What's new in C# 7.3

There are two main themes to the C# 7.3 release. One theme provides features that enable safe code to be as performant as unsafe code. The second theme provides incremental improvements to existing features. In addition, new compiler options were added in this release.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use additional generic constraints.

The following enhancements were made to existing features:

- You can test `==` and `!=` with tuple types.
- You can use expression variables in more locations.
- You may attach attributes to the backing field of auto-implemented properties.
- Method resolution when arguments differ by `in` has been improved.
- Overload resolution now has fewer ambiguous cases.

The new compiler options are:

- `-publicsign` to enable Open Source Software (OSS) signing of assemblies.
- `-pathmap` to provide a mapping for source directories.

The remainder of this article provides details and links to learn more about each of the improvements.

Enabling more performant safe code

You should be able to write C# code safely that performs as well as unsafe code. Safe code avoids classes of errors, such as buffer overruns, stray pointers, and other memory access errors. These new features expand the capabilities of verifiable safe code. Strive to write more of your code using safe constructs. These features make that easier.

Indexing `fixed` fields does not require pinning

Consider this struct:

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

In earlier versions of C#, you needed to pin a variable to access one of the integers that are part of `myFixedField`. Now, the following code compiles in a safe context:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

The variable `p` accesses one element in `myFixedField`. You don't need to declare a separate `int*` variable. Note that you still need an `unsafe` context. In earlier versions of C#, you need to declare a second fixed pointer:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

For more information, see the article on the `fixed` statement.

`ref` local variables may be reassigned

Now, `ref` locals may be reassigned to refer to different instances after being initialized. The following code now compiles:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to
different storage.
```

For more information, see the article on `ref` returns and `ref` locals, and the article on `foreach`.

`stackalloc` arrays support initializers

You've been able to specify the values for elements in an array when you initialize it:

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Now, that same syntax can be applied to arrays that are declared with `stackalloc`:

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

For more information, see the `stackalloc` statement article in the language reference.

More types support the `fixed` statement

The `fixed` statement supported a limited set of types. Starting with C# 7.3, any type that contains a `GetPinnableReference()` method that returns a `ref T` or `ref readonly T` may be `fixed`. Adding this feature means that `fixed` can be used with System.Span<T> and related types.

For more information, see the `fixed` statement article in the language reference.

Enhanced generic constraints

You can now specify the type System.Enum or System.Delegate as base class constraints for a type parameter.

You can also use the new `unmanaged` constraint, to specify that a type parameter must be an **unmanaged type**. An **unmanaged type** is a type that isn't a reference type and doesn't contain any reference type at any level of nesting.

For more information, see the articles on `where` generic constraints and constraints on type parameters.

Make existing features better

The second theme provides improvements to features in the language. These features improve productivity when writing C#.

Tuples support `==` and `!=`

The C# tuple types now support `==` and `!=`. For more information, see the section covering equality in the article on tuples.

Attach attributes to the backing fields for auto-implemented properties

This syntax is now supported:

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

The attribute `SomeThingAboutFieldAttribute` is applied to the compiler generated backing field for `SomeProperty`. For more information, see attributes in the C# programming guide.

`in` method overload resolution tiebreaker

When the `in` argument modifier was added, these two methods would cause an ambiguity:

```
static void M(S arg);
```

```
static void M(in S arg);
```

Now, the by value (first in the preceding example) overload is better than the by readonly reference version. To call the version with the readonly reference argument, you must include the `in` modifier when calling the method.

Note

This was implemented as a bug fix. This no longer is ambiguous even with the language version set to "7.2".

For more information, see the article on the `in` parameter modifier.

Extend expression variables in initializers

The syntax added in C# 7.0 to allow `out` variable declarations has been extended to include field initializers, property initializers, constructor initializers, and query clauses. It enables code such as the following example:

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

Improved overload candidates

In every release, the overload resolution rules get updated to address situations where ambiguous method invocations have an "obvious" choice. This release adds three new rules to help the compiler pick the obvious choice:

1. When a method group contains both instance and static members, the compiler discards the instance members if the method was invoked without an instance receiver or context. The compiler discards the static members if the method was invoked with an instance receiver. When there is no receiver, the compiler includes only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type, the compiler includes both. A static context, where an implicit `this` instance receiver cannot be used, includes the body of members where no `this` is defined, such as static members, as well as places where `this` cannot be used, such as field initializers and constructor-initializers.
2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

You'll only notice this change because you'll find fewer compiler errors for ambiguous method overloads when you are sure which method is better.

New compiler options

New compiler options support new build and DevOps scenarios for C# programs.

Public or Open Source signing

The `-publicsign` compiler option instructs the compiler to sign the assembly using a public key. The assembly is marked as signed, but the signature is taken from the public key. This option enables you to build signed assemblies from open-source projects using a public key.

For more information, see the -publicsign compiler option article.

pathmap

The `-pathmap` compiler option instructs the compiler to replace source paths from the build environment with mapped source paths. The `-pathmap` option controls the source path written by the compiler to PDB files or for the CallerFilePathAttribute.