Spans and Slicing

Unlike an array, a span can easily be *sliced* to represent different subsections of the same underlying data, as illustrated in Figure 23-1.

To give a practical example, suppose that you're writing a method to sum an array of integers. A micro-optimized implementation would avoid LINQ in favor of a `foreach` loop:

```
int Sum (int[] numbers)
{
  int total = 0;
  foreach (int i in numbers) total += i;
  return total;
}
```

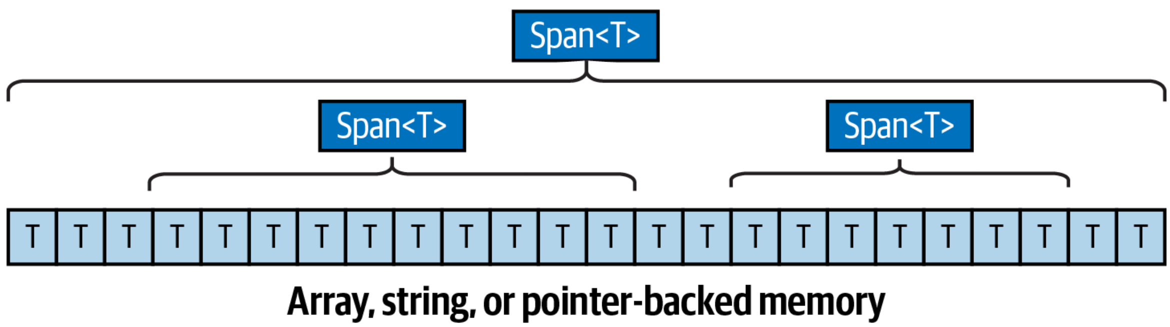

Figure 23-1. Slicing

Now imagine that you want to sum just a *portion* of the array. You have two options:

- First copy the portion of the array that you want to sum into another array
- Add additional parameters to the method (`offset` and `count`)

The first option is inefficient; the second option adds clutter and complexity (which worsens with methods that need to accept more than one array).

Spans solve this nicely. All you need to do is to change the parameter type from `int[]` to `ReadOnlySpan<int>` (everything else stays the same):

```
int Sum (ReadOnlySpan<int> numbers)
{
  int total = 0;
  foreach (int i in numbers) total += i;
  return total;
}
```

NOTE

We used `ReadOnlySpan<T>` rather than `Span<T>` because we don't need to modify the array. There's an implicit conversion from `Span<T>` to `ReadOnlySpan<T>`, so you can pass a `Span<T>` into a method that expects a `ReadOnlySpan<T>`.

We can test this method, as follows:

```
var numbers = new int [1000];
for (int i = 0; i < numbers.Length; i++) numbers [i] = i;

int total = Sum (numbers);
```

We can call `Sum` with an array because there's an implicit conversion from `T[]` to `Span<T>` and `ReadOnlySpan<T>`. Another option is to use the `AsSpan` extension method:

```
var span = numbers.AsSpan();
```

The indexer for `ReadOnlySpan<T>` uses C#'s `ref readonly` feature to reach directly into the underlying data: this allows our method to perform almost as well as the original example that used an array. But what we've gained is that we can now "slice" the array and sum just a portion of the elements as follows:

```
// Sum the middle 500 elements (starting from position 250):
int total = Sum (numbers.AsSpan (250, 500));
```

If you already have a `Span<T>` or `ReadOnlySpan<T>`, you can slice it by calling the `Slice` method:

```
Span<int> span = numbers;
int total = Sum (span.Slice (250, 500));
```

You can also use C#'s *indices* and *ranges* (from C# 8):

```
Span<int> span = numbers;
Console.WriteLine (span [^1]);            // Last element
Console.WriteLine (Sum (span [..10]));    // First 10 elements
Console.WriteLine (Sum (span [100..]));   // 100th element to end
Console.WriteLine (Sum (span [^5..]));    // Last 5 elements
```

Although `Span<T>` doesn't implement `IEnumerable<T>` (it can't implement interfaces by virtue of being a ref struct), it does implement the pattern that allows C#'s `foreach` statement to work (see "Enumeration").

CopyTo and TryCopyTo

The `CopyTo` method copies elements from one span (or `Memory<T>`) to another. In the following example, we copy all of the elements from span `x` into span `y`:

```
Span<int> x = [1, 2, 3, 4];   // Collection expression
Span<int> y = new int[4];
x.CopyTo (y);
```

NOTE

Notice that we initialized `x` with a *collection expression*. Collection expressions (from C# 12) are not only a useful shortcut, but in the case of spans, they allow the compiler the freedom to choose the underlying type. When the element count is small, the compiler may allocate memory on the stack (rather than creating an array) to avoid the overhead of a heap allocation.

