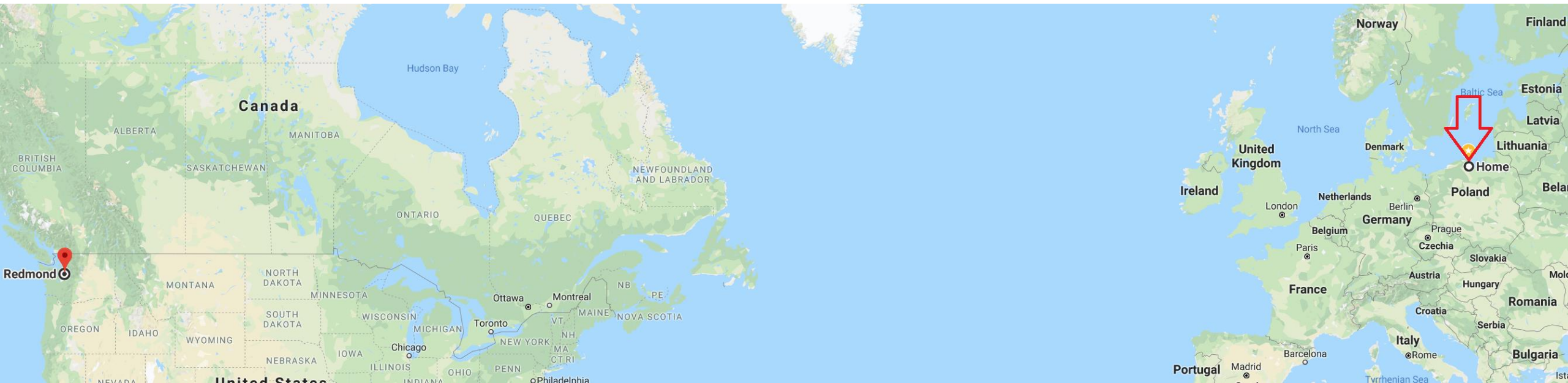


Spanification

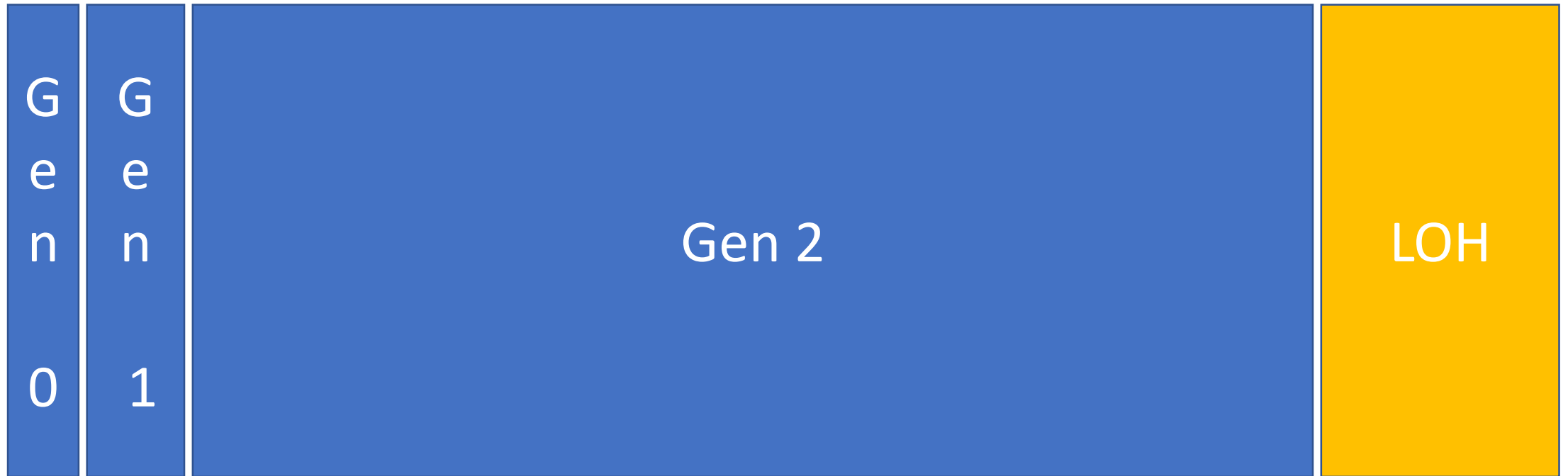
Adam Sitnik

About Myself

- Open Source Contributor
- BenchmarkDotNet maintainer
- Performance Champion on the .NET Team at Microsoft



Managed Heap: Workstation mode



LOH: allocation: find a free segment in the list



Legend: free taken

SOH: allocation: $pNext += requestedSize;$

Pointer to the next object



Legend:

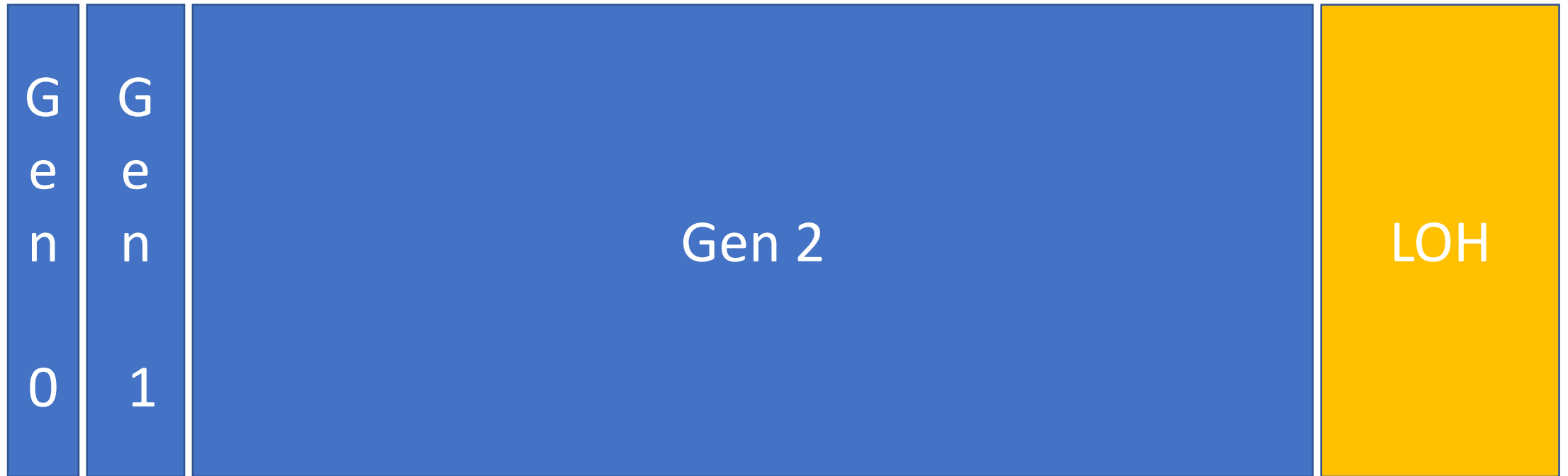
free

taken

Garbage Collection

- 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

Cleanup: LOH = GEN 2 = FULL GC



~~LOH = GEN 2 = FULL GC~~

Managed Heap: Server mode

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](#)

Unmanaged heap

- Allocate and Free on demand

```
IntPtr pointer = Marshal.AllocHGlobal(bytesCount);  
  
try  
{  
    Consume(pointer, bytesCount);  
}  
finally  
{  
    Marshal.FreeHGlobal(pointer);  
}
```

- Must not store managed references, just values
- It's developer responsibility to free the memory
- Possible fragmentation issues

Stack

- Allocated on demand, freed with stack unwind

```
public unsafe void Stack(int bytesCount)
{
    byte* pointer = stackalloc byte[bytesCount];

    Consume(pointer, bytesCount);
} // the method ends, the stack unwinds
```

- Until recently*, only values

What is the output?

