What's new in C# 11

Article • 03/15/2023

The following features were added in C# 11:

- Raw string literals
- Generic math support
- Generic attributes
- UTF-8 string literals
- Newlines in string interpolation expressions
- List patterns
- File-local types
- Required members
- Auto-default structs
- Pattern match Span<char> on a constant string
- Extended nameof scope
- Numeric IntPtr
- ref fields and scoped ref
- Improved method group conversion to delegate
- Warning wave 7

C# 11 is supported on **.NET 7**. For more information, see C# language versioning.

You can download the latest .NET 7 SDK from the .NET downloads page. You can also download Visual Studio 2022, which includes the .NET 7 SDK.

Note

We're interested in your feedback on these features. If you find issues with any of these new features, create a **new issue** in the **dotnet/roslyn** repository.

Generic attributes

You can declare a generic class whose base class is System.Attribute. This feature provides a more convenient syntax for attributes that require a System.Type parameter. Previously, you'd need to create an attribute that takes a `Type` as its constructor parameter:

C#

```
// Before C# 11:
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
```

And to apply the attribute, you use the typeof operator:

C#

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

Using this new feature, you can create a generic attribute instead:

C#

```
// C# 11 feature:
public class GenericAttribute<T> : Attribute { }
```

Then, specify the type parameter to use the attribute:

C#

```
[GenericAttribute<string>()]
public string Method() => default;
```

You must supply all type parameters when you apply the attribute. In other words, the generic type must be fully constructed. In the example above, the empty parentheses (`(` and `)`) can be omitted as the attribute does not have any arguments.

C#

```
public class GenericType<T>
{
   [GenericAttribute<T>()] // Not allowed! generic attributes must be
fully constructed types.
   public string Method() => default;
}
```

The type arguments must satisfy the same restrictions as the typeof operator. Types that require metadata annotations aren't allowed. For example, the following types aren't allowed as the type parameter:

- `dynamic`
- `string?` (or any nullable reference type)
- `(int X, int Y)` (or any other tuple types using C# tuple syntax).

These types aren't directly represented in metadata. They include annotations that describe the type. In all cases, you can use the underlying type instead:

- `object` for `dynamic`.
- `string` instead of `string?`.
- `ValueTuple<int, int>` instead of `(int X, int Y)`.

Generic math support

There are several language features that enable generic math support:

- `static virtual` members in interfaces
- checked user defined operators
- relaxed shift operators
- unsigned right-shift operator

You can add `static abstract` or `static virtual` members in interfaces to define interfaces that include overloadable operators, other static members, and static properties. The primary scenario for this feature is to use mathematical operators in generic types. For example, you can implement the `System.IAdditionOperators<TSelf, TOther, TResult>` interface in a type that implements `operator +`. Other interfaces define other mathematical operations or well-defined values. You can learn about the new syntax in the article on interfaces. Interfaces that include `static virtual` methods are typically generic interfaces. Furthermore, most will declare a constraint that the type parameter implements the declared interface.

You can learn more and try the feature yourself in the tutorial Explore static abstract interface members, or the Preview features in .NET 6 – generic math blog post.

Generic math created other requirements on the language.

- *unsigned right shift operator*: Before C# 11, to force an unsigned right-shift, you would need to cast any signed integer type to an unsigned type, perform the shift, then cast the result back to a signed type. Beginning in C# 11, you can use the `>>>`, the *unsigned shift operator*.
- *relaxed shift operator requirements*: C# 11 removes the requirement that the second operand must be an `int` or implicitly convertible to `int`. This change

allows types that implement generic math interfaces to be used in these locations.
- *checked and unchecked user defined operators*: Developers can now define `checked` and `unchecked` arithmetic operators. The compiler generates calls to the correct variant based on the current context. You can read more about `checked` operators in the article on Arithmetic operators.

Numeric `IntPtr` and `UIntPtr`

The `nint` and `nuint` types now alias System.IntPtr and System.UIntPtr, respectively.

