Chapter 23. Span<T> and Memory<T>

The `Span<T>` and `Memory<T>` structs act as low-level façades over an array, string, or any contiguous block of managed or unmanaged memory. Their main purpose is to help with certain kinds of micro-optimization—in particular, writing *low-allocation* code that minimizes managed memory allocations (thereby reducing the load on the garbage collector), without having to duplicate your code for different kinds of input. They also enable *slicing*—working with a portion of an array, string, or memory block without creating a copy.

`Span<T>` and `Memory<T>` are particularly useful in performance hotspots, such as the ASP.NET Core processing pipeline, or a JSON parser that serves an object database.

NOTE

Should you come across these types in an API and not need or care for their potential performance advantages, you can deal with them easily as follows:

- When calling a method that expects a `Span<T>`, `ReadOnlySpan<T>`, `Memory<T>`, or `ReadOnlyMemory<T>`, pass in an array instead; that is, `T[]`. (This works thanks to implicit conversion operators.)
- To convert from a span/memory *to* an array, call the `ToArray` method. And if `T` is `char`, `ToString` will convert the span/memory into a string.

From C# 12, you can also use collection initializers to create spans.

Specifically, `Span<T>` does two things:

- It provides a common array-like interface over managed arrays, strings, and pointer-backed memory. This gives you the freedom to employ stack-allocated and unmanaged memory to avoid garbage collection, without having to duplicate code or mess with pointers.
- It allows “slicing”: exposing reusable subsections of the span without making copies.

> **NOTE**
>
> `Span<T>` comprises just two fields, a pointer and a length. For this reason, it can represent only contiguous blocks of memory. (Should you need to work with noncontiguous memory, the `ReadOnlySequence<T>` class is available to serve as a linked list.)

Because `Span<T>` can wrap stack-allocated memory, there are restrictions on how you can store or pass around instances (imposed, in part, by `Span<T>` being a *ref struct*). `Memory<T>` acts as a span without those restrictions, but it cannot wrap stack-allocated memory. `Memory<T>` still provides the benefit of slicing.

Each struct comes with a read-only counterpart (`ReadOnlySpan<T>` and `ReadOnlyMemory<T>`). As well as preventing unintentional change, the read-only counterparts further improve performance by allowing the compiler and runtime additional freedom for optimization.

.NET itself (and ASP.NET Core) use these types to improve efficiency with I/O, networking, string handling, and JSON parsing.

> **NOTE**
>
> `Span<T>` and `Memory<T>`’s ability to perform array slicing make the old `ArraySegment<T>` class redundant. To help with any transition, there are implicit conversion operators from `ArraySegment<T>` to all of the span/memory structs, and from `Memory<T>` and `ReadOnlyMemory<T>` to `ArraySegment<T>`.