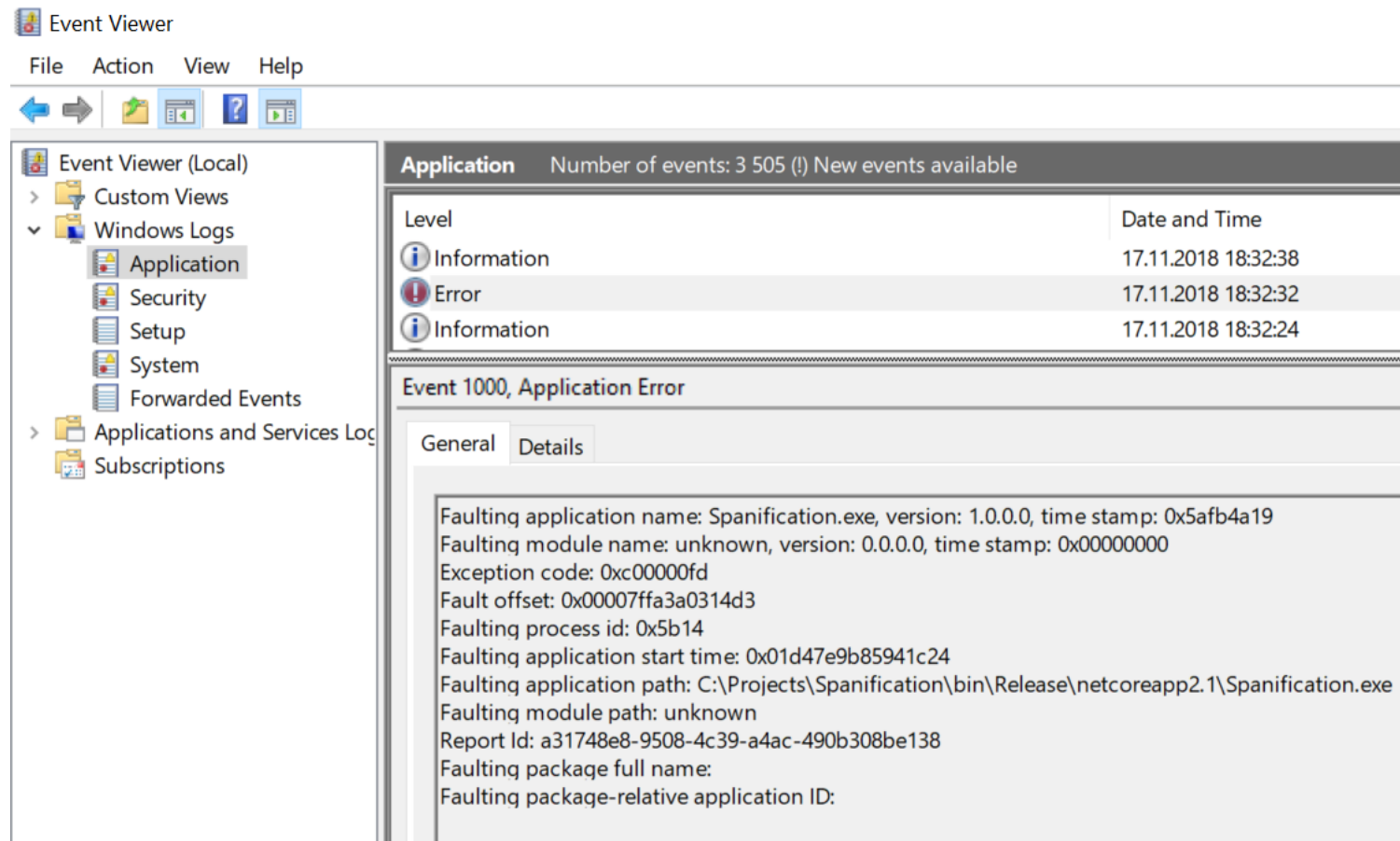
```
try
{
    byte* pointer = stackalloc byte[1_000_000 * 2]; // 2 MB

    Consume(pointer, 1_000_000 * 2);

    Console.WriteLine("OK");
}
catch (Exception e)
{
    Console.WriteLine("EXCEPTION");
}
```

```
PS C:\Projects\Spanification> dotnet run -c Release -f netcoreapp2.1
Process is terminating due to StackOverflowException.
```

Where can I find information about Unhandled Exceptions?



Summary: Available Memory

	Managed heap	Unmanaged heap	Stack
Safe	Yes	No type safety	Type safe, but can be deadly
What can be allocated	References and values	Values only	Values only*
Allocation	Cheap	Cheap	Cheap
Deallocation	Comes at a 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)
```

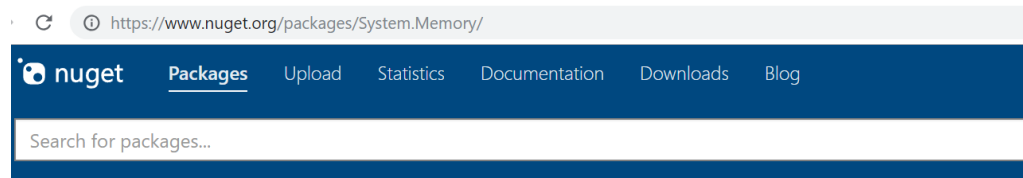
```
unsafe int Parse(char* pointer, int length)
```


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

Span<T>

- It provides a uniform API for working with:
 - Unmanaged memory buffers
 - Arrays and subarrays
 - Strings and substrings
- It's fully **type-safe** and **memory-safe**
- Almost no overhead
- It's a **read only** and **stack only Value Type**

System.Memory NuGet package





System.Memory 4.5.1

Provides types for efficient representation and pooling of managed, stack, and native memory segments and sequences of such segments, along with primitives to parse and format UTF-8 encoded text stored in those memory segments.

Commonly Used Types:

- System.Span
- System.ReadOnlySpan
- System.Memory
- System.ReadOnlyMemory
- System.Buffers.MemoryPool
- System.Buffers.ReadOnlySequence
- System.Buffers.Text.Utf8Parser
- System.Buffers.Text.Utf8Formatter

7ee84596d92e178bce54c986df31ccc52479e772

When using NuGet 3.x this package requires at least version 3.4.

Requires NuGet 2.12 or higher.

Package Manager

.NET CLI

Paket CLI

```
> dotnet add package System.Memory --version 4.5.1
```

Dependencies

[.NETCoreApp 2.0](#)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[.NETCoreApp 2.1](#)

No dependencies.

[.NETFramework 4.5](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[.NETFramework 4.6.1](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Numerics.Vectors](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[.NETStandard 1.1](#)

[NETStandard.Library](#) ($\geq 1.6.1$)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[.NETStandard 2.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Numerics.Vectors](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[MonoAndroid 1.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[MonoTouch 1.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Portable Class Library \(.NETFramework 4.5, Windows 8.0, WindowsPhoneApp 8.1\)](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[UAP 10.0.16299](#)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Windows 8.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[WindowsPhoneApp 8.1](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Xamarin.iOS 1.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Xamarin.Mac 2.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Xamarin.TVOS 1.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

[Xamarin.WatchOS 1.0](#)

[System.Buffers](#) ($\geq 4.4.0$)

[System.Runtime.CompilerServices.Unsafe](#) ($\geq 4.5.0$)

C# 7.2

```
Span<byte> stackMemory = stackalloc byte[256];
```

Error CS8107 Feature 'ref structs' is not available in C# 7.0. Please use language version 7.2 or greater.

