`ref` structure types (C# reference)

Article • 06/23/2023

You can use the `ref` modifier in the declaration of a structure type. Instances of a `ref struct` type are allocated on the stack and can't escape to the managed heap. To ensure that, the compiler limits the usage of `ref struct` types as follows:

- A `ref struct` can't be the element type of an array.
- A `ref struct` can't be a declared type of a field of a class or a non-`ref struct`.
- A `ref struct` can't implement interfaces.
- A `ref struct` can't be boxed to System.ValueType or System.Object.
- A `ref struct` can't be a type argument.
- A `ref struct` variable can't be captured by a lambda expression or a local function.
- A `ref struct` variable can't be used in an async method. However, you can use `ref struct` variables in synchronous methods, for example, in methods that return Task or Task<TResult>.
- A `ref struct` variable can't be used in iterators.

You can define a disposable `ref struct`. To do that, ensure that a `ref struct` fits the disposable pattern. That is, it has an instance `Dispose` method, which is accessible, parameterless and has a `void` return type. You can use the using statement or declaration with an instance of a disposable `ref struct`.

Typically, you define a `ref struct` type when you need a type that also includes data members of `ref struct` types:

C#

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

To declare a `ref struct` as `readonly`, combine the `readonly` and `ref` modifiers in the type declaration (the `readonly` modifier must come before the `ref` modifier):

C#

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> val-
ues)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

In .NET, examples of a `ref struct` are System.Span<T> and System.ReadOnlySpan<T>.

`ref` fields

Beginning with C# 11, you can declare a `ref` field in a `ref struct`, as the following example shows:

C#

```
public ref struct RefFieldExample
{
    private ref int number;

    public int GetNumber()
    {
        if (System.Runtime.CompilerServices.Unsafe.IsNullRef(ref num-
ber))
        {
            throw new InvalidOperationException("The number ref field
is not initialized.");
        }

        return number;
    }
}
```

A `ref` field may have the `null` value. Use the Unsafe.IsNullRef<T>(T) method to determine if a `ref` field is `null`.

You can apply the `readonly` modifier to a `ref` field in the following ways:

- `readonly ref`: You can ref reassign such a field with the `= ref` operator only inside a constructor or an init accessor. You can assign a value with the `=` operator at any point allowed by the field access modifier.

- `ref readonly`: At any point, you cannot assign a value with the `=` operator to such a field. However, you can ref reassign a field with the `= ref` operator.
- `readonly ref readonly`: You can only ref reassign such a field in a constructor or an `init` accessor. At any point, you cannot assign a value to the field.

The compiler ensures that a reference stored in a `ref` field doesn't outlive its referent.

The `ref` fields feature enables a safe implementation of types like System.Span<T>:

C#

```
public readonly ref struct Span<T>
{
    internal readonly ref T _reference;
    private readonly int _length;

    // Omitted for brevity...
}
```

The `Span<T>` type stores a reference through which it accesses the contiguous elements in memory. The use of a reference enables a `Span<T>` instance to avoid copying the storage it refers to.

C# language specification

For more information, see the following sections of the C# language specification:

- Structs: Ref modifier
- Safe context constraint for ref struct types

For more information about `ref` fields, see the Low-level struct improvements proposal note.

See also

- C# reference
- The C# type system