# Structure types (C# reference)

Article • 04/10/2023

A *structure type* (or *struct type*) is a value type that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type:

```C#
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

For information about `ref struct` and `readonly ref struct` types, see the ref structure types article.

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. For structure-type variables, an instance of the type is copied. For more information, see Value types.

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both integer and real), a Boolean value, a Unicode character, a time instance. If you're focused on the behavior of a type, consider defining a class. Class types have *reference semantics*. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Because structure types have value semantics, we recommend you define *immutable* structure types.

## `readonly` struct

You use the `readonly` modifier to declare that a structure type is immutable. All data members of a `readonly` struct must be read-only as follows:

- Any field declaration must have the readonly modifier
- Any property, including auto-implemented ones, must be read-only. In C# 9.0 and later, a property may have an init accessor.

That guarantees that no member of a `readonly` struct modifies the state of the struct. That means that other instance members except constructors are implicitly readonly.

**Note**

In a `readonly` struct, a data member of a mutable reference type still can mutate its own state. For example, you can't replace a **List<T>** instance, but you can add new elements to it.

The following code defines a `readonly` struct with init-only property setters, available in C# 9.0 and later:

C#

```csharp
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

## `readonly` instance members

You can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct. If you can't declare the whole structure type as `readonly`, use the `readonly` modifier to mark the instance members that don't modify the state of the struct.

Within a `readonly` instance member, you can't assign to structure's instance fields. However, a `readonly` member can call a non-`readonly` member. In that case, the compiler creates a copy of the structure instance and calls the non-`readonly` member on that copy. As a result, the original structure instance isn't modified.

Typically, you apply the `readonly` modifier to the following kinds of instance members:

- methods:

  C#

  ```csharp
  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  You can also apply the `readonly` modifier to methods that override methods declared in System.Object:

  C#

  ```csharp
  public readonly override string ToString() => $"({X}, {Y})";
  ```

- properties and indexers:

  C#

  ```csharp
  private int counter;
  public int Counter
  {
      readonly get => counter;
      set => counter = value;
  }
  ```

  If you need to apply the `readonly` modifier to both accessors of a property or indexer, apply it in the declaration of the property or indexer.

  ⓘ **Note**

  The compiler declares a `get` accessor of an **auto-implemented property** as `readonly`, regardless of presence of the `readonly` modifier in a property declaration.

In C# 9.0 and later, you may apply the `readonly` modifier to a property or indexer with an `init` accessor:

```C#
public readonly double X { get; init; }
```

You can apply the `readonly` modifier to static fields of a structure type, but not any other static members, such as properties or methods.

The compiler may make use of the `readonly` modifier for performance optimizations. For more information, see Avoiding allocations.

## Nondestructive mutation

Beginning with C# 10, you can use the with expression to produce a copy of a structure-type instance with the specified properties and fields modified. You use object initializer syntax to specify what members to modify and their new values, as the following example shows:

```C#
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1);  // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2);  // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3);  // output: (1, 4)
}
```

## `record` struct

Beginning with C# 10, you can define record structure types. Record types provide built-in functionality for encapsulating data. You can define both `record struct` and `readonly record struct` types. A record struct can't be a ref struct. For more information and examples, see Records.

## Inline arrays

Beginning with C# 12, you can declare *inline arrays* as a `struct` type:

C#

```csharp
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
    private char _firstElement;
}
```

An inline array is a structure that contains a contiguous block of N elements of the same type. It's a safe-code equivalent of the fixed buffer declaration available only in unsafe code. An inline array is a `struct` with the following characteristics:

- It contains a single field.
- The struct doesn't specify an explicit layout.

In addition, the compiler validates the System.Runtime.CompilerServices.InlineArrayAttribute attribute:

- The length must be greater than zero (`> 0`).
- The target type must be a struct.

In most cases, an inline array can be accessed like an array, both to read and write values. In addition, you can use the range and index operators.

There are minimal restrictions on the type of the single field. It can't be a pointer type, but it can be any reference type, or any value type. You can use inline arrays with almost any C# data structure.

Inline arrays are an advanced language feature. They're intended for high-performance scenarios where an inline, contiguous block of elements is faster than other alternative data structures. You can learn more about inline arrays from the feature speclet

## Struct initialization and default values

A variable of a `struct` type directly contains the data for that `struct`. That creates a distinction between an uninitialized `struct`, which has its default value and an initialized `struct`, which stores values set by constructing it. For example consider the following code:

C#

