This document lists known breaking changes in Roslyn after .NET 6 all the way to .NET 7.

Article • 03/17/2023

All locals of restricted types are disallowed in async methods

Introduced in Visual Studio 2022 version 17.6p1

Locals of restricted types are disallowed in async methods. But in earlier versions, the compiler failed to notice some implicitly-declared locals. For instance, in `foreach` or `using` statements or deconstructions.
Now, such implicitly-declared locals are disallowed as well.

C#

```
ref struct RefStruct { public void Dispose() { } }
public class C
{
    public async Task M()
    {
        RefStruct local = default; // disallowed
        using (default(RefStruct)) { } // now disallowed too ("error
CS9104: A using statement resource of this type cannot be used in
async methods or async lambda expressions")
    }
}
```

See https://github.com/dotnet/roslyn/pull/66264

Pointers must always be in unsafe contexts.

Introduced in Visual Studio 2022 version 17.6

In earlier SDKs, the compiler would occasionally allow locations where pointers could be referenced, without explicitly marking that location as unsafe. Now, the `unsafe` modifier must be present.
For example `using Alias = List<int*[]>;` should be changed to `using unsafe Alias = List<int*[]>;` to be legal.

A usage such as `void Method(Alias a) ...` should be changed to `unsafe void Method(Alias a) ...`.

The rule is unconditional, except for `using` alias declarations (which didn't allow an `unsafe` modifier before C# 12).
So for `using` declarations, the rule only takes effect if the language version is chosen as C# 12 or higher.

System.TypedReference considered managed

Introduced in Visual Studio 2022 version 17.6

Moving forward the `System.TypedReference` type is considered managed.

C#

```
unsafe
{
    TypedReference* r = null; // warning: This takes the address of, gets the size of, or declares a pointer to a managed type
    var a = stackalloc TypedReference[1]; // error: Cannot take the address of, get the size of, or declare a pointer to a managed type
}
```

Ref safety errors do not affect conversion from lambda expression to delegate

Introduced in Visual Studio 2022 version 17.5

Ref safety errors reported in a lambda body no longer affect whether the lambda expression is convertible to a delegate type. This change can affect overload resolution.

In the example below, the call to `M(x => ...)` is ambiguous with Visual Studio 17.5 because both `M(D1)` and `M(D2)` are now considered applicable, even though the call to `F(ref x, ref y)` within the lambda body will result in a ref safety with `M(D1)` (see the examples in `d1` and `d2` for comparison). Previously, the call bound unambiguously to `M(D2)` because the `M(D1)` overload was considered not applicable.

C#

```
using System;

ref struct R { }
```

```
delegate R D1(R r);
delegate object D2(object o);

class Program
{
    static void M(D1 d1) { }
    static void M(D2 d2) { }

    static void F(ref R x, ref Span<int> y) { }
    static void F(ref object x, ref Span<int> y) { }

    static void Main()
    {
        // error CS0121: ambiguous between: 'M(D1)' and 'M(D2)'
        M(x =>
            {
                Span<int> y = stackalloc int[1];
                F(ref x, ref y);
                return x;
            });

        D1 d1 = x1 =>
            {
                Span<int> y1 = stackalloc int[1];
                F(ref x1, ref y1); // error CS8352: 'y2' may expose
referenced variables
                return x1;
            };

        D2 d2 = x2 =>
            {
                Span<int> y2 = stackalloc int[1];
                F(ref x2, ref y2); // ok: F(ref object x, ref
Span<int> y)
                return x2;
            };
    }
}
```

To workaround the overload resolution changes, use explicit types for the lambda parameters or delegate.

C#

```
        // ok: M(D2)
        M((object x) =>
            {
                Span<int> y = stackalloc int[1];
                F(ref x, ref y); // ok: F(ref object x, ref Span<int>
y)
```

```
                return x;
            });
```

Raw string interpolations at start of line.

