# Contents

# Pattern Matching for C# 7

11/2/2020 • 15 minutes to read • Edit Online

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. The basic features are: record types, which are types whose semantic meaning is described by the shape of the data; and pattern matching, which is a new expression form that enables extremely concise multilevel decomposition of these data types. Elements of this approach are inspired by related features in the programming languages F# and Scala.

## Is expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

> Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

## Patterns

Patterns are used in the `is` operator and in a *switch_statement* to express the shape of data against which incoming data is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

declaration_pattern
    : type simple_designation
    ;

constant_pattern
    : shift_expression
    ;

var_pattern
    : 'var' simple_designation
    ;
```

> Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

#### Declaration pattern

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. If the *simple_designation* is an identifier, it introduces a local variable of the given type named by the given identifier. That local variable is *definitely assigned* when the result of the pattern-matching operation is true.

```
declaration_pattern
    : type simple_designation
    ;
```

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype), the result of the `is operator` is `true`. It declares a new local variable named by the *identifier* that is assigned the value of the left-hand operand when the result is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

> Note: In C# 7.1 we extend this to permit a pattern-matching operation if either the input type or the type `T` is an open type. This paragraph is replaced by the following:
>
> Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, **or if either `E` or `T` is an open type**. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

The declaration pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v }
```

With the slightly more concise

```
if (expr is Type v) { // code using v }
```

It is an error if *type* is a nullable value type.

The declaration pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v }
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block.

**Constant pattern**

```
constant_pattern
    : shift_expression
    ;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant, or a `typeof` expression.

If both *e* and *c* are of integral types, the pattern is considered matched if the result of the expression `e == c` is `true`.

Otherwise the pattern is considered matching if `object.Equals(e, c)` returns `true`. In this case it is a compile-time error if the static type of *e* is not *pattern compatible* with the type of the constant.

**Var pattern**

```
var_pattern
    : 'var' simple_designation
    ;
```

An expression *e* matches a *var_pattern* always. In other words, a match to a *var pattern* always succeeds. If the *simple_designation* is an identifier, then at runtime the value of *e* is bound to a newly introduced local variable. The type of the local variable is the static type of *e*.

It is an error if the name `var` binds to a type.

# Switch statement

The `switch` statement is extended to select for execution the first block having an associated pattern that matches the *switch expression*.

```
switch_label
    : 'case' complex_pattern case_guard? ':'
    | 'case' constant_expression case_guard? ':'
    | 'default' ':'
    ;

case_guard
    : 'when' expression
    ;
```

The order in which patterns are matched is not defined. A compiler is permitted to match patterns out of order, and to reuse the results of already matched patterns to compute the result of matching of other patterns.

If a *case-guard* is present, its expression is of type `bool`. It is evaluated as an additional condition that must be satisfied for the case to be considered satisfied.

It is an error if a *switch_label* can have no effect at runtime because its pattern is subsumed by previous cases.

[TODO: We should be more precise about the techniques the compiler is required to use to reach this judgment.]

A pattern variable declared in a *switch_label* is definitely assigned in its case block if and only if that case block contains precisely one *switch_label*.

[TODO: We should specify when a *switch block* is reachable.]

**Scope of pattern variables**

The scope of a variable declared in a pattern is as follows:

- If the pattern is a case label, then the scope of the variable is the *case block*.

Otherwise the variable is declared in an *is_pattern* expression, and its scope is based on the construct immediately enclosing the expression containing the *is_pattern* expression as follows:

- If the expression is in an expression-bodied lambda, its scope is the body of the lambda.
- If the expression is in an expression-bodied method or property, its scope is the body of the method or property.
- If the expression is in a `when` clause of a `catch` clause, its scope is that `catch` clause.
- If the expression is in an *iteration_statement*, its scope is just that statement.
- Otherwise if the expression is in some other statement form, its scope is the scope containing the statement.

For the purpose of determining the scope, an *embedded_statement* is considered to be in its own scope. For example, the grammar for an *if_statement* is

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

So if the controlled statement of an *if_statement* declares a pattern variable, its scope is restricted to that *embedded_statement*.

```
if (x) M(y is var z);
```

In this case the scope of `z` is the embedded statement `M(y is var z);`.

Other cases are errors for other reasons (e.g. in a parameter's default value or an attribute, both of which are an error because those contexts require a constant expression).

> In C# 7.3 we added the following contexts in which a pattern variable may be declared:
>
> - If the expression is in a *constructor initializer*, its scope is the *constructor initializer* and the constructor's body.
> - If the expression is in a field initializer, its scope is the *equals_value_clause* in which it appears.
> - If the expression is in a query clause that is specified to be translated into the body of a lambda, its scope is just that expression.

# Changes to syntactic disambiguation

There are situations involving generics where the C# grammar is ambiguous, and the language spec says how to resolve those ambiguities:

> **7.6.5.2 Grammar ambiguities**
>
> The productions for *simple-name* (§7.6.3) and *member-access* (§7.6.5) can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or > *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments. The statement

```
x = F < A > +y;
```

will be interpreted as a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple-name* with a *type-argument-list* followed by a binary plus operator. In the statement

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace-or-type-name* with a *type-argument-list*.

There are a number of changes being introduced in C# 7 that make these disambiguation rules no longer sufficient to handle the complexity of the language.

**Out variable declarations**

It is now possible to declare a variable in an out argument:

```
M(out Type name);
```

However, the type may be generic:

```
M(out A<B> name);
```

Since the language grammar for the argument uses *expression*, this context is subject to the disambiguation rule. In this case the closing `>` is followed by an *identifier*, which is not one of the tokens that permits it to be treated as a *type-argument-list*. I therefore propose to **add *identifier* to the set of tokens that triggers the disambiguation to a *type-argument-list*.**

**Tuples and deconstruction declarations**

A tuple literal runs into exactly the same issue. Consider the tuple expression

```
(A < B, C > D, E < F, G > H)
```

Under the old C# 6 rules for parsing an argument list, this would parse as a tuple with four elements, starting with `A < B` as the first. However, when this appears on the left of a deconstruction, we want the disambiguation triggered by the *identifier* token as described above:

```
(A<B,C> D, E<F,G> H) = e;
```

This is a deconstruction declaration which declares two variables, the first of which is of type `A<B,C>` and named `D`. In other words, the tuple literal contains two expressions, each of which is a declaration expression.

For simplicity of the specification and compiler, I propose that this tuple literal be parsed as a two-element tuple wherever it appears (whether or not it appears on the left-hand-side of an assignment). That would be a natural result of the disambiguation described in the previous section.

**Pattern-matching**

Pattern matching introduces a new context where the expression-type ambiguity arises. Previously the right-hand-side of an `is` operator was a type. Now it can be a type or expression, and if it is a type it may be followed by an identifier. This can, technically, change the meaning of existing code:

```
var x = e is T < A > B;
```

This could be parsed under C#6 rules as

```
var x = ((e is T) < A) > B;
```

but under under C#7 rules (with the disambiguation proposed above) would be parsed as

```
var x = e is T<A> B;
```

which declares a variable `B` of type `T<A>`. Fortunately, the native and Roslyn compilers have a bug whereby they give a syntax error on the C#6 code. Therefore this particular breaking change is not a concern.

Pattern-matching introduces additional tokens that should drive the ambiguity resolution toward selecting a type. The following examples of existing valid C#6 code would be broken without additional disambiguation rules:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;           // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];              // [
```

**Proposed change to the disambiguation rule**

I propose to revise the specification to change the list of disambiguating tokens from

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

And, in certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case`, or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

**Modified disambiguation rule**

The revised disambiguation rule would be something like this

> If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined, to see if it is
>
> - One of `( ) ] } : ; , . ? == != | ^ && || & [`; or
> - One of the relational operators `< > <= >= is as`; or
> - A contextual query keyword appearing inside a query expression; or
> - In certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case` or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.
>
> If the following token is among this list, or an identifier in such a context, then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8).

**Breaking changes due to this proposal**

No breaking changes are known due to this proposed disambiguation rule.

**Interesting examples**

Here are some interesting results of these disambiguation rules:

The expression `(A < B, C > D)` is a tuple with two elements, each a comparison.

The expression `(A<B,C> D, E)` is a tuple with two elements, the first of which is a declaration expression.

The invocation `M(A < B, C > D, E)` has three arguments.

The invocation `M(out A<B,C> D, E)` has two arguments, the first of which is an `out` declaration.

The expression `e is A<B> C` uses a declaration expression.

The case label `case A<B> C:` uses a declaration expression.

# Some examples of pattern matching

### Is-As

We can replace the idiom

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

With the slightly more concise and direct

```
if (expr is Type v) {
    // code using v
}
```

### Testing nullable

We can replace the idiom

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

With the slightly more concise and direct

```
if (x?.y?.z is Type value) {
    // code using value
}
```

### Arithmetic simplification

Suppose we define a set of recursive types to represent expressions (per a separate proposal):

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Now we can define a function to compute the (unreduced) derivative of an expression:

```
Expr Deriv(Expr e)
{
  switch (e) {
    case X(): return Const(1);
    case Const(*): return Const(0);
    case Add(var Left, var Right):
      return Add(Deriv(Left), Deriv(Right));
    case Mult(var Left, var Right):
      return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
    case Neg(var Value):
      return Neg(Deriv(Value));
  }
}
```

An expression simplifier demonstrates positional patterns:

```
Expr Simplify(Expr e)
{
  switch (e) {
    case Mult(Const(0), *): return Const(0);
    case Mult(*, Const(0)): return Const(0);
    case Mult(Const(1), var x): return Simplify(x);
    case Mult(var x, Const(1)): return Simplify(x);
    case Mult(Const(var l), Const(var r)): return Const(l*r);
    case Add(Const(0), var x): return Simplify(x);
    case Add(var x, Const(0)): return Simplify(x);
    case Add(Const(var l), Const(var r)): return Const(l+r);
    case Neg(Const(var k)): return Const(-k);
    default: return e;
  }
}
```

# Local functions

11/2/2020 • 2 minutes to read • Edit Online

We extend C# to support the declaration of functions in block scope. Local functions may use (capture) variables from the enclosing scope.

The compiler uses flow analysis to detect which variables a local function uses before assigning it a value. Every call of the function requires such variables to be definitely assigned. Similarly the compiler determines which variables are definitely assigned on return. Such variables are considered definitely assigned after the local function is invoked.

Local functions may be called from a lexical point before its definition. Local function declaration statements do not cause a warning when they are not reachable.

TODO: *WRITE SPEC*

## Syntax grammar

This grammar is represented as a diff from the current spec grammar.

```
declaration-statement
    : local-variable-declaration ';'
    | local-constant-declaration ';'
+   | local-function-declaration
    ;

+local-function-declaration
+   : local-function-header local-function-body
+   ;

+local-function-header
+   : local-function-modifiers? return-type identifier type-parameter-list?
+       ( formal-parameter-list? ) type-parameter-constraints-clauses
+   ;

+local-function-modifiers
+   : (async | unsafe)
+   ;

+local-function-body
+   : block
+   | arrow-expression-body
+   ;
```

Local functions may use variables defined in the enclosing scope. The current implementation requires that every variable read inside a local function be definitely assigned, as if executing the local function at its point of definition. Also, the local function definition must have been "executed" at any use point.

After experimenting with that a bit (for example, it is not possible to define two mutually recursive local functions), we've since revised how we want the definite assignment to work. The revision (not yet implemented) is that all local variables read in a local function must be definitely assigned at each invocation of the local function. That's actually more subtle than it sounds, and there is a bunch of work remaining to make it work. Once it is done you'll be able to move your local functions to the end of its enclosing block.

The new definite assignment rules are incompatible with inferring the return type of a local function, so we'll likely be removing support for inferring the return type.

Unless you convert a local function to a delegate, capturing is done into frames that are value types. That means you don't get any GC pressure from using local functions with capturing.

**Reachability**

We add to the spec

> The body of a statement-bodied lambda expression or local function is considered reachable.

# Out variable declarations

11/2/2020 • 2 minutes to read • Edit Online

The *out variable declaration* feature enables a variable to be declared at the location that it is being passed as an `out` argument.

```
argument_value
    : 'out' type identifier
    | ...
    ;
```

A variable declared this way is called an *out variable*. You may use the contextual keyword `var` for the variable's type. The scope will be the same as for a *pattern-variable* introduced via pattern-matching.

According to Language Specification (section 7.6.7 Element access) the argument-list of an element-access (indexing expression) does not contain ref or out arguments. However, they are permitted by the compiler for various scenarios, for example indexers declared in metadata that accept `out`.

Within the scope of a local variable introduced by an argument_value, it is a compile-time error to refer to that local variable in a textual position that precedes its declaration.

It is also an error to reference an implicitly-typed (§8.5.1) out variable in the same argument list that immediately contains its declaration.

Overload resolution is modified as follows:

We add a new conversion:

> There is a *conversion from expression* from an implicitly-typed out variable declaration to every type.

Also

> The type of an explicitly-typed out variable argument is the declared type.

and

> An implicitly-typed out variable argument has no type.

The *conversion from expression* from an implicitly-typed out variable declaration is not considered better than any other *conversion from expression*.

The type of an implicitly-typed out variable is the type of the corresponding parameter in the signature of the method selected by overload resolution.

The new syntax node `DeclarationExpressionSyntax` is added to represent the declaration in an out var argument.

# Throw expression

11/2/2020 • 2 minutes to read • Edit Online

We extend the set of expression forms to include

```
throw_expression
    : 'throw' null_coalescing_expression
    ;

null_coalescing_expression
    : throw_expression
    ;
```

The type rules are as follows:

- A *throw_expression* has no type.
- A *throw_expression* is convertible to every type by an implicit conversion.

A *throw expression* throws the value produced by evaluating the *null_coalescing_expression*, which must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

The behavior at runtime of the evaluation of a *throw expression* is the same as specified for a *throw statement*.

The flow-analysis rules are as follows:

- For every variable *v*, *v* is definitely assigned before the *null_coalescing_expression* of a *throw_expression* iff it is definitely assigned before the *throw_expression*.
- For every variable *v*, *v* is definitely assigned after *throw_expression*.

A *throw expression* is permitted in only the following syntactic contexts:

- As the second or third operand of a ternary conditional operator `?:`
- As the second operand of a null coalescing operator `??`
- As the body of an expression-bodied lambda or method.

# Binary literals

11/2/2020 • 2 minutes to read • Edit Online

There's a relatively common request to add binary literals to C# and VB. For bitmasks (e.g. flag enums) this seems genuinely useful, but it would also be great just for educational purposes.

Binary literals would look like this:

```
int nineteen = 0b10011;
```

Syntactically and semantically they are identical to hexadecimal literals, except for using `b` / `B` instead of `x` / `X`, having only digits `0` and `1` and being interpreted in base 2 instead of 16.

There's little cost to implementing these, and little conceptual overhead to users of the language.

## Syntax

The grammar would be as follows:

```
integer-literal:
    : ...
    | binary-integer-literal
    ;
binary-integer-literal:
    : `0b` binary-digits integer-type-suffix-opt
    | `0B` binary-digits integer-type-suffix-opt
    ;
binary-digits:
    : binary-digit
    | binary-digits binary-digit
    ;
binary-digit:
    : `0`
    | `1`
    ;
```

# Digit separators

11/2/2020 • 2 minutes to read • Edit Online

Being able to group digits in large numeric literals would have great readability impact and no significant downside.

Adding binary literals (#215) would increase the likelihood of numeric literals being long, so the two features enhance each other.

