Provides a type-safe and memory-safe representation of a contiguous region of arbitrary memory.

C#

```
[System.Runtime.InteropServices.Marshalling.NativeMarshalling(typeof(System.Runtime.InteropServices.Marshalling.SpanMarshaller<,>))]
public readonly ref struct Span<T>
```

Type Parameters

T

The type of items in the Span<T>.

Inheritance Object → ValueType → Span<T>

Attributes NativeMarshallingAttribute

Remarks

`Span<T>` is a ref struct that is allocated on the stack rather than on the managed heap. Ref struct types have a number of restrictions to ensure that they cannot be promoted to the managed heap, including that they can't be boxed, they can't be assigned to variables of type Object, `dynamic` or to any interface type, they can't be fields in a reference type, and they can't be used across `await` and `yield` boundaries. In addition, calls to two methods, Equals(Object) and GetHashCode, throw a NotSupportedException.

> **Important**
>
> Because it is a stack-only type, `Span<T>` is unsuitable for many scenarios that require storing references to buffers on the heap. This is true, for example, of routines that make asynchronous method calls. For such scenarios, you can use the complementary **System.Memory<T>** and **System.ReadOnlyMemory<T>** types.

For spans that represent immutable or read-only structures, use System.ReadOnlySpan<T>.

Span<T> and memory

A `Span<T>` represents a contiguous region of arbitrary memory. A `Span<T>` instance is often used to hold the elements of an array or a portion of an array. Unlike an array, however, a `Span<T>` instance can point to managed memory, native memory, or memory managed on the stack. The following example creates a `Span<Byte>` from an array:

C#

```
// Create a span over an array.
var array = new byte[100];
var arraySpan = new Span<byte>(array);

byte data = 0;
for (int ctr = 0; ctr < arraySpan.Length; ctr++)
    arraySpan[ctr] = data++;

int arraySum = 0;
foreach (var value in array)
    arraySum += value;

Console.WriteLine($"The sum is {arraySum}");
// Output:  The sum is 4950
```

The following example creates a `Span<Byte>` from 100 bytes of native memory:

C#

```
// Create a span from native memory.
var native = Marshal.AllocHGlobal(100);
Span<byte> nativeSpan;
unsafe
{
    nativeSpan = new Span<byte>(native.ToPointer(), 100);
}
byte data = 0;
for (int ctr = 0; ctr < nativeSpan.Length; ctr++)
    nativeSpan[ctr] = data++;

int nativeSum = 0;
foreach (var value in nativeSpan)
    nativeSum += value;

Console.WriteLine($"The sum is {nativeSum}");
Marshal.FreeHGlobal(native);
// Output:  The sum is 4950
```

The following example uses the C# stackalloc keyword to allocate 100 bytes of memory on

the stack:

C#

```
// Create a span on the stack.
byte data = 0;
Span<byte> stackSpan = stackalloc byte[100];
for (int ctr = 0; ctr < stackSpan.Length; ctr++)
    stackSpan[ctr] = data++;

int stackSum = 0;
foreach (var value in stackSpan)
    stackSum += value;

Console.WriteLine($"The sum is {stackSum}");
// Output:  The sum is 4950
```

Because `Span<T>` is an abstraction over an arbitrary block of memory, methods of the `Span<T>` type and methods with `Span<T>` parameters operate on any `Span<T>` object regardless of the kind of memory it encapsulates. For example, each of the separate sections of code that initialize the span and calculate the sum of its elements can be changed into single initialization and calculation methods, as the following example illustrates:

