

Optimising code using Span<T>

Mirco Vanini



Sponsor



Mirco Vanini

Microsoft MVP Developer Technologies

Consultant focused on industrial and embedded solutions using .NET and other native SDKs with over 35 years of experience, XeDotNet community co-founder, speaker and Microsoft MVP since 2012



@MircoVanini

www.proxsoft.it

<https://www.linkedin.com/in/proxsoft>

<https://sessionize.com/mirco-vanini>



Back to Basics

- .NET is a managed platform, that means the memory access and management is safe and automatic. All types are fully managed by .NET, it allocates memory either on the execution stacks, or managed heaps.
- In .NET world, there are three types of memory:
 - Managed heap memory, such as an array;
 - Stack memory, such as objects created by stackalloc;
 - Native memory, such as a native pointer reference.

Back to Basics

Value Types

- **struct** keyword
- Allocated on **stack** or embedded into a reference object
- A variable for a value type is the value itself, e.g. integer
- Passed around by value (i.e. copied)
- Assignment is a copy of the whole value

Reference Types

- **class** keyword
- Allocated on **heap**
- A variable for a reference type is a reference to the thing on the heap not the object
- Passed around by reference
- Assignment is a copy of the reference,

Back to Basics

Staff on the Stack

Instance Id	Pointer	Address	Value
0x40	0x30	0x30	33 (field 1 value)

Staff on the Managed Heap

Instance Id	Pointer	Address	Value
0x40	0x30	0x1C	(sync block address)
		0x30	0x60 (RTTI address)
		0x..	33 (field 1 value)
		0x..	Maarten (field N value)

Method Table Structure

Address	Value
0x60	
0x6C	Interface Map Table Address
0x..	Inherited virtual method addresses
0x..	Introduced virtual method addresses
0x..	Instance method addresses
0x..	Static method addresses
0x..	Static field 1 value

Back to Basics

In C# by default Value Types are passed to methods by value

In C# all parameters are passed to methods by value by default

[Method Parameters](#)

GC Managed Heap (Workstation mode)

- **Mark**

- Starting from the roots, mark all reachable objects as alive
- In Background

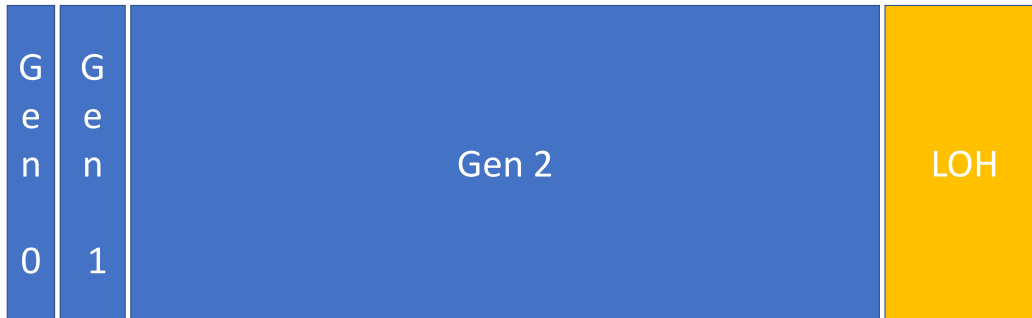
- **Sweep**

- Turning all non-reachable objects into a free space
- Blocking

- **Compact**

- To prevent fragmentation
- Not always
- Blocking

GC Managed Heap (Workstation mode)



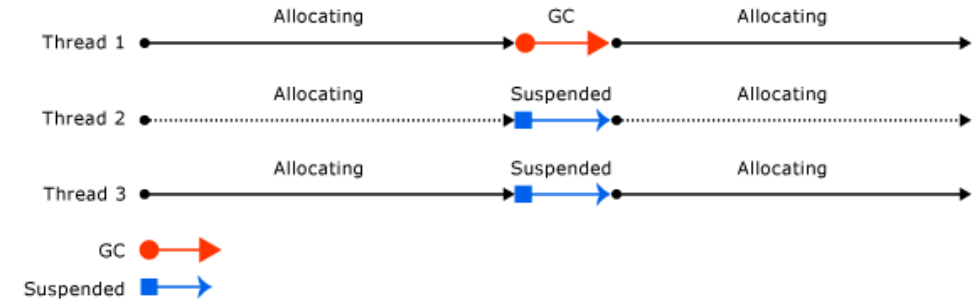
Large Object Heap (LOH)

Special segment for large objects (> 85KB)

Collected only during full garbage collection

Not compacted (by default) -> fragmentation!

