## **Structs: Value Types with Class-like Capabilities** Struktur: Tipe Nilai dengan Kemampuan Mirip Kelas

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. Sebuah struct (singkatan dari “structure”) adalah tipe nilai, yang merupakan perbedaan utama antara struct dan kelas (tipe referensi). Perbedaan ini memengaruhi cara struct disimpan di memori dan cara berperilaku saat disalin.

Key characteristics of structs:Ciri-ciri utama 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. Semantik Tipe Nilai: Saat Anda mengassign variabel struct ke variabel lain, atau meneruskannya ke metode, seluruh nilai dari struct tersebut akan disalin. Hal ini berbeda dengan tipe referensi, di mana hanya referensi (alamat memori) yang disalin.
* **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. Tidak Ada Pewarisan (Kecuali objek/ValueType): Struktur tidak dapat secara eksplisit mewarisi dari struktur atau kelas lain (kecuali secara implisit dari System.Object, atau lebih tepatnya, System.ValueType). Hal ini berarti mereka tidak berpartisipasi dalam hierarki pewarisan yang umum. Akibatnya, anggota mereka tidak dapat ditandai sebagai virtual, abstrak, atau dilindungi karena tidak ada mekanisme pewarisan subkelas.
* **No Finalizers:** Structs do not support finalizers, which are used by reference types for cleanup before garbage collection. Tidak Ada Finalizer: Struktur tidak mendukung finalizer, yang digunakan oleh tipe referensi untuk pembersihan sebelum pengumpulan sampah.
* **Members:** A struct can have almost all the same members as a class: fields, methods, properties, constructors, indexers, and events. Anggota: Sebuah struct dapat memiliki hampir semua anggota yang sama dengan sebuah kelas: bidang, metode, properti, konstruktor, indeks, dan peristiwa.

When to use a struct?

Structs are appropriate when value-type semantics are desirable. Good use cases include: Struktur (struct) cocok digunakan ketika semantik tipe nilai diinginkan. Contoh penggunaan yang baik meliputi:

* **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. Tipe numerik: Misalnya, int, decimal, dan DateTime semuanya adalah struktur. Lebih alami untuk int x = 5; int y = x; menyalin nilai 5 daripada referensi.
* **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. Tipe data kecil dan tidak dapat diubah: Ketika Anda memiliki sejumlah kecil data yang mewakili nilai tunggal (misalnya, Point, Color, atau Coordinate), dan Anda ingin penugasan menyalin seluruh nilai, struct adalah pilihan yang tepat.
* **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. Optimasi kinerja: Karena struct adalah tipe nilai, instance-nya tidak memerlukan alokasi memori heap secara individual. Saat Anda membuat banyak instance (misalnya, dalam array struct), hal ini dapat menghasilkan penghematan memori dan kinerja karena seluruh array menempati satu blok memori yang berurutan di 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). Nullability: Karena struct adalah tipe nilai, sebuah instance struct tidak dapat bernilai null. Nilai default untuk struct adalah instance “kosong”, di mana semua bidangnya ditetapkan ke nilai default masing-masing (misalnya, 0 untuk bidang numerik, false untuk bool, dan null untuk bidang tipe referensi di dalam struct).

## **Struct Construction Semantics** Semantik Konstruksi Struktur

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. Sebelum C# 11, terdapat aturan yang lebih ketat mengenai inisialisasi struct. Kini, beberapa aturan tersebut telah dilonggarkan, terutama untuk menguntungkan struct record. Namun, tetap penting untuk memahami aspek unik dalam konstruksi struct.

### **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. Setiap struct selalu memiliki konstruktor tanpa parameter yang implisit. Konstruktor ini melakukan pembersihan bitwise dari semua bidangnya, secara efektif mengaturnya ke nilai default masing-masing.

|  |
| --- |
| 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: Meskipun Anda mendefinisikan konstruktor tanpa parameter sendiri untuk sebuah struct, konstruktor tanpa parameter yang implisit tetap ada dan dapat diakses menggunakan kata kunci default:

|  |
| --- |
| 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. Perilaku konstruktor ganda ini terkadang dapat menimbulkan kebingungan. Merupakan praktik yang baik untuk mendesain struktur sedemikian rupa sehingga nilai defaultnya (semua bidang bernilai nol) merupakan keadaan yang valid dan dapat digunakan. Hal ini dapat membuat inisialisasi eksplisit atau konstruktor tanpa parameter menjadi kurang diperlukan.