Slicing makes this method much more useful. In the next example, we copy the first half of span `x` into the second half of span `y`:

```
Span<int> x = [1,  2,  3,  4 ];
Span<int> y = [10, 20, 30, 40];
```

```
x[..2].CopyTo (y[2..]);                // y is now [10, 20, 1, 2]
```

If there's not enough space in the destination to complete the copy, CopyTo throws an exception, whereas TryCopyTo returns false (without copying any elements).

The span structs also expose methods to Clear and Fill the span as well as an IndexOf method to search for an element in the span.

Searching in Spans

The MemoryExtensions class defines numerous extension methods to help with searching for values within spans such as Contains, IndexOf, LastIndexOf, and BinarySearch (as well as methods that mutate spans, such as Fill, Replace, and Reverse).

From .NET 8, there are also methods to search for any one of a number of values, such as ContainsAny, ContainsAnyExcept, IndexOfAny, and IndexOfAnyExcept. With these methods, you can specify the values to search either as a span or as a SearchValues<T> instance (in System.Buffers), which you instantiate by calling SearchValues.Create:

```
ReadOnlySpan<char> span = "The quick brown fox jumps over the lazy dog.";
var vowels = SearchValues.Create ("aeiou");
Console.WriteLine (span.IndexOfAny (vowels));   // 2
```

SearchValues<T> improves performance when the instance is reused across multiple searches.

> **NOTE**
>
> You can also utilize these methods when working with arrays or strings, simply by calling AsSpan() on the array or string.

Working with Text

Spans are designed to work well with strings, which are treated as `ReadOnlySpan<char>`. The following method counts whitespace characters:

```
int CountWhitespace (ReadOnlySpan<char> s)
{
  int count = 0;
  foreach (char c in s)
    if (char.IsWhiteSpace (c))
      count++;
  return count;
}
```

You can call such a method with a string (thanks to an implicit conversion operator):

```
int x = CountWhitespace ("Word1 Word2");   // OK
```

or with a substring:

```
int y = CountWhitespace (someString.AsSpan (20, 10));
```

The `ToString()` method converts a `ReadOnlySpan<char>` back to a string.

Extension methods ensure that some of the commonly used methods on the string class are also available to `ReadOnlySpan<char>`:

```
var span = "This ".AsSpan();                    // ReadOnlySpan<char>
Console.WriteLine (span.StartsWith ("This"));   // True
Console.WriteLine (span.Trim().Length);         // 4
```

(Note that methods such as `StartsWith` use *ordinal* comparison, whereas the corresponding methods on the `string` class use culture-sensitive comparison by default.)

Methods such as `ToUpper` and `ToLower` are available, but you must pass in a destination span with the correct length (this allows you to decide how and where to allocate the memory).

Some of `string`'s methods are unavailable, such as `Split` (which splits a string into an array of words). It's actually impossible to write the direct equivalent of `string`'s `Split` method because you cannot create an array of spans.

> **NOTE**
>
> This is because spans are defined as *ref structs*, which can exist only on the stack.
>
> (By "exist only on the stack," we mean that the struct itself can exist only on the stack. The content that the span *wraps* can—and does, in this case—exist on the heap.)

The `System.Buffers.Text` namespace contains additional types to help you work with span-based text, including the following:

- `Utf8Formatter.TryFormat` does the equivalent of calling `ToString` on built-in and simple types such as `decimal`, `DateTime`, and so on but writes to a span instead of a string.
- `Utf8Parser.TryParse` does the reverse and parses data from a span into a simple type.
- The `Base64` type provides methods for reading/writing base-64 data.

> **NOTE**
>
> From .NET 8, the .NET numeric and date/time types (as well as other simple types) allow direct formatting and parsing of UTF-8, via new `TryFormat` and `Parse`/`TryParse` methods that operate on a `Span<byte>`. The new methods are defined in the `IUtf8SpanFormattable` and `IUtf8SpanParsable<TSelf>` interfaces (the latter leverages C# 12's ability to define static abstract interface members).

Fundamental CLR methods such as `int.Parse` have also been overloaded to accept `ReadOnlySpan<char>`.