C#

```
public static void WorkWithSpans()
{
    // Create a span over an array.
    var array = new byte[100];
    var arraySpan = new Span<byte>(array);

    InitializeSpan(arraySpan);
    Console.WriteLine($"The sum is {ComputeSum(arraySpan):N0}");

    // Create an array from native memory.
    var native = Marshal.AllocHGlobal(100);
    Span<byte> nativeSpan;
    unsafe
    {
        nativeSpan = new Span<byte>(native.ToPointer(), 100);
    }

    InitializeSpan(nativeSpan);
    Console.WriteLine($"The sum is {ComputeSum(nativeSpan):N0}");

    Marshal.FreeHGlobal(native);
```

```
    // Create a span on the stack.
    Span<byte> stackSpan = stackalloc byte[100];

    InitializeSpan(stackSpan);
    Console.WriteLine($"The sum is {ComputeSum(stackSpan):N0}");
}

public static void InitializeSpan(Span<byte> span)
{
    byte value = 0;
    for (int ctr = 0; ctr < span.Length; ctr++)
        span[ctr] = value++;
}

public static int ComputeSum(Span<byte> span)
{
    int sum = 0;
    foreach (var value in span)
        sum += value;

    return sum;
}
// The example displays the following output:
//    The sum is 4,950
//    The sum is 4,950
//    The sum is 4,950
```

Span<T> and arrays

When it wraps an array, `Span<T>` can wrap an entire array, as it did in the examples in the Span<T> and memory section. Because it supports slicing, `Span<T>` can also point to any contiguous range within the array.

The following example creates a slice of the middle five elements of a 10-element integer array. Note that the code doubles the values of each integer in the slice. As the output shows, the changes made by the span are reflected in the values of the array.

C#

```
using System;

var array = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
var slice = new Span<int>(array, 2, 5);
for (int ctr = 0; ctr < slice.Length; ctr++)
```

```
for (int ctr = 0; ctr < slice.Length; ctr++)
    slice[ctr] *= 2;

// Examine the original array values.
foreach (var value in array)
    Console.Write($"{value}  ");
Console.WriteLine();

// The example displays the following output:
//       2  4  12  16  20  24  28  16  18  20
```

Span<T> and slices

`Span<T>` includes two overloads of the Slice method that form a slice out of the current span that starts at a specified index. This makes it possible to treat the data in a `Span<T>` as a set of logical chunks that can be processed as needed by portions of a data processing pipeline with minimal performance impact. For example, since modern server protocols are often text-based, manipulation of strings and substrings is particularly important. In the String class, the major method for extracting substrings is Substring. For data pipelines that rely on extensive string manipulation, its use offers some performance penalties, since it:

1. Creates a new string to hold the substring.
2. Copies a subset of the characters from the original string to the new string.

This allocation and copy operation can be eliminated by using either `Span<T>` or ReadOnlySpan<T>, as the following example shows:

C#

```
using System;

class Program2

{
    static void Main()
    {
        string contentLength = "Content-Length: 132";
        var length = GetContentLength(contentLength.ToCharArray());
        Console.WriteLine($"Content length: {length}");
    }

    private static int GetContentLength(ReadOnlySpan<char> span)
    {
        var slice = span.Slice(16);
        return int.Parse(slice);
```

```
    }
}
// Output:
//       Content length: 132
```

Constructors

Span<T>(T)	Creates a new Span<T> of length 1 around the specified reference.
Span<T>(T[])	Creates a new Span<T> object over the entirety of a specified array.
Span<T>(T[], Int32, Int32)	Creates a new Span<T> object that includes a specified number of elements of an array starting at a specified index.
Span<T>(Void*, Int32)	Creates a new Span<T> object from a specified number of `T` elements starting at a specified memory address.

Properties

Empty	Returns an empty Span<T> object.
IsEmpty	Returns a value that indicates whether the current Span<T> is empty.
Item[Int32]	Gets the element at the specified zero-based index.
Length	Returns the length of the current span.

Methods

Clear()	Clears the contents of this Span<T> object.
CopyTo(Span<T>)	Copies the contents of this Span<T> into a destination Span<T>.
Equals(Object)	**Obsolete.** Calls to this method are not supported.
Fill(T)	Fills the elements of this span with a specified value.
GetEnumerator()	Returns an enumerator for this Span<T>.