Fragmentation can cause OutOfMemoryException



CPU #0	0	1	Gen 2	LOH
CPU #1	0	1	Gen 2	LOH
CPU #2	0	1	Gen 2	LOH
CPU #3	0	1	Gen 2	LOH
CPU #4	0	1	Gen 2	LOH
(...)	0	1	Gen 2	LOH
CPU #n	0	1	Gen 2	LOH

ONE FOR ALL, ALL FOR ONE

[Middle Ground between Server and Workstation GC by Maoni](#)

[Running with Server GC in a Small Container Scenario Part 0 by Maoni](#)

Available Memory

	Managed Heap	Unmanaged Heap	Stack
Safe	Yes	No type safety	Type safe, but can be deadly
What can be allocate	References and values	Values only	Values only
Allocation	Cheap	Cheap	Cheap
Deallocation	Come at price Can be blocking	Cheap	Immediate
Who cleans up	GC	Developer	OS

How to write a code that supports all kinds of memory?

```
int Parse(string text)
```

```
int Parse(string text, int start, int length)
```

```
unsafe int Parse(char* pointer, int length)
```

```
unsafe int Parse(char* pointer, int start, int length)
```

...

The age-old problem

- In many applications, the most CPU consuming operations are string operations. If you run a profiler session against your application, you may find the fact that 95% of the CPU time is used to call string and related functions.
- Trim() or SubString() returns a new string object that is part of the original string, this is unnecessary if there is a way to slice and return a portion of the original string to save one copy.
- IsNullOrWhiteSpace() takes a string object that needs a memory copy (because string is immutable.)
- Specifically, string concatenation is expensive, it takes n string objects, makes n copy, generate $n - 1$ temporary string objects, and return a final string object, the $n - 1$ copies can be eliminated if there is a way to get direct access to the return string memory and perform sequential writes.

Reference Semantics with Value Types

- **in parameters**
 - Pass value type by reference. Called method cannot modify it
- **ref locals and ref returns (C# 7.0)**
- **ref readonly returns**
 - Return a read only value type by reference
- **readonly struct**
 - Immutable value types
- **ref struct**
 - Stack only value types

Write safe and efficient C# code

What is Span<T>?

“System.Span<T> is a new value type at the heart of .NET [that] enables the representation of contiguous regions of arbitrary memory.”

[Stephen Toub, MSDN Magazine, January 2018](#)

“Span<T> is a small, but critical, building block for a much larger effort to provide .NET APIs to enable development of high scalability server applications.”

[dotnet/corefxlab](#)


What is Span<T>?

“System.Span<T> is a **new value type** at the heart of .NET [that] enables the representation of contiguous regions of arbitrary memory.”

- Struct
- Fully type-safe and memory-safe
- Part of System.Memory, available on Nuget
- .NET Standard 1.1 so can be used in .NET 4.5+
 - Slow Span (.NET Standard)
 - Fast Span (.NET Core)
- C# 7.2-ish

[.NET API Catalog](#)

```
namespace System
{
    [DefaultMember("Item")]
    [NativeMarshalling(typeof(SpanMarshaller<?, ?>))]
    public readonly struct Span<T>
}
```



What is Span<T>?

“System.Span<T> is a new value type **at the heart of .NET** [that] enables the representation of contiguous regions of arbitrary memory.”

- High performance O(1), low (no) overhead
- Framework, CLR, JIT and GC support
- Provides memory and type safety
- Avoids the need for unsafe code
- Value Type

```
readonly ref struct Span<T>
{
    readonly ref T _pointer;
    readonly ref int _length;

    public ref T this[int index] => ...
}
```

[Span design document](#)

What is Span<T>?

“System.Span<T> is a new value type at the heart of .NET [that] enables the representation of **contiguous regions of arbitrary memory.**”

- Native/unmanaged memory (P/Invoke)
- Managed memory (.NET types)
- Stack memory (stackalloc)

```
Span<byte> stackMemory = stackalloc byte[256]; // C# 7.2
```