Memory<T>

`Span<T>` and `ReadOnlySpan<T>` are defined as *ref structs* to maximize their optimization potential as well as allowing them to work safely with stack-allocated memory (as you'll see in the next section). However, it also imposes limitations. In addition to being array-unfriendly, you cannot use them as fields in a class (this would put them on the heap). This, in turn, prevents them from appearing in lambda expressions—and as parameters in asynchronous methods, iterators, and asynchronous streams:

```
async void Foo (Span<int> notAllowed)   // Compile-time error!
```

(Remember that the compiler processes asynchronous methods and iterators by writing a private *state machine*, which means that any parameters and local variables end up as fields. The same applies to lambda expressions that close over variables: these also end up as fields in a *closure*.)

The `Memory<T>` and `ReadOnlyMemory<T>` structs work around this, acting as spans that cannot wrap stack-allocated memory, allowing their use in fields, lambda expressions, asynchronous methods, and so on.

You can obtain a `Memory<T>` or `ReadOnlyMemory<T>` from an array via an implicit conversion or the `AsMemory()` extension method:

```
Memory<int> mem1 = new int[] { 1, 2, 3 };
var mem2 = new int[] { 1, 2, 3 }.AsMemory();
```

You can easily "convert" a `Memory<T>` or `ReadOnlyMemory<T>` into a `Span<T>` or `ReadOnlySpan<T>` via its `Span` property so that you can interact with it as though it were a span. The conversion is efficient in that it doesn't perform any copying:

```
async void Foo (Memory<int> memory)
{
  Span<int> span = memory.Span;
  ...
}
```

(You can also directly slice a `Memory<T>` or `ReadOnlyMemory<T>` via its `Slice` method or a C# range, and access its length via its `Length` property.)

> **NOTE**
>
> Another way to obtain a `Memory<T>` is to rent it from a *pool*, using the `System.Buffers.MemoryPool<T>` class. This works just like array pooling (see "Array Pooling") and offers another strategy for reducing the load on the garbage collector.

We said in the previous section that you cannot write the direct equivalent of `string.Split` for spans, because you cannot create an array of spans. This limitation does not apply to `ReadOnlyMemory<char>`:

```
// Split a string into words:
IEnumerable<ReadOnlyMemory<char>> Split (ReadOnlyMemory<char> input)
{
  int wordStart = 0;
  for (int i = 0; i <= input.Length; i++)
    if (i == input.Length || char.IsWhiteSpace (input.Span [i]))
    {
      yield return input [wordStart..i];   // Slice with C# range operator
      wordStart = i + 1;
    }
}
```

This is more efficient than string's `Split` method: instead of creating new strings for each word, it returns *slices* of the original string:

```
foreach (var slice in Split ("The quick brown fox jumps over the lazy dog"))
{
  // slice is a ReadOnlyMemory<char>
}
```

> **NOTE**
>
> You can easily convert a `Memory<T>` into a `Span<T>` (via the `Span` property), but not vice versa. For this reason, it's better to write methods that accept `Span<T>` than `Memory<T>` when you have a choice.
>
> For the same reason, it's better to write methods that accept `ReadOnlySpan<T>` rather than `Span<T>`.

Forward-Only Enumerators

In the preceding section, we employed `ReadOnlyMemory<char>` as a solution to implementing a string-style `Split` method. But by giving up on `ReadOnlySpan<char>`, we lost the ability to slice spans backed by unmanaged memory. Let's revisit `ReadOnlySpan<char>` to see whether we can find another solution.

One possible option would be to write our `Split` method so that it returns *ranges*:

```
Range[] Split (ReadOnlySpan<char> input)
{
  int pos = 0;
  var list = new List<Range>();
  for (int i = 0; i <= input.Length; i++)
    if (i == input.Length || char.IsWhiteSpace (input [i]))
    {
      list.Add (new Range (pos, i));
      pos = i + 1;
    }
  return list.ToArray();
}
```

The caller could then use those ranges to slice the original span:

```
ReadOnlySpan<char> source = "The quick brown fox";
foreach (Range range in Split (source))
{
  ReadOnlySpan<char> wordSpan = source [range];
```

```
  ...
}
```

This is an improvement, but it's still imperfect. One of the reasons for using spans in the first place is to avoid memory allocations. But notice that our `Split` method creates a `List<Range>`, adds items to it, and then converts the list into an array. This incurs *at least* two memory allocations as well a memory-copy operation.