GetHashCode()	**Obsolete.** Throws a NotSupportedException.
GetPinnableReference()	Returns a reference to an object of type T that can be used for pinning. This method is intended to support .NET compilers and is not intended to be called by user code.
Slice(Int32)	Forms a slice out of the current span that begins at a specified index.
Slice(Int32, Int32)	Forms a slice out of the current span starting at a specified index for a specified length.
ToArray()	Copies the contents of this span into a new array.
ToString()	Returns the string representation of this Span<T> object.
TryCopyTo(Span<T>)	Attempts to copy the current Span<T> to a destination Span<T> and returns a value that indicates whether the copy operation succeeded.

Operators

Equality(Span<T>, Span<T>)	Returns a value that indicates whether two Span<T> objects are equal.
Implicit(ArraySegment<T> to Span<T>)	Defines an implicit conversion of an ArraySegment<T> to a Span<T>.
Implicit(Span<T> to ReadOnly Span<T>)	Defines an implicit conversion of a Span<T> to a ReadOnlySpan<T>.
Implicit(T[] to Span<T>)	Defines an implicit conversion of an array to a Span<T>.

Inequality(Span<T>, Span<T>)	Returns a value that indicates whether two Span<T> objects are not equal.

Extension Methods

ToImmutableArray<T> (Span<T>)	Converts the span to an immutable array.
BinarySearch<T>(Span<T>	Searches an entire sorted Span<T> for a value using the specified

IComparable<T>)	IComparable<T> generic interface.
BinarySearch<T,TComparer> (Span<T>, T, TComparer)	Searches an entire sorted Span<T> for a specified value using the specified `TComparer` generic type.
BinarySearch<T,TComparable> (Span<T>, TComparable)	Searches an entire sorted Span<T> for a value using the specified `TComparable` generic type.
CommonPrefixLength<T> (Span<T>, ReadOnlySpan<T>)	Finds the length of any common prefix shared between `span` and `other`.
CommonPrefixLength<T> (Span<T>, ReadOnlySpan<T>, IEqualityComparer<T>)	Finds the length of any common prefix shared between `span` and `other`.
Contains<T>(Span<T>, T)	Indicates whether a specified value is found in a span. Values are compared using IEquatable{T}.Equals(T).
EndsWith<T>(Span<T>, ReadOnlySpan<T>)	Determines whether the specified sequence appears at the end of a span.
IndexOf<T>(Span<T>, T)	Searches for the specified value and returns the index of its first occurrence. Values are compared using IEquatable{T}.Equals(T).
IndexOf<T>(Span<T>, ReadOnlySpan<T>)	Searches for the specified sequence and returns the index of its first occurrence. Values are compared using IEquatable{T}.Equals(T).
IndexOfAny<T>(Span<T>, T, T)	Searches for the first index of any of the specified values similar to calling IndexOf several times with the logical OR operator.
IndexOfAny<T>(Span<T>, T, T, T)	Searches for the first index of any of the specified values similar to calling IndexOf several times with the logical OR operator.
IndexOfAny<T>(Span<T>, ReadOnlySpan<T>)	Searches for the first index of any of the specified values similar to calling IndexOf several times with the logical OR operator.

IndexOfAnyExcept<T> (Span<T>, T)	Searches for the first index of any value other than the specified `value`.
IndexOfAnyExcept<T> (Span<T>, T, T)	Searches for the first index of any value other than the specified `value0` or `value1`.
IndexOfAnyExcept<T> (Span<T>, T, T, T)	Searches for the first index of any value other than the specified `value0`, `value1`, or `value2`.
IndexOfAnyExcept<T> (Span<T>, ReadOnlySpan<T>)	Searches for the first index of any value other than the specified `values`.