Introduced in Visual Studio 2022 version 17.5

In .NET SDK 7.0.100 or earlier the following was erroneously allowed:

C#

```
var x = $"""
    Hello
{1 + 1}
    World
    """;
```

This violated the rule that the lines content (including where an interpolation starts) must start with same whitespace as the final `"""` line. It is now required that the above be written as:

C#

```
var x = $"""
    Hello
    {1 + 1}
    World
    """;
```

Inferred delegate type for methods includes default parameter values and `params` modifier

Introduced in Visual Studio 2022 version 17.5

In .NET SDK 7.0.100 or earlier, delegate types inferred from methods ignored default parameter values and `params` modifiers as demonstrated in the following code:

C#

```
void Method(int i = 0, params int[] xs) { }
var action = Method; // System.Action<int, int[]>
DoAction(action, 1); // ok
```

```
void DoAction(System.Action<int, int[]> a, int p) => a(p, new[] { p
});
```

In .NET SDK 7.0.200 or later, such methods are inferred as anonymous synthesized delegate types with the same default parameter values and `params` modifiers. This change can break the code above as demonstrated below:

C#

```
void Method(int i = 0, params int[] xs) { }
var action = Method; // delegate void <anonymous delegate>(int arg1 =
0, params int[] arg2)
DoAction(action, 1); // error CS1503: Argument 1: cannot convert from
'<anonymous delegate>' to 'System.Action<int, int[]>'
void DoAction(System.Action<int, int[]> a, int p) => a(p, new[] { p
});
```

You can learn more about this change in the associated proposal.

For the purpose of definite assignment analysis, invocations of async local functions are no longer treated as being awaited

Introduced in Visual Studio 2022 version 17.5

For the purpose of definite assignment analysis, invocations of an async local function is no longer treated as being awaited and, therefore, the local function is not considered to be fully executed. See https://github.com/dotnet/roslyn/issues/43697 for the rationale.

The code below is now going to report a definite assignment error:

C#

```
    public async Task M()
    {
        bool a;
        await M1();
        Console.WriteLine(a); // error CS0165: Use of unassigned lo-
cal variable 'a'

        async Task M1()
        {
            if ("" == String.Empty)
            {
                throw new Exception();
```

```
            }
            else
            {
                a = true;
            }
        }
    }
```

`INoneOperation` nodes for attributes are now `IAttributeOperation` nodes.

Introduced in Visual Studio 2022 version 17.5, .NET SDK version 7.0.200

In previous versions of the compiler, the `IOperation` tree for an attribute was rooted with an `INoneOperation` node. We have added native support for attributes, which means that the root of the tree is now an `IAttributeOperation`. Some analyzers, including older versions of the .NET SDK analyzers, are not expecting this tree shape, and will incorrectly warn (or potentially fail to warn) when encountering it. The workarounds for this are:

- Update your analyzer version, if possible. If using the .NET SDK or older versions of Microsoft.CodeAnalysis.FxCopAnalyzers, update to Microsoft.CodeAnalysis.NetAnalyzers 7.0.0-preview1.22464.1 or newer.
- Suppress any false-positives from the analyzers until they can be updated with a version that takes this change into account.

Type tests for `ref` structs are not supported.

Introduced in Visual Studio 2022 version 17.4

When a `ref` struct type is used in an 'is' or 'as' operator, in some scenarios compiler was previously reporting an erroneous warning about the type test always failing at runtime, omitting the actual type check, and leading to incorrect behavior. When incorrect behavior at execution time was possible, compiler will now produce an error instead.

C#

```
ref struct G<T>
{
    public void Test()
    {
        if (this is G<int>) // Will now produce an error, used to be
```

```
treated as always `false`.
        {
```

Unused results from ref local are dereferences.

Introduced in Visual Studio 2022 version 17.4

When a `ref` local variable is referenced by value, but the result is not used (such as being assigned to a discard), the result was previously ignored. The compiler will now dereference that local, ensuring that any side effects are observed.

