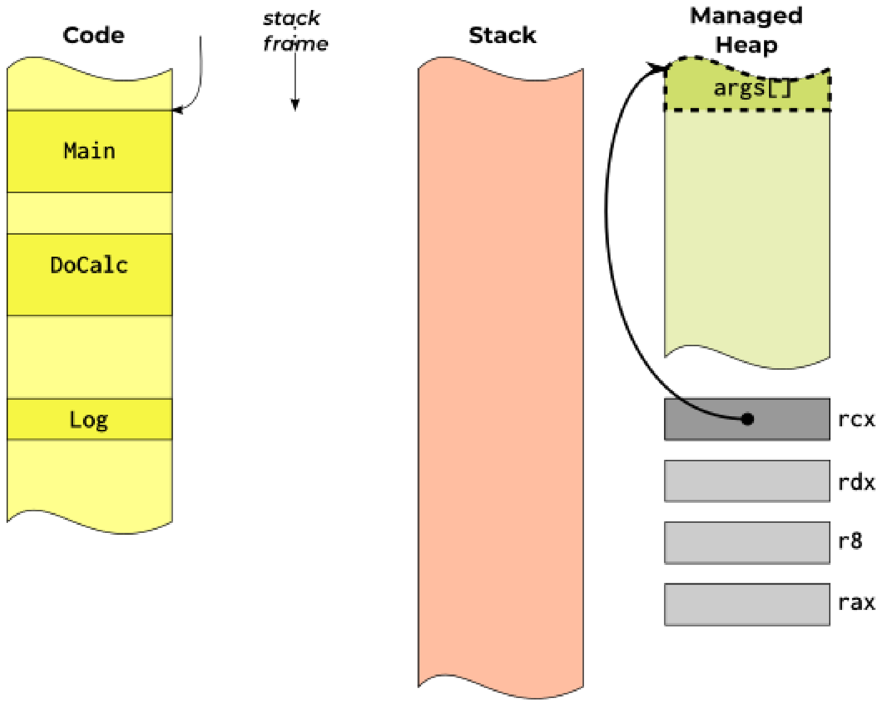


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

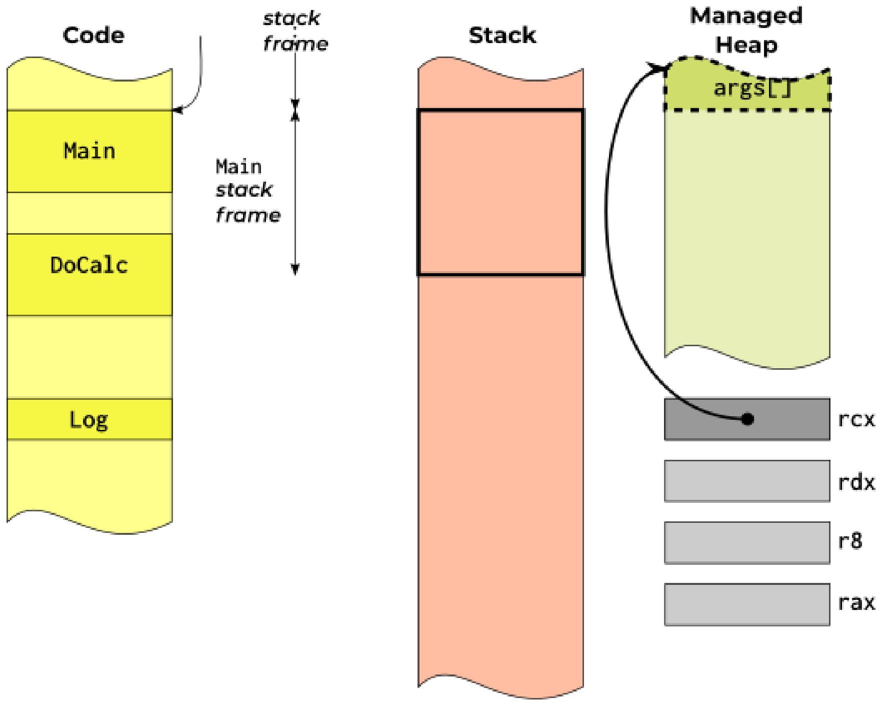


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

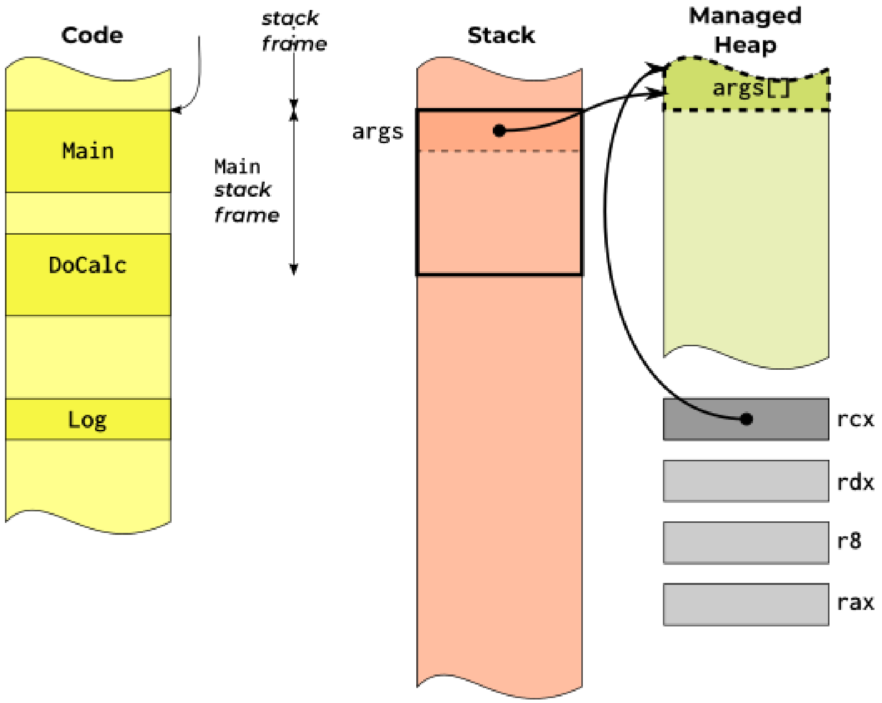


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

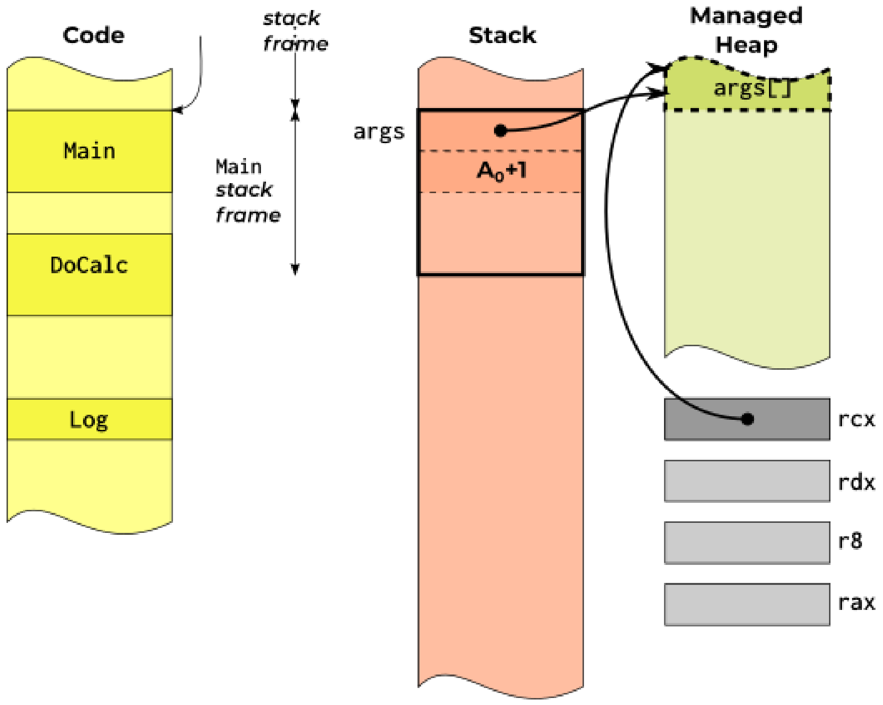


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

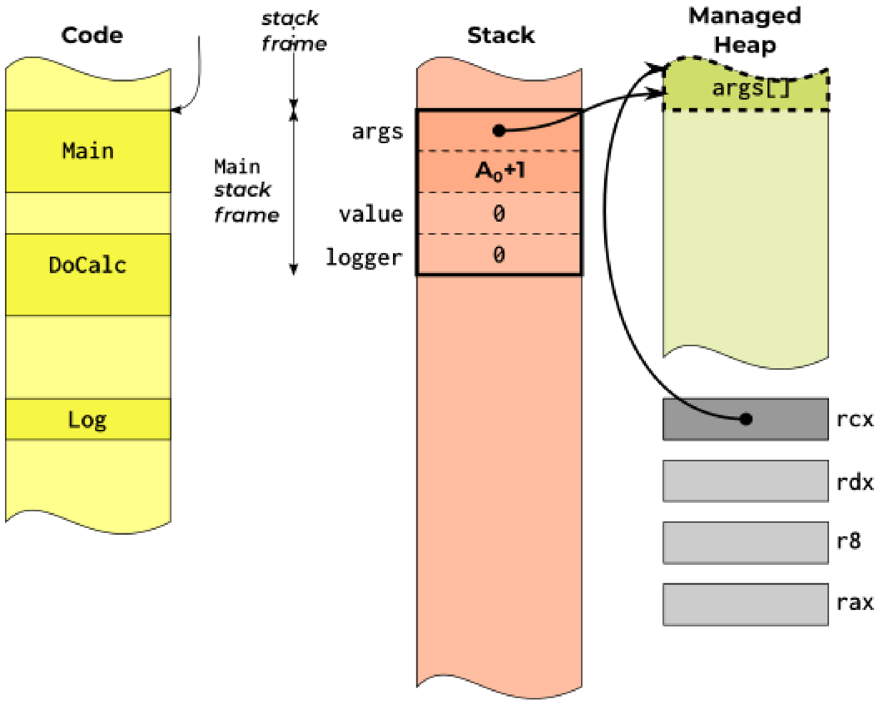


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

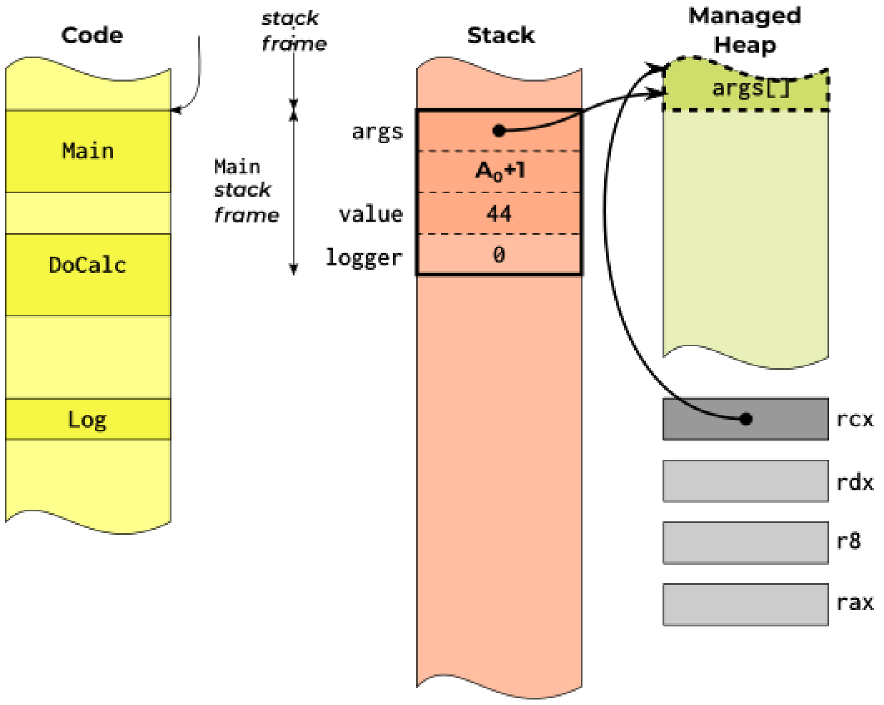


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

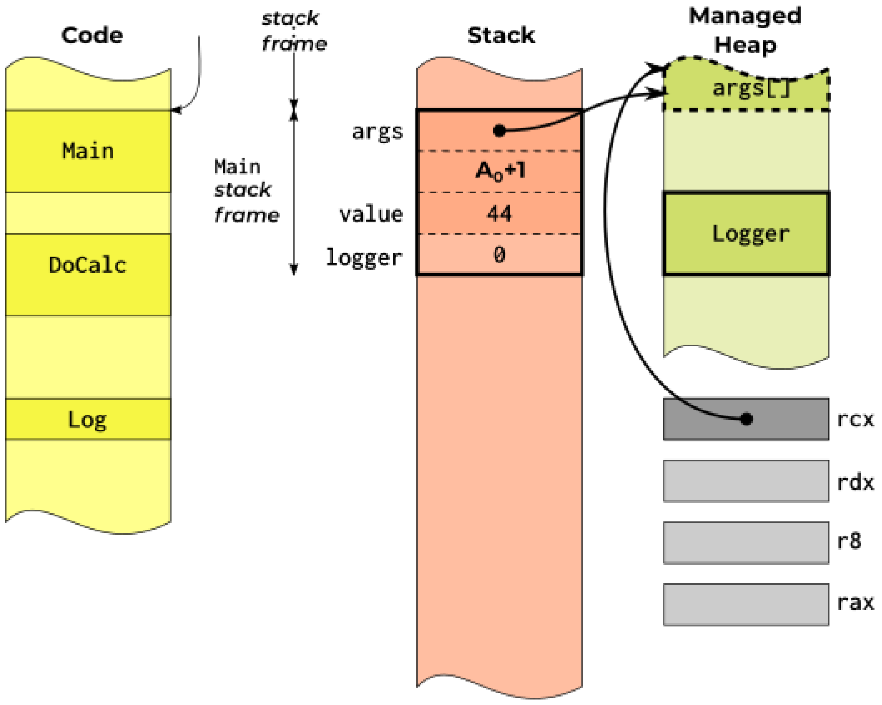