LastIndexOf<T>(Span<T>, T)	Searches for the specified value and returns the index of its last occurrence. Values are compared using IEquatable{T}.Equals(T).
LastIndexOf<T>(Span<T>, ReadOnlySpan<T>)	Searches for the specified sequence and returns the index of its last occurrence. Values are compared using IEquatable{T}.Equals(T).
LastIndexOfAny<T>(Span<T>, T, T)	Searches for the last index of any of the specified values similar to calling LastIndexOf several times with the logical OR operator.
LastIndexOfAny<T>(Span<T>, T, T, T)	Searches for the last index of any of the specified values similar to calling LastIndexOf several times with the logical OR operator.
LastIndexOfAny<T>(Span<T>, ReadOnlySpan<T>)	Searches for the last index of any of the specified values similar to calling LastIndexOf several times with the logical OR operator.
LastIndexOfAnyExcept<T>(Span<T>, T)	Searches for the last index of any value other than the specified `value`.
LastIndexOfAnyExcept<T>(Span<T>, T, T)	Searches for the last index of any value other than the specified `value0` or `value1`.
LastIndexOfAnyExcept<T>(Span<T>, T, T, T)	Searches for the last index of any value other than the specified `value0`, `value1`, or `value2`.
LastIndexOfAnyExcept<T>(Span<T>, ReadOnlySpan<T>)	Searches for the last index of any value other than the specified `values`.
Overlaps<T>(Span<T>, ReadOnlySpan<T>)	Determines whether a span and a read-only span overlap in memory.
Overlaps<T>(Span<T>, ReadOnlySpan<T>, Int32)	Determines whether a span and a read-only span overlap in memory and outputs the element offset.
Reverse<T>(Span<T>)	Reverses the sequence of the elements in the entire span.

SequenceCompareTo<T>(Span<T>, ReadOnlySpan<T>)	Determines the relative order of a span and a read-only span by comparing the elements using IComparable{T}.CompareTo(T).
SequenceEqual<T>(Span<T>, ReadOnlySpan<T>)	Determines whether a span and a read-only span are equal by comparing the elements using IEquatable{T}.Equals(T).
SequenceEqual<T>(Span<T>, ReadOnlySpan<T>, IEqualityComparer<T>)	Determines whether two sequences are equal by comparing the elements using an IEqualityComparer<T>.
Sort<T>(Span<T>)	Sorts the elements in the entire Span<T> using the

	IComparable<T> implementation of each element of the Span<T>.
Sort<T>(Span<T>, Comparison<T>)	Sorts the elements in the entire Span<T> using the specified Comparison<T>.
Sort<T,TComparer>(Span<T>, TComparer)	Sorts the elements in the entire Span<T> using the `TComparer`.
Sort<TKey,TValue>(Span<TKey>, Span<TValue>)	Sorts a pair of spans (one containing the keys and the other containing the corresponding items) based on the keys in the first Span<T> using the IComparable<T> implementation of each key.
Sort<TKey,TValue>(Span<TKey>, Span<TValue>, Comparison<TKey>)	Sorts a pair of spans (one containing the keys and the other containing the corresponding items) based on the keys in the first Span<T> using the specified comparison.
Sort<TKey,TValue,TComparer>(Span<TKey>, Span<TValue>, TComparer)	Sorts a pair of spans (one containing the keys and the other containing the corresponding items) based on the keys in the first Span<T> using the specified comparer.
StartsWith<T>(Span<T>, ReadOnlySpan<T>)	Determines whether a specified sequence appears at the start of a span.
Trim<T>(Span<T>, T)	Removes all leading and trailing occurrences of a specified element from a span.
Trim<T>(Span<T>, ReadOnlySpan<T>)	Removes all leading and trailing occurrences of a set of elements specified in a read-only span from a span.
TrimEnd<T>(Span<T>, T)	Removes all trailing occurrences of a specified element from a span.
TrimEnd<T>(Span<T>, ReadOnlySpan<T>)	Removes all trailing occurrences of a set of elements specified in a read-only span from a span.

TrimStart<T>(Span<T>, T)	Removes all leading occurrences of a specified element from the span.
TrimStart<T>(Span<T>, ReadOnlySpan<T>)	Removes all leading occurrences of a set of elements specified in a read-only span from the span.

Applies to

Product	Versions

Product	Versions
.NET	Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Standard	2.1

See also

- Memory- and span-related types
- Memory<T> and Span<T> usage guidelines