C#

```
ref int local = Unsafe.NullRef<int>();
_ = local; // Will now produce a `NullReferenceException`
```

Types cannot be named `scoped`

Introduced in Visual Studio 2022 version 17.4. Starting in C# 11, types cannot be named `scoped`. The compiler will report an error on all such type names. To work around this, the type name and all usages must be escaped with an `@`:

C#

```
class scoped {} // Error CS9056
class @scoped {} // No error
```

C#

```
ref scoped local; // Error
ref scoped.nested local; // Error
ref @scoped local2; // No error
```

This was done as `scoped` is now a modifier for variable declarations and reserved following `ref` in a ref type.

Types cannot be named `file`

Introduced in Visual Studio 2022 version 17.4. Starting in C# 11, types cannot be named `file`. The compiler will report an error on all such type names. To work around this, the type name and all usages must be escaped with an `@`:

C#

```
class file {} // Error CS9056
class @file {} // No error
```

This was done as `file` is now a modifier for type declarations.

You can learn more about this change in the associated csharplang issue.

Required spaces in #line span directives

Introduced in .NET SDK 6.0.400, Visual Studio 2022 version 17.3.

When the `#line` span directive was introduced in C# 10, it required no particular spacing.
For example, this would be valid: `#line(1,2)-(3,4)5"file.cs"`.

In Visual Studio 17.3, the compiler requires spaces before the first parenthesis, the character offset, and the file name.
So the above example fails to parse unless spaces are added: `#line (1,2)-(3,4) 5 "file.cs"`.

Checked operators on System.IntPtr and System.UIntPtr

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.3.

When the platform supports **numeric** `IntPtr` and `UIntPtr` types (as indicated by the presence of `System.Runtime.CompilerServices.RuntimeFeature.NumericIntPtr`) the built-in operators from `nint` and `nuint` apply to those underlying types. This means that on such platforms, `IntPtr` and `UIntPtr` have built-in `checked` operators, which can now throw when an overflow occurs.

C#

```
IntPtr M(IntPtr x, int y)
{
    checked
    {
        return x + y; // may now throw
    }
}
```

```
unsafe IntPtr M2(void* ptr)
{
    return checked((IntPtr)ptr); // may now throw
}
```

Possible workarounds are:

1. Specify `unchecked` context
2. Downgrade to a platform/TFM without numeric `IntPtr`/`UIntPtr` types

Also, implicit conversions between `IntPtr`/`UIntPtr` and other numeric types are treated as standard conversions on such platforms. This can affect overload resolution in some cases.

These changes could cause a behavioral change if the user code was depending on overflow exceptions in an unchecked context, or if it was not expecting overflow exceptions in a checked context. An analyzer was added in 7.0 to help detect such behavioral changes and take appropriate action. The analyzer will produce diagnostics on potential behavioral changes, which default to info severity but can be upgraded to warnings via editorconfig.

Addition of System.UIntPtr and System.Int32

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.3.

When the platform supports **numeric** `IntPtr` and `UIntPtr` types (as indicated by the presence of `System.Runtime.CompilerServices.RuntimeFeature.NumericIntPtr`), the operator `+(UIntPtr, int)` defined in `System.UIntPtr` can no longer be used. Instead, adding expressions of types `System.UIntPtr` and a `System.Int32` results in an error:

C#

```
UIntPtr M(UIntPtr x, int y)
{
    return x + y; // error: Operator '+' is ambiguous on operands of
type 'nuint' and 'int'
}
```

Possible workarounds are:

1. Use the `UIntPtr.Add(UIntPtr, int)` method: `UIntPtr.Add(x, y)`
2. Apply an unchecked cast to type `nuint` on the second operand: `x + unchecked((nuint)y)`

Nameof operator in attribute on method or local function

Introduced in .NET SDK 6.0.400, Visual Studio 2022 version 17.3.

When the language version is C# 11 or later, a `nameof` operator in an attribute on a method brings the type parameters of that method in scope. The same applies for local functions.
A `nameof` operator in an attribute on a method, its type parameters or parameters brings the parameters of that method in scope. The same applies to local functions, lambdas, delegates and indexers.