We would follow Java and others, and use an underscore `_` as a digit separator. It would be able to occur everywhere in a numeric literal (except as the first and last character), since different groupings may make sense in different scenarios and especially for different numeric bases:

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

Any sequence of digits may be separated by underscores, possibly more than one underscore between two consecutive digits. They are allowed in decimals as well as exponents, but following the previous rule, they may not appear next to the decimal ( `10_.0` ), next to the exponent character ( `1.1e_1` ), or next to the type specifier ( `10_f` ). When used in binary and hexadecimal literals, they may not appear immediately following the `0x` or `0b` .

The syntax is straightforward, and the separators have no semantic impact - they are simply ignored.

This has broad value and is easy to implement.

# Async Task Types in C#

11/24/2020 • 3 minutes to read • Edit Online

Extend `async` to support *task types* that match a specific pattern, in addition to the well known types `System.Threading.Tasks.Task` and `System.Threading.Tasks.Task<T>`.

## Task Type

A *task type* is a `class` or `struct` with an associated *builder type* identified with `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. The *task type* may be non-generic, for async methods that do not return a value, or generic, for methods that return a value.

To support `await`, the *task type* must have a corresponding, accessible `GetAwaiter()` method that returns an instance of an *awaiter type* (see *C# 7.7.7.1 Awaitable expressions*).

```
[AsyncMethodBuilder(typeof(MyTaskMethodBuilder<>))]
class MyTask<T>
{
    public Awaiter<T> GetAwaiter();
}

class Awaiter<T> : INotifyCompletion
{
    public bool IsCompleted { get; }
    public T GetResult();
    public void OnCompleted(Action completion);
}
```

## Builder Type

The *builder type* is a `class` or `struct` that corresponds to the specific *task type*. The *builder type* can have at most 1 type parameter and must not be nested in a generic type. The *builder type* has the following `public` methods. For non-generic *builder types*, `SetResult()` has no parameters.

```
class MyTaskMethodBuilder<T>
{
    public static MyTaskMethodBuilder<T> Create();

    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;

    public void SetStateMachine(IAsyncStateMachine stateMachine);
    public void SetException(Exception exception);
    public void SetResult(T result);

    public void AwaitOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : ICriticalNotifyCompletion
        where TStateMachine : IAsyncStateMachine;

    public MyTask<T> Task { get; }
}
```

## Execution

The types above are used by the compiler to generate the code for the state machine of an `async` method. (The generated code is equivalent to the code generated for async methods that return `Task`, `Task<T>`, or `void`. The difference is, for those well known types, the *builder types* are also known to the compiler.)

`Builder.Create()` is invoked to create an instance of the *builder type*.

If the state machine is implemented as a `struct`, then `builder.SetStateMachine(stateMachine)` is called with a boxed instance of the state machine that the builder can cache if necessary.

`builder.Start(ref stateMachine)` is invoked to associate the builder with compiler-generated state machine instance. The builder must call `stateMachine.MoveNext()` either in `Start()` or after `Start()` has returned to advance the state machine. After `Start()` returns, the `async` method calls `builder.Task` for the task to return from the async method.

Each call to `stateMachine.MoveNext()` will advance the state machine. If the state machine completes successfully, `builder.SetResult()` is called, with the method return value if any. If an exception is thrown in the state machine, `builder.SetException(exception)` is called.

If the state machine reaches an `await expr` expression, `expr.GetAwaiter()` is invoked. If the awaiter implements `ICriticalNotifyCompletion` and `IsCompleted` is false, the state machine invokes `builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine)`. `AwaitUnsafeOnCompleted()` should call `awaiter.OnCompleted(action)` with an action that calls `stateMachine.MoveNext()` when the awaiter completes. Similarly for `INotifyCompletion` and `builder.AwaitOnCompleted()`.

## Overload Resolution

Overload resolution is extended to recognize *task types* in addition to `Task` and `Task<T>`.

An `async` lambda with no return value is an exact match for an overload candidate parameter of non-generic *task type*, and an `async` lambda with return type `T` is an exact match for an overload candidate parameter of generic *task type*.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, or an

exact match for both, and there is an implicit conversion from one candidate type to the other, the from candidate wins. Otherwise recursively evaluate the types `A` and `B` within `Task1<A>` and `Task2<B>` for better match.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, but one candidate is a more specialized type than the other, the more specialized candidate wins.

# Async Main

11/2/2020 • 2 minutes to read • Edit Online

- [x] Proposed
- [ ] Prototype
- [ ] Implementation
- [ ] Specification

## Summary

Allow `await` to be used in an application's Main / entrypoint method by allowing the entrypoint to return `Task` / `Task<int>` and be marked `async`.

## Motivation

It is very common when learning C#, when writing console-based utilities, and when writing small test apps to want to call and `await` `async` methods from Main. Today we add a level of complexity here by forcing such `await` 'ing to be done in a separate async method, which causes developers to need to write boilerplate like the following just to get started:

```
public static void Main()
{
    MainAsync().GetAwaiter().GetResult();
}

private static async Task MainAsync()
{
    ... // Main body here
}
```

We can remove the need for this boilerplate and make it easier to get started simply by allowing Main itself to be `async` such that `await`s can be used in it.

## Detailed design

The following signatures are currently allowed entrypoints:

```
static void Main()
static void Main(string[])
static int Main()
static int Main(string[])
```

We extend the list of allowed entrypoints to include:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

To avoid compatibility risks, these new signatures will only be considered as valid entrypoints if no overloads of the previous set are present. The language / compiler will not require that the entrypoint be marked as `async`, though

we expect the vast majority of uses will be marked as such.

When one of these is identified as the entrypoint, the compiler will synthesize an actual entrypoint method that calls one of these coded methods:

- `static Task Main()` will result in the compiler emitting the equivalent of `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` will result in the compiler emitting the equivalent of `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` will result in the compiler emitting the equivalent of `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` will result in the compiler emitting the equivalent of `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

Example usage:

```
using System;
using System.Net.Http;

class Test
{
    static async Task Main(string[] args) =>
     Console.WriteLine(await new HttpClient().GetStringAsync(args[0]));
}
```

## Drawbacks

The main drawback is simply the additional complexity of supporting additional entrypoint signatures.

## Alternatives

Other variants considered:

Allowing `async void`. We need to keep the semantics the same for code calling it directly, which would then make it difficult for a generated entrypoint to call it (no Task returned). We could solve this by generating two other methods, e.g.

```
public static async void Main()
{
   ... // await code
}
```

becomes

```
public static async void Main() => await $MainTask();

private static void $EntrypointMain() => Main().GetAwaiter().GetResult();

private static async Task $MainTask()
{
    ... // await code
}
```

There are also concerns around encouraging usage of `async void`.

Using "MainAsync" instead of "Main" as the name. While the async suffix is recommended for Task-returning

methods, that's primarily about library functionality, which Main is not, and supporting additional entrypoint names beyond "Main" is not worth it.

## Unresolved questions

n/a

## Design meetings

n/a

# Target-typed "default" literal

11/2/2020 • 2 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype
- [x] Implementation
- [ ] Specification

## Summary

The target-typed `default` feature is a shorter form variation of the `default(T)` operator, which allows the type to be omitted. Its type is inferred by target-typing instead. Aside from that, it behaves like `default(T)`.

## Motivation

The main motivation is to avoid typing redundant information.

For instance, when invoking `void Method(ImmutableArray<SomeType> array)`, the *default* literal allows `M(default)` in place of `M(default(ImmutableArray<SomeType>))`.

This is applicable in a number of scenarios, such as:

- declaring locals (`ImmutableArray<SomeType> x = default;`)
- ternary operations (`var x = flag ? default : ImmutableArray<SomeType>.Empty;`)
- returning in methods and lambdas (`return default;`)
- declaring default values for optional parameters (`void Method(ImmutableArray<SomeType> arrayOpt = default)`)
- including default values in array creation expressions (`var x = new[] { default, ImmutableArray.Create(y) };`)

## Detailed design

A new expression is introduced, the *default* literal. An expression with this classification can be implicitly converted to any type, by a *default literal conversion*.

The inference of the type for the *default* literal works the same as that for the *null* literal, except that any type is allowed (not just reference types).

This conversion produces the default value of the inferred type.

The *default* literal may have a constant value, depending on the inferred type. So `const int x = default;` is legal, but `const int? y = default;` is not.

The *default* literal can be the operand of equality operators, as long as the other operand has a type. So `default == x` and `x == default` are valid expressions, but `default == default` is illegal.

## Drawbacks

A minor drawback is that *default* literal can be used in place of *null* literal in most contexts. Two of the exceptions are `throw null;` and `null == null`, which are allowed for the *null* literal, but not the *default* literal.

## Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the default operator with an explicit type.
- Extending the null literal: This is the VB approach with `Nothing`. We could allow `int x = null;`.

## Unresolved questions

- [x] Should *default* be allowed as the operand of the *is* or *as* operators? Answer: disallow `default is T`, allow `x is default`, allow `default as RefType` (with always-null warning)

## Design meetings

- LDM 3/7/2017
- LDM 3/28/2017
- LDM 5/31/2017

# Infer tuple names (aka. tuple projection initializers)

11/2/2020 • 2 minutes to read • Edit Online

## Summary

In a number of common cases, this feature allows the tuple element names to be omitted and instead be inferred. For instance, instead of typing `(f1: x.f1, f2: x?.f2)`, the element names "f1" and "f2" can be inferred from `(x.f1, x?.f2)`.

This parallels the behavior of anonymous types, which allow inferring member names during creation. For instance, `new { x.f1, y?.f2 }` declares members "f1" and "f2".

This is particularly handy when using tuples in LINQ:

```
// "c" and "result" have element names "f1" and "f2"
var result = list.Select(c => (c.f1, c.f2)).Where(t => t.f2 == 1);
```

## Detailed design

There are two parts to the change:

1. Try to infer a candidate name for each tuple element which does not have an explicit name:
   - Using same rules as name inference for anonymous types.
     - In C#, this allows three cases: `y` (identifier), `x.y` (simple member access) and `x?.y` (conditional access).
     - In VB, this allows for additional cases, such as `x.y()`.
   - Rejecting reserved tuple names (case-sensitive in C#, case-insensitive in VB), as they are either forbidden or already implicit. For instance, such as `ItemN`, `Rest`, and `ToString`.
   - If any candidate names are duplicates (case-sensitive in C#, case-insensitive in VB) within the entire tuple, we drop those candidates,
2. During conversions (which check and warn about dropping names from tuple literals), inferred names would not produce any warnings. This avoids breaking existing tuple code.

Note that the rule for handling duplicates is different than that for anonymous types. For instance, `new { x.f1, x.f1 }` produces an error, but `(x.f1, x.f1)` would still be allowed (just without any inferred names). This avoids breaking existing tuple code.

For consistency, the same would apply to tuples produced by deconstruction-assignments (in C#):

```
// tuple has element names "f1" and "f2"
var tuple = ((x.f1, x?.f2) = (1, 2));
```

The same would also apply to VB tuples, using the VB-specific rules for inferring name from expression and case-insensitive name comparisons.

When using the C# 7.1 compiler (or later) with language version "7.0", the element names will be inferred (despite the feature not being available), but there will be a use-site error for trying to access them. This will limit additions of new code that would later face the compatibility issue (described below).

## Drawbacks

The main drawback is that this introduces a compatibility break from C# 7.0:

```
Action y = () => M();
var t = (x: x, y);
t.y(); // this might have previously picked up an extension method called “y”, but would now call the lambda.
```

The compatibility council found this break acceptable, given that it is limited and the time window since tuples shipped (in C# 7.0) is short.

## References

- LDM April 4th 2017
- Github discussion (thanks @alrz for bringing this issue up)
- Tuples design

# pattern-matching with generics

11/2/2020 • 2 minutes to read • Edit Online

- [x] Proposed
- [ ] Prototype:
- [ ] Implementation:
- [ ] Specification:

## Summary

The specification for the existing C# as operator permits there to be no conversion between the type of the operand and the specified type when either is an open type. However, in C# 7 the `Type identifier` pattern requires there be a conversion between the type of the input and the given type.

We propose to relax this and change `expression is Type identifier`, in addition to being permitted in the conditions when it is permitted in C# 7, to also be permitted when `expression as Type` would be allowed. Specifically, the new cases are cases where the type of the expression or the specified type is an open type.

## Motivation

Cases where pattern-matching should "obviously" be permitted currently fail to compile. See, for example, https://github.com/dotnet/roslyn/issues/16195.

## Detailed design

We change the paragraph in the pattern-matching specification (the proposed addition is shown in bold):

> Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, **or if either `E` or `T` is an open type**. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

## Drawbacks

None.

## Alternatives

None.

## Unresolved questions

None.

## Design meetings

LDM considered this question and felt it was a bug-fix level change. We are treating it as a separate language feature because just making the change after the language has been released would introduce a forward

incompatibility. Using the proposed change requires that the programmer specify language version 7.1.

# Readonly references

11/2/2020 • 24 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype
- [x] Implementation: Started
- [ ] Specification: Not Started

## Summary

The "readonly references" feature is actually a group of features that leverage the efficiency of passing variables by reference, but without exposing the data to modifications:

- `in` parameters
- `ref readonly` returns
- `readonly` structs
- `ref` / `in` extension methods
- `ref readonly` locals
- `ref` conditional expressions

## Passing arguments as readonly references.

There is an existing proposal that touches this topic https://github.com/dotnet/roslyn/issues/115 as a special case of readonly parameters without going into many details. Here I just want to acknowledge that the idea by itself is not very new.

#### Motivation

Prior to this feature C# did not have an efficient way of expressing a desire to pass struct variables into method calls for readonly purposes with no intention of modifying. Regular by-value argument passing implies copying, which adds unnecessary costs. That drives users to use by-ref argument passing and rely on comments/documentation to indicate that the data is not supposed to be mutated by the callee. It is not a good solution for many reasons.
The examples are numerous - vector/matrix math operators in graphics libraries like XNA are known to have ref operands purely because of performance considerations. There is code in Roslyn compiler itself that uses structs to avoid allocations and then passes them by reference to avoid copying costs.

#### Solution ( `in` parameters)

Similarly to the `out` parameters, `in` parameters are passed as managed references with additional guarantees from the callee.
Unlike `out` parameters which *must* be assigned by the callee before any other use, `in` parameters cannot be assigned by the callee at all.

As a result `in` parameters allow for effectiveness of indirect argument passing without exposing arguments to mutations by the callee.

#### Declaring `in` parameters

`in` parameters are declared by using `in` keyword as a modifier in the parameter signature.

For all purposes the `in` parameter is treated as a `readonly` variable. Most of the restrictions on the use of `in`

parameters inside the method are the same as with `readonly` fields.

> Indeed an `in` parameter may represent a `readonly` field. Similarity of restrictions is not a coincidence.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly` variables .

```
static Vector3 Add (in Vector3 v1, in Vector3 v2)
{
    // not OK!!
    v1 = default(Vector3);

    // not OK!!
    v1.X = 0;

    // not OK!!
    foo(ref v1.X);

    // OK
    return new Vector3(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
}
```

- `in` parameters are allowed anywhere where ordinary byval parameters are allowed. This includes indexers, operators (including conversions), delegates, lambdas, local functions.

```
(in int x) => x                                                 // lambda expression
TValue this[in TKey index];                                     // indexer
public static Vector3 operator +(in Vector3 x, in Vector3 y) => ... // operator
```

- `in` is not allowed in combination with `out` or with anything that `out` does not combine with.
- It is not permitted to overload on `ref` / `out` / `in` differences.
- It is permitted to overload on ordinary byval and `in` differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `in` behaves similarly to an `out` parameter. All the same rules apply. For example the overriding method will have to match `in` parameters with `in` parameters of an identity-convertible type.
- For the purpose of delegate/lambda/method group conversions, `in` behaves similarly to an `out` parameter. Lambdas and applicable method group conversion candidates will have to match `in` parameters of the target delegate with `in` parameters of an identity-convertible type.
- For the purpose of generic variance, `in` parameters are nonvariant.