```
<PropertyGroup>  
  <LangVersion>7.2</LangVersion>  
</PropertyGroup>
```

Supports **any** memory

```
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();
```

Span<T> API

```
public int Length { get; }  
public ref T this[int index] { get; set; }  
  
public Span<T> Slice(int start);  
public Span<T> Slice(int start, int length);  
  
public void Clear();  
public void Fill(T value);  
  
public void CopyTo(Span<T> destination);  
public bool TryCopyTo(Span<T> destination);
```

ReadOnlySpan<T> API

```
public int Length { get; }
```

```
public readonly ref T this[int index] { get; }
```

```
public ReadOnlySpan<T> Slice(int start);
```

```
public ReadOnlySpan<T> Slice(int start, int length);
```

```
public void CopyTo(Span<T> destination);
```

```
public bool TryCopyTo(Span<T> destination);
```

https://apisof.net

← → ↻ 🔒 https://apisof.net/catalog/System.ReadOnlySpan<T>.Item[Int32] ☆

.NET API Catalog Search

📖 Catalog / {} System / 📖 ReadOnlySpan<T> / 📖 Item[Int32]

📖 ReadOnlySpan<T>

📖 Enumerator

🔍 .ctor(T[])

🔍 .ctor(T[], Int32, Int32)

🔍 .ctor(Void*, Int32)

🔍 Empty

🔍 IsEmpty

🔍 Item[Index]

🔍 Item[Int32]

🔍 Item[Range]

🔍 Length

🔍 CopyTo(Span<T>)

🔍 Equals(Object)

🔍 GetEnumerator()

🔍 GetHashCode()

🔍 GetPinnableReference()

🔍 op_Equality(ReadOnlySpan<T>, ReadOnlySpan<T>)

🔍 op_Implicit(ArraySegment<T>)

🔍 op_Implicit(T[])

🔍 op_Inequality(ReadOnlySpan<T>, ReadOnlySpan<T>)

🔍 Slice(Int32)

🔍 Slice(Int32, Int32)

🔍 ToArray()

🔍 ToString()

🔍 TryCopyTo(Span<T>)

Item[Int32] Property

```
// .NET Standard 2.1
// netstandard, Version=2.0.0.0, PublicKeyToken=cc7b13ffcd2ddd51