For instance, these will now be errors:

C#

```
class C
{
  class TParameter
  {
    internal const string Constant = """";
  }
  [MyAttribute(nameof(TParameter.Constant))]
  void M<TParameter>() { }
}
```

C#

```
class C
{
  class parameter
  {
    internal const string Constant = """";
  }
  [MyAttribute(nameof(parameter.Constant))]
  void M(int parameter) { }
}
```

Possible workarounds are:

1. Rename the type parameter or parameter to avoid shadowing the name from outer scope.
2. Use a string literal instead of the `nameof` operator.

Cannot return an out parameter by reference

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.3.

With language version C# 11 or later, or with .NET 7.0 or later, an `out` parameter cannot be returned by reference.

```
C#
```

```
static ref T ReturnOutParamByRef<T>(out T t)
{
    t = default;
    return ref t; // error CS8166: Cannot return a parameter by ref-
erence 't' because it is not a ref parameter
}
```

Possible workarounds are:

1. Use `System.Diagnostics.CodeAnalysis.UnscopedRefAttribute` to mark the reference as unscoped.

   ```
   C#
   ```

   ```
   static ref T ReturnOutParamByRef<T>([UnscopedRef] out T t)
   {
       t = default;
       return ref t; // ok
   }
   ```

2. Change the method signature to pass the parameter by `ref`.

   ```
   C#
   ```

   ```
   static ref T ReturnRefParamByRef<T>(ref T t)
   {
       t = default;
       return ref t; // ok
   }
   ```

Instance method on ref struct may capture unscoped ref parameters

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.4.

With language version C# 11 or later, or with .NET 7.0 or later, a `ref struct` instance method invocation is assumed to capture unscoped `ref` or `in` parameters.

C#

```
R<int> Use(R<int> r)
{
    int i = 42;
    r.MayCaptureArg(ref i); // error CS8350: may expose variables
referenced by parameter 't' outside of their declaration scope
    return r;
}

ref struct R<T>
{
    public void MayCaptureArg(ref T t) { }
}
```

A possible workaround, if the `ref` or `in` parameter is not captured in the `ref struct` instance method, is to declare the parameter as `scoped ref` or `scoped in`.

C#

```
R<int> Use(R<int> r)
{
    int i = 42;
    r.CannotCaptureArg(ref i); // ok
    return r;
}

ref struct R<T>
{
    public void CannotCaptureArg(scoped ref T t) { }
}
```

Method ref struct return escape analysis depends on ref escape of ref arguments

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.4.

With language version C# 11 or later, or with .NET 7.0 or later, a `ref struct` returned from a method invocation, either as a return value or in an `out` parameters, is only *safe-to-escape* if all the `ref` and `in` arguments to the method invocation are *ref-safe-to-escape*. *The `in` arguments may include implicit default parameter values.*

C#

```
ref struct R { }
```

```
static R MayCaptureArg(ref int i) => new R();

static R MayCaptureDefaultArg(in int i = 0) => new R();

static R Create()
{
    int i = 0;
    // error CS8347: Cannot use a result of 'MayCaptureArg(ref int)'
because it may expose
    // variables referenced by parameter 'i' outside of their decla-
ration scope
    return MayCaptureArg(ref i);
}

static R CreateDefault()
{
    // error CS8347: Cannot use a result of 'MayCaptureDefaultArg(in
int)' because it may expose
    // variables referenced by parameter 'i' outside of their decla-
ration scope
    return MayCaptureDefaultArg();
}
```

A possible workaround, if the `ref` or `in` argument is not captured in the `ref struct` return value, is to declare the parameter as `scoped ref` or `scoped in`.

C#

```
static R CannotCaptureArg(scoped ref int i) => new R();

static R Create()
{
    int i = 0;
    return CannotCaptureArg(ref i); // ok
}
```

`ref` to `ref struct` argument considered unscoped in `__arglist`

Introduced in .NET SDK 7.0.100, Visual Studio 2022 version 17.4.