> NOTE: There are no warnings on `in` parameters that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `Method(in T param)` when `T` was substituted to be `int`, or when having methods like `Volatile.Read(in int location)`
>
> It is conceivable to have an analyzer that warns in cases of inefficient use of `in` parameters, but the rules for such analysis would be too fuzzy to be a part of a language specification.

### Use of `in` at call sites. (`in` arguments)

There are two ways to pass arguments to `in` parameters.

**`in` arguments can match `in` parameters:**

An argument with an `in` modifier at the call site can match `in` parameters.

```
int x = 1;

void M1<T>(in T x)
{
  // . . .
}

var x = M1(in x);  // in argument to a method

class D
{
    public string this[in Guid index];
}

D dictionary = . . . ;
var y = dictionary[in Guid.Empty]; // in argument to an indexer
```

- `in` argument must be a *readable* LValue(*). Example: `M1(in 42)` is invalid

> (*) The notion of LValue/RValue vary between languages.
> Here, by LValue I mean an expression that represent a location that can be referred to directly. And RValue means an expression that yields a temporary result which does not persist on its own.

- In particular it is valid to pass `readonly` fields, `in` parameters or other formally `readonly` variables as `in` arguments. Example: `dictionary[in Guid.Empty]` is legal. `Guid.Empty` is a static readonly field.
- `in` argument must have type *identity-convertible* to the type of the parameter. Example: `M1<object>(in Guid.Empty)` is invalid. `Guid.Empty` is not *identity-convertible* to `object`

The motivation for the above rules is that `in` arguments guarantee *aliasing* of the argument variable. The callee always receives a direct reference to the same location as represented by the argument.

- in rare situations when `in` arguments must be stack-spilled due to `await` expressions used as operands of the same call, the behavior is the same as with `out` and `ref` arguments - if the variable cannot be spilled in referentially-transparent manner, an error is reported.

Examples:

1. `M1(in staticField, await SomethingAsync())` is valid. `staticField` is a static field which can be accessed more than once without observable side effects. Therefore both the order of side effects and aliasing requirements can be provided.
2. `M1(in RefReturningMethod(), await SomethingAsync())` will produce an error. `RefReturningMethod()` is a `ref` returning method. A method call may have observable side effects, therefore it must be evaluated before the `SomethingAsync()` operand. However the result of the invocation is a reference that cannot be preserved across the `await` suspension point which make the direct reference requirement impossible.

> NOTE: the stack spilling errors are considered to be implementation-specific limitations. Therefore they do not have effect on overload resolution or lambda inference.

#### Ordinary byval arguments can match `in` parameters:

Regular arguments without modifiers can match `in` parameters. In such case the arguments have the same relaxed constraints as an ordinary byval arguments would have.

The motivation for this scenario is that `in` parameters in APIs may result in inconveniences for the user when arguments cannot be passed as a direct reference - ex: literals, computed or `await`-ed results or arguments that

happen to have more specific types.
All these cases have a trivial solution of storing the argument value in a temporary local of appropriate type and passing that local as an `in` argument.
To reduce the need for such boilerplate code compiler can perform the same transformation, if needed, when `in` modifier is not present at the call site.

In addition, in some cases, such as invocation of operators, or `in` extension methods, there is no syntactical way to specify `in` at all. That alone requires specifying the behavior of ordinary byval arguments when they match `in` parameters.

In particular:

- it is valid to pass RValues. A reference to a temporary is passed in such case. Example:

```
Print("hello");      // not an error.

void Print<T>(in T x)
{
  //. . .
}
```

- implicit conversions are allowed.

> This is actually a special case of passing an RValue

A reference to a temporary holding converted value is passed in such case. Example:

```
Print<int>(Short.MaxValue)     // not an error.
```

- in a case of a receiver of an `in` extension method (as opposed to `ref` extension methods), RValues or implicit *this-argument-conversions* are allowed. A reference to a temporary holding converted value is passed in such case. Example:

```
public static IEnumerable<T> Concat<T>(in this (IEnumerable<T>, IEnumerable<T>) arg)  => . . .;

("aa", "bb").Concat<char>()    // not an error.
```

More information on `ref` / `in` extension methods is provided further in this document.

- argument spilling due to `await` operands could spill "by-value", if necessary. In scenarios where providing a direct reference to the argument is not possible due to intervening `await` a copy of the argument's value is spilled instead.
  Example:

```
M1(RefReturningMethod(), await SomethingAsync())   // not an error.
```

Since the result of a side-effecting invocation is a reference that cannot be preserved across `await` suspension, a temporary containing the actual value will be preserved instead (as it would in an ordinary byval parameter case).

#### Omitted optional arguments

It is permitted for an `in` parameter to specify a default value. That makes the corresponding argument optional.

Omitting optional argument at the call site results in passing the default value via a temporary.

```
Print("hello");      // not an error, same as
Print("hello", c: Color.Black);

void Print(string s, in Color c = Color.Black)
{
    // . . .
}
```

**Aliasing behavior in general**

Just like `ref` and `out` variables, `in` variables are references/aliases to existing locations.

While callee is not allowed to write into them, reading an `in` parameter can observe different values as a side effect of other evaluations.

Example:

```
static Vector3 v = Vector3.UnitY;

static void Main()
{
    Test(v);
}

static void Test(in Vector3 v1)
{
    Debug.Assert(v1 == Vector3.UnitY);
    // changes v1 deterministically (no races required)
    ChangeV();
    Debug.Assert(v1 == Vector3.UnitX);
}

static void ChangeV()
{
    v = Vector3.UnitX;
}
```

**`in` parameters and capturing of local variables.**

For the purpose of lambda/async capturing `in` parameters behave the same as `out` and `ref` parameters.

- `in` parameters cannot be captured in a closure
- `in` parameters are not allowed in iterator methods
- `in` parameters are not allowed in async methods

**Temporary variables.**

Some uses of `in` parameter passing may require indirect use of a temporary local variable:

- `in` arguments are always passed as direct aliases when call-site uses `in`. Temporary is never used in such case.
- `in` arguments are not required to be direct aliases when call-site does not use `in`. When argument is not an LValue, a temporary may be used.
- `in` parameter may have default value. When corresponding argument is omitted at the call site, the default value are passed via a temporary.
- `in` arguments may have implicit conversions, including those that do not preserve identity. A temporary is used in those cases.
- receivers of ordinary struct calls may not be writeable LValues (**existing case!**). A temporary is used in those cases.

The life time of the argument temporaries matches the closest encompassing scope of the call-site.

The formal life time of temporary variables is semantically significant in scenarios involving escape analysis of variables returned by reference.

**Metadata representation of `in` parameters.**

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a byref parameter, it means that the parameter is an `in` parameter.

In addition, if the method is *abstract* or *virtual*, then the signature of such parameters (and only such parameters) must have `modreq[System.Runtime.InteropServices.InAttribute]`.

**Motivation**: this is done to ensure that in a case of method overriding/implementing the `in` parameters match.

Same requirements apply to `Invoke` methods in delegates.

**Motivation**: this is to ensure that existing compilers cannot simply ignore `readonly` when creating or assigning delegates.

# Returning by readonly reference.

**Motivation**

The motivation for this sub-feature is roughly symmetrical to the reasons for the `in` parameters - avoiding copying, but on the returning side. Prior to this feature, a method or an indexer had two options: 1) return by reference and be exposed to possible mutations or 2) return by value which results in copying.

**Solution (`ref readonly` returns)**

The feature allows a member to return variables by reference without exposing them to mutations.

**Declaring `ref readonly` returning members**

A combination of modifiers `ref readonly` on the return signature is used to to indicate that the member returns a readonly reference.

For all purposes a `ref readonly` member is treated as a `readonly` variable - similar to `readonly` fields and `in` parameters.

For example fields of `ref readonly` member which has a struct type are all recursively classified as `readonly` variables. - It is permitted to pass them as `in` arguments, but not as `ref` or `out` arguments.

```
ref readonly Guid Method1()
{
}

Method2(in Method1()); // valid. Can pass as `in` argument.

Method3(ref Method1()); // not valid. Cannot pass as `ref` argument
```

- `ref readonly` returns are allowed in the same places were `ref` returns are allowed. This includes indexers, delegates, lambdas, local functions.
- It is not permitted to overload on `ref` / `ref readonly` / differences.
- It is permitted to overload on ordinary byval and `ref readonly` return differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `ref readonly` is similar but distinct from `ref`. For example the a method that overrides `ref readonly` one, must itself be `ref readonly` and have identity-convertible type.

- For the purpose of delegate/lambda/method group conversions, `ref readonly` is similar but distinct from `ref`. Lambdas and applicable method group conversion candidates have to match `ref readonly` return of the target delegate with `ref readonly` return of the type that is identity-convertible.
- For the purpose of generic variance, `ref readonly` returns are nonvariant.

> NOTE: There are no warnings on `ref readonly` returns that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `ref readonly T Method()` when `T` was substituted to be `int`.
>
> It is conceivable to have an analyzer that warns in cases of inefficient use of `ref readonly` returns, but the rules for such analysis would be too fuzzy to be a part of a language specification.

#### Returning from `ref readonly` members

Inside the method body the syntax is the same as with regular ref returns. The `readonly` will be inferred from the containing method.

The motivation is that `return ref readonly <expression>` is unnecessary long and only allows for mismatches on the `readonly` part that would always result in errors. The `ref` is, however, required for consistency with other scenarios where something is passed via strict aliasing vs. by value.

> Unlike the case with `in` parameters, `ref readonly` returns never return via a local copy. Considering that the copy would cease to exist immediately upon returning such practice would be pointless and dangerous. Therefore `ref readonly` returns are always direct references.

Example:

```
struct ImmutableArray<T>
{
    private readonly T[] array;

    public ref readonly T ItemRef(int i)
    {
        // returning a readonly reference to an array element
        return ref this.array[i];
    }
}
```

- An argument of `return ref` must be an LValue (**existing rule**)
- An argument of `return ref` must be "safe to return" (**existing rule**)
- In a `ref readonly` member an argument of `return ref` is *not required to be writeable*. For example such member can ref-return a readonly field or one of its `in` parameters.

#### Safe to Return rules.

Normal safe to return rules for references will apply to readonly references as well.

Note that a `ref readonly` can be obtained from a regular `ref` local/parameter/return, but not the other way around. Otherwise the safety of `ref readonly` returns is inferred the same way as for regular `ref` returns.

Considering that RValues can be passed as `in` parameter and returned as `ref readonly` we need one more rule - **RValues are not safe-to-return by reference**.

> Consider the situation when an RValue is passed to an `in` parameter via a copy and then returned back in a form of a `ref readonly`. In the context of the caller the result of such invocation is a reference to local data and

> as such is unsafe to return. Once RValues are not safe to return, the existing rule `#6` already handles this case.

Example:

```
ref readonly Vector3 Test1()
{
    // can pass an RValue as "in" (via a temp copy)
    // but the result is not safe to return
    // because the RValue argument was not safe to return by reference
    return ref Test2(default(Vector3));
}

ref readonly Vector3 Test2(in Vector3 r)
{
    // this is ok, r is returnable
    return ref r;
}
```

Updated `safe to return` rules:

1. **refs to variables on the heap are safe to return**
2. **ref/in parameters are safe to return** `in` parameters naturally can only be returned as readonly.
3. **out parameters are safe to return** (but must be definitely assigned, as is already the case today)
4. **instance struct fields are safe to return as long as the receiver is safe to return**
5. **'this' is not safe to return from struct members**
6. **a ref, returned from another method is safe to return if all refs/outs passed to that method as formal parameters were safe to return.** *Specifically it is irrelevant if receiver is safe to return, regardless whether receiver is a struct, class or typed as a generic type parameter.*
7. **RValues are not safe to return by reference.** *Specifically RValues are safe to pass as in parameters.*

> NOTE: There are additional rules regarding safety of returns that come into play when ref-like types and ref-reassignments are involved. The rules equally apply to `ref` and `ref readonly` members and therefore are not mentioned here.

**Aliasing behavior.**

`ref readonly` members provide the same aliasing behavior as ordinary `ref` members (except for being readonly). Therefore for the purpose of capturing in lambdas, async, iterators, stack spilling etc... the same restrictions apply. - I.E. due to inability to capture the actual references and due to side-effecting nature of member evaluation such scenarios are disallowed.

> It is permitted and required to make a copy when `ref readonly` return is a receiver of regular struct methods, which take `this` as an ordinary writeable reference. Historically in all cases where such invocations are applied to readonly variable a local copy is made.

**Metadata representation.**

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to the return of a byref returning method, it means that the method returns a readonly reference.

In addition, the result signature of such methods (and only those methods) must have `modreq[System.Runtime.CompilerServices.IsReadOnlyAttribute]`.

**Motivation**: this is to ensure that existing compilers cannot simply ignore `readonly` when invoking methods with `ref readonly` returns

# Readonly structs

In short - a feature that makes `this` parameter of all instance members of a struct, except for constructors, an `in` parameter.

### Motivation

Compiler must assume that any method call on a struct instance may modify the instance. Indeed a writeable reference is passed to the method as `this` parameter and fully enables this behavior. To allow such invocations on `readonly` variables, the invocations are applied to temp copies. That could be unintuitive and sometimes forces people to abandon `readonly` for performance reasons.
Example: https://codeblog.jonskeet.uk/2014/07/16/micro-optimization-the-surprising-inefficiency-of-readonly-fields/

After adding support for `in` parameters and `ref readonly` returns the problem of defensive copying will get worse since readonly variables will become more common.

### Solution

Allow `readonly` modifier on struct declarations which would result in `this` being treated as `in` parameter on all struct instance methods except for constructors.

```
static void Test(in Vector3 v1)
{
    // no need to make a copy of v1 since Vector3 is a readonly struct
    System.Console.WriteLine(v1.ToString());
}

readonly struct Vector3
{
    . . .

    public override string ToString()
    {
        // not OK!!  `this` is an `in` parameter
        foo(ref this.X);

        // OK
        return $"X: {X}, Y: {Y}, Z: {Z}";
    }
}
```

### Restrictions on members of readonly struct

- Instance fields of a readonly struct must be readonly.
  **Motivation:** can only be written to externally, but not through members.
- Instance autoproperties of a readonly struct must be get-only.
  **Motivation:** consequence of restriction on instance fields.
- Readonly struct may not declare field-like events.
  **Motivation:** consequence of restriction on instance fields.

### Metadata representation.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a value type, it means that the type is a `readonly struct`.

In particular:

- The identity of the `IsReadOnlyAttribute` type is unimportant. In fact it can be embedded by the compiler in the containing assembly if needed.

# `ref` / `in` extension methods

There is actually an existing proposal (https://github.com/dotnet/roslyn/issues/165) and corresponding prototype PR (https://github.com/dotnet/roslyn/pull/15650). I just want to acknowledge that this idea is not entirely new. It is, however, relevant here since `ref readonly` elegantly removes the most contentious issue about such methods - what to do with RValue receivers.