namespace System
{
    public readonly ref struct ReadOnlySpan<T>
    {
        public ref readonly T this[int index] { get; }
    }
}
```

Usage

% apps

API Port Telemetry

nuget.org

Metadata

ID

P:System.ReadOnlySpan`1.Item(System.Int32)

Declarations

Platform	Version	Assembly
.NET Core	2.1	System.Runtime, Version=4.2.1.0, PublicKeyToken=b03f5f7f11d50a3a
	3.0	System.Runtime, Version=4.2.1.0, PublicKeyToken=b03f5f7f11d50a3a
.NET Core + Platform Extensions	2.1	System.Runtime, Version=4.2.1.0, PublicKeyToken=b03f5f7f11d50a3a
.NET Standard	2.1	netstandard, Version=2.0.0.0, PublicKeyToken=cc7b13ffcd2ddd51

API Simplicity

~~int Parse(string text)~~

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

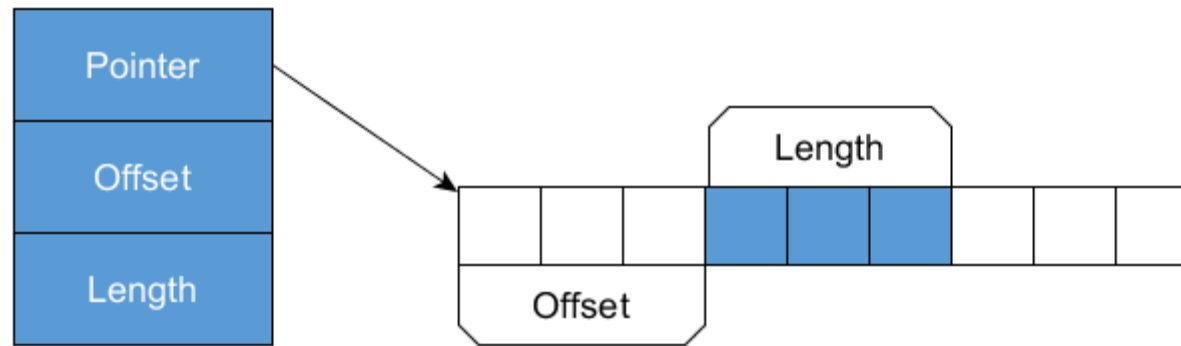
~~unsafe int Parse(char* pointer, int length)~~

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

int Parse(Span<char> input)

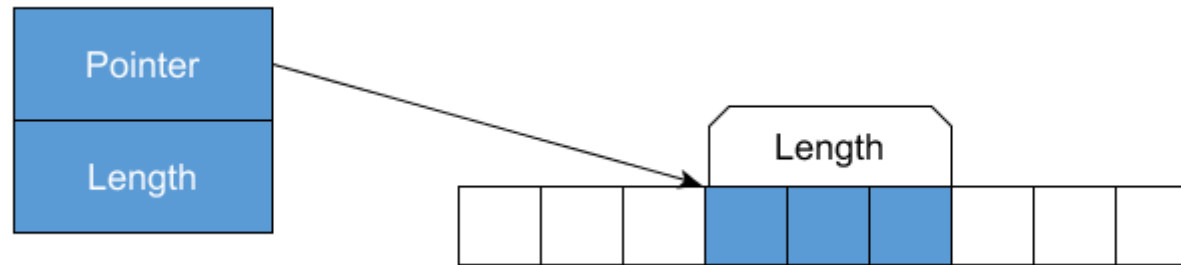
Span for existing runtimes

.NET Standard 1.0 (.NET 4.5+)

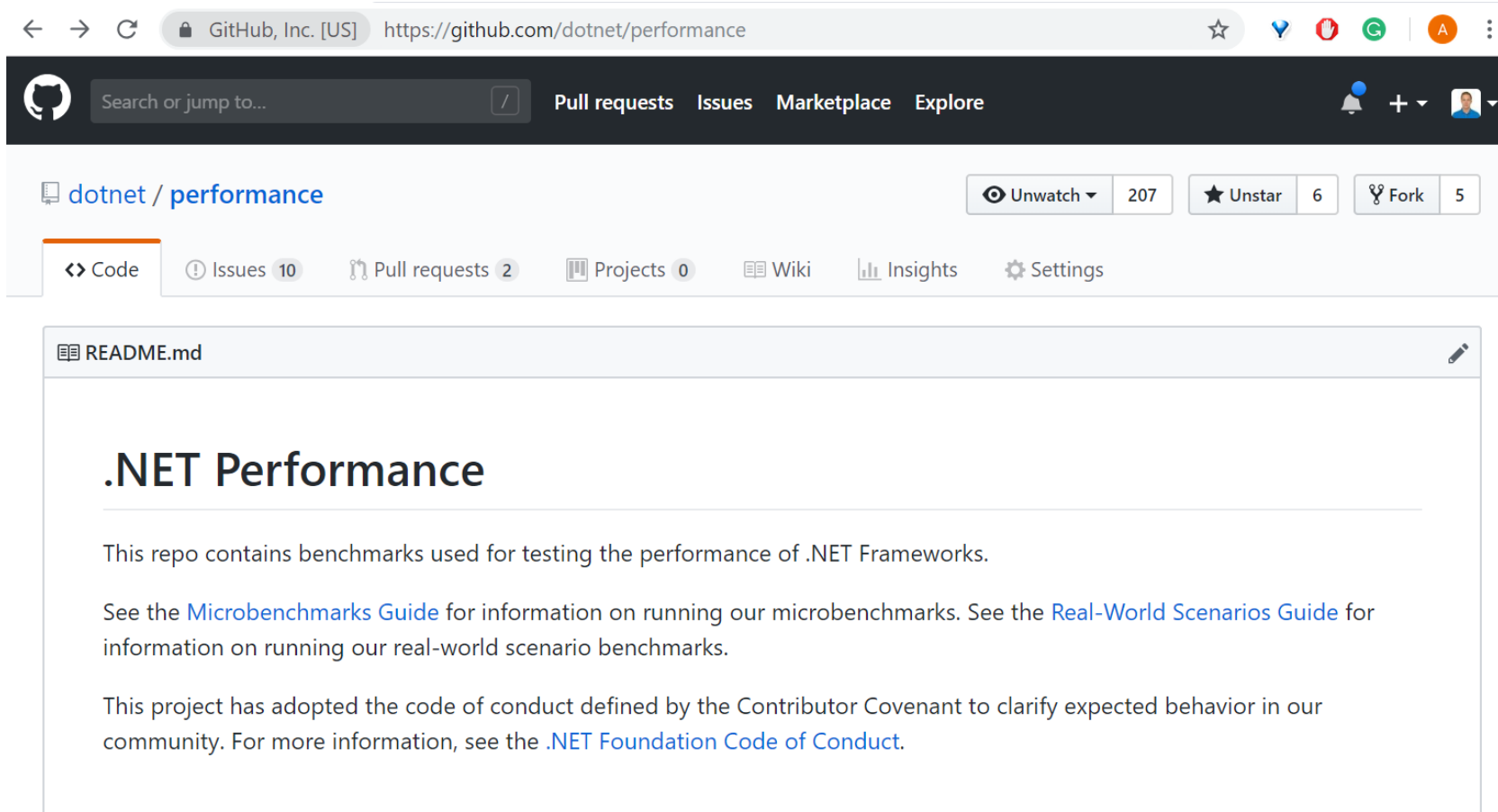


Span for new runtimes

.NET Core 2.0 and any other runtime supporting by-ref fields



<https://github.com/dotnet/performance>



The screenshot shows the GitHub repository page for `dotnet/performance`. The browser address bar displays the URL `https://github.com/dotnet/performance`. The GitHub navigation bar includes a search bar, links for Pull requests, Issues, Marketplace, and Explore, and a user profile icon. The repository header shows the name `dotnet / performance`, a watch button (Unwatch) with 207 watchers, an unstar button (Unstar) with 6 stars, and a fork button (Fork) with 5 forks. Below the header, a tab bar shows the repository is currently viewed as code, with other tabs for Issues (10), Pull requests (2), Projects (0), Wiki, Insights, and Settings. The main content area displays the README.md file, which has the title `.NET Performance`. The README text states: "This repo contains benchmarks used for testing the performance of .NET Frameworks." It also provides links to the [Microbenchmarks Guide](#) and the [Real-World Scenarios Guide](#). Finally, it mentions the project's adoption of the Contributor Covenant code of conduct, with a link to the [.NET Foundation Code of Conduct](#).

← → ↻ GitHub, Inc. [US] https://github.com/dotnet/performance ☆ 🔍 📄 📄 📄 📄

Search or jump to... / Pull requests Issues Marketplace Explore 🔔 + 👤

`dotnet / performance` 🔍 Unwatch 207 ★ Unstar 6 🍴 Fork 5

<> Code ⓘ Issues 10 🔗 Pull requests 2 📁 Projects 0 📖 Wiki 📊 Insights ⚙ Settings

📖 README.md ✎

.NET Performance

This repo contains benchmarks used for testing the performance of .NET Frameworks.

See the [Microbenchmarks Guide](#) for information on running our microbenchmarks. See the [Real-World Scenarios Guide](#) for information on running our real-world scenario benchmarks.

This project has adopted the code of conduct defined by the Contributor Covenant to clarify expected behavior in our community. For more information, see the [.NET Foundation Code of Conduct](#).

--list flat --allCategories Span

```
Span.IndexerBench.Ref
Span.IndexerBench.Fixed1
Span.IndexerBench.Fixed2
Span.IndexerBench.Indexer1
Span.IndexerBench.Indexer2
Span.IndexerBench.Indexer3
Span.IndexerBench.Indexer4
Span.IndexerBench.Indexer5
Span.IndexerBench.Indexer6
Span.IndexerBench.ReadOnlyIndexer1
Span.IndexerBench.ReadOnlyIndexer2
Span.IndexerBench.WriteVialIndexer1
Span.IndexerBench.WriteVialIndexer2
Span.IndexerBench.KnownSizeArray
Span.IndexerBench.KnownSizeCtor
Span.IndexerBench.KnownSizeCtor2
Span.IndexerBench.SameIndex1
Span.IndexerBench.SameIndex2
Span.IndexerBench.CoveredIndex1
Span.IndexerBench.CoveredIndex2
Span.IndexerBench.CoveredIndex3
Span.Sorting.QuickSortSpan
Span.Sorting.BubbleSortSpan
System.Collections.Clear<Int32>.Span
System.Collections.Clear<String>.Span
System.Collections.CopyTo<Int32>.Span
System.Collections.CopyTo<Int32>.ReadOnlySpan
System.Collections.CopyTo<Int32>.Memory
System.Collections.CopyTo<Int32>.ReadOnlyMemory
System.Collections.CopyTo<String>.Span
System.Collections.CopyTo<String>.ReadOnlySpan
System.Collections.CopyTo<String>.Memory
System.Collections.CopyTo<String>.ReadOnlyMemory
System.Collections.IndexerSet<Int32>.Span
System.Collections.IndexerSet<String>.Span
System.Collections.IndexerSetReverse<Int32>.Span
System.Collections.IndexerSetReverse<String>.Span
System.Collections.IterateFor<Int32>.Span
System.Collections.IterateFor<Int32>.ReadOnlySpan
System.Collections.IterateFor<String>.Span
System.Collections.IterateFor<String>.ReadOnlySpan
System.Collections.IterateForEach<Int32>.Span
System.Collections.IterateForEach<Int32>.ReadOnlySpan
System.Collections.IterateForEach<String>.Span
System.Collections.IterateForEach<String>.ReadOnlySpan
System.Memory.Constructors._ValueTypesOnly<Byte>.SpanFromPointerLength
System.Memory.Constructors._ValueTypesOnly<Byte>.ReadOnlyFromPointerLength
System.Memory.Constructors._ValueTypesOnly<Int32>.SpanFromPointerLength
System.Memory.Constructors._ValueTypesOnly<Int32>.ReadOnlyFromPointerLength
System.Memory.Constructors<Byte>.SpanFromArray
System.Memory.Constructors<Byte>.ReadOnlySpanFromArray
System.Memory.Constructors<Byte>.SpanFromArrayStartLength
System.Memory.Constructors<Byte>.ReadOnlySpanFromArrayStartLength
System.Memory.Constructors<Byte>.SpanFromMemory
System.Memory.Constructors<Byte>.ReadOnlySpanFromMemory
System.Memory.Constructors<Byte>.ReadOnlySpanFromMemory
System.Memory.Constructors<Byte>.SpanImplicitCastFromArray
System.Memory.Constructors<Byte>.ReadOnlySpanImplicitCastFromArray
System.Memory.Constructors<Byte>.SpanImplicitCastFromArraySegment
System.Memory.Constructors<Byte>.ReadOnlySpanImplicitCastFromArraySegment
```