With language version C# 11 or later, or with .NET 7.0 or later, a `ref` to a `ref struct` type is considered an unscoped reference when passed as an argument to `__arglist`.

C#

```
ref struct R { }
```

```
class Program
{
    static void MayCaptureRef(__arglist) { }

    static void Main()
    {
        var r = new R();
        MayCaptureRef(__arglist(ref r)); // error: may expose vari-
ables outside of their declaration scope
    }
}
```

Unsigned right shift operator

Introduced in .NET SDK 6.0.400, Visual Studio 2022 version 17.3. The language added support for an "Unsigned Right Shift" operator (`>>>`). This disables the ability to consume methods implementing user-defined "Unsigned Right Shift" operators as regular methods.

For example, there is an existing library developed in some language (other than VB or C#) that exposes an "Unsigned Right Shift" user-defined operator for type `C1`. The following code used to compile successfully before:

C#

```
static C1 Test1(C1 x, int y) => C1.op_UnsignedRightShift(x, y); //er-
ror CS0571: 'C1.operator >>>(C1, int)': cannot explicitly call opera-
tor or accessor
```

A possible workaround is to switch to using `>>>` operator:

C#

```
static C1 Test1(C1 x, int y) => x >>> y;
```

Foreach enumerator as a ref struct

Introduced in .NET SDK 6.0.300, Visual Studio 2022 version 17.2. A `foreach` using a ref struct enumerator type reports an error if the language version is set to 7.3 or earlier.

This fixes a bug where the feature was supported in newer compilers targeting a version of C# prior to its support.

Possible workarounds are:

1. Change the `ref struct` type to a `struct` or `class` type.
2. Upgrade the `<LangVersion>` element to 7.3 or later.

Async `foreach` prefers pattern based `DisposeAsync` to an explicit interface implementation of `IAsyncDisposable.DisposeAsync()`

Introduced in .NET SDK 6.0.300, Visual Studio 2022 version 17.2. An async `foreach` prefers to bind using a pattern-based `DisposeAsync()` method rather than `IAsyncDisposable.DisposeAsync()`.

For instance, the `DisposeAsync()` will be picked, rather than the `IAsyncEnumerator<int>.DisposeAsync()` method on `AsyncEnumerator`:

C#

```
await foreach (var i in new AsyncEnumerable())
{
}

struct AsyncEnumerable
{
    public AsyncEnumerator GetAsyncEnumerator() => new
AsyncEnumerator();
}

struct AsyncEnumerator : IAsyncDisposable
{
    public int Current => 0;
    public async ValueTask<bool> MoveNextAsync()
    {
        await Task.Yield();
        return false;
    }
    public async ValueTask DisposeAsync()
    {
        Console.WriteLine("PICKED");
        await Task.Yield();
    }
    ValueTask IAsyncDisposable.DisposeAsync() => throw null; // no
longer picked
}
```

This change fixes a spec violation where the public `DisposeAsync` method is visible on the declared type, whereas the explicit interface implementation is only visible using a reference to the interface type.

To workaround this error, remove the pattern based `DisposeAsync` method from your type.

Disallow converted strings as a default argument

Introduced in .NET SDK 6.0.300, Visual Studio 2022 version 17.2. The C# compiler would accept incorrect default argument values involving a reference conversion of a string constant, and would emit `null` as the constant value instead of the default value specified in source. In Visual Studio 17.2, this becomes an error. See roslyn#59806.

This change fixes a spec violation in the compiler. Default arguments must be compile time constants. Previous versions allowed the following code:

```
C#

void M(IEnumerable<char> s = "hello")
```

The preceding declaration required a conversion from `string` to `IEnumerable<char>`. The compiler allowed this construct, and would emit `null` as the value of the argument. The preceding code produces a compiler error starting in 17.2.

To work around this change, you can make one of the following changes:

1. Change the parameter type so a conversion isn't required.
2. Change the value of the default argument to `null` to restore the previous behavior.

