C# 7 Span<T> and universal memory management

Background

.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 the event of interop or low-level development, you may want the access to the native objects and system memory, here is why the interop part comes, there are types that can marshal into the native world, invoke native APIs, convert managed/native types and define a native structure from the managed code.

Problem 1: Memory access patterns

In .NET world, there are three types of memory you may be interested:

- Managed heap memory, such as an array;
- Stack memory, such as objects created by `stackalloc`;
- Native memory, such as a native pointer reference.

Each type of memory access may need to use language features that are designed for it:

- To access heap memory, use the `fixed` (pinned) pointer on supported types (like `string`), or use other appropriate .NET types that have access to it, such as an array or a buffer;
- To access stack memory, use pointers with `stackalloc`;
- To access unmanaged system memory, use pointers with `Marshal` APIs.

You see, different access pattern needs different code, no single built-in type for all contiguous memory access.

Problem 2: Performance

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`, `IsNullOrWhiteSpace`, and `SubString` may be the most frequently used string APIs, and they are also very heavy:

- `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.

Span<T>

`System.Span<T>` is a stack-only type (`ref struct`) that wraps all memory access patterns, it is the type for universal contiguous memory access. You can think the implementation of the Span<T> contains a dummy reference and a length, accepting all 3 memory access types.

You can create a Span<T> using its constructor overloads or implicit operators from array, stackalloc'd pointers and unmanaged pointers.

```
// Use implicit operator Span<char>(char[]).
Span<char> span1 = new char[] { 's', 'p', 'a', 'n' };

// Use stackalloc.
Span<byte> span2 = stackalloc byte[50];

// Use constructor.
IntPtr array = new IntPtr();
Span<int> span3 = new Span<int>(array.ToPointer(), 1);
```

Once you have a Span<T> object, you can set value with a specified index, or return a portion of the span:

```
// Create an instance.
Span<char> span = new char[] { 's', 'p', 'a', 'n' };
// Access the reference of the first element.
ref char first = ref span[0];
// Assign the reference with a new value.
first = 'S';
// You get "Span".
Console.WriteLine(span.ToArray());

// Return a new span with start index = 1 and end index = span.Length - 1.
// You get "pan".
Span<char> span2 = span.Slice(1);
Console.WriteLine(span2.ToArray());
```

You can then use the `Slice()` method to write a high performance `Trim()` method:

```
private static void Main(string[] args)
{
    string test = "   Hello, World! ";
    Console.WriteLine(Trim(test.ToCharArray()).ToArray());
}

private static Span<char> Trim(Span<char> source)
{
    if (source.IsEmpty)
    {
        return source;
    }

    int start = 0, end = source.Length - 1;
    char startChar = source[start], endChar = source[end];

    while ((start < end) && (startChar == ' ' || endChar == ' '))
    {
        if (startChar == ' ')
        {
            start++;
        }

        if (endChar == ' ')
        {
            end–;
        }

        startChar = source[start];
        endChar = source[end];
    }

    return source.Slice(start, end - start + 1);
}
```

The above code does not copy over strings, nor generate new strings, it returns a portion of the original string by calling the `Slice()`method.

Because Span<T> is a ref struct, all ref struct restrictions apply. i.e. you cannot use Span<T> in fields, properties, iterator and async methods.

Memory<T>

`System.Memory<T>` is a wrapper of `System.Span<T>`, make it accessible in iterator and async methods. Use the `Span` property on the Memory<T> to access the underlying memory, this is extremely helpful in the asynchronous scenarios like File Streams and network communications (`HttpClient` etc..)

The following code shows simple usage of this type.

```
private static async Task Main(string[] args)
{
    Memory<byte> memory = new Memory<byte>(new byte[50]);
    int count = await ReadFromUrlAsync("https://www.microsoft.com", memory).Configure
Await(false);
    Console.WriteLine("Bytes written: {0}", count);
}

private static async ValueTask<int> ReadFromUrlAsync(string url, Memory<byte> memory)
{
    using (HttpClient client = new HttpClient())
    {
        Stream stream = await client.GetStreamAsync(new Uri(url)).ConfigureAwait(fals
e);
        return await stream.ReadAsync(memory).ConfigureAwait(false);
    }
}
```

The Framework Class Library/Core Framework (FCL/CoreFx) will add APIs based on the span-like types for Streams, strings and more in .NET Core 2.1.