The general idea is allowing extension methods to take the `this` parameter by reference, as long as the type is known to be a struct type.

```
public static void Extension(ref this Guid self)
{
    // do something
}
```

The reasons for writing such extension methods are primarily:

1. Avoid copying when receiver is a large struct
2. Allow mutating extension methods on structs

The reasons why we do not want to allow this on classes

1. It would be of very limited purpose.
2. It would break long standing invariant that a method call cannot turn non-`null` receiver to become `null` after invocation.

> In fact, currently a non-`null` variable cannot become `null` unless *explicitly* assigned or passed by `ref` or `out`. That greatly aids readability or other forms of "can this be a null here" analysis. 3. It would be hard to reconcile with "evaluate once" semantics of null-conditional accesses. Example: `obj.stringField?.RefExtension(...)` - need to capture a copy of `stringField` to make the null check meaningful, but then assignments to `this` inside RefExtension would not be reflected back to the field.

An ability to declare extension methods on **structs** that take the first argument by reference was a long-standing request. One of the blocking consideration was "what happens if receiver is not an LValue?".

- There is a precedent that any extension method could also be called as a static method (sometimes it is the only way to resolve ambiguity). It would dictate that RValue receivers should be disallowed.
- On the other hand there is a practice of making invocation on a copy in similar situations when struct instance methods are involved.

The reason why the "implicit copying" exists is because the majority of struct methods do not actually modify the struct while not being able to indicate that. Therefore the most practical solution was to just make the invocation on a copy, but this practice is known for harming performance and causing bugs.

Now, with availability of `in` parameters, it is possible for an extension to signal the intent. Therefore the conundrum can be resolved by requiring `ref` extensions to be called with writeable receivers while `in` extensions permit implicit copying if necessary.

```
// this can be called on either RValue or an LValue
public static void Reader(in this Guid self)
{
    // do something nonmutating.
    WriteLine(self == default(Guid));
}

// this can be called only on an LValue
public static void Mutator(ref this Guid self)
{
    // can mutate self
    self = new Guid();
}
```

**`in` extensions and generics.**

The purpose of `ref` extension methods is to mutate the receiver directly or by invoking mutating members. Therefore `ref this T` extensions are allowed as long as `T` is constrained to be a struct.

On the other hand `in` extension methods exist specifically to reduce implicit copying. However any use of an `in T` parameter will have to be done through an interface member. Since all interface members are considered mutating, any such use would require a copy. - Instead of reducing copying, the effect would be the opposite. Therefore `in this T` is not allowed when `T` is a generic type parameter regardless of constraints.

**Valid kinds of extension methods (recap):**

The following forms of `this` declaration in an extension method are now allowed:

1. `this T arg` - regular byval extension. (**existing case**)

- T can be any type, including reference types or type parameters. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues.
- `in this T self` - `in` extension. T must be an actual struct type. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues (may be invoked on a temp if needed).
- `ref this T self` - `ref` extension. T must be a struct type or a generic type parameter constrained to be a struct. Instance may be written to by the invocation. Allows only identity conversions. Must be called on writeable LValue. (never invoked via a temp).

# Readonly ref locals.

**Motivation.**

Once `ref readonly` members were introduced, it was clear from the use that they need to be paired with appropriate kind of local. Evaluation of a member may produce or observe side effects, therefore if the result must be used more than once, it needs to be stored. Ordinary `ref` locals do not help here since they cannot be assigned a `readonly` reference.

**Solution.**

Allow declaring `ref readonly` locals. This is a new kind of `ref` locals that is not writeable. As a result `ref readonly` locals can accept references to readonly variables without exposing these variables to writes.

**Declaring and using `ref readonly` locals.**

The syntax of such locals uses `ref readonly` modifiers at declaration site (in that specific order). Similarly to ordinary `ref` locals, `ref readonly` locals must be ref-initialized at declaration. Unlike regular `ref` locals, `ref readonly` locals can refer to `readonly` LValues like `in` parameters, `readonly` fields, `ref readonly` methods.

For all purposes a `ref readonly` local is treated as a `readonly` variable. Most of the restrictions on the use are the same as with `readonly` fields or `in` parameters.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly` variables .

```
static readonly ref Vector3 M1() => . . .

static readonly ref Vector3 M1_Trace()
{
    // OK
    ref readonly var r1 = ref M1();

    // Not valid. Need an LValue
    ref readonly Vector3 r2 = ref default(Vector3);

    // Not valid. r1 is readonly.
    Mutate(ref r1);

    // OK.
    Print(in r1);

    // OK.
    return ref r1;
}
```

**Restrictions on use of `ref readonly` locals**

Except for their `readonly` nature, `ref readonly` locals behave like ordinary `ref` locals and are subject to exactly same restrictions.
For example restrictions related to capturing in closures, declaring in `async` methods or the `safe-to-return` analysis equally applies to `ref readonly` locals.

## Ternary `ref` expressions. (aka "Conditional LValues")

**Motivation**

Use of `ref` and `ref readonly` locals exposed a need to ref-initialize such locals with one or another target variable based on a condition.

A typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
         return ref consequence;
    }
    else
    {
         return ref alternative;
    }
}
```

Note that `Choice` is not an exact replacement of a ternary since *all* arguments must be evaluated at the call site, which was leading to unintuitive behavior and bugs.

The following will not work as expected:

```
// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

**Solution**

Allow special kind of conditional expression that evaluates to a reference to one of LValue argument based on a condition.

**Using `ref` ternary expression.**

The syntax for the `ref` flavor of a conditional expression is `<condition> ? ref <consequence> : ref <alternative>;`

Just like with the ordinary conditional expression only `<consequence>` or `<alternative>` is evaluated depending on result of the boolean condition expression.

Unlike ordinary conditional expression, `ref` conditional expression:

- requires that `<consequence>` and `<alternative>` are LValues.
- `ref` conditional expression itself is an LValue and
- `ref` conditional expression is writeable if both `<consequence>` and `<alternative>` are writeable LValues

Examples:
`ref` ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;

// error. readOnlyField is readonly and thus conditional expression is readonly
(arr != null ? ref arr[0]: ref obj.readOnlyField) = 1;
```

Can be used as a receiver of a method call and skip copying if necessary.

```
// no copies
(arr != null ? ref arr[0]: ref otherArr[0]).StructMethod();

// invoked on a copy.
// The receiver is `readonly` because readOnlyField is readonly.
(arr != null ? ref arr[0]: ref obj.readOnlyField).StructMethod();

// no copies. `ReadonlyStructMethod` is a method on a `readonly` struct
// and can be invoked directly on a readonly receiver
(arr != null ? ref arr[0]: ref obj.readOnlyField).ReadonlyStructMethod();
```

`ref` ternary can be used in a regular (not ref) context as well.

```
// only an example
// a regular ternary could work here just the same
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

**Drawbacks**

I can see two major arguments against enhanced support for references and readonly references:

1. The problems that are solved here are very old. Why suddenly solve them now, especially since it would not help existing code?

As we find C# and .Net used in new domains, some problems become more prominent.
As examples of environments that are more critical than average about computation overheads, I can list

- cloud/datacenter scenarios where computation is billed for and responsiveness is a competitive advantage.
- Games/VR/AR with soft-realtime requirements on latencies

This feature does not sacrifice any of the existing strengths such as type-safety, while allowing to lower overheads in some common scenarios.

2. Can we reasonably guarantee that the callee will play by the rules when it opts into `readonly` contracts?

We have similar trust when using `out`. Incorrect implementation of `out` can cause unspecified behavior, but in reality it rarely happens.

Making the formal verification rules familiar with `ref readonly` would further mitigate the trust issue.

**Alternatives**

The main competing design is really "do nothing".

**Unresolved questions**

**Design meetings**

https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-02-22.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-03-01.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-08-28.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-25.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-27.md

# Compile time enforcement of safety for ref-like types

11/2/2020 • 16 minutes to read • Edit Online

## Introduction

The main reason for the additional safety rules when dealing with types like `Span<T>` and `ReadOnlySpan<T>` is that such types must be confined to the execution stack.

There are two reasons why `Span<T>` and similar types must be a stack-only types.

1. `Span<T>` is semantically a struct containing a reference and a range - `(ref T data, int length)`. Regardless of actual implementation, writes to such struct would not be atomic. Concurrent "tearing" of such struct would lead to the possibility of `length` not matching the `data`, causing out-of-range accesses and type-safety violations, which ultimately could result in GC heap corruption in seemingly "safe" code.
2. Some implementations of `Span<T>` literally contain a managed pointer in one of its fields. Managed pointers are not supported as fields of heap objects and code that manages to put a managed pointer on the GC heap typically crashes at JIT time.

All the above problems would be alleviated if instances of `Span<T>` are constrained to exist only on the execution stack.

An additional problem arises due to composition. It would be generally desirable to build more complex data types that would embed `Span<T>` and `ReadOnlySpan<T>` instances. Such composite types would have to be structs and would share all the hazards and requirements of `Span<T>`. As a result the safety rules described here should be viewed as applicable to the whole range of ***ref-like types***.

The draft language specification is intended to ensure that values of a ref-like type occur only on the stack.

## Generalized `ref-like` types in source code

`ref-like` structs are explicitly marked in the source code using `ref` modifier:

```
ref struct TwoSpans<T>
{
 // can have ref-like instance fields
 public Span<T> first;
 public Span<T> second;
}

// error: arrays of ref-like types are not allowed.
TwoSpans<T>[] arr = null;
```

Designating a struct as ref-like will allow the struct to have ref-like instance fields and will also make all the requirements of ref-like types applicable to the struct.

## Metadata representation or ref-like structs

Ref-like structs will be marked with **System.Runtime.CompilerServices.IsRefLikeAttribute** attribute.

The attribute will be added to common base libraries such as `mscorlib`. In a case if the attribute is not available, compiler will generate an internal one similarly to other embedded-on-demand attributes such as

`IsReadOnlyAttribute`.

An additional measure will be taken to prevent the use of ref-like structs in compilers not familiar with the safety rules (this includes C# compilers prior to the one in which this feature is implemented).

Having no other good alternatives that work in old compilers without servicing, an `Obsolete` attribute with a known string will be added to all ref-like structs. Compilers that know how to use ref-like types will ignore this particular form of `Obsolete`.

A typical metadata representation:

```
[IsRefLike]
[Obsolete("Types with embedded references are not supported in this version of your compiler.")]
public struct TwoSpans<T>
{
   // . . . .
}
```

NOTE: it is not the goal to make it so that any use of ref-like types on old compilers fails 100%. That is hard to achieve and is not strictly necessary. For example there would always be a way to get around the `Obsolete` using dynamic code or, for example, creating an array of ref-like types through reflection.

In particular, if user wants to actually put an `Obsolete` or `Deprecated` attribute on a ref-like type, we will have no choice other than not emitting the predefined one since `Obsolete` attribute cannot be applied more than once..

## Examples:

```
SpanLikeType M1(ref SpanLikeType x, Span<byte> y)
{
    // this is all valid, unconcerned with stack-referring stuff
    var local = new SpanLikeType(y);
    x = local;
    return x;
}

void Test1(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    // this is allowed
    stackReferring2 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    stackReferring2 = M1(ref param1, stackReferring1);

    // this is NOT allowed
    param1 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    param2 = stackReferring1.Slice(10);

    // this is allowed
    param1 = new SpanLikeType(param2);

    // this is allowed
    stackReferring2 = param1;
}

ref SpanLikeType M2(ref SpanLikeType x)
{
    return ref x;
}

ref SpanLikeType Test2(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    ref var stackReferring3 = M2(ref stackReferring2);

    // this is allowed
    stackReferring3 = M1(ref stackReferring2, stackReferring1);

    // this is allowed
    M2(ref stackReferring3) = stackReferring2;

    // this is NOT allowed
    M1(ref param1) = stackReferring2;

    // this is NOT allowed
    param1 = stackReferring3;

    // this is NOT allowed
    return ref stackReferring3;

    // this is allowed
    return ref param1;
}
```

# Draft language specification

Below we describe a set of safety rules for ref-like types (`ref struct`s) to ensure that values of these types occur only on the stack. A different, simpler set of safety rules would be possible if locals cannot be passed by reference. This specification would also permit the safe reassignment of ref locals.

**Overview**

We associate with each expression at compile-time the concept of what scope that expression is permitted to escape to, "safe-to-escape". Similarly, for each lvalue we maintain a concept of what scope a reference to it is permitted to escape to, "ref-safe-to-escape". For a given lvalue expression, these may be different.

These are analogous to the "safe to return" of the ref locals feature, but it is more fine-grained. Where the "safe-to-return" of an expression records only whether (or not) it may escape the enclosing method as a whole, the safe-to-escape records which scope it may escape to (which scope it may not escape beyond). The basic safety mechanism is enforced as follows. Given an assignment from an expression E1 with a safe-to-escape scope S1, to an (lvalue) expression E2 with safe-to-escape scope S2, it is an error if S2 is a wider scope than S1. By construction, the two scopes S1 and S2 are in a nesting relationship, because a legal expression is always safe-to-return from some scope enclosing the expression.

For the time being it is sufficient, for the purpose of the analysis, to support just two scopes - external to the method, and top-level scope of the method. That is because ref-like values with inner scopes cannot be created and ref locals do not support re-assignment. The rules, however, can support more than two scope levels.

The precise rules for computing the *safe-to-return* status of an expression, and the rules governing the legality of expressions, follow.

**ref-safe-to-escape**

The *ref-safe-to-escape* is a scope, enclosing an lvalue expression, to which it is safe for a ref to the lvalue to escape to. If that scope is the entire method, we say that a ref to the lvalue is *safe to return* from the method.

**safe-to-escape**

The *safe-to-escape* is a scope, enclosing an expression, to which it is safe for the value to escape to. If that scope is the entire method, we say that the value is *safe to return* from the method.

An expression whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method. Otherwise we refer to the rules below.

**Parameters**

An lvalue designating a formal parameter is *ref-safe-to-escape* (by reference) as follows:

- If the parameter is a `ref`, `out`, or `in` parameter, it is *ref-safe-to-escape* from the entire method (e.g. by a `return ref` statement); otherwise
- If the parameter is the `this` parameter of a struct type, it is *ref-safe-to-escape* to the top-level scope of the method (but not from the entire method itself); Sample
- Otherwise the parameter is a value parameter, and it is *ref-safe-to-escape* to the top-level scope of the method (but not from the method itself).

An expression that is an rvalue designating the use of a formal parameter is *safe-to-escape* (by value) from the entire method (e.g. by a `return` statement). This applies to the `this` parameter as well.

**Locals**

An lvalue designating a local variable is *ref-safe-to-escape* (by reference) as follows:

- If the variable is a `ref` variable, then its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of its initializing expression; otherwise
- The variable is *ref-safe-to-escape* the scope in which it was declared.

An expression that is an rvalue designating the use of a local variable is *safe-to-escape* (by value) as follows:

- But the general rule above, a local whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method.
- If the variable is an iteration variable of a `foreach` loop, then the variable's *safe-to-escape* scope is the same as the *safe-to-escape* of the `foreach` loop's expression.
- A local of `ref struct` type and uninitialized at the point of declaration is *safe-to-return* from the entire enclosing method.
- Otherwise the variable's type is a `ref struct` type, and the variable's declaration requires an initializer. The variable's *safe-to-escape* scope is the same as the *safe-to-escape* of its initializer.