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

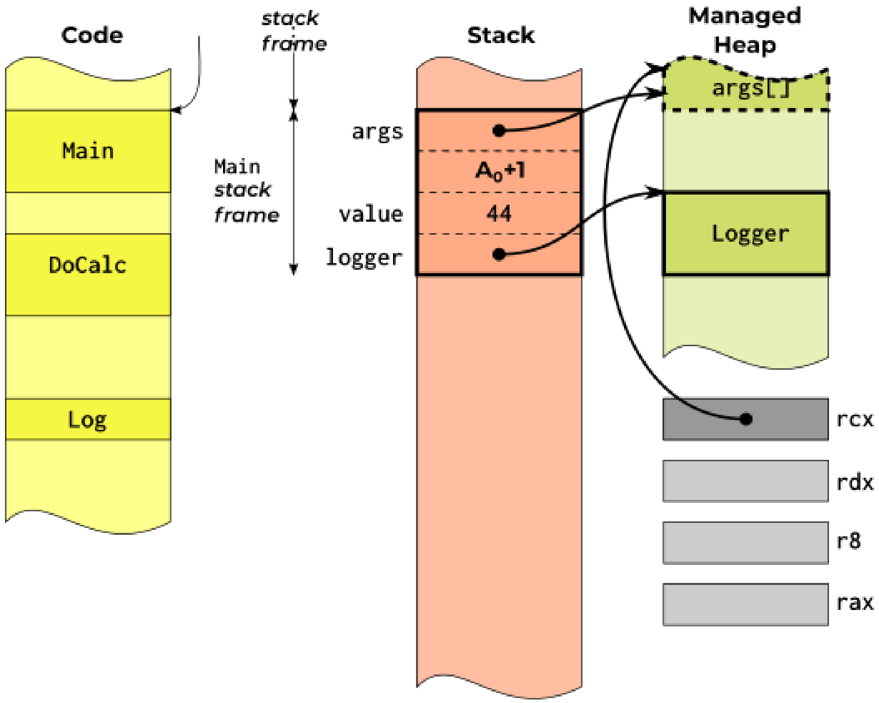


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

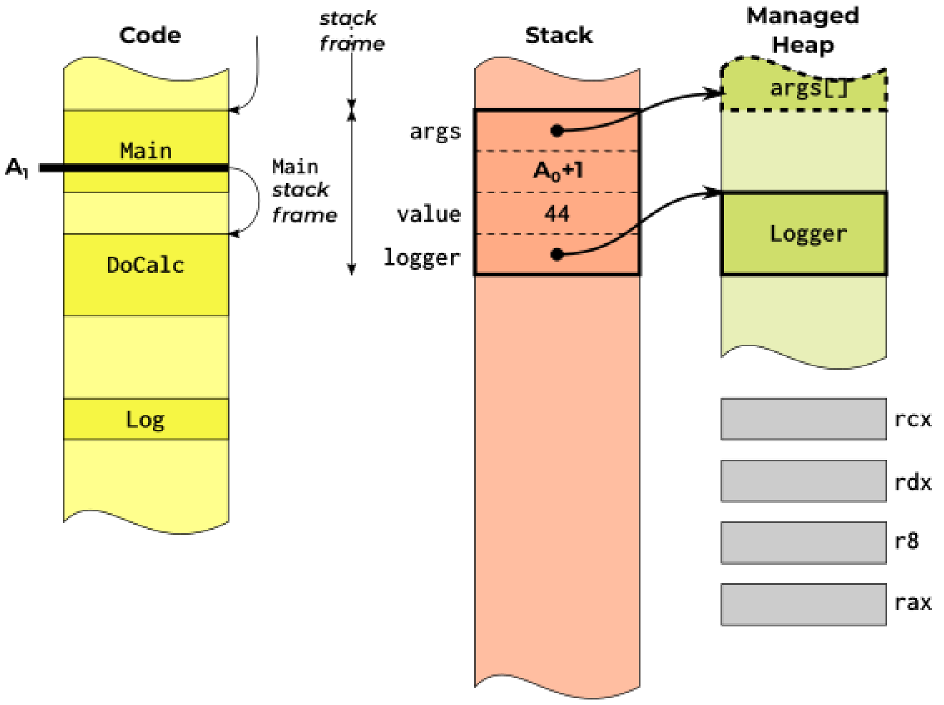


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

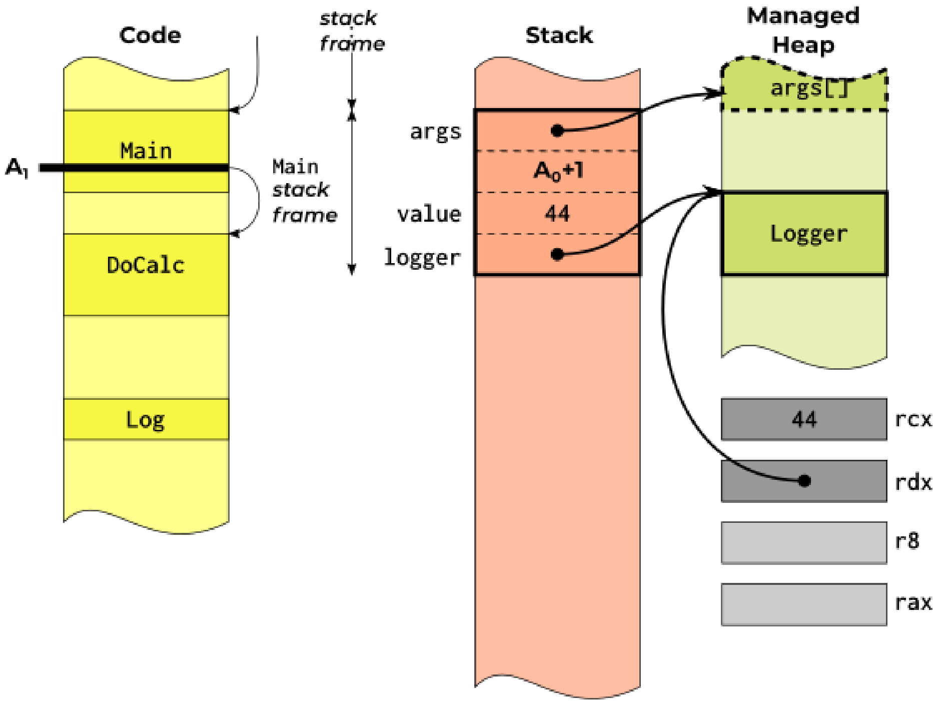


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

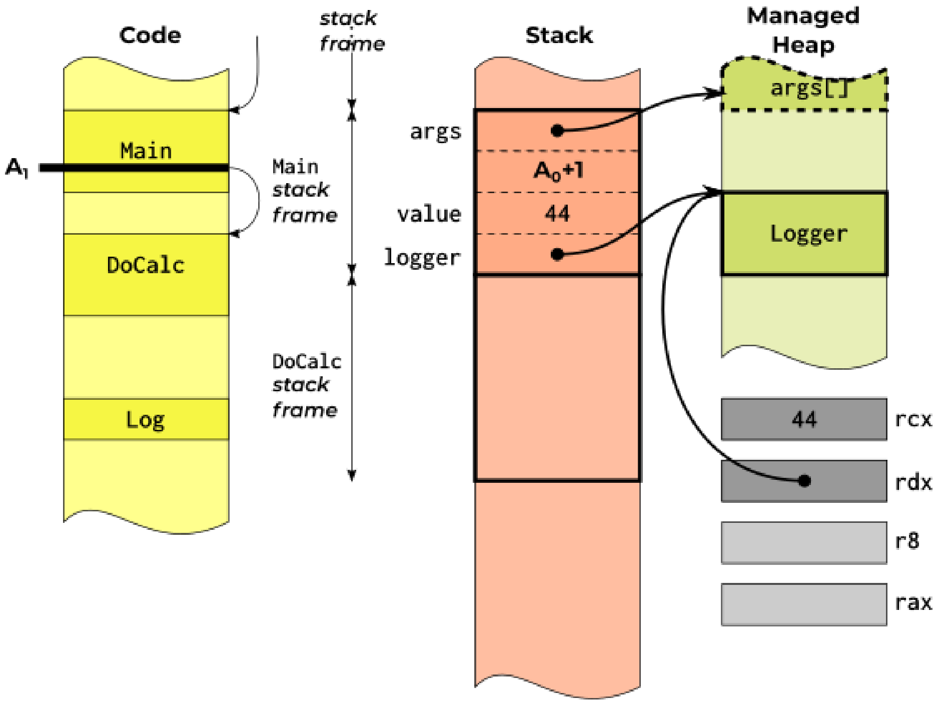


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

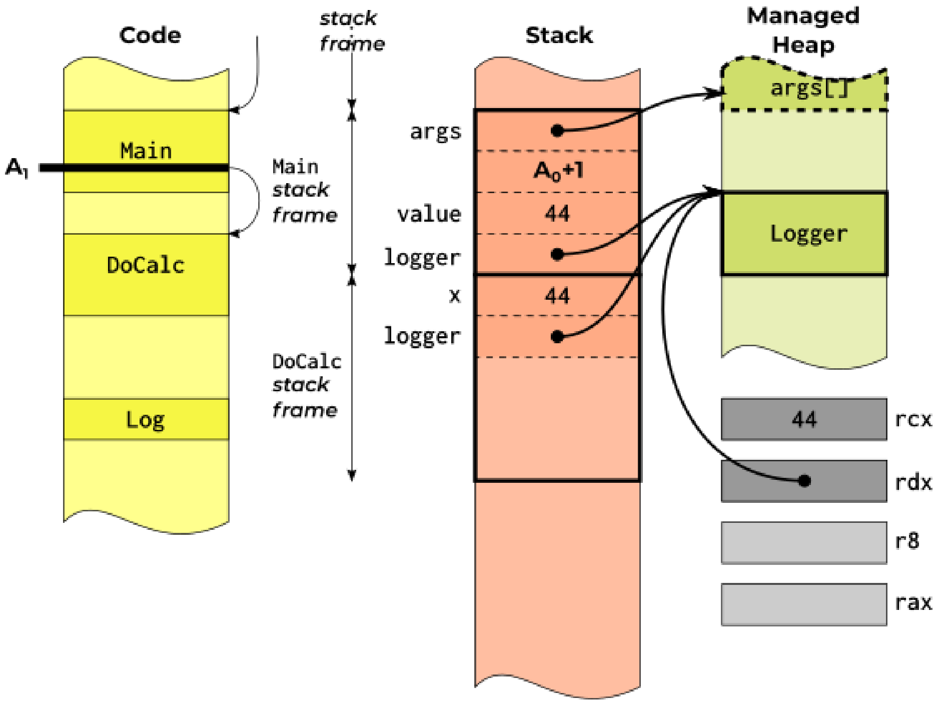


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

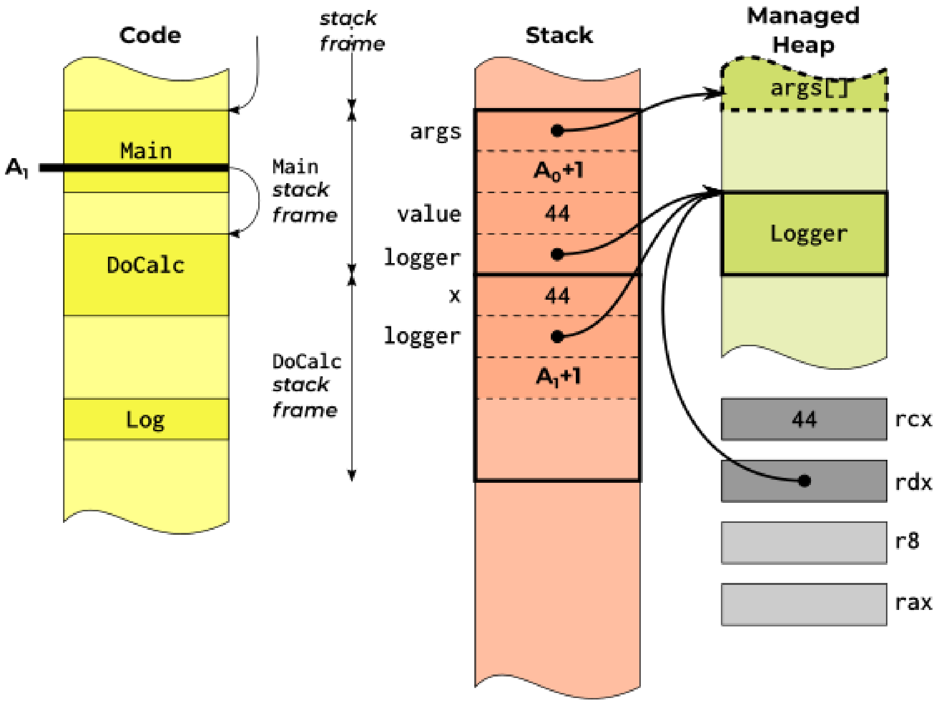


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