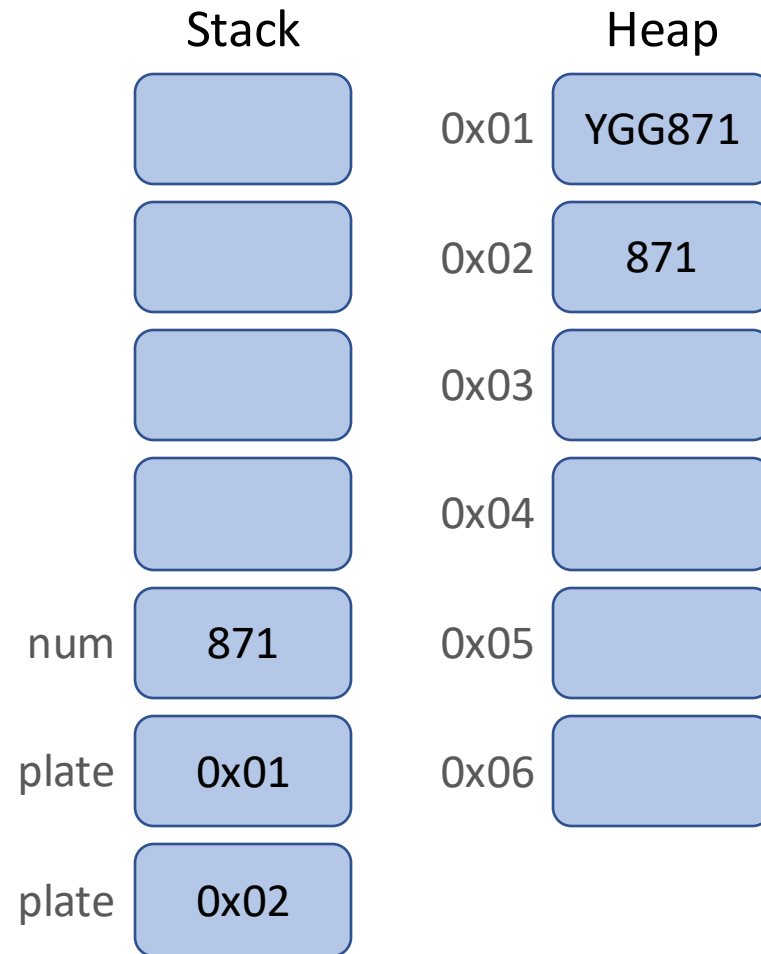
```
IntPtr unmanagedHandle = Marshal.AllocHGlobal(256);  
Span<byte> unmanaged = new Span<byte>(unmanagedHandle.ToPointer(), 256);
```

```
char[] array = new char[] { 'i', 'm', 'p', 'l', 'i', 'c', 'i', 't' };  
Span<char> fromArray = array; // implicit cast
```

```
ReadOnlySpan<char> fromString = "Spanification".AsSpan();
```

What is Span<T>?

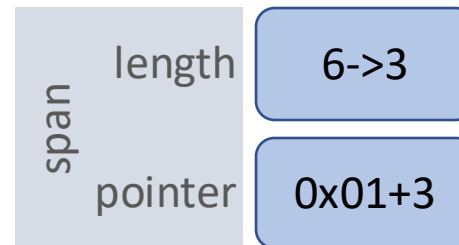
```
string plate = "YGG871";  
plate = plate.Substring(3);  
int num = int.Parse(plate);
```



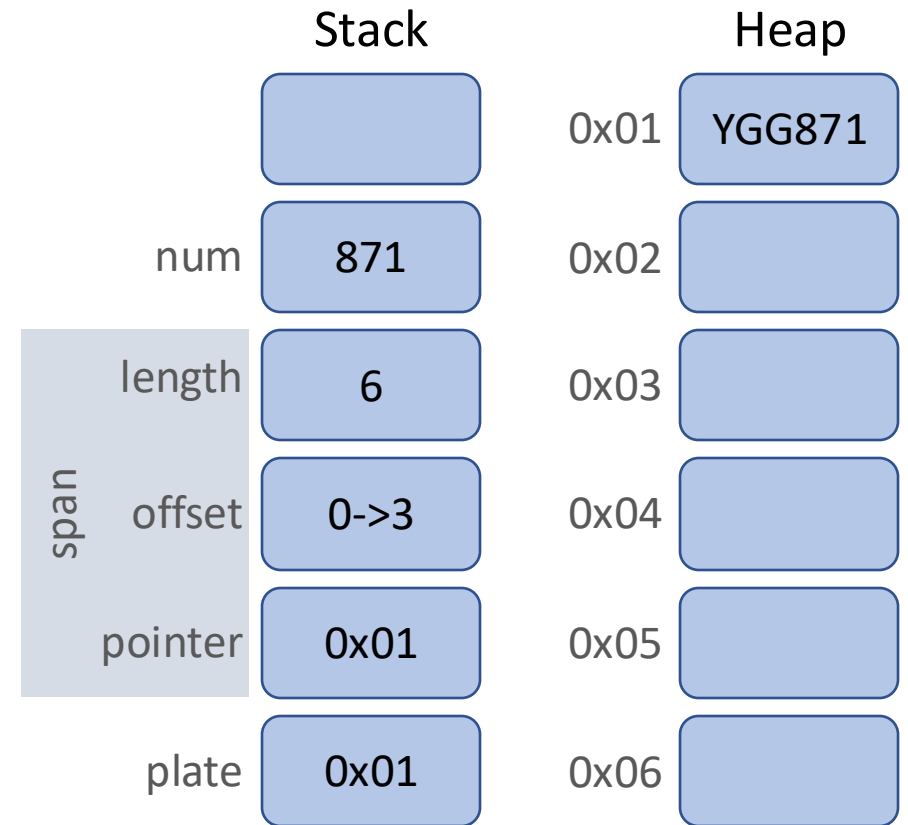
What is Span<T>?

```
string plate = "YGG871";  
ReadOnlySpan<char> s = plate.AsSpan();  
s = s.Slice(3);  
int num = int.Parse(s);
```

```
readonly ref struct Span<T>  
{  
    readonly ref T _pointer;  
    readonly int _length;  
  
    public ref T this[int index] => ...  
}
```