#### Field reference

An lvalue designating a reference to a field, `e.F`, is *ref-safe-to-escape* (by reference) as follows:

- If `e` is of a reference type, it is *ref-safe-to-escape* from the entire method; otherwise
- If `e` is of a value type, its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of `e`.

An rvalue designating a reference to a field, `e.F`, has a *safe-to-escape* scope that is the same as the *safe-to-escape* of `e`.

#### Operators including `?:`

The application of a user-defined operator is treated as a method invocation.

For an operator that yields an rvalue, such as `e1 + e2` or `c ? e1 : e2`, the *safe-to-escape* of the result is the narrowest scope among the *safe-to-escape* of the operands of the operator. As a consequence, for a unary operator that yields an rvalue, such as `+e`, the *safe-to-escape* of the result is the *safe-to-escape* of the operand.

For an operator that yields an lvalue, such as `c ? ref e1 : ref e2`

- the *ref-safe-to-escape* of the result is the narrowest scope among the *ref-safe-to-escape* of the operands of the operator.
- the *safe-to-escape* of the operands must agree, and that is the *safe-to-escape* of the resulting lvalue.

#### Method invocation

An lvalue resulting from a ref-returning method invocation `e1.M(e2, ...)` is *ref-safe-to-escape* the smallest of the following scopes:

- The entire enclosing method
- the *ref-safe-to-escape* of all `ref` and `out` argument expressions (excluding the receiver)
- For each `in` parameter of the method, if there is a corresponding expression that is an lvalue, its *ref-safe-to-escape*, otherwise the nearest enclosing scope
- the *safe-to-escape* of all argument expressions (including the receiver)

> Note: the last bullet is necessary to handle code such as
>
> ```
> var sp = new Span(...)
> return ref sp[0];
> ```
>
> or
>
> ```
> return ref M(sp, 0);
> ```

An rvalue resulting from a method invocation `e1.M(e2, ...)` is *safe-to-escape* from the smallest of the following scopes:

- The entire enclosing method
- the *safe-to-escape* of all argument expressions (including the receiver)

#### An Rvalue

An rvalue is *ref-safe-to-escape* from the nearest enclosing scope. This occurs for example in an invocation such as `M(ref d.Length)` where `d` is of type `dynamic`. It is also consistent with (and perhaps subsumes) our handling of arguments corresponding to `in` parameters.

#### Property invocations

A property invocation (either `get` or `set`) it treated as a method invocation of the underlying method by the above rules.

`stackalloc`

A stackalloc expression is an rvalue that is *safe-to-escape* to the top-level scope of the method (but not from the entire method itself).

#### Constructor invocations

A `new` expression that invokes a constructor obeys the same rules as a method invocation that is considered to return the type being constructed.

In addition *safe-to-escape* is no wider than the smallest of the *safe-to-escape* of all arguments/operands of the object initializer expressions, recursively, if initializer is present.

#### Span constructor

The language relies on `Span<T>` not having a constructor of the following form:

```
void Example(ref int x)
{
    // Create a span of length one
    var span = new Span<int>(ref x);
}
```

Such a constructor makes `Span<T>` which are used as fields indistinguishable from a `ref` field. The safety rules described in this document depend on `ref` fields not being a valid construct in C# or .NET.

`default` **expressions**

A `default` expression is *safe-to-escape* from the entire enclosing method.

## Language Constraints

We wish to ensure that no `ref` local variable, and no variable of `ref struct` type, refers to stack memory or variables that are no longer alive. We therefore have the following language constraints:

- Neither a ref parameter, nor a ref local, nor a parameter or local of a `ref struct` type can be lifted into a lambda or local function.
- Neither a ref parameter nor a parameter of a `ref struct` type may be an argument on an iterator method or an `async` method.
- Neither a ref local, nor a local of a `ref struct` type may be in scope at the point of a `yield return` statement or an `await` expression.
- A `ref struct` type may not be used as a type argument, or as an element type in a tuple type.
- A `ref struct` type may not be the declared type of a field, except that it may be the declared type of an instance field of another `ref struct`.

- A `ref struct` type may not be the element type of an array.
- A value of a `ref struct` type may not be boxed:
  - There is no conversion from a `ref struct` type to the type `object` or the type `System.ValueType`.
  - A `ref struct` type may not be declared to implement any interface
  - No instance method declared in `object` or in `System.ValueType` but not overridden in a `ref struct` type may be called with a receiver of that `ref struct` type.
  - No instance method of a `ref struct` type may be captured by method conversion to a delegate type.
- For a ref reassignment `ref e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.
- For a ref return statement `return ref e1`, the *ref-safe-to-escape* of `e1` must be *ref-safe-to-escape* from the entire method. (TODO: Do we also need a rule that `e1` must be *safe-to-escape* from the entire method, or is that redundant?)
- For a return statement `return e1`, the *safe-to-escape* of `e1` must be *safe-to-escape* from the entire method.
- For an assignment `e1 = e2`, if the type of `e1` is a `ref struct` type, then the *safe-to-escape* of `e2` must be at least as wide a scope as the *safe-to-escape* of `e1`.
- For a method invocation if there is a `ref` or `out` argument of a `ref struct` type (including the receiver), with *safe-to-escape* E1, then no argument (including the receiver) may have a narrower *safe-to-escape* than E1. Sample
- A local function or anonymous function may not refer to a local or parameter of `ref struct` type declared in an enclosing scope.

> ***Open Issue:*** We need some rule that permits us to produce an error when needing to spill a stack value of a `ref struct` type at an await expression, for example in the code
>
> ```
> Foo(new Span<int>(...), await e2);
> ```

# Explanations

These explanations and samples help explain why many of the safety rules above exist

**Method Arguments Must Match**

When invoking a method where there is an `out`, `ref` parameter that is a `ref struct` including the receiver then all of the `ref struct` need to have the same lifetime. This is necessary because C# must make all of its decisions around lifetime safety based on the information available in the signature of the method and the lifetime of the values at the call site.

When there are `ref` parameters that are `ref struct` then there is the possiblity they could swap around their contents. Hence at the call site we must ensure all of these **potential** swaps are compatible. If the language didn't enforce that then it will allow for bad code like the following.

```
void M1(ref Span<int> s1)
{
    Span<int> s2 = stackalloc int[1];
    Swap(ref s1, ref s2);
}

void Swap(ref Span<int> x, ref Span<int> y)
{
    // This will effectively assign the stackalloc to the s1 parameter and allow it
    // to escape to the caller of M1
    ref x = ref y;
}
```

The restriction on the receiver is necessary because while none of its contents are ref-safe-to-escape it can store the provided values. This means with mismatched lifetimes you could create a type safety hole in the following way:

```
ref struct S
{
    public Span<int> Span;

    public void Set(Span<int> span)
    {
        Span = span;
    }
}

void Broken(ref S s)
{
    Span<int> span = stackalloc int[1];

    // The result of a stackalloc is now stored in s.Span and escaped to the caller
    // of Broken
    s.Set(span);
}
```

**Struct This Escape**

When it comes to span safety rules, the `this` value in an instance member is modeled as a parameter to the member. Now for a `struct` the type of `this` is actually `ref S` where in a `class` it's simply `S` (for members of a `class / struct` named S).

Yet `this` has different escaping rules than other `ref` parameters. Specifically it is not ref-safe-to-escape while other parameters are:

```
ref struct S
{
    int Field;

    // Illegal because `this` isn't safe to escape as ref
    ref int Get() => ref Field;

    // Legal
    ref int GetParam(ref int p) => ref p;
}
```

The reason for this restriction actually has little to do with `struct` member invocation. There are some rules that need to be worked out with respect to member invocation on `struct` members where the receiver is an rvalue. But that is very approachable.

The reason for this restriction is actually about interface invocation. Specifically it comes down to whether or not the following sample should or should not compile;

```
interface I1
{
    ref int Get();
}

ref int Use<T>(T p)
    where T : I1
{
    return ref p.Get();
}
```

Consider the case where `T` is instantiated as a `struct`. If the `this` parameter is ref-safe-to-escape then the return of `p.Get` could point to the stack (specifically it could be a field inside of the instantiated type of `T`). That means the language could not allow this sample to compile as it could be returning a `ref` to a stack location. On the other hand if `this` is not ref-safe-to-escape then `p.Get` cannot refer to the stack and hence it's safe to return.

This is why the escapability of `this` in a `struct` is really all about interfaces. It can absolutely be made to work but it has a trade off. The design eventually came down in favor of making interfaces more flexible.

There is potential for us to relax this in the future though.

## Future Considerations

**Length one Span<T> over ref values**

Though not legal today there are cases where creating a length one `Span<T>` instance over a value would be beneficial:

```
void RefExample()
{
    int x = ...;

    // Today creating a length one Span<int> requires a stackalloc and a new
    // local
    Span<int> span1 = stackalloc [] { x };
    Use(span1);
    x = span1[0];

    // Simpler to just allow length one span
    var span2 = new Span<int>(ref x);
    Use(span2);
}
```

This feature gets more compelling if we lift the restrictions on fixed sized buffers as it would allow for `Span<T>` instances of even greater length.

If there is ever a need to go down this path then the language could accommodate this by ensuring such `Span<T>` instances were downward facing only. That is they were only ever *safe-to-escape* to the scope in which they were created. This ensure the language never had to consider a `ref` value escaping a method via a `ref struct` return or field of `ref struct`. This would likely also require further changes to recognize such constructors as capturing a `ref` parameter in this way though.

# Non-trailing named arguments

11/2/2020 • 3 minutes to read • Edit Online

## Summary

Allow named arguments to be used in non-trailing position, as long as they are used in their correct position. For example: `DoSomething(isEmployed:true, name, age);`.

## Motivation

The main motivation is to avoid typing redundant information. It is common to name an argument that is a literal (such as `null`, `true`) for the purpose of clarifying the code, rather than of passing arguments out-of-order. That is currently disallowed (`CS1738`) unless all the following arguments are also named.

```
DoSomething(isEmployed:true, name, age); // currently disallowed, even though all arguments are in position
// CS1738 "Named argument specifications must appear after all fixed arguments have been specified"
```

Some additional examples:

```
public void DoSomething(bool isEmployed, string personName, int personAge) { ... }

DoSomething(isEmployed:true, name, age); // currently CS1738, but would become legal
DoSomething(true, personName:name, age); // currently CS1738, but would become legal
DoSomething(name, isEmployed:true, age); // remains illegal
DoSomething(name, age, isEmployed:true); // remains illegal
DoSomething(true, personAge:age, personName:name); // already legal
```

This would also work with params:

```
public class Task
{
    public static Task When(TaskStatus all, TaskStatus any, params Task[] tasks);
}
Task.When(all: TaskStatus.RanToCompletion, any: TaskStatus.Faulted, task1, task2)
```

## Detailed design

In §7.5.1 (Argument lists), the spec currently says:

> An *argument* with an *argument-name* is referred to as a **named argument**, whereas an *argument* without an *argument-name* is a **positional argument**. It is an error for a positional argument to appear after a named argument in an *argument-list*.

The proposal is to remove this error and update the rules for finding the corresponding parameter for an argument (§7.5.1.1):

Arguments in the argument-list of instance constructors, methods, indexers and delegates:

- [existing rules]
- An unnamed argument corresponds to no parameter when it is after an out-of-position named argument or a

named params argument.

In particular, this prevents invoking `void M(bool a = true, bool b = true, bool c = true, );` with `M(c: false, valueB);`. The first argument is used out-of-position (the argument is used in first position, but the parameter named "c" is in third position), so the following arguments should be named.

In other words, non-trailing named arguments are only allowed when the name and the position result in finding the same corresponding parameter.

# Drawbacks

This proposal exacerbates existing subtleties with named arguments in overload resolution. For instance:

```
void M(int x, int y) { }
void M<T>(T y, int x) { }

void M2()
{
    M(3, 4);
    M(y: 3, x: 4); // Invokes M(int, int)
    M(y: 3, 4); // Invokes M<T>(T, int)
}
```

You could get this situation today by swapping the parameters:

```
void M(int y, int x) { }
void M<T>(int x, T y) { }

void M2()
{
    M(3, 4);
    M(x: 3, y: 4); // Invokes M(int, int)
    M(3, y: 4); // Invokes M<T>(int, T)
}
```

Similarly, if you have two methods `void M(int a, int b)` and `void M(int x, string y)`, the mistaken invocation `M(x: 1, 2)` will produce a diagnostic based on the second overload ("cannot convert from 'int' to 'string'"). This problem already exists when the named argument is used in a trailing position.

# Alternatives

There are a couple of alternatives to consider:

- The status quo
- Providing IDE assistance to fill-in all the names of trailing arguments when you type specific a name in the middle.

Both of those suffer from more verbosity, as they introduce multiple named arguments even if you just need one name of a literal at the beginning of the argument list.

# Unresolved questions

# Design meetings

The feature was briefly discussed in LDM on May 16th 2017, with approval in principle (ok to move to proposal/prototype). It was also briefly discussed on June 28th 2017.

Relates to initial discussion https://github.com/dotnet/csharplang/issues/518 Relates to championed issue https://github.com/dotnet/csharplang/issues/570

# private protected

11/2/2020 • 9 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype: Complete
- [x] Implementation: Complete
- [x] Specification: Complete

## Summary

Expose the CLR `protectedAndInternal` accessibility level in C# as `private protected`.

## Motivation

There are many circumstances in which an API contains members that are only intended to be implemented and used by subclasses contained in the assembly that provides the type. While the CLR provides an accessibility level for that purpose, it is not available in C#. Consequently API owners are forced to either use `internal` protection and self-discipline or a custom analyzer, or to use `protected` with additional documentation explaining that, while the member appears in the public documentation for the type, it is not intended to be part of the public API. For examples of the latter, see members of Roslyn's `CSharpCompilationOptions` whose names start with `Common`.

Directly providing support for this access level in C# enables these circumstances to be expressed naturally in the language.

## Detailed design

**`private protected` access modifier**

We propose to add a new access modifier combination `private protected` (which can appear in any order among the modifiers). This maps to the CLR notion of protectedAndInternal, and borrows the same syntax currently used in C++/CLI.

A member declared `private protected` can be accessed within a subclass of its container if that subclass is in the same assembly as the member.

We modify the language specification as follows (additions in bold). Section numbers are not shown below as they may vary depending on which version of the specification it is integrated into.

---

> The declared accessibility of a member can be one of the following:

- Public, which is selected by including a public modifier in the member declaration. The intuitive meaning of public is "access not limited".
- Protected, which is selected by including a protected modifier in the member declaration. The intuitive meaning of protected is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an internal modifier in the member declaration. The intuitive meaning of internal is "access limited to this assembly".
- Protected internal, which is selected by including both a protected and an internal modifier in the member declaration. The intuitive meaning of protected internal is "accessible within this assembly as well as types derived from the containing class".

- **Private protected, which is selected by including both a private and a protected modifier in the member declaration. The intuitive meaning of private protected is "accessible within this assembly by types derived from the containing class".**

---

> Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have public declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared directly in compilation units or namespaces (as opposed to within other types) can have public or internal declared accessibility and default to internal declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to private declared accessibility. [Note: A type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Struct members can have public, internal, or private declared accessibility and default to private declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have protected*,* ~~or~~ protected internal**, or private protected** declared accessibility. [Note: A type declared as a member of a struct can have public, internal, or private declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Interface members implicitly have public declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have public declared accessibility. No access modifiers are allowed on enumeration member declarations.