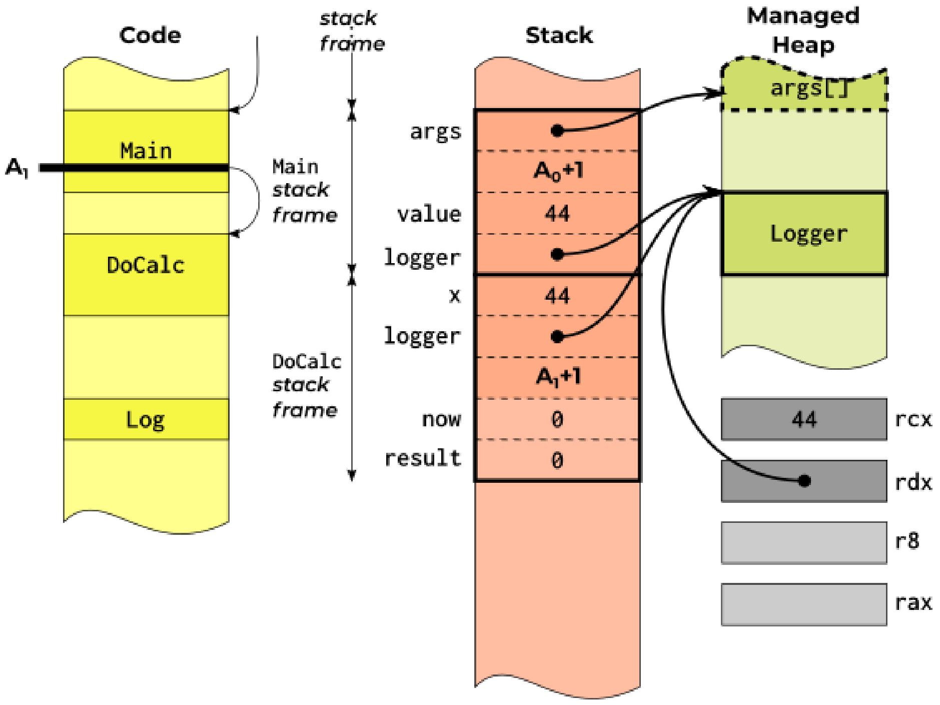


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```



`DateTime` sidenote

`DateTime` is struct with only single **`ulong`** field (8 bytes):

DateTime **sidenote**

DateTime is struct with only single **ulong** field (8 bytes):

```
public readonly partial struct DateTime : IComparable, ISpanFormattable, IConvertible,
                                          IComparable<DateTime>, IEquatable<DateTime>, ISerializable

        // The data is stored as an unsigned 64-bit integer
        //   Bits 01-62: The value of 100-nanosecond ticks where 0 represents 1/1/0001 12:00am, up until the value
        //               12/31/9999 23:59:59.9999999
        //   Bits 63-64: A four-state value that describes the DateTimeKind value of the date time, with a 2nd
        //               value for the rare case where the date time is local, but is in an overlapped daylight
        //               savings time hour and it is in daylight savings time. This allows distinction of these
        //               otherwise ambiguous local times and prevents data loss when round tripping from Local to
        //               UTC time.
        private readonly ulong _dateData;
```