.NET Core 2.0+ – fast span



.NET Framework 4.5+ - slow span

* Legends *

ItemCount : Value of the 'ItemCount' parameter
Mean : Arithmetic mean of all measurements
Error : Half of 99.9% confidence interval
StdDev : Standard deviation of all measurements
Ratio : Mean of the ratio distribution ([Current]/[Baseline])
RatioSD : Standard deviation of the ratio distribution ([Current]/[Baseline])
Rank : Relative position of current benchmark mean among all benchmarks (Arabic style)
Gen 0 : GC Generation 0 collects per 1000 operations
Gen 1 : GC Generation 1 collects per 1000 operations
Gen 2 : GC Generation 2 collects per 1000 operations
Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
1 ms : 1 Millisecond (0.001 sec)



DEMO

Span<T>



BenchmarkDotNet
Powerful .NET library for benchmarking

Benchmark

BenchmarkDotNet v0.14.0, Windows 11 (10.0.22621.4169/22H2/2022Update/SunValley2)

Intel Core i9-9880H CPU 2.30GHz, 1 CPU, 8 logical and 8 physical cores

.NET SDK 8.0.400

[Host] : .NET 8.0.8 (8.0.824.36612), X64 RyuJIT AVX2

DefaultJob : .NET 8.0.8 (8.0.824.36612), X64 RyuJIT AVX2

Method	Mean	Error	StdDev	Ratio	Rank	Gen0	Allocated	Alloc Ratio
GetLastNameWithSpan	2.987 ns	0.0673 ns	0.0630 ns	0.03	1	-	-	0.00
GetLastNameUsingSubstring	21.156 ns	0.3696 ns	0.3276 ns	0.19	2	0.0048	40 B	0.28
GetLastName	109.893 ns	1.0109 ns	0.8442 ns	1.00	3	0.0172	144 B	1.00

Method	CharactersCount	Mean	Error	StdDev	Ratio	RatioSD	Rank	Gen0	Allocated	Alloc Ratio
Slice	10	0.3635 ns	0.0366 ns	0.0342 ns	1.01	0.13	1	-	-	NA
Substring	10	8.3607 ns	0.2053 ns	0.2364 ns	23.20	2.25	2	0.0038	32 B	NA
Slice	10000	0.4554 ns	0.0391 ns	0.0495 ns	1.01	0.18	1	-	-	NA
Substring	10000	555.2479 ns	14.1533 ns	41.7313 ns	1,237.51	195.50	2	1.1969	10024 B	NA

Method	ItemCount	Mean	Error	StdDev	Median	Ratio	RatioSD	Rank	Gen0	Gen1	Gen2	Allocated	Alloc Ratio
StringParseSum	10	NA	NA	NA	NA	?	?	?	NA	NA	NA	NA	?
StringParseSumWithSpan	10	NA	NA	NA	NA	?	?	?	NA	NA	NA	NA	?
StringParseSumWithSpan	100000	1.957 ms	0.0742 ms	0.2081 ms	1.884 ms	0.21	0.03	1	-	-	-	3 B	0.000
StringParseSum	100000	9.265 ms	0.2344 ms	0.6134 ms	9.328 ms	1.00	0.10	2	562.5000	546.8750	187.5000	4000133 B	1.000