ReadOnlySpan<T> and ReadOnlyMemory<T>

`System.ReadOnlySpan<T>` is the read-only version of the `System.Span<T>` struct where the indexer returns a `readonly ref` object instead of `ref` object. You get read-only memory access when using `System.ReadOnlySpan<T>` `readonly ref` struct.

This is useful for `string` type, because string is immutable, it is treated as read-only span.

We can rewrite the above code to implement the `Trim()` method using ReadOnlySpan<T>:

```
private static void Main(string[] args)
{
    // Implicit operator ReadOnlySpan(string).
    ReadOnlySpan<char> test = "   Hello, World! ";
    Console.WriteLine(Trim(test).ToArray());
}

private static ReadOnlySpan<char> Trim(ReadOnlySpan<char> source)
{
    if (source.IsEmpty)
    {
        return source;
    }

    int start = 0, end = source.Length - 1;
    char startChar = source[start], endChar = source[end];

    while ((start < end) && (startChar == ' ' || endChar == ' '))
    {
        if (startChar == ' ')
     {
            start++;
        }

        if (endChar == ' ')
        {
            end—;
        }

        startChar = source[start];
        endChar = source[end];
    }

    return source.Slice(start, end - start + 1);
}
```

As you can see, Nothing is changed in the method body; I just changed the parameter type from `Span<T>` to `ReadOnlySpan<T>`, and used the implicit operator to convert a `string` literal to `ReadOnlySpan<char>`.

`System.ReadOnlyMemory<T>` is the read-only version of `System.Memory<T>` struct where the `Span` property is a `ReadOnlySpan<T>`. When using this type, you get read-only access to the memory and you can use it with an iterator method or async method.

Memory Extensions

The `System.MemoryExtensions` class contains extension methods for different types that manipulates with span types, here is a list of commonly used extension methods, many of them are the equivalent implementations for existing APIs using the span types.

- **AsSpan, AsMemory**: Convert arrays into Span<T> or Memory<T> or their read-only counterparts.
- **BinarySearch, IndexOf, LastIndexOf**: Search elements and indexes.
- **IsWhiteSpace, Trim, TrimStart, TrimEnd, ToUpper, ToUpperInvariant, ToLower, ToLowerInvariant**: `Span<char>` operations similar to `string`.

Memory Marshal

In some case, you probably want to have lower level access to the memory types and system buffers, and convert between spans and read-only spans.
The `System.Runtime.InteropServices.MemoryMarshal` static class provides such functionalities to allow you control these access scenarios. The following code shows to title case a string using the span types, this is high performant because there is no temporary string allocations.

```
private static void Main(string[] args)
{
    string source = "span like types are awesome!";
    // source.ToMemory() converts source from string to ReadOnlyMemory<char>,
    // and MemoryMarshal.AsMemory converts ReadOnlyMemory<char> to Memory<char>
    // so you can modify the elements.
    TitleCase(MemoryMarshal.AsMemory(source.AsMemory()));
    // You get "Span like types are awesome!";
    Console.WriteLine(source);
}

private static void TitleCase(Memory<char> memory)
{
    if (memory.IsEmpty)
    {
        return;
    }

    ref char first = ref memory.Span[0];
    if (first >= 'a' && first <= 'z')
    {
        first = (char)(first - 32);
    }
}
```

Conclusion

Span<T> and Memory<T> enables a uniform way to access contiguous memory, regardless how the memory is allocated. It is very helpful for native development scenarios, as well as high performance scenarios. Especially, you will gain significant performance improvements while using span types to work with strings. It is a very nice feature innovated in C# 7.2.

NOTE: To use this feature, you will need to use Visual Studio 2017.5 and language version 7.2 or latest.

Tags C# CLR

August 12, 2018 at 10:12 pm

Nice article. Though I can't help but wondering, in the last code snippet, you seem to have changed the actual content of a string instance. This can lead to some weird situations, especially if `source` has already been interned (which is usually the case).

Suppose we have

string source = "span like types are awesome!";
string source2 = "span like types are awesome!";
Debug.Assert(object.ReferenceEquals(s1, s2));
TitleCase(MemoryMarshal.AsMemory(source.AsMemory()));
Console.WriteLine(source);
Console.WriteLine(source2);

here. By modifying the CONTENT of `source` string, actually you modified `source2` at the same time. I don't think it's a good idea somehow…

September 20, 2018 at 8:43 pm

When I try to compile the example code for Memory section, I give me an compile error that says there's no overloaded version of ReadAsync that has only one parameter.
And the next exmaple code emits compile error where you use the implicit converter. It seems that the implicit converter between string type and ReadOnlySpan type is no longer available, and you have to use the AsSpan() method in this case.