`DateTime` sidenote

`DateTime` is struct with only single **`ulong`** field (8 bytes):

```
public readonly partial struct DateTime : IComparable, ISpanFormattable, IConvertible,
                                          IComparable<DateTime>, IEquatable<DateTime>, ISerializable

        // The data is stored as an unsigned 64-bit integer
        //   Bits 01-62: The value of 100-nanosecond ticks where 0 represents 1/1/0001 12:00am, up until the value
        //               12/31/9999 23:59:59.9999999
        //   Bits 63-64: A four-state value that describes the DateTimeKind value of the date time, with a 2nd
        //               value for the rare case where the date time is local, but is in an overlapped daylight
        //               savings time hour and it is in daylight savings time. This allows distinction of these
        //               otherwise ambiguous local times and prevents data loss when round tripping from Local to
        //               UTC time.
        private readonly ulong _dateData;
```

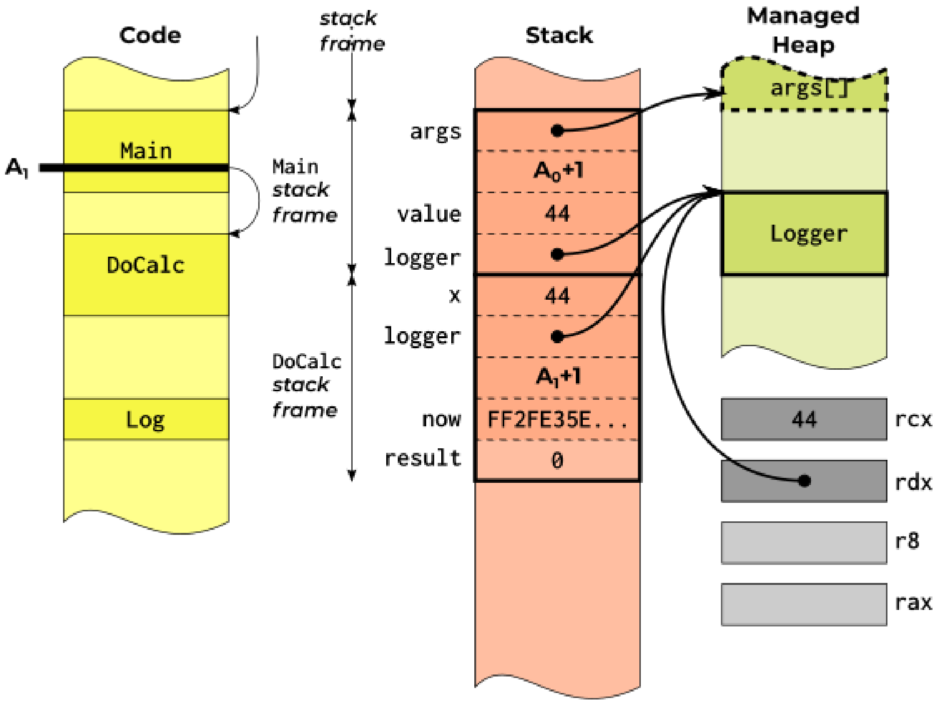
Thus, the whole struct is also 8 bytes.

Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

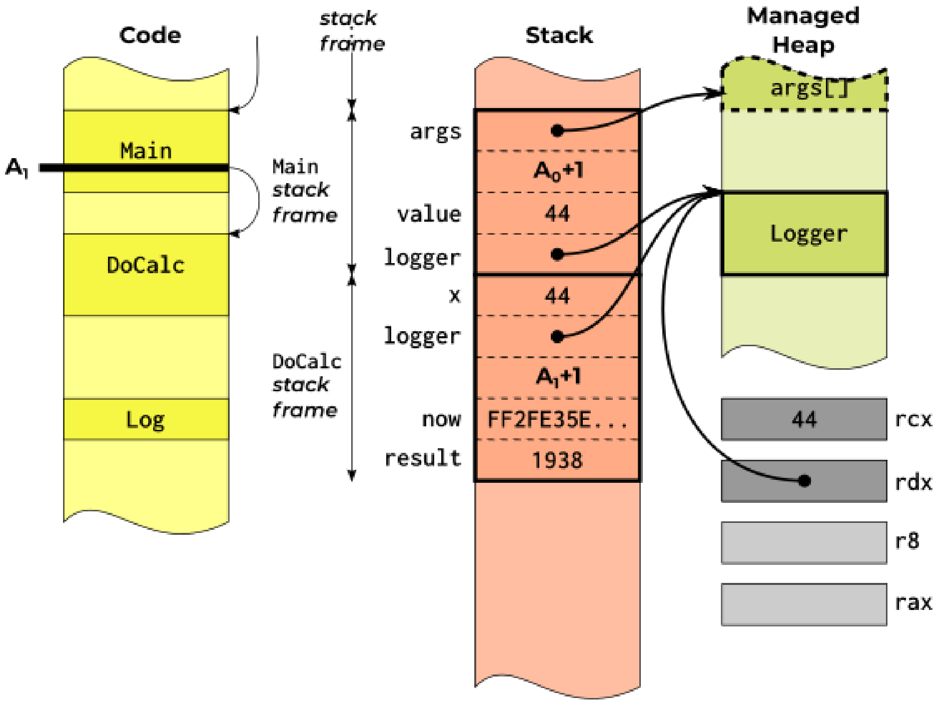


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

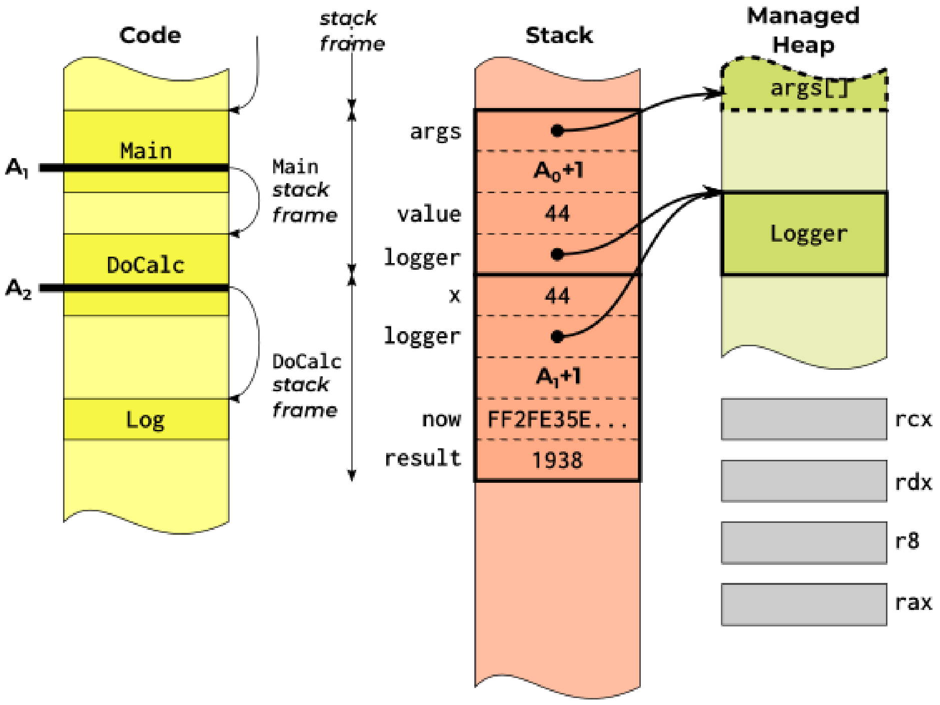


Memory locations - sample (*Debuggish* version)

```csharp
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

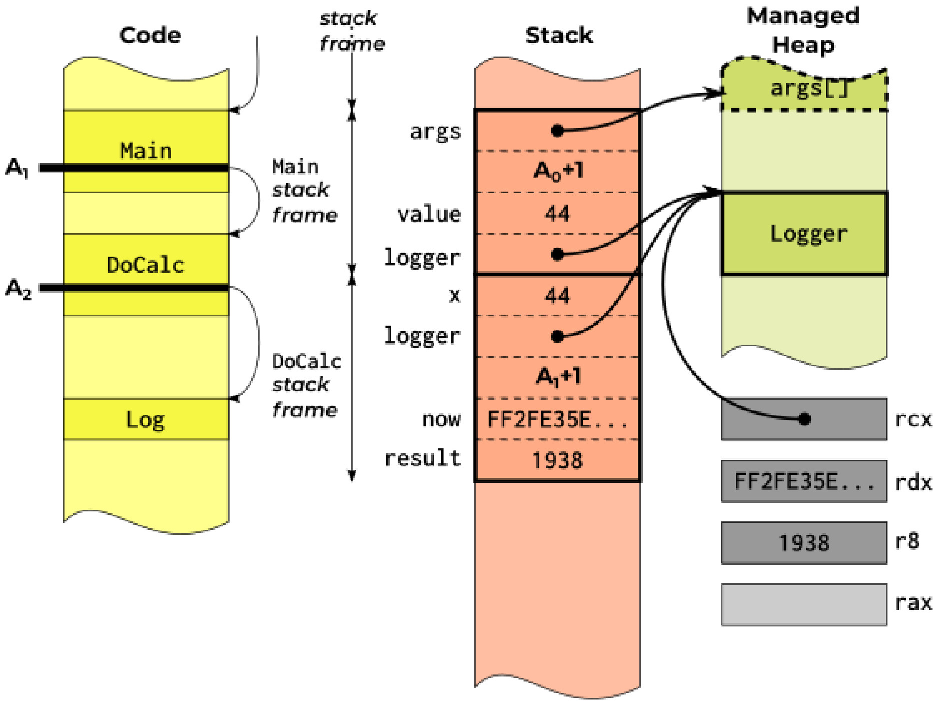


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

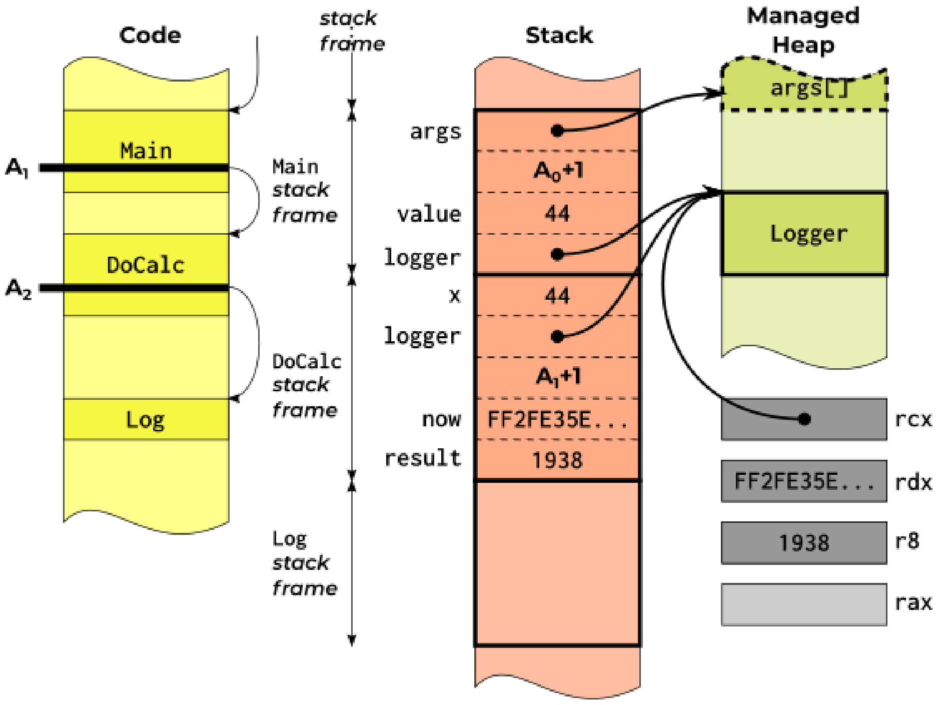


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

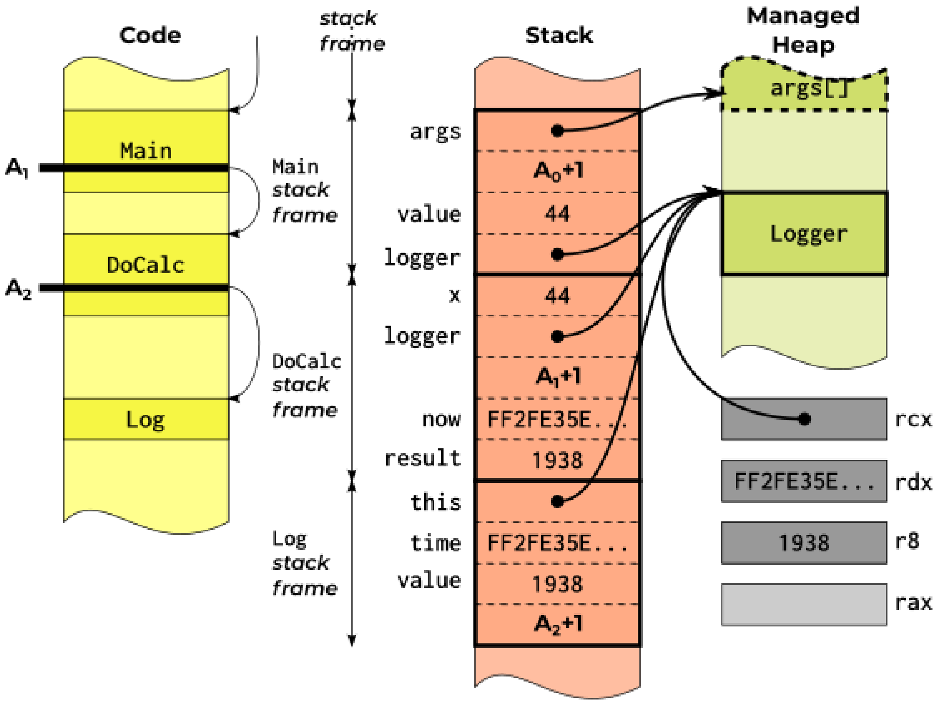


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

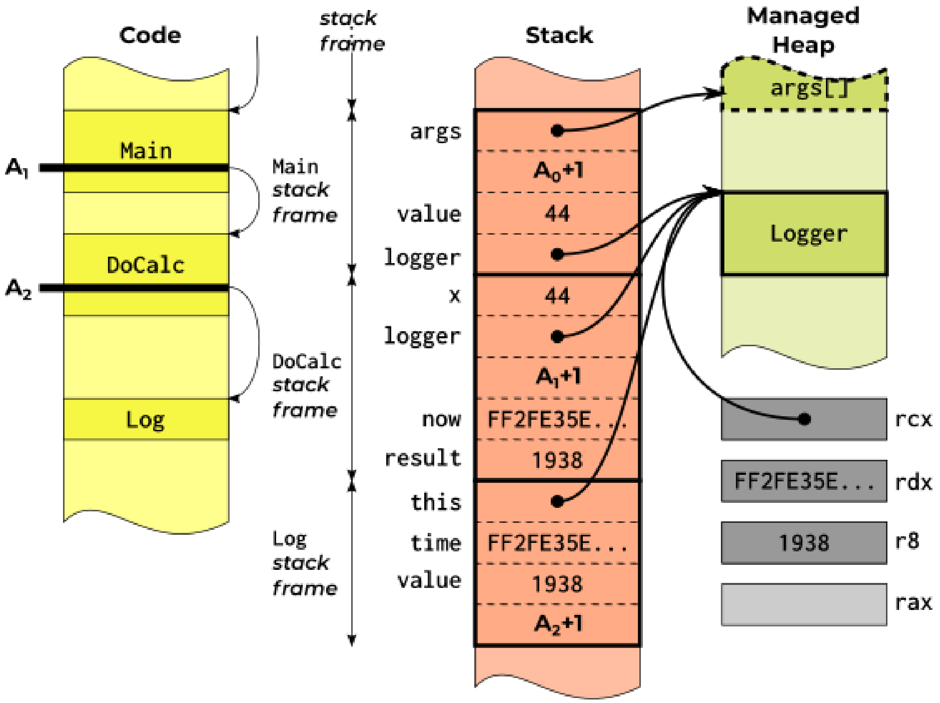


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

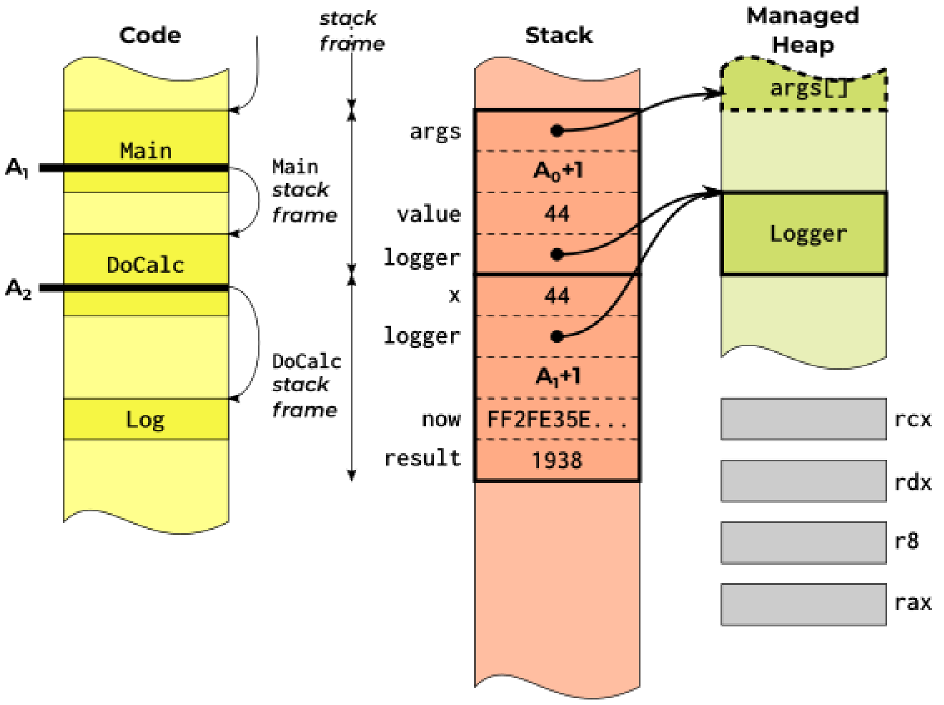