```
System.Memory.Constructors<Byte>.ReadOnlySpanImplicitCastFromArraySegment
System.Memory.Constructors<Byte>.ReadOnlySpanImplicitCastFromSpan
System.Memory.Constructors<Byte>.MemoryFromArray
System.Memory.Constructors<Byte>.ReadOnlyMemoryFromArray
System.Memory.Constructors<Byte>.MemoryFromArrayStartLength
System.Memory.Constructors<Byte>.ReadOnlyMemoryFromArrayStartLength
System.Memory.Constructors<Byte>.MemoryMarshalCreateSpan
System.Memory.Constructors<Byte>.MemoryMarshalCreateReadOnlySpan
System.Memory.Constructors<String>.SpanFromArray
System.Memory.Constructors<String>.ReadOnlySpanFromArray
System.Memory.Constructors<String>.SpanFromArrayStartLength
System.Memory.Constructors<String>.ReadOnlySpanFromArrayStartLength
System.Memory.Constructors<String>.SpanFromMemory
System.Memory.Constructors<String>.ReadOnlySpanFromMemory
System.Memory.Constructors<String>.SpanImplicitCastFromArray
System.Memory.Constructors<String>.ReadOnlySpanImplicitCastFromArray
System.Memory.Constructors<String>.SpanImplicitCastFromArraySegment
System.Memory.Constructors<String>.ReadOnlySpanImplicitCastFromArraySegment
System.Memory.Constructors<String>.ReadOnlySpanImplicitCastFromSpan
System.Memory.Constructors<String>.MemoryFromArray
System.Memory.Constructors<String>.ReadOnlyMemoryFromArray
System.Memory.Constructors<String>.MemoryFromArrayStartLength
System.Memory.Constructors<String>.ReadOnlyMemoryFromArrayStartLength
System.Memory.Constructors<String>.MemoryMarshalCreateSpan
System.Memory.Constructors<String>.MemoryMarshalCreateReadOnlySpan
System.Memory.Memory<Byte>.Pin
System.Memory.Memory<Byte>.ToArray
System.Memory.Memory<Char>.Pin
System.Memory.Memory<Char>.ToArray
System.Memory.MemoryMarshal<Byte>.GetReference
System.Memory.MemoryMarshal<Byte>.AsBytes
System.Memory.MemoryMarshal<Byte>.CastToByte
System.Memory.MemoryMarshal<Byte>.CastToInt
System.Memory.MemoryMarshal<Byte>.TryGetArray
System.Memory.MemoryMarshal<Byte>.Read
System.Memory.MemoryMarshal<Int32>.GetReference
System.Memory.MemoryMarshal<Int32>.AsBytes
System.Memory.MemoryMarshal<Int32>.CastToByte
System.Memory.MemoryMarshal<Int32>.CastToInt
System.Memory.MemoryMarshal<Int32>.TryGetArray
System.Memory.MemoryMarshal<Int32>.Read
System.Memory.ReadOnlyMemory<Byte>.Pin
System.Memory.ReadOnlyMemory<Byte>.ToArray
System.Memory.ReadOnlyMemory<Char>.Pin
System.Memory.ReadOnlyMemory<Char>.ToArray
System.Memory.ReadOnlySpan.StringAsSpan
System.Memory.ReadOnlySpan.GetPinnableReference
System.Memory.ReadOnlySpan.IndexOfString
System.Memory.Slice<Byte>.SpanStart
System.Memory.Slice<Byte>.SpanStartLength
System.Memory.Slice<Byte>.ReadOnlySpanStart
System.Memory.Slice<Byte>.ReadOnlySpanStartLength
System.Memory.Slice<Byte>.MemoryStart
System.Memory.Slice<Byte>.MemoryStartLength
System.Memory.Slice<Byte>.ReadOnlyMemoryStart
System.Memory.Slice<Byte>.ReadOnlyMemoryStartLength
System.Memory.Slice<String>.SpanStart
System.Memory.Slice<String>.SpanStartLength
```

```
System.Memory.Slice<String>.ReadOnlySpanStart
System.Memory.Slice<String>.ReadOnlySpanStartLength
System.Memory.Slice<String>.MemoryStart
System.Memory.Slice<String>.MemoryStartLength
System.Memory.Slice<String>.ReadOnlyMemoryStart
System.Memory.Slice<String>.ReadOnlyMemoryStartLength
System.Memory.Span<Byte>.Clear
System.Memory.Span<Byte>.Fill
System.Memory.Span<Byte>.Reverse
System.Memory.Span<Byte>.ToArray
System.Memory.Span<Byte>.SequenceEqual
System.Memory.Span<Byte>.SequenceCompareTo
System.Memory.Span<Byte>.StartsWith
System.Memory.Span<Byte>.EndsWith
System.Memory.Span<Byte>.IndexOfValue
System.Memory.Span<Byte>.LastIndexOfValue
System.Memory.Span<Byte>.LastIndexOfAnyValues
System.Memory.Span<Byte>.BinarySearch
System.Memory.Span<Byte>.GetPinnableReference
System.Memory.Span<Char>.Clear
System.Memory.Span<Char>.Fill
System.Memory.Span<Char>.Reverse
System.Memory.Span<Char>.ToArray
System.Memory.Span<Char>.SequenceEqual
System.Memory.Span<Char>.SequenceCompareTo
System.Memory.Span<Char>.StartsWith
System.Memory.Span<Char>.EndsWith
System.Memory.Span<Char>.IndexOfValue
System.Memory.Span<Char>.LastIndexOfValue
System.Memory.Span<Char>.LastIndexOfAnyValues
System.Memory.Span<Char>.BinarySearch
System.Memory.Span<Char>.GetPinnableReference
System.Memory.Span<Int32>.Clear
System.Memory.Span<Int32>.Fill
System.Memory.Span<Int32>.Reverse
System.Memory.Span<Int32>.ToArray
System.Memory.Span<Int32>.SequenceEqual
System.Memory.Span<Int32>.SequenceCompareTo
System.Memory.Span<Int32>.StartsWith
System.Memory.Span<Int32>.EndsWith
System.Memory.Span<Int32>.IndexOfValue
System.Memory.Span<Int32>.LastIndexOfValue
System.Memory.Span<Int32>.LastIndexOfAnyValues
System.Memory.Span<Int32>.BinarySearch
System.Memory.Span<Int32>.GetPinnableReference
```

„Fast” vs „Slow” Span

dotnet run -c Release -f netcoreapp2.1 --filter System.Memory.Span<Char>.Reverse --runtimes net472 netcoreapp2.1

Method	Runtime	Size	Mean	Ratio	Allocated Memory/Op
Reverse	Clr	512	217.8 ns	1.00	-
Reverse	Core	512	206.4 ns	0.95	-

dotnet run -c Release -f netcoreapp2.1 --filter System.Memory.Span<Char>.Clear --runtimes net472 netcoreapp2.1

Method	Runtime	Size	Mean	Ratio	Allocated Memory/Op
Clear	Clr	512	41.16 ns	1.00	-
Clear	Core	512	17.00 ns	0.41	-

Creating substrings **before Span** (pseudocode)