C# - All About Span: Exploring a New .NET Mainstay

Imagine you're exposing a specialized sort routine to operate in-place on data in memory. You'd likely expose a method that takes an array and provide an implementation that operates over that T[]. That’s great if your method’s caller has an array and wants the whole array sorted, but what if the caller only wants part of it sorted? You’d probably then also expose an overload that took an offset and a count. But what if you wanted to support data in memory that wasn’t in an array, but instead came from native code, for example, or lived on the stack and you only had a pointer and a length? How could you write your sort method that operated on such an arbitrary region of memory, and yet worked equally well with full arrays or with subsets of arrays, and that also worked equally well with managed arrays and unmanaged pointers?

Or take another example. You’re implementing an operation over System.String, such as a specialized parsing method. You’d likely expose a method that takes a string and provide an implementation that operates on strings. But what if you wanted to support operating over a subset of that string? String.Substring could be used to carve out just the piece that’s interesting to them, but that’s a relatively expensive operation, involving a string allocation and memory copy. You could, as mentioned in the array example, take an offset and a count, but then what if the caller doesn’t have a string but instead has a char[]? Or what if the caller has a char*, like one they created with stackalloc to use some space on the stack, or as the result of a call to native code? How could you write your parsing method in a way that didn’t force the caller to do any allocations or copies, and yet worked equally well with inputs of type string, char[] and char*?

In both situations, you might be able to use unsafe code and pointers, exposing an implementation that accepted a pointer and a length. That, however, eliminates the safety guarantees that are core to .NET and opens you up to problems like buffer overruns and access violations that for most .NET developers are a thing of the past. It also invites additional performance penalties, such as needing to pin managed objects for the duration of the operation so that the pointer you retrieve remains valid. And depending on the type of data involved, getting a pointer at all may not be practical.

There’s an answer to this conundrum, and its name is Span<T>.

What Is Span<T>?

System.Span<T> is a new value type at the heart of .NET. It enables the representation of contiguous regions of arbitrary memory, regardless of whether that memory is associated with a managed object, is provided by native code via interop, or is on the stack. And it does so while still providing safe access with performance characteristics like that of arrays.

For example, you can create a Span<T> from an array:

```
var arr = new byte[10];
Span<byte> bytes = arr; // Implicit cast from T[] to Span<T>
```

From there, you can easily and efficiently create a span to represent/point to just a subset of this array, utilizing an overload of the span's Slice method. From there you can index into the resulting span to write and read data in the relevant portion of the original array:

```
Span<byte> slicedBytes = bytes.Slice(start: 5, length: 2);
slicedBytes[0] = 42;
slicedBytes[1] = 43;
Assert.Equal(42, slicedBytes[0]);
Assert.Equal(43, slicedBytes[1]);
Assert.Equal(arr[5], slicedBytes[0]);
Assert.Equal(arr[6], slicedBytes[1]);
slicedBytes[2] = 44; // Throws IndexOutOfRangeException
bytes[2] = 45; // OK
Assert.Equal(arr[2], bytes[2]);
Assert.Equal(45, arr[2]);
```

As mentioned, spans are more than just a way to access and subset arrays. They can also be used to refer to data on the stack. For example:

```
Span<byte> bytes = stackalloc byte[2]; // Using C# 7.2 stackalloc support for spans
bytes[0] = 42;
bytes[1] = 43;
Assert.Equal(42, bytes[0]);
Assert.Equal(43, bytes[1]);
bytes[2] = 44; // throws IndexOutOfRangeException
```

More generally, they can be used to refer to arbitrary pointers and lengths, such as to memory allocated from a native heap, like so:

```
IntPtr ptr = Marshal.AllocHGlobal(1);
try
{
  Span<byte> bytes;
  unsafe { bytes = new Span<byte>((byte*)ptr, 1); }
  bytes[0] = 42;
  Assert.Equal(42, bytes[0]);
  Assert.Equal(Marshal.ReadByte(ptr), bytes[0]);
  bytes[1] = 43; // Throws IndexOutOfRangeException
}
finally { Marshal.FreeHGlobal(ptr); }
```

