Declaration statements

Article • 06/21/2023

A declaration statement declares a new local variable, local constant, or local reference variable. To declare a local variable, specify its type and provide its name. You can declare multiple variables of the same type in one statement, as the following example shows:

C#

```
string greeting;
int a, b, c;
List<double> xs;
```

In a declaration statement, you can also initialize a variable with its initial value:

C#

```
string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

The preceding examples explicitly specify the type of a variable. You can also let the compiler infer the type of a variable from its initialization expression. To do that, use the `var` keyword instead of a type's name. For more information, see the Implicitly-typed local variables section.

To declare a local constant, use the const keyword, as the following example shows:

C#

```
const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

When you declare a local constant, you must also initialize it.

For information about local reference variables, see the Reference variables section.

Implicitly-typed local variables

When you declare a local variable, you can let the compiler infer the type of the variable from the initialization expression. To do that use the `var` keyword instead of the name

of a type:

```
C#
```

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType());  // output: System.String

var a = 32;
Console.WriteLine(a.GetType());  // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType());  // output:
System.Collections.Generic.List`1[System.Double]
```

As the preceding example shows, implicitly-typed local variables are strongly typed.

> **Note**
>
> When you use `var` in the enabled **nullable aware context** and the type of an initialization expression is a reference type, the compiler always infers a **nullable** reference type even if the type of an initialization expression isn't nullable.

A common use of `var` is with a constructor invocation expression. The use of `var` allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

```
C#
```

```
var xs = new List<int>();
```

Beginning with C# 9.0, you can use a target-typed new expression as an alternative:

```
C#
```

```
List<int> xs = new();
List<int>? ys = new();
```

When you work with anonymous types, you must use implicitly-typed local variables. The following example shows a query expression that uses an anonymous type to hold a customer's name and phone number:

```
C#
```

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
```

```
                    select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone=
{customer.Phone}");
}
```

In the preceding example, you can't explicitly specify the type of the `fromPhoenix` variable. The type is IEnumerable<T> but in this case `T` is an anonymous type and you can't provide its name. That's why you need to use `var`. For the same reason, you must use `var` when you declare the `customer` iteration variable in the `foreach` statement.

For more information about implicitly-typed local variables, see Implicitly-typed local variables.

In pattern matching, the `var` keyword is used in a var pattern.

Reference variables

When you declare a local variable and add the `ref` keyword before the variable's type, you declare a *reference variable*, or a `ref` local:

C#

```
ref int alias = ref variable;
```

A reference variable is a variable that refers to another variable, which is called the *referent*. That is, a reference variable is an *alias* to its referent. When you assign a value to a reference variable, that value is assigned to the referent. When you read the value of a reference variable, the referent's value is returned. The following example demonstrates that behavior:

C#

```
int a = 1;
ref int alias = ref a;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a,
alias) is (1, 1)

a = 2;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a,
alias) is (2, 2)

alias = 3;
```

```
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a,
alias) is (3, 3)
```

Use the ref assignment operator `= ref` to change the referent of a reference variable, as the following example shows:

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = { 0, 0, 0 };
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

In the preceding example, the `element` reference variable is initialized as an alias to the first array element. Then it's `ref` reassigned to refer to the last array element.

You can define a `ref readonly` local variable. You can't assign a value to a `ref readonly` variable. However you can `ref` reassign such a reference variable, as the following example shows:

C#

```
int[] xs = { 1, 2, 3 };

ref readonly int element = ref xs[0];
// element = 100;  error CS0131: The left-hand side of an assignment
must be a variable, property or indexer
Console.WriteLine(element);  // output: 1

element = ref xs[^1];
Console.WriteLine(element);  // output: 3
```

You can assign a reference return to a reference variable, as the following example shows:

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = { 1, 30, 7, 1557, 381, 63, 1027,
2550, 511, 1023 };

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence:  {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence:  1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

In the preceding example, the `GetReferenceToMax` method is a *returns-by-ref* method. It doesn't return the maximum value itself, but a reference return that is an alias to the array element that holds the maximum value. The `Run` method assigns a reference return to the `max` reference variable. Then, by assigning to `max`, it updates the internal storage of the `store` instance. You can also define a `ref readonly` method. The callers of a `ref readonly` method can't assign a value to its reference return.

The iteration variable of the `foreach` statement can be a reference variable. For more information, see the foreach statement section of the Iteration statements article.

In performance-critical scenarios, the use of reference variables and returns might increase performance by avoiding potentially expensive copy operations.

The compiler ensures that a reference variable doesn't outlive its referent and stays valid for the whole of its lifetime. For more information, see the Ref safe contexts section of the C# language specification.

For information about the `ref` fields, see the ref fields section of the ref structure types article.

scoped ref

The contextual keyword `scoped` restricts the lifetime of a value. The `scoped` modifier restricts the *ref-safe-to-escape* or *safe-to-escape* lifetime, respectively, to the current method. Effectively, adding the `scoped` modifier asserts that your code won't extend the lifetime of the variable.

You can apply `scoped` to a parameter or local variable. The `scoped` modifier may be applied to parameters and locals when the type is a ref struct. Otherwise, the `scoped` modifier may be applied only to local reference variables. That includes local variables declared with the `ref` modifier and parameters declared with the `in`, `ref` or `out` modifiers.

The `scoped` modifier is implicitly added to `this` in methods declared in a `struct`, `out` parameters, and `ref` parameters when the type is a `ref struct`.

C# language specification

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

- Declaration statements
- Reference variables and returns

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

See also

- C# reference
- Object and collection initializers
- ref keyword

- Reduce memory allocations using new C# features
- 'var' preferences (style rules IDE0007 and IDE0008)