```csharp
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1);  // output: NaN (Undefined)

    var m2 = default(Measurement);
    Console.WriteLine(m2);  // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms));  // output: 0 (), 0 ()
}
```

As the preceding example shows, the default value expression ignores a parameterless constructor and produces the default value of the structure type. Structure-type array instantiation also ignores a parameterless constructor and produces an array populated with the default values of a structure type.

The most common situation where you see default values is in arrays or in other collections where internal storage includes blocks of variables. The following example creates an array of 30 `TemperatureRange` structures, each of which has the default value:

```C#
// All elements have default values of 0:
TemperatureRange[] lastMonth = new TemperatureRange[30];
```

All of a struct's member fields must be *definitely assigned* when it's created because `struct` types directly store their data. The `default` value of a struct has *definitely assigned* all fields to 0. All fields must be definitely assigned when a constructor is invoked. You initialize fields using the following mechanisms:

- You can add *field initializers* to any field or auto implemented property.
- You can initialize any fields, or auto properties, in the body of the constructor.

Beginning with C# 11, if you don't initialize all fields in a struct, the compiler adds code to the constructor that initializes those fields to the default value. The compiler performs its usual definite assignment analysis. Any fields that are accessed before being assigned, or not definitely assigned when the constructor finishes executing are assigned their default values before the constructor body executes. If `this` is accessed before all fields are assigned, the struct is initialized to the default value before the constructor body executes.

```C#
public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public Measurement(string description)
    {
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary
measurement";
```

```
        public override string ToString() => $"{Value} ({Description})";
    }

    public static void Main()
    {
        var m1 = new Measurement(5);
        Console.WriteLine(m1);  // output: 5 (Ordinary measurement)

        var m2 = new Measurement();
        Console.WriteLine(m2);  // output: 0 ()

        var m3 = default(Measurement);
        Console.WriteLine(m3);  // output: 0 ()
    }
```

Every `struct` has a `public` parameterless constructor. If you write a parameterless constructor, it must be public. If a struct declares any field initializers, it must explicitly declare a constructor. That constructor need not be parameterless. If a struct declares a field initializer but no constructors, the compiler reports an error. Any explicitly declared constructor (with parameters, or parameterless) executes all field initializers for that struct. All fields without a field initializer or an assignment in a constructor are set to the default value. For more information, see the Parameterless struct constructors feature proposal note.

Beginning with C# 12, `struct` types can define a primary constructor as part of its declaration. Primary constructors provides a concise syntax for constructor parameters that can be used throughout the `struct` body, in any member declaration for that struct.

If all instance fields of a structure type are accessible, you can also instantiate it without the `new` operator. In that case you must initialize all instance fields before the first use of the instance. The following example shows how to do that:

C#

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

In the case of the built-in value types, use the corresponding literals to specify a value of the type.

## Limitations with the design of a structure type

Structs have most of the capabilities of a class type. There are some exceptions, and some exceptions that have been removed in more recent versions:

- A structure type can't inherit from other class or structure type and it can't be the base of a class. However, a structure type can implement interfaces.
- You can't declare a finalizer within a structure type.
- Prior to C# 11, a constructor of a structure type must initialize all instance fields of the type.
- Prior to C# 10, you can't declare a parameterless constructor.
- Prior to C# 10, you can't initialize an instance field or property at its declaration.

## Passing structure-type variables by reference

When you pass a structure-type variable to a method as an argument or return a structure-type value from a method, the whole instance of a structure type is copied. Pass by value can affect the performance of your code in high-performance scenarios that involve large structure types. You can avoid value copying by passing a structure-type variable by reference. Use the ref, out, or in method parameter modifiers to indicate that an argument must be passed by reference. Use ref returns to return a method result by reference. For more information, see Avoid allocations.

## struct constraint

You also use the `struct` keyword in the struct constraint to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct` constraint.

## Conversions

For any structure type (except ref struct types), there exist boxing and unboxing conversions to and from the System.ValueType and System.Object types. There exist also boxing and unboxing conversions between a structure type and any interface that it implements.

## C# language specification

For more information, see the Structs section of the C# language specification.

For more information about `struct` features, see the following feature proposal notes:

- C# 7.2 - Readonly structs
- C# 8 - Readonly instance members
- C# 10 - Parameterless struct constructors
- C# 10 - Allow with expression on structs
- C# 10 - Record structs
- C# 11 - Auto default structs

## See also

- C# reference
- The C# type system
- Design guidelines - Choosing between class and struct
- Design guidelines - Struct design