The Span<T> indexer takes advantage of a C# language feature introduced in C# 7.0 called ref returns. The indexer is declared with a "ref T" return type, which provides semantics like that of indexing into arrays, returning a reference to the actual storage location rather than returning a copy of what lives at that location:

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

The impact of this ref-returning indexer is most obvious via example, such as by comparing it with the List<T> indexer, which is not ref returning. Here's an example:

```
struct MutableStruct { public int Value; }
...
Span<MutableStruct> spanOfStructs = new MutableStruct[1];
spanOfStructs[0].Value = 42;
Assert.Equal(42, spanOfStructs[0].Value);
var listOfStructs = new List<MutableStruct> { new MutableStruct() };
listOfStructs[0].Value = 42; // Error CS1612: the return value is not a variable
```

A second variant of Span<T>, called System.ReadOnlySpan<T>, enables read-only access. This type is just like Span<T>, except its indexer takes advantage of a new C# 7.2 feature to return a "ref readonly T" instead of a "ref T," enabling it to work with immutable data types like System.String. ReadOnlySpan<T> makes it very efficient to slice strings without allocating or copying, as shown here:

```
string str = "hello, world";
string worldString = str.Substring(startIndex: 7, length: 5); // Allocates
ReadOnlySpan<char> worldSpan =
  str.AsSpan().Slice(start: 7, length: 5); // No allocation
Assert.Equal('w', worldSpan[0]);
worldSpan[0] = 'a'; // Error CS0200: indexer cannot be assigned to
```

Spans provide a multitude of benefits beyond those already mentioned. For example, spans support the notion of reinterpret casts, meaning you can cast a Span<byte> to be a Span<int> (where the 0th index into the Span<int> maps to the first four bytes of the Span<byte>). That way if you read a buffer of bytes, you can pass it off to methods that operate on grouped bytes as ints safely and efficiently.

How Is Span<T> Implemented?

Developers generally don't need to understand how a library they're using is implemented. However, in the case of Span<T>, it's worthwhile to have at least a basic understanding of the details behind it, as those details imply something about both its performance and its usage constraints.

First, Span<T> is a value type containing a ref and a length, defined approximately as follows:

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

The concept of a ref T field may be strange at first—in fact, one can't actually declare a ref T field in C# or even in MSIL. But Span<T> is actually written to use a special internal type in the runtime that's treated as a just-in-time (JIT) intrinsic, with the JIT generating for it the equivalent of a ref T field. Consider a ref usage that's likely much more familiar:

```
public static void AddOne(ref int value) => value += 1;
...
var values = new int[] { 42, 84, 126 };
AddOne(ref values[2]);
Assert.Equal(127, values[2]);
```

This code passes a slot in the array by reference, such that (optimizations aside) you have a ref T on the stack. The ref T in the Span<T> is the same idea, simply encapsulated inside a struct. Types that contain such refs directly or indirectly are called ref-like types, and the C# 7.2 compiler allows declaration of such ref-like types by using ref struct in the signature.

From this brief description, two things should be clear:

1. Span<T> is defined in such a way that operations can be as efficient as on arrays: indexing into a span doesn't require computation to determine the beginning from a pointer and its starting offset, as the ref field itself already encapsulates both. (By contrast, ArraySegment<T> has a separate offset field, making it more expensive both to index into and to pass around.)

2. The nature of Span<T> as a ref-like type brings with it some constraints due to its ref T field.

This second item has some interesting ramifications that result in .NET containing a second and related set of types, led by Memory<T>.

What Is Memory<T> and Why Do You Need It?

Span<T> is a ref-like type as it contains a ref field, and ref fields can refer not only to the beginning of objects like arrays, but also to the middle of them:

```
var arr = new byte[100];
Span<byte> interiorRef1 = arr.AsSpan(start: 20);
Span<byte> interiorRef2 = new Span<byte>(arr, 20, arr.Length - 20);
Span<byte> interiorRef3 =
  MemoryMarshal.CreateSpan<byte>(arr, ref arr[20], arr.Length - 20);
```

These references are called interior pointers, and tracking them is a relatively expensive operation for the .NET runtime's garbage collector. As such, the runtime constrains these refs to only live on the stack, as it provides an implicit low limit on the number of interior pointers that might be in existence.