The contextual keyword `var` as an explicit lambda return type

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. The contextual keyword var cannot be used as an explicit lambda return type.

This change enables potential future features by ensuring that the `var` remains the natural type for the return type of a lambda expression.

You can encounter this error if you have a type named `var` and define a lambda expression using an explicit return type of `var` (the type).

C#

```
using System;

F(var () => default);  // error CS8975: The contextual keyword 'var'
cannot be used as an explicit lambda return type
F(@var () => default); // ok
F(() => default);      // ok: return type is inferred from the para-
meter to F()

static void F(Func<var> f) { }

public class var
{
}
```

Workarounds include the following changes:

1. Use `@var` as the return type.
2. Remove the explicit return type so that the compiler determines the return type.

Interpolated string handlers and indexer initialization

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. Indexers that take an interpolated string handler and require the receiver as an input for the constructor cannot be used in an object initializer.

This change disallows an edge case scenario where indexer initializers use an interpolated string handler and that interpolated string handler takes the receiver of the indexer as a parameter of the constructor. The reason for this change is that this scenario could result in accessing variables that haven't yet been initialized. Consider this example:

C#

```
using System.Runtime.CompilerServices;

// error: Interpolated string handler conversions that reference
// the instance being indexed cannot be used in indexer member ini-
tializers.
var c = new C { [$""] = 1 };
```

```
class C
{
    public int this[[InterpolatedStringHandlerArgument("")]
CustomHandler c]
    {
        get => ...;
        set => ...;
    }
}

[InterpolatedStringHandler]
class CustomHandler
{
    // The constructor of the string handler takes a "C" instance:
    public CustomHandler(int literalLength, int formattedCount, C c)
{}
}
```

Workarounds include the following changes:

1. Remove the receiver type from the interpolated string handler.
2. Change the argument to the indexer to be a `string`

ref, readonly ref, in, out not allowed as parameters or return on methods with Unmanaged callers only

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. `ref` / `ref readonly` / `in` / `out` are not allowed to be used on return/parameters of a method attributed with `UnmanagedCallersOnly`.

This change is a bug fix. Return values and parameters aren't blittable. Passing arguments or return values by reference can cause undefined behavior. None of the following declarations will compile:

C#

```
using System.Runtime.InteropServices;
[UnmanagedCallersOnly]
static ref int M1() => throw null; // error CS8977: Cannot use 'ref',
'in', or 'out' in a method attributed with 'UnmanagedCallersOnly'.

[UnmanagedCallersOnly]
static ref readonly int M2() => throw null; // error CS8977: Cannot
use 'ref', 'in', or 'out' in a method attributed with 'Unmanaged-
CallersOnly'.
```

```
[UnmanagedCallersOnly]
static void M3(ref int o) => throw null; // error CS8977: Cannot use
'ref', 'in', or 'out' in a method attributed with 'UnmanagedCallers-
Only'.

[UnmanagedCallersOnly]
static void M4(in int o) => throw null; // error CS8977: Cannot use
'ref', 'in', or 'out' in a method attributed with 'UnmanagedCallers-
Only'.

[UnmanagedCallersOnly]
static void M5(out int o) => throw null; // error CS8977: Cannot use
'ref', 'in', or 'out' in a method attributed with
'UnmanagedCallersOnly'.
```

The workaround is to remove the by reference modifier.

Length, Count assumed to be non-negative in patterns

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. `Length` and `Count` properties on countable and indexable types are assumed to be non-negative for purpose of subsumption and exhaustiveness analysis of patterns and switches. Those types can be used with implicit Index indexer and list patterns.