---

> The accessibility domain of a nested member M declared in a type T within a program P, is defined as follows (noting that M itself might possibly be a type):

- If the declared accessibility of M is public, the accessibility domain of M is the accessibility domain of T.
- If the declared accessibility of M is protected internal, let D be the union of the program text of P and the program text of any type derived from T, which is declared outside P. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- **If the declared accessibility of M is private protected, let D be the intersection of the program text of P and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.**
- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

---

> When a protected **or private protected** instance member is accessed outside the program text of the class in which it is declared, and when a protected internal instance member is accessed outside the program text of the program in which it is declared, the access shall take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

> The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§9.5.2):

- Types declared in compilation units or namespaces can have public or internal access. The default is internal access.
- Types declared in classes can have public, protected internal, **private protected**, protected, internal, or private access. The default is private access.
- Types declared in structs can have public, internal, or private access. The default is private access.

---

> A static class declaration is subject to the following restrictions:

- A static class shall not include a sealed or abstract modifier. (However, since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.)
- A static class shall not include a class-base specification (§16.2.5) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type object.
- A static class shall only contain static members (§16.4.8). [Note: All constants and nested types are classified as static members. end note]
- A static class shall not have members with protected**, private protected** or protected internal declared accessibility.

> It is a compile-time error to violate any of these restrictions.

---

> A class-member-declaration can have any one of the ~~five~~**six** possible kinds of declared accessibility (§9.5.2): public, **private protected**, protected internal, protected, internal, or private. Except for the protected internal **and private protected** combination**s**, it is a compile-time error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, private is assumed.

---

> Non-nested types can have public or internal declared accessibility and have internal declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of ~~five~~**six** forms of declared accessibility (public, **private protected**, protected internal, protected, internal, or private) and, like other class members, defaults to private declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (public, internal, or private) and, like other struct members, defaults to private declared accessibility.

---

> The method overridden by an override declaration is known as the overridden base method For an override method M declared in a class C, the overridden base method is determined by examining each base class type of C, starting with the direct base class type of C and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as M after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is **either** internal **or private protected** and declared in the same program as C.

---

> The use of accessor-modifiers is governed by the following restrictions:

- An accessor-modifier shall not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no override modifier, an accessor-modifier is permitted only if the property or indexer has both a get and set accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an override modifier, an accessor shall match the accessor-modifier, if any, of the accessor being overridden.
- The accessor-modifier shall declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of public, the accessor-modifier may be either **private protected**, , protected internal, internal, protected, or private.
  - If the property or indexer has a declared accessibility of protected internal, the accessor-modifier may be either **private protected**, internal, protected, or private.
  - If the property or indexer has a declared accessibility of internal or protected, the accessor-modifier shall be **either private protected or** private.
  - **If the property or indexer has a declared accessibility of private protected, the accessor-modifier shall be private.**
  - If the property or indexer has a declared accessibility of private, no accessor-modifier may be used.

---

> Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be protected, **private protected**, or protected internal.

---

## Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

## Alternatives

An alternative would be the provision of an API combining an attribute and an analyzer. The attribute is placed by the programmer on an `internal` member to indicates that the member is intended to be used only in subclasses, and the analyzer checks that those restrictions are obeyed.

## Unresolved questions

The implementation is largely complete. The only open work item is drafting a corresponding specification for VB.

## Design meetings

TBD

# Conditional ref expressions

11/2/2020 • 2 minutes to read • Edit Online

The pattern of binding a ref variable to one or another expression conditionally is not currently expressible in C#.

The typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
         return ref consequence;
    }
    else
    {
         return ref alternative;
    }
}
```

Note that this is not an exact replacement of a ternary since all arguments must be evaluated at the call site.

The following will not work as expected:

```
       // will crash with NRE because 'arr[0]' will be executed unconditionally
      ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

The proposed syntax would look like:

```
     <condition> ? ref <consequence> : ref <alternative>;
```

The above attempt with "Choice" can be *correctly* written using ref ternary as:

```
     ref var r = ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

The difference from Choice is that consequence and alternative expressions are accessed in a *truly* conditional manner, so we do not see a crash if `arr == null`

The ternary ref is just a ternary where both alternative and consequence are refs. It will naturally require that consequence/alternative operands are LValues. It will also require that consequence and alternative have types that are identity convertible to each other.

The type of the expression will be computed similarly to the one for the regular ternary. I.E. in a case if consequence and alternative have identity convertible, but different types, the existing type-merging rules will apply.

Safe-to-return will be assumed conservatively from the conditional operands. If either is unsafe to return the whole thing is unsafe to return.

Ref ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;
```

Ref ternary can be used in a regular (not ref) context as well. Although it would not be common since you could as well just use a regular ternary.

```
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

---

Implementation notes:

The complexity of the implementation would seem to be the size of a moderate-to-large bug fix. - I.E not very expensive. I do not think we need any changes to the syntax or parsing. There is no effect on metadata or interop. The feature is completely expression based. No effect on debugging/PDB either

# Allow digit separator after 0b or 0x

11/2/2020 • 2 minutes to read • Edit Online

In C# 7.2, we extend the set of places that digit separators (the underscore character) can appear in integral literals. Beginning in C# 7.0, separators are permitted between the digits of a literal. Now, in C# 7.2, we also permit digit separators before the first significant digit of a binary or hexadecimal literal, after the prefix.

```
123      // permitted in C# 1.0 and later
1_2_3    // permitted in C# 7.0 and later
0x1_2_3  // permitted in C# 7.0 and later
0b101    // binary literals added in C# 7.0
0b1_0_1  // permitted in C# 7.0 and later

// in C# 7.2, _ is permitted after the `0x` or `0b`
0x_1_2   // permitted in C# 7.2 and later
0b_1_0_1 // permitted in C# 7.2 and later
```

We do not permit a decimal integer literal to have a leading underscore. A token such as `_123` is an identifier.

# Unmanaged type constraint

11/2/2020 • 4 minutes to read • Edit Online

## Summary

The unmanaged constraint feature will give language enforcement to the class of types known as "unmanaged types" in the C# language spec. This is defined in section 18.2 as a type which is not a reference type and doesn't contain reference type fields at any level of nesting.

## Motivation

The primary motivation is to make it easier to author low level interop code in C#. Unmanaged types are one of the core building blocks for interop code, yet the lack of support in generics makes it impossible to create re-usable routines across all unmanaged types. Instead developers are forced to author the same boiler plate code for every unmanaged type in their library:

```
int Hash(Point point) { ... }
int Hash(TimeSpan timeSpan) { ... }
```

To enable this type of scenario the language will be introducing a new constraint: unmanaged:

```
void Hash<T>(T value) where T : unmanaged
{
    ...
}
```

This constraint can only be met by types which fit into the unmanaged type definition in the C# language spec. Another way of looking at it is that a type satisfies the unmanaged constraint iff it can also be used as a pointer.

```
Hash(new Point()); // Okay
Hash(42); // Okay
Hash("hello") // Error: Type string does not satisfy the unmanaged constraint
```

Type parameters with the unmanaged constraint can use all the features available to unmanaged types: pointers, fixed, etc ...

```
void Hash<T>(T value) where T : unmanaged
{
    // Okay
    fixed (T* p = &value)
    {
        ...
    }
}
```

This constraint will also make it possible to have efficient conversions between structured data and streams of bytes. This is an operation that is common in networking stacks and serialization layers:

```
Span<byte> Convert<T>(ref T value) where T : unmanaged
{
    ...
}
```

Such routines are advantageous because they are provably safe at compile time and allocation free. Interop authors today can not do this (even though it's at a layer where perf is critical). Instead they need to rely on allocating routines that have expensive runtime checks to verify values are correctly unmanaged.

## Detailed design

The language will introduce a new constraint named `unmanaged`. In order to satisfy this constraint a type must be a struct and all the fields of the type must fall into one of the following categories:

- Have the type `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `IntPtr` or `UIntPtr`.
- Be any `enum` type.
- Be a pointer type.
- Be a user defined struct that satisfies the `unmanaged` constraint.

Compiler generated instance fields, such as those backing auto-implemented properties, must also meet these constraints.

For example:

```
// Unmanaged type
struct Point
{
    int X;
    int Y {get; set;}
}

// Not an unmanaged type
struct Student
{
    string FirstName;
    string LastName;
}
```

The `unmanaged` constraint cannot be combined with `struct`, `class` or `new()`. This restriction derives from the fact that `unmanaged` implies `struct` hence the other constraints do not make sense.

The `unmanaged` constraint is not enforced by CLR, only by the language. To prevent mis-use by other languages, methods which have this constraint will be protected by a mod-req. This will prevent other languages from using type arguments which are not unmanaged types.

The token `unmanaged` in the constraint is not a keyword, nor a contextual keyword. Instead it is like `var` in that it is evaluated at that location and will either:

- Bind to user defined or referenced type named `unmanaged`: This will be treated just as any other named type constraint is treated.
- Bind to no type: This will be interpreted as the `unmanaged` constraint.

In the case there is a type named `unmanaged` and it is available without qualification in the current context, then there will be no way to use the `unmanaged` constraint. This parallels the rules surrounding the feature `var` and user defined types of the same name.

## Drawbacks

The primary drawback of this feature is that it serves a small number of developers: typically low level library authors or frameworks. Hence it's spending precious language time for a small number of developers.

Yet these frameworks are often the basis for the majority of .NET applications out there. Hence performance / correctness wins at this level can have a ripple effect on the .NET ecosystem. This makes the feature worth considering even with the limited audience.

## Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the implicit opt in behavior.

## Questions

**Metadata Representation**

The F# language encodes the constraint in the signature file which means C# cannot re-use their representation. A new attribute will need to be chosen for this constraint. Additionally a method which has this constraint must be protected by a mod-req.

**Blittable vs. Unmanaged**

The F# language has a very similar feature which uses the keyword unmanaged. The blittable name comes from the use in Midori. May want to look to precedence here and use unmanaged instead.

**Resolution** The language decide to use unmanaged

**Verifier**

Does the verifier / runtime need to be updated to understand the use of pointers to generic type parameters? Or can it simply work as is without changes?

**Resolution** No changes needed. All pointer types are simply unverifiable.

## Design meetings

n/a

# Indexing `fixed` fields should not require pinning regardless of the movable/unmovable context.

11/2/2020 • 2 minutes to read • Edit Online

The change has the size of a bug fix. It can be in 7.3 and does not conflict with whatever direction we take further. This change is only about allowing the following scenario to work even though `s` is moveable. It is already valid when `s` is not moveable.

NOTE: in either case, it still requires `unsafe` context. It is possible to read uninitialized data or even out of range. That is not changing.

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

class Program
{
    static S s;

    unsafe static void Main()
    {
        int p = s.myFixedField[5]; // indexing fixed-size array fields would be ok
    }
}
```

The main "challenge" that I see here is how to explain the relaxation in the spec. In particular, since the following would still need pinning. (because `s` is moveable and we explicitly use the field as a pointer)

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

class Program
{
    static S s;

    unsafe static void Main()
    {
        int* ptr = s.myFixedField; // taking a pointer explicitly still requires pinning.
        int p = ptr[5];
    }
}
```

One reason why we require pinning of the target when it is movable is the artifact of our code generation strategy, - we always convert to an unmanaged pointer and thus force the user to pin via `fixed` statement. However, conversion to unmanaged is unnecessary when doing indexing. The same unsafe pointer math is equally applicable when we have the receiver in the form of a managed pointer. If we do that, then the intermediate ref is managed (GC-tracked) and the pinning is unnecessary.

The change https://github.com/dotnet/roslyn/pull/24966 is a prototype PR that relaxes this requirement.

# Pattern-based `fixed` statement

11/2/2020 • 3 minutes to read • Edit Online

## Summary

Introduce a pattern that would allow types to participate in `fixed` statements.

## Motivation

The language provides a mechanism for pinning managed data and obtain a native pointer to the underlying buffer.

```
fixed(byte* ptr = byteArray)
{
   // ptr is a native pointer to the first element of the array
   // byteArray is protected from being moved/collected by the GC for the duration of this block
}
```

The set of types that can participate in `fixed` is hardcoded and limited to arrays and `System.String`. Hardcoding "special" types does not scale when new primitives such as `ImmutableArray<T>`, `Span<T>`, `Utf8String` are introduced.

In addition, the current solution for `System.String` relies on a fairly rigid API. The shape of the API implies that `System.String` is a contiguous object that embeds UTF16 encoded data at a fixed offset from the object header. Such approach has been found problematic in several proposals that could require changes to the underlying layout. It would be desirable to be able to switch to something more flexible that decouples `System.String` object from its internal representation for the purpose of unmanaged interop.

## Detailed design

### *Pattern*

A viable pattern-based “fixed” need to:

- Provide the managed references to pin the instance and to initialize the pointer (preferably this is the same reference)
- Convey unambiguously the type of the unmanaged element (i.e. “char” for “string”)
- Prescribe the behavior in "empty" case when there is nothing to refer to.
- Should not push API authors toward design decisions that hurt the use of the type outside of `fixed`.

I think the above could be satisfied by recognizing a specially named ref-returning member: `ref [readonly] T GetPinnableReference()`.

In order to be used by the `fixed` statement the following conditions must be met:

1. There is only one such member provided for a type.
2. Returns by `ref` or `ref readonly`. (`readonly` is permitted so that authors of immutable/readonly types could implement the pattern without adding writeable API that could be used in safe code)
3. T is an unmanaged type. (since `T*` becomes the pointer type. The restriction will naturally expand if/when the notion of "unmanaged" is expanded)

4. Returns managed `nullptr` when there is no data to pin – probably the cheapest way to convey emptiness. (note that "" string returns a ref to '\0' since strings are null-terminated)

Alternatively for the `#3` we can allow the result in empty cases be undefined or implementation-specific. That, however, may make the API more dangerous and prone to abuse and unintended compatibility burdens.

## *Translation*

```
fixed(byte* ptr = thing)
{
    // <BODY>
}
```

becomes the following pseudocode (not all expressible in C#)

```
byte* ptr;
// specially decorated "pinned" IL local slot, not visible to user code.
pinned ref byte _pinned;

try
{
    // NOTE: null check is omitted for value types
    // NOTE: `thing` is evaluated only once (temporary is introduced if necessary)
    if (thing != null)
    {
        // obtain and "pin" the reference
        _pinned = ref thing.GetPinnableReference();

        // unsafe cast in IL
        ptr = (byte*)_pinned;
    }
    else
    {
        ptr = default(byte*);
    }

    // <BODY>
}
finally   // finally can be omitted when not observable
{
    // "unpin" the object
    _pinned = nullptr;
}
```

# Drawbacks

- GetPinnableReference is intended to be used only in `fixed`, but nothing prevents its use in safe code, so implementor must keep that in mind.

# Alternatives

Users can introduce GetPinnableReference or similar member and use it as

```
fixed(byte* ptr = thing.GetPinnableReference())
{
    // <BODY>
}
```

There is no solution for `System.String` if alternative solution is desired.

## Unresolved questions

- [ ] Behavior in "empty" state. - `nullptr` or `undefined` ?
- [ ] Should the extension methods be considered ?
- [ ] If a pattern is detected on `System.String`, should it win over ?

## Design meetings

None yet.

# Ref Local Reassignment

11/2/2020 • 2 minutes to read • Edit Online

In C# 7.3, we add support for rebinding the referent of a ref local variable or a ref parameter.

We add the following to the set of `assignment_operator`s.