```
string Substring(string text, int startIndex, int length)
{
    string result = new string(length); // ALLOCATION!

    Memory.Copy(text, result, startIndex, length); // COPYING

    return result;
}
```

Creating substrings **without allocation!** (pseudocode)

```
ReadOnlySpan<char> Slice(string text, int startIndex, int length)
=> new ReadOnlySpan<char>(
    ref text[0] + (startIndex * sizeof(char)),
    length);
```

Substring vs Slice: Benchmark

```
public class Slicing
{
    public IEnumerable<object> Arguments()
    {
        yield return "Substring vs Slice";
        yield return string.Join(", ", Enumerable.Repeat("Substring vs Slice", 1000));
    }

    [Benchmark(Baseline = true)]
    [ArgumentsSource(nameof(Arguments))]
    public string Substring(string text) => text.Substring(startIndex: text.Length / 2);

    [Benchmark]
    [ArgumentsSource(nameof(Arguments))]
    public ReadOnlySpan<char> Slice(string text) => text.AsSpan().Slice(start: text.Length / 2);
}
```

Substring vs Slice: Benchmark results

Method	text	Mean	StdDev	Ratio	Gen 0 /1k Op	Gen 1 /1k Op	Allocated Memory
Substring	Substring vs Slice	10.860 ns	0.0981 ns	1.00	0.0076	-	48 B
Slice	Substring vs Slice	1.151 ns	0.0151 ns	0.11	-	-	-

Substring vs Slice: Benchmark results

Method	text	Mean	StdDev	Ratio	Gen 0 /1k Op	Gen 1 /1k Op	Allocated Memory
Substring	Substring vs Slice	10.860 ns	0.0981 ns	1.00	0.0076	-	48 B
Slice	Substring vs Slice	1.151 ns	0.0151 ns	0.11	-	-	-
Substring	Substring vs Slice x1000	1,703.534 ns	8.8209 ns	1.000	3.1815	0.1984	20024 B
Slice	Substring vs Slice x1000	1.145 ns	0.0135 ns	0.001	-	-	-

Slice is $O(1)$

Method	text	Mean	StdDev	Ratio	Gen 0 /1k Op	Gen 1 /1k Op	Allocated Memory
Substring	Substring vs Slice	10.860 ns	0.0981 ns	1.00	0.0076	-	48 B
Slice	Substring vs Slice	1.151 ns	0.0151 ns	0.11	-	-	-
Substring	Substring vs Slice x1000	1,703.534 ns	8.8209 ns	1.000	3.1815	0.1984	20024 B
Slice	Substring vs Slice x1000	1.145 ns	0.0135 ns	0.001	-	-	-

Summary: Span basics

- `Span<T>` allows working with any kind of memory
- `ReadOnlySpan<T>` allows for read-only access
- Memory and type safe
- `System.Memory` package, C# 7.2
- .NET Standard implementation is fast, .NET Core is very fast
- Slice is $O(1)$

Let's parse a Utf8 line of floats..

```
[Benchmark(Baseline = true)]
public float OldWay()
{
    float result = 0;

    string line = Encoding.UTF8.GetString(utf8line);

    string[] splitted = line.Split(' ', '\t');

    foreach (string toParse in splitted)
        if (float.TryParse(toParse, out float parsed))
            result += parsed;

    return result;
}
```

Let's Spanify it!

```
[Benchmark]
public float NewWay()
{
    float result = 0;

    ReadOnlySpan<byte> toParse = new ReadOnlySpan<byte>(utf8line);
    while (!toParse.IsEmpty)
    {
        if (!Utf8Parser.TryParse(toParse, out float parsed, out int bytesConsumed))
            break;

        result += parsed;
        toParse = toParse.Slice(start: bytesConsumed + 1); // 1 is for ' '
    }

    return result;
}
```

18% boost, no allocations

Method	Count	Mean	Ratio	Gen 0/1k Op	Allocated Memory
OldWay	100	15.74 us	1.00	3.5706	7496 B
NewWay	100	12.67 us	0.81	-	-
OldWay	1000	153.90 us	1.00	35.1563	74096 B
NewWay	1000	126.33 us	0.82	-	-

Utf8Parser

```
class Utf8Parser
{
    bool TryParse(ReadOnlySpan<byte> source, out bool value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out byte value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out DateTime value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out DateTimeOffset value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out decimal value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out double value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out System.Guid value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out short value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out int value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out long value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out sbyte value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out float value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out TimeSpan value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out ushort value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out uint value, out int bytesConsumed, char standardFormat = '\0')
    bool TryParse(ReadOnlySpan<byte> source, out ulong value, out int bytesConsumed, char standardFormat = '\0')
}
```

Utf8Formatter

```
class Utf8Formatter
{
    bool TryFormat(bool value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(byte value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(DateTime value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(DateTimeOffset value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(decimal value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(double value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(Guid value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(short value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(int value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(long value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(sbyte value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(float value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(TimeSpan value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(ushort value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(uint value, Span<byte> destination, out int bytesWritten, StandardFormat format)
    bool TryFormat(ulong value, Span<byte> destination, out int bytesWritten, StandardFormat format)
}
```

Let's parse 6.5GB ML file line by line

```
[Benchmark(Baseline = true)]
public float OldWay()
{
    float result = 0;

    foreach (var line in File.ReadLines(filePath))
    {
        string[] splitted = line.Split(' ');
        foreach (string toParse in splitted)
            if (float.TryParse(toParse, out float parsed))
                result += parsed;
    }

    return result;
}
```


Let's Spanify it!