When do mere mortals use it?

```
var str = "EGOR 3.14 1234 7/3/2018";
```

Allocated on heap: 168 bytes

```
string name = str.Substring(0, 4);  
float pi = float.Parse(str.Substring(5, 4));  
int number = int.Parse(str.Substring(10, 4));  
DateTime date = DateTime.Parse(str.Substring(15, 8));
```

```
var str = "EGOR 3.14 1234 7/3/2018".AsSpan();
```

Allocated on heap: 0 bytes

```
string name = str.Slice(0, 4);  
float pi = float.Parse(str.Slice(5, 4));  
int number = int.Parse(str.Slice(10, 4));  
DateTime date = DateTime.Parse(str.Slice(15, 8));
```

Span - The Limitations

- Can only live on the stack
- Implemented as a ref struct
- Can only be contained by ref structs

ref structs

Structs that can exist only on the stack. New in C# 7.2

- Can't implement interfaces
- Can't be used as generic type arguments
- Can't be boxed to object
- Can't be passed in to - or used in places inside - of async methods, iterators, nested functions or query expressions

Memory<T>

- “Normal” struct
- Not as performant as Span
- Can be used in more places than Span (ie, doesn’t have the limitations)
- Use the Span property to access the underlying memory

```
readonly struct Memory<T>
{
    readonly object _object;
    readonly int _index;
    readonly int _length;

    public Span<T> Span => ...
    public Memory<T>(T[] array, int index, int length) => ...
}
```


Memory<T>

```
async Task DoSomethingAsync(Span<byte> buffer) // <-- error
{
    buffer[0] = 0;
    await Something(); // Bang!
    buffer[0] = 1;
}
```

```
async Task DoSomethingAsync(Memory<byte> buffer)
{
    buffer.Span[0] = 0;
    await Something(); // Totally fine
    buffer.Span[0] = 1;
}
```



DEMO

Memory<T>

System.Buffers Namespace

Version

.NET 8

Search

System.Buffers

- > [ArrayBufferWriter<T>](#)
- > [ArrayPool<T>](#)
- > [BuffersExtensions](#)
- > [IBufferWriter<T>](#)
- > [IMemoryOwner<T>](#)
- > [IPinnable](#)
- > [MemoryHandle](#)
- > [MemoryManager<T>](#)
- > [MemoryPool<T>](#)
- > [NIndex](#)
- > [NRange](#)
- > [OperationStatus](#)
- > [ReadOnlySequence<T>.Enumerator](#)
- > [ReadOnlySequence<T>](#)
- > [ReadOnlySequenceSegment<T>](#)
- > [ReadOnlySpanAction<T,TArg>](#)
- > [SearchValues](#)
- > [SearchValues<T>](#)
- > [SequenceReader<T>](#)
- > [SequenceReaderExtensions](#)
- > [SpanAction<T,TArg>](#)
- > [StandardFormat](#)

Download PDF

System.Buffers Namespace

Reference

[Feedback](#)

Contains types used in creating and managing memory buffers, such as those represented by [Span<T>](#) and [Memory<T>](#).

Classes

[Expand table](#)

ArrayBufferWriter<T>	Represents a heap-based, array-backed output sink into which T data can be written.
ArrayPool<T>	Provides a resource pool that enables reusing instances of type T [].
BuffersExtensions	Provides extension methods for ReadOnlySequence<T> .
MemoryManager<T>	An abstract base class that is used to replace the implementation of Memory<T> .
MemoryPool<T>	Represents a pool of memory blocks.
ReadOnlySequenceSegment<T>	Represents a linked list of ReadOnlyMemory<T> nodes.
SearchValues	Provides a set of initialization methods for instances of the SearchValues<T> class.
SearchValues<T>	Provides an immutable, read-only set of values optimized for efficient searching. Instances are created by Create(ReadOnlySpan<Byte>) or Create(ReadOnlySpan<Char>) .
SequenceReaderExtensions	Provides extended functionality for the SequenceReader<T> class that allows reading of endian specific numeric values from binary data.

Structs

[Expand table](#)

MemoryHandle	Provides a memory handle for a block of memory.
NIndex	Represent a type can be used to index a collection either from the start or the end.
NRange	Represent a range that has start and end indices.
ReadOnly	Represents an enumerator over a ReadOnlySequence<T> .

Recap

- `Span<T>` makes it easy and safe to use any kind of memory
- `System.Memory` package, C# ≥ 7.2
- `Memory<T>` has no stack-only limitations
- Don't copy memory! Slice it!
- Prefer read-only versions over mutable ones
- Prefer safe managed memory over native memory
- `Span<T>/ReadOnlySpan<T>` for Synchronous Methods
- `Memory<T>/ReadOnlyMemory<T>` for Asynchronous Methods

Performance is a Feature!