```
assignment_operator
    : '=' 'ref'
    ;
```

The `=ref` operator is called the ***ref assignment operator***. It is not a *compound assignment operator*. The left operand must be an expression that binds to a ref local variable, a ref parameter (other than `this`), or an out parameter. The right operand must be an expression that yields an lvalue designating a value of the same type as the left operand.

The right operand must be definitely assigned at the point of the ref assignment.

When the left operand binds to an `out` parameter, it is an error if that `out` parameter has not been definitely assigned at the beginning of the ref assignment operator.

If the left operand is a writeable ref (i.e. it designates anything other than a `ref readonly` local or `in` parameter), then the right operand must be a writeable lvalue.

The ref assignment operator yields an lvalue of the assigned type. It is writeable if the left operand is writeable (i.e. not `ref readonly` or `in`).

The safety rules for this operator are:

- For a ref reassignment `e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.

Where *ref-safe-to-escape* is defined in Safety for ref-like types

# Stackalloc array initializers

11/2/2020 • 2 minutes to read • Edit Online

## Summary

Allow array initializer syntax to be used with `stackalloc`.

## Motivation

Ordinary arrays can have their elements initialized at creation time. It seems reasonable to allow that in `stackalloc` case.

The question of why such syntax is not allowed with `stackalloc` arises fairly frequently.
See, for example, #1112

## Detailed design

Ordinary arrays can be created through the following syntax:

```
new int[3]
new int[3] { 1, 2, 3 }
new int[] { 1, 2, 3 }
new[] { 1, 2, 3 }
```

We should allow stack allocated arrays be created through:

```
stackalloc int[3]    // currently allowed
stackalloc int[3] { 1, 2, 3 }
stackalloc int[] { 1, 2, 3 }
stackalloc[] { 1, 2, 3 }
```

The semantics of all cases is roughly the same as with arrays.
For example: in the last case the element type is inferred from the initializer and must be an "unmanaged" type.

NOTE: the feature is not dependent on the target being a `Span<T>`. It is just as applicable in `T*` case, so it does not seem reasonable to predicate it on `Span<T>` case.

## Translation

The naive implementation could just initialize the array right after creation through a series of element-wise assignments.

Similarly to the case with arrays, it might be possible and desirable to detect cases where all or most of the elements are blittable types and use more efficient techniques by copying over the pre-created state of all the constant elements.

## Drawbacks

## Alternatives

This is a convenience feature. It is possible to just do nothing.

## Unresolved questions

## Design meetings

None yet.

# Auto-Implemented Property Field-Targeted Attributes

11/2/2020 • 2 minutes to read • Edit Online

## Summary

This feature intends to allow developers to apply attributes directly to the backing fields of auto-implemented properties.

## Motivation

Currently it is not possible to apply attributes to the backing fields of auto-implemented properties. In those cases where the developer must use a field-targeting attribute they are forced to declare the field manually and use the more verbose property syntax. Given that C# has always supported field-targeted attributes on the generated backing field for events it makes sense to extend the same functionality to their property kin.

## Detailed design

In short, the following would be legal C# and not produce a warning:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public string MySecret { get; set; }
}
```

This would result in the field-targeted attributes being applied to the compiler-generated backing field:

```
[Serializable]
public class Foo
{
    [NonSerialized]
    private string _mySecretBackingField;

    public string MySecret
    {
        get { return _mySecretBackingField; }
        set { _mySecretBackingField = value; }
    }
}
```

As mentioned, this brings parity with event syntax from C# 1.0 as the following is already legal and behaves as expected:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public event EventHandler MyEvent;
}
```

## Drawbacks

There are two potential drawbacks to implementing this change:

1. Attempting to apply an attribute to the field of an auto-implemented property produces a compiler warning that the attributes in that block will be ignored. If the compiler were changed to support those attributes they would be applied to the backing field on a subsequent recompilation which could alter the behavior of the program at runtime.
2. The compiler does not currently validate the AttributeUsage targets of the attributes when attempting to apply them to the field of the auto-implemented property. If the compiler were changed to support field-targeted attributes and the attribute in question cannot be applied to a field the compiler would emit an error instead of a warning, breaking the build.

## Alternatives

## Unresolved questions

## Design meetings

# Expression variables in initializers

11/2/2020 • 2 minutes to read • Edit Online

## Summary

We extend the features introduced in C# 7 to permit expressions containing expression variables (out variable declarations and declaration patterns) in field initializers, property initializers, ctor-initializers, and query clauses.

## Motivation

This completes a couple of the rough edges left in the C# language due to lack of time.

## Detailed design

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a ctor-initializer. Such a declared variable is in scope throughout the body of the constructor.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a field or property initializer. Such a declared variable is in scope throughout the initializing expression.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a query expression clause that is translated into the body of a lambda. Such a declared variable is in scope throughout that expression of the query clause.

## Drawbacks

None.

## Alternatives

The appropriate scope for expression variables declared in these contexts is not obvious, and deserves further LDM discussion.

## Unresolved questions

- [ ] What is the appropriate scope for these variables?

## Design meetings

None.

# Support for == and != on tuple types

11/2/2020 • 4 minutes to read • Edit Online

Allow expressions `t1 == t2` where `t1` and `t2` are tuple or nullable tuple types of same cardinality, and evaluate them roughly as `temp1.Item1 == temp2.Item1 && temp1.Item2 == temp2.Item2` (assuming `var temp1 = t1; var temp2 = t2;`).

Conversely it would allow `t1 != t2` and evaluate it as `temp1.Item1 != temp2.Item1 || temp1.Item2 != temp2.Item2`.

In the nullable case, additional checks for `temp1.HasValue` and `temp2.HasValue` are used. For instance, `nullableT1 == nullableT2` evaluates as `temp1.HasValue == temp2.HasValue ? (temp1.HasValue ? ... : true) : false`.

When an element-wise comparison returns a non-bool result (for instance, when a non-bool user-defined `operator ==` or `operator !=` is used, or in a dynamic comparison), then that result will be either converted to `bool` or run through `operator true` or `operator false` to get a `bool`. The tuple comparison always ends up returning a `bool`.

As of C# 7.2, such code produces an error (`error CS0019: Operator '==' cannot be applied to operands of type '(...)' and '(...)'`), unless there is a user-defined `operator==`.

## Details

When binding the `==` (or `!=`) operator, the existing rules are: (1) dynamic case, (2) overload resolution, and (3) fail. This proposal adds a tuple case between (1) and (2): if both operands of a comparison operator are tuples (have tuple types or are tuple literals) and have matching cardinality, then the comparison is performed element-wise. This tuple equality is also lifted onto nullable tuples.

Both operands (and, in the case of tuple literals, their elements) are evaluated in order from left to right. Each pair of elements is then used as operands to bind the operator `==` (or `!=`), recursively. Any elements with compile-time type `dynamic` cause an error. The results of those element-wise comparisons are used as operands in a chain of conditional AND (or OR) operators.

For instance, in the context of `(int, (int, int)) t1, t2;`, `t1 == (1, (2, 3))` would evaluate as `temp1.Item1 == temp2.Item1 && temp1.Item2.Item1 == temp2.Item2.Item1 && temp1.Item2.Item2 == temp2.Item2.Item2`.

When a tuple literal is used as operand (on either side), it receives a converted tuple type formed by the element-wise conversions which are introduced when binding the operator `==` (or `!=`) element-wise.

For instance, in `(1L, 2, "hello") == (1, 2L, null)`, the converted type for both tuple literals is `(long, long, string)` and the second literal has no natural type.

### Deconstruction and conversions to tuple

In `(a, b) == x`, the fact that `x` can deconstruct into two elements does not play a role. That could conceivably be in a future proposal, although it would raise questions about `x == y` (is this a simple comparison or an element-wise comparison, and if so using what cardinality?). Similarly, conversions to tuple play no role.

### Tuple element names

When converting a tuple literal, we warn when an explicit tuple element name was provided in the literal, but it doesn't match the target tuple element name. We use the same rule in tuple comparison, so that assuming `(int a, int b) t` we warn on `d` in `t == (c, d: 0)`.

**Non-bool element-wise comparison results**

If an element-wise comparison is dynamic in a tuple equality, we use a dynamic invocation of the operator `false` and negate that to get a `bool` and continue with further element-wise comparisons.

If an element-wise comparison returns some other non-bool type in a tuple equality, there are two cases:

- if the non-bool type converts to `bool`, we apply that conversion,
- if there is no such conversion, but the type has an operator `false`, we'll use that and negate the result.

In a tuple inequality, the same rules apply except that we'll use the operator `true` (without negation) instead of the operator `false`.

Those rules are similar to the rules involved for using a non-bool type in an `if` statement and some other existing contexts.

## Evaluation order and special cases

The left-hand-side value is evaluated first, then the right-hand-side value, then the element-wise comparisons from left to right (including conversions, and with early exit based on existing rules for conditional AND/OR operators).

For instance, if there is a conversion from type `A` to type `B` and a method `(A, A) GetTuple()`, evaluating `(new A(1), (new B(2), new B(3))) == (new B(4), GetTuple())` means:

- `new A(1)`
- `new B(2)`
- `new B(3)`
- `new B(4)`
- `GetTuple()`
- then the element-wise conversions and comparisons and conditional logic is evaluated (convert `new A(1)` to type `B`, then compare it with `new B(4)`, and so on).

**Comparing `null` to `null`**

This is a special case from regular comparisons, that carries over to tuple comparisons. The `null == null` comparison is allowed, and the `null` literals do not get any type. In tuple equality, this means, `(0, null) == (0, null)` is also allowed and the `null` and tuple literals don't get a type either.

**Comparing a nullable struct to `null` without `operator==`**

This is another special case from regular comparisons, that carries over to tuple comparisons. If you have a `struct S` without `operator==`, the `(S?)x == null` comparison is allowed, and it is interpreted as `((S?).x).HasValue`. In tuple equality, the same rule is applied, so `(0, (S?)x) == (0, null)` is allowed.

## Compatibility

If someone wrote their own `ValueTuple` types with an implementation of the comparison operator, it would have previously been picked up by overload resolution. But since the new tuple case comes before overload resolution, we would handle this case with tuple comparison instead of relying on the user-defined comparison.

---

Relates to relational and type testing operators Relates to #190

# Improved overload candidates

11/2/2020 • 2 minutes to read • Edit Online

## Summary

The overload resolution rules have been updated in nearly every C# language update to improve the experience for programmers, making ambiguous invocations select the "obvious" choice. This has to be done carefully to preserve backward compatibility, but since we are usually resolving what would otherwise be error cases, these enhancements usually work out nicely.

1. When a method group contains both instance and static members, we discard the instance members if invoked without an instance receiver or context, and discard the static members if invoked with an instance receiver. When there is no receiver, we include only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type due to a color-color situation, we include 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.

# Nullable reference types in C#

11/2/2020 • 7 minutes to read • Edit Online

The goal of this feature is to:

- Allow developers to express whether a variable, parameter or result of a reference type is intended to be null or not.
- Provide warnings when such variables, parameters and results are not used according to that intent.

## Expression of intent

The language already contains the `T?` syntax for value types. It is straightforward to extend this syntax to reference types.

It is assumed that the intent of an unadorned reference type `T` is for it to be non-null.

## Checking of nullable references

A flow analysis tracks nullable reference variables. Where the analysis deems that they would not be null (e.g. after a check or an assignment), their value will be considered a non-null reference.

A nullable reference can also explicitly be treated as non-null with the postfix `x!` operator (the "damnit" operator), for when flow analysis cannot establish a non-null situation that the developer knows is there.

Otherwise, a warning is given if a nullable reference is dereferenced, or is converted to a non-null type.

A warning is given when converting from `S[]` to `T?[]` and from `S?[]` to `T[]`.

A warning is given when converting from `C<S>` to `C<T?>` except when the type parameter is covariant ( `out` ), and when converting from `C<S?>` to `C<T>` except when the type parameter is contravariant ( `in` ).

A warning is given on `C<T?>` if the type parameter has non-null constraints.

## Checking of non-null references

A warning is given if a null literal is assigned to a non-null variable or passed as a non-null parameter.

A warning is also given if a constructor does not explicitly initialize non-null reference fields.

We cannot adequately track that all elements of an array of non-null references are initialized. However, we could issue a warning if no element of a newly created array is assigned to before the array is read from or passed on. That might handle the common case without being too noisy.

We need to decide whether `default(T)` generates a warning, or is simply treated as being of the type `T?` .

## Metadata representation

Nullability adornments should be represented in metadata as attributes. This means that downlevel compilers will ignore them.

We need to decide if only nullable annotations are included, or there's also some indication of whether non-null was "on" in the assembly.

# Generics

If a type parameter `T` has non-nullable constraints, it is treated as non-nullable within its scope.

If a type parameter is unconstrained or has only nullable constraints, the situation is a little more complex: this means that the corresponding type argument could be *either* nullable or non-nullable. The safe thing to do in that situation is to treat the type parameter as *both* nullable and non-nullable, giving warnings when either is violated.

It is worth considering whether explicit nullable reference constraints should be allowed. Note, however, that we cannot avoid having nullable reference types *implicitly* be constraints in certain cases (inherited constraints).

The `class` constraint is non-null. We can consider whether `class?` should be a valid nullable constraint denoting "nullable reference type".

# Type inference

In type inference, if a contributing type is a nullable reference type, the resulting type should be nullable. In other words, nullness is propagated.

We should consider whether the `null` literal as a participating expression should contribute nullness. It doesn't today: for value types it leads to an error, whereas for reference types the null successfully converts to the plain type.

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

# Null guard guidance

As a feature, nullable reference types allow developers to express their intent, and provide warnings through flow analysis if that intent is contradicted. There is a common question as to whether or not null guards are necessary.

**Example of null guard**

```
public void DoWork(Worker worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

In the previous example, the `DoWork` function accepts a `Worker` and guards against it potentially being `null`. If the `worker` argument is `null`, the `DoWork` function will `throw`. With nullable reference types, the code in the previous example makes the intent that the `Worker` parameter would *not* be `null`. If the `DoWork` function was a public API, such as a NuGet package or a shared library - as guidance you should leave null guards in place. As a public API, the only guarantee that a caller isn't passing `null` is to guard against it.