```
using (var fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
{
    byte[] buffer = new byte[16000 * 8]; // 128 KB
    int bytesRead = 0;

    while ((bytesRead = fileStream.Read(buffer, 0, buffer.Length)) > 0)
    {
        var slice = new ReadOnlySpan<byte>(buffer, start: 0, length: bytesRead); // bytesRead != buffer.Length

        int newLineLength = 0;
        while ((newLineLength = slice.IndexOf((byte)0x0A)) > 0) // 0x0A = new line
        {
            ReadOnlySpan<byte> utf8Line = slice.Slice(0, newLineLength);

            result += Parse(utf8Line);

            slice = slice.Slice(start: newLineLength + 1);
        }

        fileStream.Seek(-1 * slice.Length, SeekOrigin.Current); // we go back where the last line started
    }
}
```

The difference

Method	Mean	Ratio	Gen 0/Op	Gen 1/Op	Gen 2/1 Op	Allocated Memory
OldWay	138.9 s	1.00	11562	636	1	72 760 971 232 B
NewWay	104.3 s	0.75	-	-	-	128 200 B

72760971232 B - 128200 B = **72.76 GB**

MemoryExtensions

AsMemory
AsSpan
BinarySearch
CompareTo
Contains
CopyTo
EndsWith
Equals
IndexOf
IndexOfAny
IsWhiteSpace
LastIndexOf
LastIndexOfAny

Overlaps
Reverse
SequenceCompareTo
SequenceEqual
StartsWith
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
Trim
TrimEnd
TrimStart

What can be improved here?

```
using (var fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
{
    byte[] buffer = new byte[16000 * 8]; // 128 KB
    int bytesRead = 0;

    while ((bytesRead = fileStream.Read(buffer, 0, buffer.Length)) > 0)
    {
        var slice = new ReadOnlySpan<byte>(buffer, start: 0, length: bytesRead); // bytesRead != buffer.Length

        int newLineLength = 0;
        while ((newLineLength = slice.IndexOf((byte)0x0A)) > 0) // 0x0A = new line
        {
            ReadOnlySpan<byte> utf8Line = slice.Slice(0, newLineLength);

            result += Parse(utf8Line);

            slice = slice.Slice(start: newLineLength + 1);
        }

        fileStream.Seek(-1 * slice.Length, SeekOrigin.Current); // we go back where the last line started
    }
}
```

ArrayPool

```
using (var fileStream = new FileStream(FilePath, FileMode.Open, FileAccess.Read))
{
    ArrayPool<byte> pool = ArrayPool<byte>.Shared;
    byte[] buffer = pool.Rent(16000 * 8); // 128 KB

    try
    {
        // (...)
    }
    finally
    {
        pool.Return(buffer);
    }
}
```

Parsing 6.5 GB file allocating just 176 B

Method	Mean	Ratio	Gen 0/Op	Gen 1/Op	Gen 2/1 Op	Allocated Memory
OldWay	138.9 s	1.00	11562	636	1	72 760 971 232 B
NewWay	104.3 s	0.75	-	-	-	128 200 B
NewWayPool	104.2 s	0.75	-	-	-	176 B

Async

```
public float NewWayArrayPool()  
{  
    // (...)  
    while ((bytesRead = fileStream.Read(buffer, 0, buffer.Length)) > 0)  
  
public async Task<float> NewWayArrayPoolAsync()  
{  
    // (...)  
    while ((bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
```

error CS4012: Parameters or locals of type 'ReadOnlySpan<byte>' cannot be declared in async methods or lambda expressions.

Span<T> is a Stack Only type (ref struct)

- Why:
 - Span<T> can point to stack-allocated memory
 - When the method ends or throws everything that was allocated on the stack is destroyed
- Advantages:
 - Safe Concurrency – only one thread has access at the same time (no Struct Tearing)
 - Short lifetime - fewer interior pointers for GC to track

Stack Only: Must not be stored on a heap

```
class SomeClass  
{  
    StackOnly<byte> field;  
}
```

```
async Task Method(StackOnly<byte> bytes)
```

Memory<T>

```
public readonly struct Memory<T>
{
    private readonly object _object; // String, Array or OwnedMemory
    private readonly int _index;
    private readonly int _length;

    public Span<T> Span { get; }

    public Memory<T> Slice(int start)
    public Memory<T> Slice(int start, int length)

    public MemoryHandle Pin()
}
```

Pass Memory from async to sync to get Span

```
public async Task<float> NewWayArrayPoolAsync()
{
    using (var fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
    {
        byte[] buffer = ArrayPool<byte>.Shared.Rent(16000 * 8); // 128 KB

        while ((bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
        {
            ParseBlock(new ReadOnlyMemory<byte>(buffer, start: 0, length: bytesRead), fileStream, ref result);
        }

        ArrayPool<byte>.Shared.Return(buffer);
    }
}

private void ParseBlock(ReadOnlyMemory<byte> memory, FileStream fileStream, ref float result)
{
    ReadOnlySpan<byte> slice = memory.Span;

    // using Span from here
}
```

Parsing 6.5 GB in asynchronous way

Method	Mean	Ratio	Gen 0/Op	Gen 1/Op	Gen 2/1 Op	Allocated Memory
OldWay	138.9 s	1.00	11562	636	1	72 760 971 232 B
NewWay	104.3 s	0.75	-	-	-	128 200 B
NewWayPool	104.2 s	0.75	-	-	-	176 B
NewWayPoolAsync	107.5 s	0.77	-	-	-	131 848 B

Generic Parser

```
public List<T> ParseFile<T>(string path, Func<ReadOnlySpan<byte>, T> lineParser)
{
    // (...)
    ReadOnlySpan<byte> utf8Line = slice.Slice(0, newLineLength);

    list.Add(lineParser(utf8Line));
}
```

error CS0306: The type 'ReadOnlySpan<byte>' may not be used as a type argument

Stack Only: No Heap Limitations

```
void NonConstrained<T>(IEnumerable<T> collection)
```

```
struct SomeValueType<T> : IEnumerable<T> { }
```

```
void Demo()
```

```
{
```

```
    var value = new SomeValueType<int>();
```

```
    NonConstrained(value);
```

```
}
```

Boxing == Heap. Heap != Stack

```
.method private hidebysig
instance void Demo () cil managed
{
    // Method begins at RVA 0x2054
    // Code size 21 (0x15)
    .maxstack 2
    .locals init (
        [0] valuetype Sample.SomeValueType`1<int32> 'value'
    )

    IL_0000: ldloc.s 'value'
    IL_0002: initobj valuetype Sample.SomeValueType`1<int32>
    IL_0008: ldarg.0
    IL_0009: ldloc.0
    IL_000a: box valuetype Sample.SomeValueType`1<int32>
    IL_000f: call instance void Sample.Program::NonConstrained<int32>(class
    IL_0014: ret
} // end of method Program::Demo
```

What Func and Action are?