The implicit default constructor is also invoked when:Konstruktor default implisit juga dipanggil ketika:

* An array of structs is created (var points = new Point[10]; will create an array where each Point is (0,0)). Sebuah array dari struct dibuat (var points = new Point[10]; akan membuat array di mana setiap Point adalah (0,0)).
* A class contains a struct as a field, and the class is instantiated without explicitly assigning a value to that struct field. Sebuah kelas mengandung struktur sebagai bidang, dan kelas tersebut diinstansiasi tanpa secara eksplisit menetapkan nilai ke bidang struktur tersebut.

## **Read-Only Structs and FunctionsStruktur dan Fungsi Hanya Baca**

You can enforce immutability within structs: Anda dapat menerapkan keimmutabilitas dalam 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. Menggunakan modifikator readonly pada deklarasi struct memastikan bahwa semua bidangnya harus bersifat readonly. Hal ini memberikan jaminan yang kuat terhadap ketidakubahannya dan memungkinkan compiler untuk melakukan optimasi yang lebih banyak.

|  |
| --- |
| 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.  
Mulai dari C# 8, Anda dapat menerapkan modifikator readonly pada fungsi-fungsi struktural (metode, properti, indeks) secara individual. Hal ini memastikan bahwa fungsi tersebut tidak mengubah bidang-bidang struktural. Jika fungsi readonly mencoba mengubah bidang, kesalahan kompilasi akan dihasilkan.

|  |
| --- |
| 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.Jika fungsi readonly memanggil fungsi non-readonly, compiler akan menghasilkan peringatan dan secara defensif membuat salinan struktur untuk memastikan tidak terjadi mutasi.

## **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>. Ref structs adalah fitur khusus yang diperkenalkan dalam C# 7.2, terutama untuk optimasi kinerja tingkat lanjut dalam skenario seperti System.Span<T> dan System.ReadOnlySpan<T>.

**Memory Location:**

* **Reference types** (classes) always reside on the **heap**.Tipe referensi (kelas) selalu berada di heap.
* **Value types** (structs) typically reside **in-place**:Tipe nilai (struct) biasanya berada di tempat:
  + If declared as a local variable or parameter, they reside on the **stack**.Jika dinyatakan sebagai variabel lokal atau parameter, mereka berada di tumpukan.
  + If declared as a field within a class, they reside on the **heap** (as part of the class instance).Jika dinyatakan sebagai bidang dalam suatu kelas, mereka berada di heap (sebagai bagian dari instance kelas).
  + Arrays of structs and boxed structs also reside on the **heap**.Array dari struct dan struct yang dibungkus juga berada di 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: Menambahkan modifikator ref ke deklarasi struct (ref struct MyStruct { ... }) memastikan bahwa instansinya hanya dapat berada di tumpukan. Ini berarti struct ref tidak dapat disimpan di tumpukan. Setiap upaya untuk menggunakan struct ref dengan cara yang dapat menyebabkan penyimpanannya di tumpukan akan menghasilkan kesalahan kompilasi:

|  |
| --- |
| 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.Keuntungan utama dari ref structs adalah untuk membungkus memori yang dialokasikan di tumpukan dengan aman. Span<T> dan ReadOnlySpan<T> menggunakan ini untuk menyediakan akses yang aman dan berperforma tinggi ke wilayah memori yang berurutan tanpa alokasi memori di tumpukan.

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:Karena ref structs secara ketat dibatasi pada stack, mereka tidak dapat berpartisipasi dalam fitur C# apa pun yang secara langsung atau tidak langsung menyebabkan mereka disimpan di heap. Hal ini termasuk:

* **Asynchronous functions (async)Fungsi asinkron (async) dua atau lebih proses tidak dieksekusi diwaktu yang sama. Dalam pemrograman memungkinkan suatu tugas dimulai tanpa harus menunggu tugas lain selesai**
* **Iterators (yield return)**Iterator (yield return)
* **Lambda expressions** (these features often involve compiler-generated classes with fields that might capture data on the heap).Ekspresi lambda (fitur-fitur ini sering melibatkan kelas yang dihasilkan oleh compiler dengan bidang yang mungkin menyimpan data di heap).
* They cannot be fields of non-ref structs.Mereka tidak dapat berupa bidang dari struktur non-ref.
* They cannot implement interfaces (because interface implementations often rely on boxing, which would move the struct to the heap).Mereka tidak dapat mengimplementasikan antarmuka (karena implementasi antarmuka sering kali bergantung pada boxing, yang akan memindahkan struktur ke 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.Struktur data, terutama struktur data readonly dan ref, menyediakan alat yang kuat untuk mengoptimalkan penggunaan memori dan kinerja dalam skenario tertentu. Memilih antara kelas dan struktur data merupakan keputusan desain yang penting, yang bergantung pada semantik tipe data, ukuran, persyaratan mutabilitas, dan pertimbangan kinerja.