Further, Span<T> as previously shown is larger than the machine's word size, which means reading and writing a span is not an atomic operation. If multiple threads read and write a span's fields on the heap at the same time, there's a risk of "tearing." Imagine an already initialized span containing a valid reference and a corresponding _length of 50. One thread starts writing a new span over it and gets as far as writing the new _pointer value. Then, before it can set the corresponding _length to 20, a second thread reads the span, including the new _pointer but the old (and longer) _length.

As a result, Span<T> instances can only live on the stack, not on the heap. This means you can't box spans (and thus can't use Span<T> with existing reflection invoke APIs, for example, as they require boxing). It means you can't have Span<T> fields in classes, or even in non-ref-like structs. It means you can't use spans in places where they might implicitly become fields on classes, for instance by capturing them into lambdas or as locals in async methods or iterators (as those "locals" may end up being fields on the compiler-generated state machines.) It also means you can't use Span<T> as a generic argument, as instances of that type argument could end up getting boxed or otherwise stored to the heap (and there's currently no "where T : ref struct" constraint available).

These limitations are immaterial for many scenarios, in particular for compute-bound and synchronous processing functions. But asynchronous functionality is another story. Most of the issues cited at the beginning of this article around arrays, array slices, native memory, and so on exist whether dealing with synchronous or asynchronous operations. Yet, if Span<T> can't be stored to the heap and thus can't be persisted across asynchronous operations, what's the answer? Memory<T>.

Memory<T> looks very much like an ArraySegment<T>:

```
public readonly struct Memory<T>
{
  private readonly object _object;
  private readonly int _index;
  private readonly int _length;
  ...
}
```

You can create a Memory<T> from an array and slice it just as you would a span, but it's a (non-ref-like) struct and can live on the heap. Then, when you want to do synchronous processing, you can get a Span<T> from it, for example:

```
static async Task<int> ChecksumReadAsync(Memory<byte> buffer, Stream stre
am)
{
  int bytesRead = await stream.ReadAsync(buffer);
  return Checksum(buffer.Span.Slice(0, bytesRead));
  // Or buffer.Slice(0, bytesRead).Span
}
static int Checksum(Span<byte> buffer) { ... }
```

As with Span<T> and ReadOnlySpan<T>, Memory<T> has a read-only equivalent, ReadOnlyMemory<T>. And as you'd expect, its Span property returns a ReadOnlySpan<T>. See **Figure 1**for a quick summary of built-in mechanisms for converting between these types.

Figure 1 Non-Allocating/Non-Copying Conversions Between Span-Related Types

From	To	Mechanism
ArraySegment<T>	Memory<T>	Implicit cast, AsMemory method
ArraySegment<T>	ReadOnlyMemory<T>	Implicit cast, AsMemory method
ArraySegment<T>	ReadOnlySpan<T>	Implicit cast, AsSpan method

ArraySegment<T>	Span<T>	Implicit cast, AsSpan method
ArraySegment<T>	T[]	Array property
Memory<T>	ArraySegment<T>	MemoryMarshal.TryGetArray method
Memory<T>	ReadOnlyMemory<T>	Implicit cast, AsMemory method
Memory<T>	Span<T>	Span property
ReadOnlyMemory<T>	ArraySegment<T>	MemoryMarshal.TryGetArray method
ReadOnlyMemory<T>	ReadOnlySpan<T>	Span property
ReadOnlySpan<T>	ref readonly T	Indexer get accessor, marshaling methods
Span<T>	ReadOnlySpan<T>	Implicit cast, AsSpan method
Span<T>	ref T	Indexer get accessor, marshaling methods
String	ReadOnlyMemory<char>	AsMemory method
String	ReadOnlySpan<char>	Implicit cast, AsSpan method
T[]	ArraySegment<T>	Ctor, Implicit cast
T[]	Memory<T>	Ctor, Implicit cast, AsMemory method
T[]	ReadOnlyMemory<T>	Ctor, Implicit cast, AsMemory method
T[]	ReadOnlySpan<T>	Ctor, Implicit cast, AsSpan method
T[]	Span<T>	Ctor, Implicit cast, AsSpan method
void*	ReadOnlySpan<T>	Ctor
void*	Span<T>	Ctor

You’ll notice that Memory<T>’s _object field isn’t strongly typed as T[]; rather, it’s stored as an object. This highlights that Memory<T> can wrap things other than arrays, like System.Buffers.OwnedMemory<T>. OwnedMemory<T> is an abstract class that can be used to wrap data that needs to have its lifetime tightly managed, such as memory retrieved from a pool. That's a more advanced topic beyond the scope of this article, but it's how Memory<T> can be used to, for example, wrap pointers into native memory. ReadOnlyMemory<char> can also be used with strings, just as can ReadOnlySpan<char>.