```
namespace System
{
    public delegate void Action();
    public delegate void Action<in T>(T obj);
    public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
    public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
    public delegate void Action<in T1, in T2, in T3, in T4>(T1 arg1, T2 arg2, T3 arg3, T4 arg4);
    public delegate void Action<in T1, in T2, in T3, in T4, in T5>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5);
    public delegate void Action<in T1, in T2, in T3, in T4, in T5, in T6>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6);
    public delegate void Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6, T7 arg7);
    public delegate void Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6, T7 arg7, T8 arg8);

    public delegate TResult Func<out TResult>();
    public delegate TResult Func<in T, out TResult>(T arg);
    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
    public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);
    public delegate TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4);
    public delegate TResult Func<in T1, in T2, in T3, in T4, in T5, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5);
    public delegate TResult Func<in T1, in T2, in T3, in T4, in T5, in T6, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6);
    public delegate TResult Func<in T1, in T2, in T3, in T4, in T5, in T6, in T7, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6, T7 arg7);
    public delegate TResult Func<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4, T5 arg5, T6 arg6, T7 arg7, T8 arg8);

    public delegate int Comparison<in T>(T x, T y);

    public delegate TOutput Converter<in TInput, out TOutput>(TInput input);

    public delegate bool Predicate<in T>(T obj);
}
```


Generic Parser

```
public delegate T ParsingFunc<T>(ReadOnlySpan<byte> input);

public List<T> ParseFile<T>(string path, ParsingFunc<T> lineParser)
{
    // (...)
    ReadOnlySpan<byte> utf8Line = slice.Slice(0, newLineLength);

    list.Add(lineParser(utf8Line));
}
```

How do I convert an array of floats to a byte[] and back?



I have an array of Floats that need to be converted to a byte array and back to a float[]... can anyone help me do this correctly?

34



I'm working with the bitConverter class and found myself stuck trying to append the results.



The reason I'm doing this is so I can save runtime values into a IO Stream. The target storage is Azure Page blobs in case that matters. I don't care about what endian this is stored in, as long as it input matches the output.

11

MemoryMarshal.Cast

```
private float[] arrayOfFloats;  
// (...) setup  
  
[Benchmark(Baseline = true)]  
public byte[] OldWay()  
{  
    var byteArray = new byte[arrayOfFloats.Length * 4];  
  
    Buffer.BlockCopy(arrayOfFloats, 0, byteArray, 0, byteArray.Length);  
  
    return byteArray;  
}  
  
[Benchmark]  
public Span<byte> NewWay() => MemoryMarshal.Cast<float, byte>(arrayOfFloats);
```

MemoryMarshal.Cast is O(1)

Method	Count	Mean	Ratio	Gen 0/1k Op	Allocated Memory/Op
OldWay	4	13.2849 ns	1.00	0.0190	40 B
NewWay	4	0.6356 ns	0.05	-	-
OldWay	1000	311.4184 ns	1.000	1.9155	4024 B
NewWay	1000	0.6123 ns	0.002	-	-

MemoryMarshal.Read is also O(1)

```
[StructLayout(LayoutKind.Explicit)]
public struct Bid
{
    [FieldOffset(0)]
    public float Value;

    [FieldOffset(4)]
    public long ProductId;

    [FieldOffset(12)]
    public long UserId;

    [FieldOffset(20)]
    public DateTime Time;
}

public class BinaryRead
{
    public Bid Deserialize(ReadOnlySpan<byte> serialized) => MemoryMarshal.Read<Bid>(serialized);
}
```

MemoryMarshal API

```
class MemoryMarshal
{
    ReadOnlySpan<byte> AsBytes<T>(ReadOnlySpan<T> span) where T : struct
    Span<byte> AsBytes<T>(Span<T> span) where T : struct
    Memory<T> AsMemory<T>(ReadOnlyMemory<T> memory)
    ref readonly T AsRef<T>(ReadOnlySpan<byte> span) where T : struct
    ref T AsRef<T>(Span<byte> span) where T : struct
    ReadOnlySpan<TTo> Cast<TFrom, TTo>(ReadOnlySpan<TFrom> span) where TFrom : struct where TTo : struct
    Span<TTo> Cast<TFrom, TTo>(Span<TFrom> span) where TFrom : struct where TTo : struct
    Memory<T> CreateFromPinnedArray<T>(T[] array, int start, int length)
    ReadOnlySpan<T> CreateReadOnlySpan<T>(ref T reference, int length)
    Span<T> CreateSpan<T>(ref T reference, int length)
    ref T GetReference<T>(ReadOnlySpan<T> span)
    ref T GetReference<T>(Span<T> span)
    T Read<T>(ReadOnlySpan<byte> source) where T : struct
    IEnumerable<T> ToEnumerable<T>(ReadOnlyMemory<T> memory)
    bool TryGetArray<T>(ReadOnlyMemory<T> memory, out ArraySegment<T> segment)
    bool TryGetMemoryManager<T, TManager>(ReadOnlyMemory<T> memory, out TManager manager)
    bool TryGetString(ReadOnlyMemory<char> memory, out string text, out int start, out int length)
    bool TryRead<T>(ReadOnlySpan<byte> source, out T value) where T : struct
    bool TryWrite<T>(Span<byte> destination, ref T value) where T : struct
    void Write<T>(Span<byte> destination, ref T value) where T : struct
}
```



.NET Core 2.1

Search

System.Buffers

- > ArrayPool<T>
- > BuffersExtensions
- > IBufferWriter<T>
- > IMemoryOwner<T>
- > IPinnable
- > MemoryHandle
- > MemoryManager<T>
- > MemoryPool<T>
- OperationStatus
- > ReadOnlySequence<T>.Enumerator
- > ReadOnlySequence<T>
- > ReadOnlySequence

System.Buffers Namespace

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

Classes

[ArrayPool<T>](#) Provides a resource pool that enables reusing instances of type T[].

[BuffersExtensions](#)

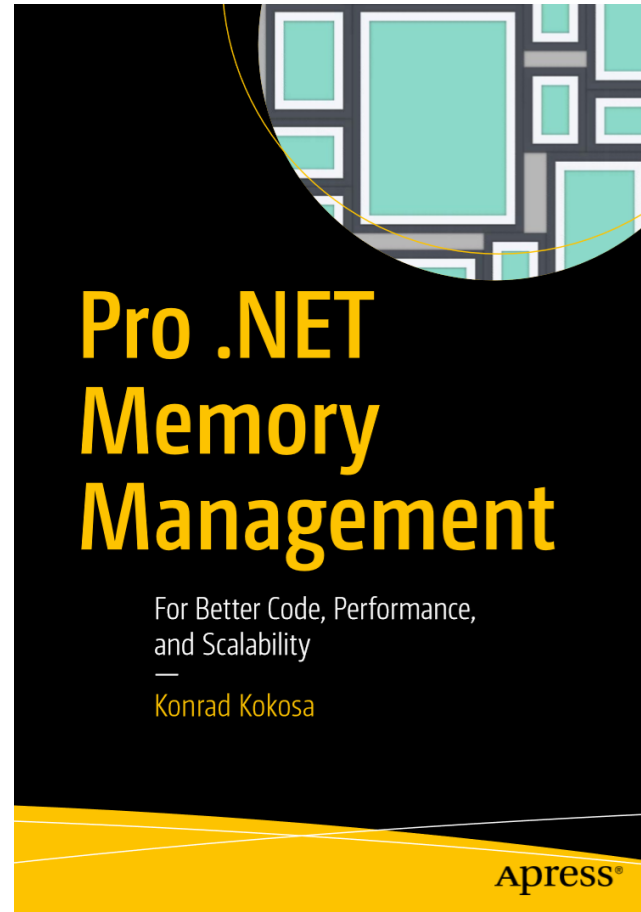
[MemoryManager<T>](#)

[MemoryPool<T>](#)

[ReadOnlySequence](#)

[Segment<T>](#)

Chapter XIV: Advanced Techniques



Summary

- `Span<T>` makes it easy and safe to use any kind of memory
- Don't copy memory! Slice it!
- `Memory<T>` has no stack-only limitations
- `System.Memory` package, C# 7.2
- Use `Utf8Parser` and `Utf8Formatter` when working with UTF8
- Prefer read-only versions over mutable ones
- Prefer safe managed memory over native memory

Questions?

Thank you!