Newlines in string interpolations

The text inside the `{` and `}` characters for a string interpolation can now span multiple lines. The text between the `{` and `}` markers is parsed as C#. Any legal C#, including newlines, is allowed. This feature makes it easier to read string interpolations that use longer C# expressions, like pattern matching `switch` expressions, or LINQ queries.

You can learn more about the newlines feature in the string interpolations article in the language reference.

List patterns

List patterns extend pattern matching to match sequences of elements in a list or an array. For example, `sequence is [1, 2, 3]` is `true` when the `sequence` is an array or a list of three integers (1, 2, and 3). You can match elements using any pattern, including constant, type, property and relational patterns. The discard pattern (`_`) matches any single element, and the new *range pattern* (`..`) matches any sequence of zero or more elements.

You can learn more details about list patterns in the pattern matching article in the language reference.

Improved method group conversion to delegate

The C# standard on Method group conversions now includes the following item:

> - The conversion is permitted (but not required) to use an existing delegate instance that already contains these references.

Previous versions of the standard prohibited the compiler from reusing the delegate object created for a method group conversion. The C# 11 compiler caches the delegate object created from a method group conversion and reuses that single delegate object. This feature was first available in Visual Studio 2022 version 17.2 as a preview feature, and in .NET 7 Preview 2.

Raw string literals

Raw string literals are a new format for string literals. Raw string literals can contain arbitrary text, including whitespace, new lines, embedded quotes, and other special characters without requiring escape sequences. A raw string literal starts with at least three double-quote (""") characters. It ends with the same number of double-quote characters. Typically, a raw string literal uses three double quotes on a single line to start the string, and three double quotes on a separate line to end the string. The newlines following the opening quote and preceding the closing quote aren't included in the final content:

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

Any whitespace to the left of the closing double quotes will be removed from the string literal. Raw string literals can be combined with string interpolation to include braces in the output text. Multiple `$` characters denote how many consecutive braces start and end the interpolation:

C#

```
var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

The preceding example specifies that two braces start and end an interpolation. The third repeated opening and closing brace are included in the output string.

You can learn more about raw string literals in the article on strings in the programming guide, and the language reference articles on string literals and interpolated strings.

Auto-default struct

The C# 11 compiler ensures that all fields of a `struct` type are initialized to their default value as part of executing a constructor. This change means any field or auto property not initialized by a constructor is automatically initialized by the compiler. Structs where the constructor doesn't definitely assign all fields now compile, and any fields not explicitly initialized are set to their default value. You can read more about how this change affects struct initialization in the article on structs.

Pattern match `Span<char>` or `ReadOnlySpan<char>` on a constant `string`

You've been able to test if a `string` had a specific constant value using pattern matching for several releases. Now, you can use the same pattern matching logic with variables that are `Span<char>` or `ReadOnlySpan<char>`.

Extended nameof scope

Type parameter names and parameter names are now in scope when used in a `nameof` expression in an attribute declaration on that method. This feature means you can use the `nameof` operator to specify the name of a method parameter in an attribute on the method or parameter declaration. This feature is most often useful to add attributes for nullable analysis.

UTF-8 string literals

You can specify the `u8` suffix on a string literal to specify UTF-8 character encoding. If your application needs UTF-8 strings, for HTTP string constants or similar text protocols, you can use this feature to simplify the creation of UTF-8 strings.

You can learn more about UTF-8 string literals in the string literal section of the article on builtin reference types.

Required members

You can add the required modifier to properties and fields to enforce constructors and callers to initialize those values. The

System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute can be added to constructors to inform the compiler that a constructor initializes *all* required members.

For more information on required members, See the init-only section of the properties article.

`ref` fields and `ref scoped` variables

You can declare `ref` fields inside a ref struct. This supports types such as System.Span<T> without special attributes or hidden internal types.

You can add the scoped modifier to any `ref` declaration. This limits the scope where the reference can escape to.

File local types

Beginning in C# 11, you can use the `file` access modifier to create a type whose visibility is scoped to the source file in which it is declared. This feature helps source generator authors avoid naming collisions. You can learn more about this feature in the article on file-scoped types in the language reference.

See also

- What's new in .NET 7