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

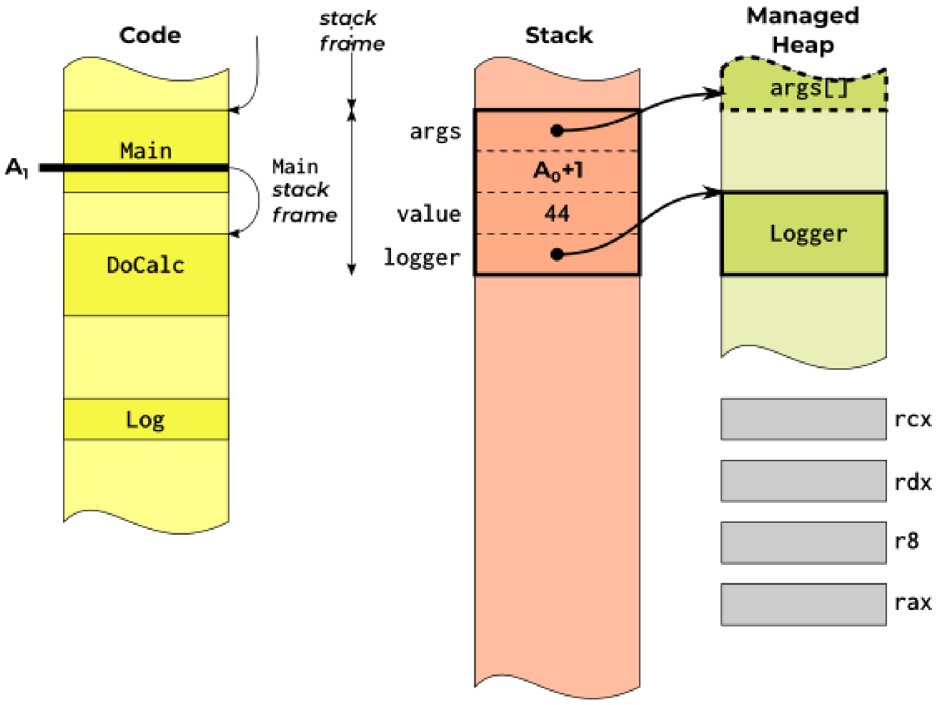


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

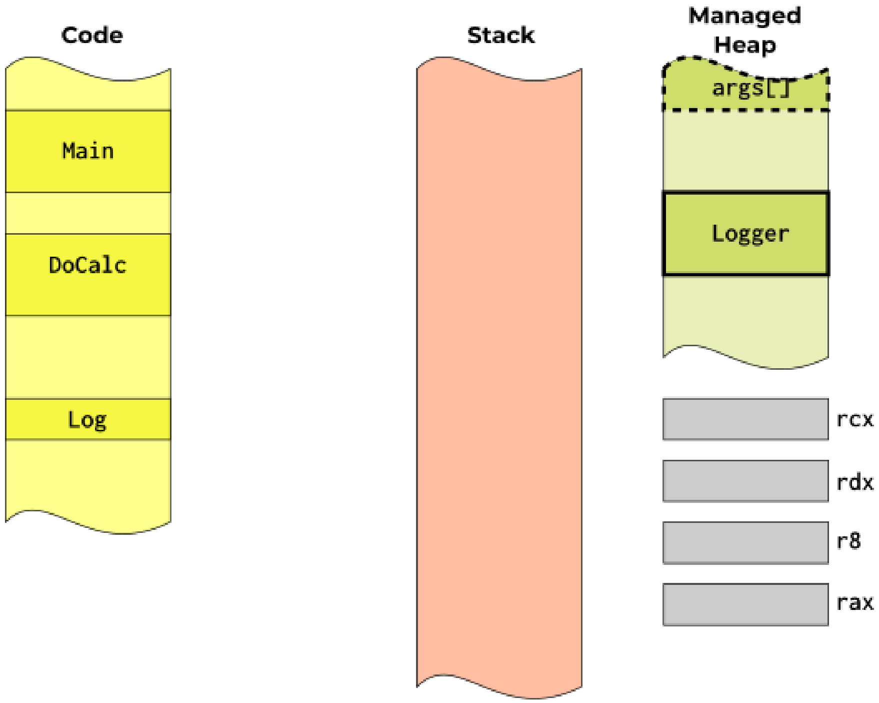


Memory locations - sample (*Debuggish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```



Memory locations - sample (*Release-ish* version)

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

Well…

- stack frames are created to call **`System.Number.ParseInt32`** or **`System.DateTime.get_Now()`**
- but…
- everything else uses CPU registers (including **`DateTime`**, as it fits)
- even no stack is used for locals/arguments 🥰

`.maxstack` sidenote

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

