Structs: Value Types with Class-like Capabilities

A **struct** (short for "structure") is a **value type**, which is its most significant difference from a class (a reference type). This distinction impacts how structs are stored in memory and how they behave when copied.

Key characteristics of structs:

- **Value Type Semantics:** When you assign a struct variable to another, or pass it to a method, the *entire value* of the struct is copied. This is in contrast to reference types, where only the *reference* (memory address) is copied.
- **No Inheritance (Except `object`/`ValueType`):** Structs cannot explicitly inherit from other structs or classes (except implicitly from `System.Object`, or more precisely, `System.ValueType`). This means they don't participate in typical inheritance hierarchies. Consequently, their members cannot be marked as `virtual`, `abstract`, or `protected` because there's no subclassing mechanism.
- **No Finalizers:** Structs do not support finalizers, which are used by reference types for cleanup before garbage collection.
- **Members:** A struct can have almost all the same members as a class: fields, methods, properties, constructors, indexers, and events.

When to use a struct?

Structs are appropriate when value-type semantics are desirable. Good use cases include:

- **Numeric types:** For example, `int`, `decimal`, `DateTime` are all structs. It's more natural for `int x = 5; int y = x;` to copy the value `5` rather than a reference.
- **Small, immutable data types:** When you have a small amount of data that represents a single value (e.g., a `Point`, `Color`, or `Coordinate`), and you want assignment to copy the entire value, a struct is a good fit.
- **Performance optimization:** Because structs are value types, their instances do not require individual heap allocations. When you create many instances (e.g., in an array of structs), this can result in memory and performance savings because the entire array occupies a single contiguous block on the heap.

Nullability: Because structs are value types, an instance of a struct **cannot be null**. The default value for a struct is an "empty" instance, where all its fields are set to their default values (e.g., 0 for numeric fields, false for bool, null for reference type fields within the struct).

Struct Construction Semantics

Prior to C# 11, there were stricter rules about struct initialization. Now, some of these rules have been relaxed, primarily to benefit **record structs**. However, it's still worth understanding the unique aspects of struct construction.

The Default Constructor

Every struct always has an **implicit parameterless constructor**. This constructor performs a bitwise-zeroing of all its fields, effectively setting them to their default values.

```
struct Point { int x, y; }
Point p = new Point(); // p.x and p.y will both be 0 (default int value)
```

Even if you define your own parameterless constructor for a struct, the implicit parameterless constructor *still exists* and can be accessed using the default keyword:

```
struct Point
{
    int x = 1;      // Field initializer
    int y;
    public Point() => y = 1; // Explicit parameterless constructor
}

Point p1 = new Point();   // Calls explicit constructor: p1.x will be 1, p1.y
will be 1
Point p2 = default;       // Calls implicit default constructor: p2.x will be
0, p2.y will be 0
```

This dual-constructor behavior can sometimes lead to confusion. It's often a good practice to design structs so that their default value (all fields zeroed) is a valid and usable state. This can make explicit initializers or parameterless constructors less necessary.

The implicit default constructor is also invoked when:

- An array of structs is created (`var points = new Point[10];` will create an array where each `Point` is (0,0)).
- A class contains a struct as a field, and the class is instantiated without explicitly assigning a value to that struct field.

Read-Only Structs and Functions

You can enforce immutability within structs:

`readonly` Structs

Applying the `readonly` modifier to a struct declaration ensures that **all its fields must be `readonly`**. This provides a strong guarantee of immutability and allows the compiler to perform more optimizations.

```
readonly struct Point
{
    public readonly int X, Y; // Fields must be readonly
}
// Any attempt to modify X or Y after construction will result in a
compile-time error.
```

`readonly` Functions (C# 8+)

From C# 8, you can apply the `readonly` modifier to individual **struct functions** (methods, properties, indexers). This ensures that the function does not modify any fields of the struct. If a `readonly` function attempts to modify a field, a compile-time error is generated.

```
struct Point
{
    public int X, Y;
    public readonly void ResetX() => X = 0; // Error! Cannot modify X in a
readonly function
}
```

If a `readonly` function calls a non-`readonly` function, the compiler generates a warning and defensively creates a copy of the struct to ensure no mutation occurs.

Ref Structs (C# 7.2+)

Ref structs are a specialized feature introduced in C# 7.2, primarily for advanced performance optimizations in scenarios like `System.Span<T>` and `System.ReadOnlySpan<T>`.

Memory Location:

- **Reference types** (classes) always reside on the **heap**.
- **Value types** (structs) typically reside **in-place**:
    - If declared as a local variable or parameter, they reside on the **stack**.
    - If declared as a field within a class, they reside on the **heap** (as part of the class instance).
    - Arrays of structs and boxed structs also reside on the **heap**.

Adding the **`ref` modifier** to a struct's declaration (`ref struct MyStruct { ... }`) enforces that its instances **can only ever reside on the stack**. This means a `ref struct` cannot be stored on the heap. Any attempt to use a `ref struct` in a way that *could* lead to it being stored on the heap results in a compile-time error:

```
ref struct Point { public int X, Y; }

// var points = new Point[100];            // Error: Arrays live on the heap
// class MyClass { Point P; }               // Error: Class fields live on the
heap
// object obj = new Point();                // Error: Boxing sends to the heap
```

Why ref struct?

The main benefit of ref structs is to safely wrap stack-allocated memory. Span<T> and ReadOnlySpan<T> use this to provide high-performance, memory-safe access to contiguous memory regions without heap allocations.

Limitations of ref structs:

Because ref structs are strictly confined to the stack, they cannot participate in any C# feature that might directly or indirectly cause them to be stored on the heap. This includes:

- **Asynchronous functions (`async`)**
- **Iterators (`yield return`)**
- **Lambda expressions** (these features often involve compiler-generated classes with fields that might capture data on the heap).
- They cannot be fields of non-`ref` structs.
- They cannot implement interfaces (because interface implementations often rely on boxing, which would move the struct to the heap).

Structs, particularly `readonly` and `ref` structs, offer powerful tools for optimizing memory and performance in specific scenarios. Choosing between a class and a struct is a crucial design decision that depends on your type's semantics, size, mutability requirements, and performance considerations.