The solution to this is to eschew the list and array in favor of a forward-only enumerator. An enumerator is clumsier to work with, but it can be made allocation-free with the use of structs:

```
// We must define this as a ref struct, because _input is a ref struct.
public readonly ref struct CharSpanSplitter
{
  readonly ReadOnlySpan<char> _input;
  public CharSpanSplitter (ReadOnlySpan<char> input) => _input = input;
  public Enumerator GetEnumerator() => new Enumerator (_input);

  public ref struct Enumerator   // Forward-only enumerator
  {
    readonly ReadOnlySpan<char> _input;
    int _wordPos;
    public ReadOnlySpan<char> Current { get; private set; }

    public Rator (ReadOnlySpan<char> input)
    {
      _input = input;
      _wordPos = 0;
      Current = default;
    }

    public bool MoveNext()
    {
      for (int i = _wordPos; i <= _input.Length; i++)
        if (i == _input.Length || char.IsWhiteSpace (_input [i]))
        {
          Current = _input [_wordPos..i];
          _wordPos = i + 1;
          return true;
        }
      return false;
    }
```

```
  }
}

public static class CharSpanExtensions
{
  public static CharSpanSplitter Split (this ReadOnlySpan<char> input)
    => new CharSpanSplitter (input);

  public static CharSpanSplitter Split (this Span<char> input)
    => new CharSpanSplitter (input);
}
```

Here’s how you would call it:

```
var span = "the quick brown fox".AsSpan();
foreach (var word in span.Split())
{
  // word is a ReadOnlySpan<char>
}
```

By defining a `Current` property and a `MoveNext` method, our enumerator can work with C#’s `foreach` statement (see “Enumeration”). We don’t have to implement the `IEnumerable<T>`/`IEnumerator<T>` interfaces (in fact, we can’t; ref structs can’t implement interfaces). We’re sacrificing abstraction for micro-optimization.

Working with Stack-Allocated and Unmanaged Memory

Another effective micro-optimization technique is to reduce the load on the garbage collector by minimizing heap-based allocations. This means making greater use of stack-based memory—or even unmanaged memory.

Unfortunately, this normally requires that you rewrite code to use pointers. In the case of our previous example that summed elements in an array, we would need to write another version as follows:

```
unsafe int Sum (int* numbers, int length)
{
```

```
  int total = 0;
  for (int i = 0; i < length; i++) total += numbers [i];
  return total;
}
```

so that we could do this:

```
int* numbers = stackalloc int [1000];   // Allocate array on the stack
int total = Sum (numbers, 1000);
```

Spans solve this problem: you can construct a `Span<T>` or `ReadOnlySpan<T>` directly from a pointer:

```
int* numbers = stackalloc int [1000];
var span = new Span<int> (numbers, 1000);
```

Or in one step:

```
Span<int> numbers = stackalloc int [1000];
```

(Note that this doesn't require the use of `unsafe`). Recall the `Sum` method that we wrote previously:

```
int Sum (ReadOnlySpan<int> numbers)
{
  int total = 0;
  int len = numbers.Length;
  for (int i = 0; i < len; i++) total += numbers [i];
  return total;
}
```

This method works equally well for a stack-allocated span. We have gained on three counts:

- The same method works with both arrays and stack-allocated memory
- We can use stack-allocated memory with minimal use of pointers
- The span can be sliced

> **NOTE**
>
> The compiler is smart enough to prevent you from writing a method that allocates memory on the stack and returns it to the caller via Span<T> or ReadOnlySpan<T>.
>
> (In other scenarios, however, you can legally return a Span<T> or ReadOnlySpan<T>.)

You can also use spans to wrap memory that you allocate from the unmanaged heap. In the following example, we allocate unmanaged memory using the Marshal.AllocHGlobal function, wrap it in a Span<char>, and then copy a string into the unmanaged memory. Finally, we employ the CharSpanSplitter struct that we wrote in the preceding section to split the unmanaged string into words:

```
var source = "The quick brown fox".AsSpan();
var ptr = Marshal.AllocHGlobal (source.Length * sizeof (char));
try
{
  var unmanaged = new Span<char> ((char*)ptr, source.Length);
  source.CopyTo (unmanaged);
  foreach (var word in unmanaged.Split())
    Console.WriteLine (word.ToString());
}
finally { Marshal.FreeHGlobal (ptr); }
```

A nice bonus is that Span<T>'s indexer performs bounds-checking, preventing a buffer overrun. This protection applies if you correctly instantiate Span<T>: in our example, you would lose this protection if you wrongly obtained the span:

```
var span = new Span<char> ((char*)ptr, source.Length * 2);
```

There's also no protection from the equivalent of a dangling pointer, so you must take care not to access the span after releasing its unmanaged memory with Marshal.FreeHGlobal.