How Do Span<T> and Memory<T> Integrate with .NET Libraries?

In the previous Memory<T> code snippet, you'll notice a call to Stream.ReadAsync that's passing in a Memory<byte>. But Stream.ReadAsync in .NET today is defined to accept a byte[]. How does that work?

In support of Span<T> and friends, hundreds of new members and types are being added across .NET. Many of these are overloads of existing array-based and string-based methods, while others are entirely new types focused on specific areas of processing. For example, all the primitive types like Int32 now have Parse overloads that accept a ReadOnlySpan<char> in addition to the existing overloads that take strings. Imagine a situation where you're expecting a string that contains two numbers separated by a comma (such as "123,456"), and you want to parse out those two numbers. Today you might write code like this:

```
string input = ...;
int commaPos = input.IndexOf(',');
int first = int.Parse(input.Substring(0, commaPos));
int second = int.Parse(input.Substring(commaPos + 1));
```

That, however, incurs two string allocations. If you're writing performance-sensitive code, that may be two string allocations too many. Instead, you can now write this:

```
string input = ...;
ReadOnlySpan<char> inputSpan = input;
int commaPos = input.IndexOf(',');
int first = int.Parse(inputSpan.Slice(0, commaPos));
int second = int.Parse(inputSpan.Slice(commaPos + 1));
```

By using the new Span-based Parse overloads, you've made this whole operation allocation-free. Similar parsing and formatting methods exist for primitives like Int32 up through core types like DateTime, TimeSpan and Guid, and even up to higher-level types like BigInteger and IPAddress.

In fact, many such methods have been added across the framework. From System.Random to System.Text.StringBuilder to System.Net.Sockets, overloads have been added to make working with {ReadOnly}Span<T> and

{ReadOnly}Memory<T> simple and efficient. Some of these even carry with them additional benefits. For example, Stream now has this method:

```
public virtual ValueTask<int> ReadAsync(
  Memory<byte> destination,
  CancellationToken cancellationToken = default) { ... }
```

You'll notice that unlike the existing ReadAsync method that accepts a byte[] and returns a Task<int>, this overload not only accepts a Memory<byte> instead of a byte[], but also returns a ValueTask<int> instead of a Task<int>. ValueTask<T> is a struct that helps avoid allocations in cases where an asynchronous method is frequently expected to return synchronously, and where it's unlikely we can cache a completed task for all common return values. For instance, the runtime can cache a completed Task<bool> for a result of true and one for a result of false, but it can't cache four billion task objects for all possible result values of a Task<int>.

Because it's quite common for Stream implementations to buffer in a way that makes ReadAsync calls complete synchronously, this new ReadAsync overload returns a ValueTask<int>. This means asynchronous Stream read operations that complete synchronously can be entirely allocation-free. ValueTask<T> is also used in other new overloads, such as in overloads of Socket.ReceiveAsync, Socket.SendAsync, WebSocket.ReceiveAsync and TextReader.ReadAsync.

In addition, there are places where Span<T> allows the framework to include methods that in the past raised memory safety concerns. Consider a situation where you want to create a string containing a randomly generated value, such as for an ID of some kind. Today you might write code that requires allocating a char array, like this:

```
int length = ...;
Random rand = ...;
var chars = new char[length];
for (int i = 0; i < chars.Length; i++)
{
  chars[i] = (char)(rand.Next(0, 10) + '0');
}
string id = new string(chars);
```

You could instead use stack-allocation, and even take advantage of Span<char>, to avoid needing to use unsafe code. This approach also takes advantage of the new string constructor that accepts a ReadOnlySpan<char>, like so:

```
int length = ...;
Random rand = ...;
Span<char> chars = stackalloc char[length];
```

```
for (int i = 0; i < chars.Length; i++)
{
  chars[i] = (char)(rand.Next(0, 10) + '0');
}
string id = new string(chars);
```