The `Length` and `Count` properties, even though typed as `int`, are assumed to be non-negative when analyzing patterns. Consider this sample method:

C#

```
string SampleSizeMessage<T>(IList<T> samples)
{
    return samples switch
    {
        // This switch arm prevents a warning before 17.1, but will
never happen in practice.
        // Starting with 17.1, this switch arm produces a compiler
error.
        // Removing it won't introduce a warning.
        { Count: < 0 }    => throw new InvalidOperationException(),
        { Count:  0 }     => "Empty collection",
        { Count: < 5 }    => "Too small",
        { Count: < 20 }   => "reasonable for the first pass",
        { Count: < 100 }  => "reasonable",
        { Count: >= 100 } => "fine",
    };
}

void M(int[] i)
```

```
{
    if (i is { Length: -1 }) {} // error: impossible under assumption
of non-negative length
}
```

Prior to 17.1, The first switch arm, testing that `Count` is negative was necessary to avoid a warning that all possible values weren't covered. Starting with 17.1, the first switch arm generates a compiler error. The workaround is to remove the switch arms added for the invalid cases.

This change was made as part of adding list patterns. The processing rules are more consistent if every use of a `Length` or `Count` property on a collection are considered non-negative. You can read more details about the change in the language design issue.

The workaround is to remove the switch arms with unreachable conditions.

Adding field initializers to a struct requires an explicitly declared constructor

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. `struct` type declarations with field initializers must include an explicitly declared constructor. Additionally, all fields must be definitely assigned in `struct` instance constructors that do not have a `: this()` initializer so any previously unassigned fields must be assigned from the added constructor or from field initializers. See dotnet/csharplang#5552, dotnet/roslyn#58581.

There are two ways to initialize a variable to its default value in C#: `new()` and `default`. For classes, the difference is evident since `new` creates a new instance and `default` returns `null`. The difference is more subtle for structs, since for `default`, structs return an instance with each field/property set to its own default. We added field initializers for structs in C# 10. Field initializers are executed only when an explicitly declared constructor runs. Significantly, they don't execute when you use `default` or create an array of any `struct` type.

In 17.0, if there are field initializers but no declared constructors, a parameterless constructor is synthesized that runs field initializers. However, that meant adding or removing a constructor declaration may affect whether a parameterless constructor is synthesized, and as a result, may change the behavior of `new()`.

To address the issue, in .NET SDK 6.0.200 (VS 17.1) the compiler no longer synthesizes a parameterless constructor. If a `struct` contains field initializers and no explicit constructors, the compiler generates an error. If a `struct` has field initializers it must declare a constructor, because otherwise the field initializers are never executed.

Additionally, all fields that do not have field initializers must be assigned in each `struct` constructor unless the constructor has a `: this()` initializer.

For instance:

C#

```
struct S // error CS8983: A 'struct' with field initializers must in-
clude an explicitly declared constructor.
{
    int X = 1;
    int Y;
}
```

The workaround is to declare a constructor. Unless fields were previously unassigned, this constructor can, and often will, be an empty parameterless constructor:

C#

```
struct S
{
    int X = 1;
    int Y;

    public S() { Y = 0; } // ok
}
```

Format specifiers can't contain curly braces

Introduced in .NET SDK 6.0.200, Visual Studio 2022 version 17.1. Format specifiers in interpolated strings can not contain curly braces (either `{` or `}`). In previous versions `{{` was interpreted as an escaped `{` and `}}` was interpreted as an escaped `}` char in the format specifier. Now the first `}` char in a format specifier ends the interpolation, and any `{` char is an error.

This makes interpolated string processing consistent with the processing for `System.String.Format`:

C#

```
using System;
Console.WriteLine($"{{{12:X}}}");
//prints now: "{C}" - not "{X}}"
```

`X` is the format for uppercase hexadecimal and `C` is the hexadecimal value for 12.

The workaround is to remove the extra braces in the format string.

You can learn more about this change in the associated roslyn issue.

Types cannot be named `required`

Introduced in Visual Studio 2022 version 17.3. Starting in C# 11, types cannot be named `required`. The compiler will report an error on all such type names. To work around this, the type name and all usages must be escaped with an `@`:

C#

```
class required {} // Error CS9029
class @required {} // No error
```

This was done as `required` is now a member modifier for properties and fields.

You can learn more about this change in the associated csharplang issue.