```
.method public hidebysig static void DoCalc
  (int32 x, class Logger logger) {
    .maxstack 3
    .locals init (
        [0] valuetype System.DateTime now,
        [1] int32 result)
    ...
}

.method public hidebysig instance void Log
  (valuetype System.DateTime time, int32 'value') {
    .maxstack 8
    ...
}
```

.maxstack sidenote

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

```
.method public hidebysig static void DoCalc
  (int32 x, class Logger logger) {
    .maxstack 3
    .locals init (
        [0] valuetype System.DateTime now,
        [1] int32 result)
    ...
}

.method public hidebysig instance void Log
  (valuetype System.DateTime time, int32 'value') {
    .maxstack 8
    ...
}
```

- **.maxstack** is metadata information about the maximum expected depth of the **evaluation stack**

`.maxstack` **sidenote**

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

```
.method public hidebysig static void DoCalc
  (int32 x, class Logger logger) {
    .maxstack 3
    .locals init (
        [0] valuetype System.DateTime now,
        [1] int32 result)
    ...
}

.method public hidebysig instance void Log
  (valuetype System.DateTime time, int32 'value') {
    .maxstack 8
    ...
}
```

- `.maxstack` is metadata information about the maximum expected depth of the **evaluation stack**
- if not specified in metadata, it is assumed `8`

`.maxstack` sidenote

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

```
.method public hidebysig static void DoCalc
  (int32 x, class Logger logger) {
    .maxstack 3
    .locals init (
        [0] valuetype System.DateTime now,
        [1] int32 result)
    ...
}

.method public hidebysig instance void Log
  (valuetype System.DateTime time, int32 'value') {
    .maxstack 8
    ...
}
```

- `.maxstack` is metadata information about the maximum expected depth of the **evaluation stack**
- if not specified in metadata, it is assumed `8`
- may be used during (optional) **IL verification** process - not during JIT

`.maxstack` sidenote

```
public class Program {
  static public void Main(string[] args)
  {
    var value = int.Parse(args[0]); // assume 44
    var logger = new Logger();
    DoCalc(value, logger);
  }

  static public void DoCalc(int x, Logger logger) {
    DateTime now = DateTime.Now;
    int result = /* do some calculations */
    logger.Log(now, result);
  }
}

public class Logger {
  public void Log(DateTime time, int value) {
    Console.WriteLine($"[{time}] {value}");
  }
}
```

```
.method public hidebysig static void DoCalc
  (int32 x, class Logger logger) {
    .maxstack 3
    .locals init (
        [0] valuetype System.DateTime now,
        [1] int32 result)
    ...
}

.method public hidebysig instance void Log
  (valuetype System.DateTime time, int32 'value') {
    .maxstack 8
    ...
}
```

- **`.maxstack`** is metadata information about the maximum expected depth of the **evaluation stack**
- if not specified in metadata, it is assumed **`8`**
- may be used during (optional) **IL verification** process - not during JIT
- thus, it has **nothing** in common with the thread stack

Escape analysis

From the *Object Stack Allocation* runtime doc:

> *"If the lifetime of an object **is bounded by the lifetime of the allocating method, the allocation may be moved to the stack**. The benefits of this optimization:*
>
> - *The pressure on the garbage collector is reduced because the GC heap becomes smaller. The garbage collector doesn't have to be involved in allocating or deallocating these objects.*
> - *Object field accesses may become cheaper if the compiler is able to do scalar replacement of the fields of the stack-allocated object (i.e., if the fields can be promoted).*
> - *Some field zero-initializations may be elided by the compiler."*

Escape analysis

```
public class EscapeAnalysis
{
    private int _x = 10;
    private int _y = 20;

    public int UseCalculator()
    {
        Calculator calc = new Calculator();
        calc.X = _x;
        calc.Y = _y;
        return calc.Add();
    }
}

public class Calculator
{
    public int X { get; set; }
    public int Y { get; set; }

    public int Add() => X + Y;
}
```

Escape analysis

```
public class EscapeAnalysis
{
    private int _x = 10;
    private int _y = 20;

    public int UseCalculator()
    {
        Calculator calc = new Calculator();
        calc.X = _x;
        calc.Y = _y;
        return calc.Add();
    }
}

public class Calculator
{
    public int X { get; set; }
    public int Y { get; set; }

    public int Add() => X + Y;
}
```

COMPlus_JitObjectStackAllocation=0:

```
UseCalculator():
mov     rcx,7FF7C10955B0h
call    00007ff8`208f7840      ; allocate Calculator
mov     edx,dword ptr [rsi+8]
mov     dword ptr [rax+8],edx   ; set X
mov     edx,dword ptr [rsi+0Ch]
mov     dword ptr [rax+0Ch],edx ; set Y
mov     edx,dword ptr [rax+8]
add     edx,dword ptr [rax+0Ch] ; inlined Add
mov     eax,edx
```

Escape analysis

```
public class EscapeAnalysis
{
    private int _x = 10;
    private int _y = 20;

    public int UseCalculator()
    {
        Calculator calc = new Calculator();
        calc.X = _x;
        calc.Y = _y;
        return calc.Add();
    }
}

public class Calculator
{
    public int X { get; set; }
    public int Y { get; set; }

    public int Add() => X + Y;
}
```

COMPlus_JitObjectStackAllocation=0:

```
UseCalculator():
mov     rcx,7FF7C10955B0h
call    00007ff8`208f7840       ; allocate Calculator
mov     edx,dword ptr [rsi+8]
mov     dword ptr [rax+8],edx   ; set X
mov     edx,dword ptr [rsi+0Ch]
mov     dword ptr [rax+0Ch],edx ; set Y
mov     edx,dword ptr [rax+8]
add     edx,dword ptr [rax+0Ch] ; inlined Add
mov     eax,edx
```

COMPlus_JitObjectStackAllocation=1:

```
UseCalculator():
mov     eax,dword ptr [rcx+8]
mov     edx,dword ptr [rcx+0Ch]
add     eax,edx
```

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but… might be allocated **on the heap** - fe. as a field of reference type, or boxed, or…
  - but… might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique
- important **mostly** in high performance code, when we want to take leverage of those details

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique
- important **mostly** in high performance code, when we want to take leverage of those details

Semantic difference:

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique
- important **mostly** in high performance code, when we want to take leverage of those details

Semantic difference:

- value types - passed by value (copy!)

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique
- important **mostly** in high performance code, when we want to take leverage of those details

Semantic difference:

- value types - passed by value (copy!)
- reference types - passed by reference

"value types are allocated on the stack and reference types are allocated on the heap"

Implementation details:

- value types:
  - might be allocated **on the stack** (or even into **CPU registers**)
  - but... might be allocated **on the heap** - fe. as a field of reference type, or boxed, or...
  - but... might be allocated in a special **statics blob**
- reference types:
  - typically allocated **on the heap**
  - but sometimes optimized away or allocated on **on the stack** (or **CPU registers**) by *object stack allocation* technique
- important **mostly** in high performance code, when we want to take leverage of those details

Semantic difference:

- value types - passed by value (copy!)
- reference types - passed by reference

We will return to those differences when talking about allocations and low-level optimizations.

Materials

- x64 Assembly - Brent's Website
- x64 calling convention
- Object Stack Allocation/Escape Analysis in .NET runtime doc