This is better, in that you've avoided the heap allocation, but you're still forced to copy into the string the data that was generated on the stack. This approach also only works when the amount of space required is something small enough for the stack. If the length is short, like 32 bytes, that's fine, but if it's thousands of bytes, it could easily lead to a stack overflow situation. What if you could write to the string's memory directly instead? Span<T> allows you to do that. In addition to string's new constructor, string now also has a Create method:

```
public static string Create<TState>(
  int length, TState state, SpanAction<char, TState> action);
...
public delegate void SpanAction<T, in TArg>(Span<T> span, TArg arg);
```

This method is implemented to allocate the string and then hand out a writable span you can write to in order to fill in the contents of the string while it's being constructed. Note that the stack-only nature of Span<T> is beneficial in this case, guaranteeing that the span (which refers to the string's internal storage) will cease to exist before the string's constructor completes, making it impossible to use the span to mutate the string after the construction is complete:

```
int length = ...;
Random rand = ...;
string id = string.Create(length, rand, (Span<char> chars, Random r) =>
{
  for (int i = 0; chars.Length; i++)
  {
    chars[i] = (char)(r.Next(0, 10) + '0');
  }
});
```

Now, not only have you avoided the allocation, you're writing directly into the string's memory on the heap, which means you're also avoiding the copy and you're not constrained by size limitations of the stack.

Beyond core framework types gaining new members, many new .NET types are being developed to work with spans for efficient processing in specific scenarios. For example, developers looking to write high-performance microservices and Web sites heavy in text processing can earn a significant performance win if they don't have to encode to and decode from strings when working in UTF-8. To enable this, new types like System.Buffers.Text.Base64, System.Buffers.Text.Utf8Parser and

System.Buffers.Text.Utf8Formatter are being added. These operate on spans of bytes, which not only avoids the Unicode encoding and decoding, but enables them to work with native buffers that are common in the very lowest levels of various networking stacks:

```
ReadOnlySpan<byte> utf8Text = ...;
if (!Utf8Parser.TryParse(utf8Text, out Guid value,
  out int bytesConsumed, standardFormat = 'P'))
  throw new InvalidDataException();
```

All this functionality isn't just for public consumption; rather the framework itself is able to utilize these new Span<T>-based and Memory<T>-based methods for better performance. Call sites across .NET Core have switched to using the new ReadAsync overloads to avoid unnecessary allocations. Parsing that had been done by allocating substrings now takes advantage of allocation-free parsing. Even niche types like Rfc2898DeriveBytes have gotten in on the action, taking advantage of the new Span<byte>-based TryComputeHash method on System.Security.Cryptography.Hash-Algorithm to achieve a monstrous savings on allocation (a byte array per iteration of the algorithm, which might iterate thousands of times), as well as a throughput improvement.

This doesn't stop at the level of the core .NET libraries; it continues all the way up the stack. ASP.NET Core now has a heavy dependency on spans, for example, with the Kestrel server's HTTP parser written on top of them. In the future, it's likely that spans will be exposed out of public APIs in the lower levels of ASP.NET Core, such as in its middleware pipeline.

What About the .NET Runtime?

One of the ways the .NET runtime provides safety is by ensuring that indexing into an array doesn't allow going beyond the length of the array, a practice known as bounds checking. For example, consider this method:

```
[MethodImpl(MethodImplOptions.NoInlining)]
static int Return4th(int[] data) => data[3];
```

On the x64 machine on which I'm typing this article, the generated assembly for this method looks like the following:

```
sub      rsp, 40
       cmp      dword ptr [rcx+8], 3
       jbe      SHORT G_M22714_IG04
       mov      eax, dword ptr [rcx+28]
       add      rsp, 40
       ret
G_M22714_IG04:
       call     CORINFO_HELP_RNGCHKFAIL
       int3
```

That cmp instruction is comparing the length of the data array against the index 3, and the subsequent jbe instruction is then jumping to the range check failure routine if 3 is out of range (for an exception to be thrown). The JIT needs to generate code that ensures such accesses don't go outside the bounds of the array, but that doesn't mean that every individual array access needs a bound check. Consider this Sum method:

```
static int Sum(int[] data)
{
  int sum = 0;
  for (int i = 0; i < data.Length; i++) sum += data[i];
  return sum;
}
```

The JIT needs to generate code here that ensures the accesses to data[i] don't go outside the bounds of the array, but because the JIT can tell from the structure of the loop that i will always be in range (the loop iterates through each element from beginning to end), the JIT can optimize away the bounds checks on the array. Thus, the assembly code generated for the loop looks like the following:

```
G_M33811_IG03:
       movsxd   r9, edx
       add      eax, dword ptr [rcx+4*r9+16]
       inc      edx
       cmp      r8d, edx
       jg       SHORT G_M33811_IG03
```

A cmp instruction is still in the loop, but simply to compare the value of i (as stored in the edx register) against the length of the array (as stored in the r8d register); no additional bounds checking.

The runtime applies similar optimizations to span (both Span<T> and ReadOnlySpan<T>). Compare the previous example to the following code, where the only change is on the parameter type:

```
static int Sum(Span<int> data)
{
  int sum = 0;
  for (int i = 0; i < data.Length; i++) sum += data[i];
  return sum;
}
```

The generated assembly for this code is almost identical:

```
G_M33812_IG03:
       movsxd   r9, r8d
       add      ecx, dword ptr [rax+4*r9]
       inc      r8d
       cmp      r8d, edx
       jl       SHORT G_M33812_IG03
```

The assembly code is so similar in part because of the elimination of bounds checks. But also relevant is the JIT's recognition of the span indexer as an intrinsic, meaning that the JIT generates special code for the indexer, rather than translating its actual IL code into assembly.

All of this is to illustrate that the runtime can apply for spans the same kinds of optimizations it does for arrays, making spans an efficient mechanism for accessing data. More details are available in the blog post at bit.ly/2zywvyI.

What About the C# Language and Compiler?

I've already alluded to features added to the C# language and compiler to help make Span<T> a first-class citizen in .NET. Several features of C# 7.2 are related to spans (and in fact the C# 7.2 compiler will be required to use Span<T>). Let's look at three such features.

Ref structs. As noted earlier, Span<T> is a ref-like type, which is exposed in C# as of version 7.2 as ref struct. By putting the ref keyword before struct, you tell the C# compiler to allow you to use other ref struct types like Span<T> as fields, and in doing so also sign up for the associated constraints to be assigned to your type. For example, if you wanted to write a struct Enumerator for a Span<T>, that Enumerator would need to store the Span<T> and, thus, would itself need to be a ref struct, like this:

```
public ref struct Enumerator
{
  private readonly Span<char> _span;
  private int _index;
  ...
}
```

Stackalloc initialization of spans. In previous versions of C#, the result of stackalloc could only be stored into a pointer local variable. As of C# 7.2, stackalloc can now be used as part of an expression and can target a span, and that can be done without using the unsafe keyword. Thus, instead of writing:

```
Span<byte> bytes;
```

```
unsafe
{
  byte* tmp = stackalloc byte[length];
  bytes = new Span<byte>(tmp, length);
}
```

You can write simply:

```
Span<byte> bytes = stackalloc byte[length];
```

This is also extremely useful in situations where you need some scratch space to perform an operation, but want to avoid allocating heap memory for relatively small sizes. Previously you had two choices:

- Write two completely different code paths, allocating and operating over stack-based memory and over heap-based memory.
- Pin the memory associated with the managed allocation and then delegate to an implementation also used for the stack-based memory and written with pointer manipulation in unsafe code.

Now, the same thing can be accomplished without code duplication, with safe code and with minimal ceremony:

```
Span<byte> bytes = length <= 128 ? stackalloc byte[length] : new byte[leng
th];
... // Code that operates on the Span<byte>
```

Span usage validation. Because spans can refer to data that might be associated with a given stack frame, it can be dangerous to pass spans around in a way that might enable referring to memory that's no longer valid. For example, imagine a method that tried to do the following:

```
static Span<char> FormatGuid(Guid guid)
{
  Span<char> chars = stackalloc char[100];
  bool formatted = guid.TryFormat(chars, out int charsWritten, "d");
  Debug.Assert(formatted);
  return chars.Slice(0, charsWritten); // Uh oh
}
```

Here space is being allocated from the stack and then trying to return a reference to that space, but the moment you return, that space will no longer be valid for use. Thankfully the C# compiler detects such invalid usage with ref structs and fails the compilation with an error:

error CS8352: Cannot use local 'chars' in this context because it may expose referenced variables outside of their declaration scope