**Express intent**

A more compelling use of the previous example is to express that the `Worker` parameter could be `null`, thus making the null guard more appropriate. If you remove the null guard in the following example, the compiler warns that you may be dereferencing null. Regardless, both null guards are still valid.

```
public void DoWork(Worker? worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

For non-public APIs, such as source code entirely in control by a developer or dev team - the nullable reference types could allow for the safe removal of null guards where the developers can guarantee it is not necessary. The feature can help with warnings, but it cannot guarantee that at runtime code execution could result in a `NullReferenceException`.

## Breaking changes

Non-null warnings are an obvious breaking change on existing code, and should be accompanied with an opt-in mechanism.

Less obviously, warnings from nullable types (as described above) are a breaking change on existing code in certain scenarios where the nullability is implicit:

- Unconstrained type parameters will be treated as implicitly nullable, so assigning them to `object` or accessing e.g. `ToString` will yield warnings.
- if type inference infers nullness from `null` expressions, then existing code will sometimes yield nullable rather than non-nullable types, which can lead to new warnings.

So nullable warnings also need to be optional

Finally, adding annotations to an existing API will be a breaking change to users who have opted in to warnings, when they upgrade the library. This, too, merits the ability to opt in or out. "I want the bug fixes, but I am not ready to deal with their new annotations"

In summary, you need to be able to opt in/out of:

- Nullable warnings
- Non-null warnings
- Warnings from annotations in other files

The granularity of the opt-in suggests an analyzer-like model, where swaths of code can opt in and out with pragmas and severity levels can be chosen by the user. Additionally, per-library options ("ignore the annotations from JSON.NET until I'm ready to deal with the fall out") may be expressible in code as attributes.

The design of the opt-in/transition experience is crucial to the success and usefulness of this feature. We need to make sure that:

- Users can adopt nullability checking gradually as they want to
- Library authors can add nullability annotations without fear of breaking customers
- Despite these, there is not a sense of "configuration nightmare"

## Tweaks

We could consider not using the `?` annotations on locals, but just observing whether they are used in accordance with what gets assigned to them. I don't favor this; I think we should uniformly let people express their intent.

We could consider a shorthand `T! x` on parameters, that auto-generates a runtime null check.

Certain patterns on generic types, such as `FirstOrDefault` or `TryGet`, have slightly weird behavior with non-nullable type arguments, because they explicitly yield default values in certain situations. We could try to nuance the type system to accommodate these better. For instance, we could allow `?` on unconstrained type parameters, even though the type argument could already be nullable. I doubt that it is worth it, and it leads to weirdness related to interaction with nullable *value* types.

## Nullable value types

We could consider adopting some of the above semantics for nullable value types as well.

We already mentioned type inference, where we could infer `int?` from `(7, null)`, instead of just giving an error.

Another opportunity is to apply the flow analysis to nullable value types. When they are deemed non-null, we could actually allow using as the non-nullable type in certain ways (e.g. member access). We just have to be careful that the things that you can *already* do on a nullable value type will be preferred, for back compat reasons.

# Nullable Reference Types Specification

11/24/2020 • 18 minutes to read • Edit Online

*This is a work in progress - several parts are missing or incomplete.*

This feature adds two new kinds of nullable types (nullable reference types and nullable generic types) to the existing nullable value types, and introduces a static flow analysis for purpose of null-safety.

## Syntax

**Nullable reference types and nullable type parameters**

Nullable reference types and nullable type parameters have the same syntax `T?` as the short form of nullable value types, but do not have a corresponding long form.

For the purposes of the specification, the current `nullable_type` production is renamed to `nullable_value_type`, and `nullable_reference_type` and `nullable_type_parameter` productions are added:

```
type
    : value_type
    | reference_type
    | nullable_type_parameter
    | type_parameter
    | type_unsafe
    ;

reference_type
    : ...
    | nullable_reference_type
    ;

nullable_reference_type
    : non_nullable_reference_type '?'
    ;

non_nullable_reference_type
    : reference_type
    ;

nullable_type_parameter
    : non_nullable_non_value_type_parameter '?'
    ;

non_nullable_non_value_type_parameter
    : type_parameter
    ;
```

The `non_nullable_reference_type` in a `nullable_reference_type` must be a nonnullable reference type (class, interface, delegate or array).

The `non_nullable_non_value_type_parameter` in `nullable_type_parameter` must be a type parameter that isn't constrained to be a value type.

Nullable reference types and nullable type parameters cannot occur in the following positions:

- as a base class or interface
- as the receiver of a `member_access`

- as the `type` in an `object_creation_expression`
- as the `delegate_type` in a `delegate_creation_expression`
- as the `type` in an `is_expression`, a `catch_clause` or a `type_pattern`
- as the `interface` in a fully qualified interface member name

A warning is given on a `nullable_reference_type` and `nullable_type_parameter` in a *disabled* nullable annotation context.

#### `class` and `class?` constraint

The `class` constraint has a nullable counterpart `class?`:

```
primary_constraint
    : ...
    | 'class' '?'
    ;
```

A type parameter constrained with `class` (in an *enabled* annotation context) must be instantiated with a nonnullable reference type.

A type parameter constrained with `class?` (or `class` in a *disabled* annotation context) may either be instantiated with a nullable or nonnullable reference type.

A warning is given on a `class?` constraint in a *disabled* annotation context.

#### `notnull` constraint

A type parameter constrained with `notnull` may not be a nullable type (nullable value type, nullable reference type or nullable type parameter).

```
primary_constraint
    : ...
    | 'notnull'
    ;
```

#### `default` constraint

The `default` constraint can be used on a method override or explicit implementation to disambiguate `T?` meaning "nullable type parameter" from "nullable value type" (`Nullable<T>`). Lacking the `default` constraint a `T?` syntax in an override or explicit implementation will be interpreted as `Nullable<T>`

See https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint

### The null-forgiving operator

The post-fix `!` operator is called the null-forgiving operator. It can be applied on a *primary_expression* or within a *null_conditional_expression*.

```
primary_expression
    : ...
    | null_forgiving_expression
    ;

null_forgiving_expression
    : primary_expression '!'
    ;

null_conditional_expression
    : primary_expression null_conditional_operations_no_suppression suppression?
    ;

null_conditional_operations_no_suppression
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;

null_conditional_operations
    : null_conditional_operations_no_suppression suppression?
    ;

suppression
    : '!'
    ;
```

For example:

```
var v = expr!;
expr!.M();
_ = a?.b!.c;
```

The `primary_expression` and `null_conditional_operations_no_suppression` must be of a nullable type.

The postfix `!` operator has no runtime effect - it evaluates to the result of the underlying expression. Its only role is to change the null state of the expression to "not null", and to limit warnings given on its use.

**Nullable compiler directives**

`#nullable` directives control the nullable annotation and warning contexts.

```
pp_directive
    : ...
    | pp_nullable
    ;

pp_nullable
    : whitespace? '#' whitespace? 'nullable' whitespace nullable_action (whitespace nullable_target)?
pp_new_line
    ;

nullable_action
    : 'disable'
    | 'enable'
    | 'restore'
    ;

nullable_target
    : 'warnings'
    | 'annotations'
    ;
```

`#pragma warning` directives are expanded to allow changing the nullable warning context:

```
pragma_warning_body
    : ...
    | 'warning' whitespace warning_action whitespace 'nullable'
    ;
```

For example:

```
#pragma warning disable nullable
```

## Nullable contexts

Every line of source code has a *nullable annotation context* and a *nullable warning context*. These control whether nullable annotations have effect, and whether nullability warnings are given. The annotation context of a given line is either *disabled* or *enabled*. The warning context of a given line is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code), or anywhere within a source file via `#nullable` pre-processor directives. If no project level settings are provided the default is for both contexts to be *disabled*.

The `#nullable` directive controls the annotation and warning contexts within the source text, and take precedence over the project-level settings.

A directive sets the context(s) it controls for subsequent lines of code, until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable`: Sets the nullable annotation and warning contexts to *disabled*
- `#nullable enable`: Sets the nullable annotation and warning contexts to *enabled*
- `#nullable restore`: Restores the nullable annotation and warning contexts to project settings
- `#nullable disable annotations`: Sets the nullable annotation context to *disabled*
- `#nullable enable annotations`: Sets the nullable annotation context to *enabled*
- `#nullable restore annotations`: Restores the nullable annotation context to project settings

- `#nullable disable warnings`: Sets the nullable warning context to *disabled*
- `#nullable enable warnings`: Sets the nullable warning context to *enabled*
- `#nullable restore warnings`: Restores the nullable warning context to project settings

# Nullability of types

A given type can have one of three nullabilities: *oblivious*, *nonnullable*, and *nullable*.

*Nonnullable* types may cause warnings if a potential `null` value is assigned to them. *Oblivious* and *nullable* types, however, are "*null-assignable*" and can have `null` values assigned to them without warnings.

Values of *oblivious* and *nonnullable* types can be dereferenced or assigned without warnings. Values of *nullable* types, however, are "*null-yielding*" and may cause warnings when dereferenced or assigned without proper null checking.

The *default null state* of a null-yielding type is "maybe null" or "maybe default". The default null state of a non-null-yielding type is "not null".

The kind of type and the nullable annotation context it occurs in determine its nullability:

- A nonnullable value type `S` is always *nonnullable*
- A nullable value type `S?` is always *nullable*
- An unannotated reference type `C` in a *disabled* annotation context is *oblivious*
- An unannotated reference type `C` in an *enabled* annotation context is *nonnullable*
- A nullable reference type `C?` is *nullable* (but a warning may be yielded in a *disabled* annotation context)

Type parameters additionally take their constraints into account:

- A type parameter `T` where all constraints (if any) are either nullable types or the `class?` constraint is *nullable*
- A type parameter `T` where at least one constraint is either *oblivious* or *nonnullable* or one of the `struct` or `class` or `notnull` constraints is
  - *oblivious* in a *disabled* annotation context
  - *nonnullable* in an *enabled* annotation context
- A nullable type parameter `T?` is *nullable*, but a warning is yielded in a *disabled* annotation context if `T` isn't a value type

**Oblivious vs nonnullable**

A `type` is deemed to occur in a given annotation context when the last token of the type is within that context.

Whether a given reference type `C` in source code is interpreted as oblivious or nonnullable depends on the annotation context of that source code. But once established, it is considered part of that type, and "travels with it" e.g. during substitution of generic type arguments. It is as if there is an annotation like `?` on the type, but invisible.

# Constraints

Nullable reference types can be used as generic constraints.

`class?` is a new constraint denoting "possibly nullable reference type", whereas `class` in an *enabled* annotation context denotes "nonnullable reference type".

`default` is a new constraint denoting a type parameter that isn't known to be a reference or value type. It can only be used on overridden and explicitly implemented methods. With this constraint, `T?` means a nullable type parameter, as opposed to being a shorthand for `Nullable<T>`.

`notnull` is a new constraint denoting a type parameter that is nonnullable.

The nullability of a type argument or of a constraint does not impact whether the type satisfies the constraint, except where that is already the case today (nullable value types do not satisfy the `struct` constraint). However, if the type argument does not satisfy the nullability requirements of the constraint, a warning may be given.

# Null state and null tracking

Every expression in a given source location has a *null state*, which indicated whether it is believed to potentially evaluate to null. The null state is either "not null", "maybe null", or "maybe default". The null state is used to determine whether a warning should be given about null-unsafe conversions and dereferences.

The distinction between "maybe null" and "maybe default" is subtle and applies to type parameters. The distinction is that a type parameter `T` which has the state "maybe null" means the value is in the domain of legal values for `T` however that legal value may include `null`. Where as a "maybe default" means that the value may be outside the legal domain of values for `T`.

Example:

```
// The value `t` here has the state "maybe null". It's possible for `T` to be instantiated
// with `string?` in which case `null` would be within the domain of legal values here. The
// assumption though is the value provided here is within the legal values of `T`. Hence
// if `T` is `string` then `null` will not be a value, just as we assume that `null` is not
// provided for a normal `string` parameter
void M<T>(T t)
{
    // There is no guarantee that default(T) is within the legal values for T hence the
    // state *must* be "maybe-default" and hence `local` must be `T?`
    T? local = default(T);
}
```

### Null tracking for variables

For certain expressions denoting variables, fields or properties, the null state is tracked between occurrences, based on assignments to them, tests performed on them and the control flow between them. This is similar to how definite assignment is tracked for variables. The tracked expressions are the ones of the following form:

```
tracked_expression
    : simple_name
    | this
    | base
    | tracked_expression '.' identifier
    ;
```

Where the identifiers denote fields or properties.

The null state for tracked variables is "not null" in unreachable code. This follows other decisions around unreachable code like considering all locals to be definitely assigned.

*Describe null state transitions similar to definite assignment*

### Null state for expressions

The null state of an expression is derived from its form and type, and from the null state of variables involved in it.

### Literals

The null state of a `null` literal depends on the target type of the expression. If the target type is a type parameter constrained to a reference type then it's "maybe default". Otherwise it is "maybe null".

The null state of a `default` literal depends on the target type of the `default` literal. A `default` literal with target type `T` has the same null state as the `default(T)` expression.

The null state of any other literal is "not null".

### Simple names

If a `simple_name` is not classified as a value, its null state is "not null". Otherwise it is a tracked expression, and its null state is its tracked null state at this source location.

### Member access

If a `member_access` is not classified as a value, its null state is "not null". Otherwise, if it is a tracked expression, its null state is its tracked null state at this source location. Otherwise its null state is the default null state of its type.

```
var person = new Person();

// The receiver is a tracked expression hence the member_access of the property
// is tracked as well
if (person.FirstName is not null)
{
    Use(person.FirstName);
}

// The return of an invocation is not a tracked expression hence the member_access
// of the return is also not tracked
if (GetAnonymous().FirstName is not null)
{
    // Warning: Cannot convert null literal to non-nullable reference type.
    Use(GetAnonymous().FirstName);
}

void Use(string s)
{
    // ...
}

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    private static Person s_anonymous = new Person();
    public static Person GetAnonymous() => s_anonymous;
}
```

### Invocation expressions

If an `invocation_expression` invokes a member that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

The null state of an `invocation_expression` is not tracked by the compiler.

```
// The result of an invocation_expression is not tracked
if (GetText() is not null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    string s = GetText();
    // Warning: Dereference of a possibly null reference.
    Use(s);
}

// Nullable friendly pattern
if (GetText() is string s)
{
    Use(s);
}

string? GetText() => ...
Use(string s) {  }
```

#### Element access

If an `element_access` invokes an indexer that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

```
object?[] array = ...;
if (array[0] != null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    object o = array[0];
    // Warning: Dereference of a possibly null reference.
    Console.WriteLine(o.ToString());
}

// Nullable friendly pattern
if (array[0] is {} o)
{
    Console.WriteLine(o.ToString());
}
```

#### Base access

If `B` denotes the base type of the enclosing type, `base.I` has the same null state as `((B)this).I` and `base[E]` has the same null state as `((B)this)[E]`.

#### Default expressions

`default(T)` has the null state based on the properties of the type `T`:

- If the type is a *nonnullable* type then it has the null state "not null"
- Else if the type is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### Null-conditional expressions ?.

A `null_conditional_expression` has the null state based on the expression type. Note that this refers to the type of the `null_conditional_expression`, not the original type of the member being invoked:

- If the type is a *nullable* value type then it has the null state "maybe null"
- Else if the type is a *nullable* type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### Cast expressions

If a cast expression `(T)E` invokes a user-defined conversion, then the null state of the expression is the default null state for the type of the user-defined conversion. Otherwise:

- If `T` is a *nonnullable* value type then `T` has the null state "not null"
- Else if `T` is a *nullable* value type then `T` has the null state "maybe null"
- Else if `T` is a *nullable* type in the form `U?` where `U` is a type parameter then `T` has the null state "maybe default"
- Else if `T` is a *nullable* type, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe null"
- Else if `T` is a type parameter, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe default"
- Else `T` has the same null state as `E`

#### Unary and binary operators

If a unary or binary operator invokes an user-defined operator then the null state of the expression is the default null state for the type of the user-defined operator. Otherwise it is the null state of the expression.

*Something special to do for binary `+` over strings and delegates?*

#### Await expressions

The null state of `await E` is the default null state of its type.

#### The `as` operator

The null state of an `E as T` expression depends first on properties of the type `T`. If the type of `T` is *nonnullable* then the null state is "not null". Otherwise the null state depends on the conversion from the type of `E` to type `T`:

- If the conversion is an identity, boxing, implicit reference, or implicit nullable conversion, then the null state is the null state of `E`
- Else if `T` is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### The null-coalescing operator

The null state of `E1 ?? E2` is the null state of `E2`

#### The conditional operator

The null state of `E1 ? E2 : E3` is based on the null state of `E2` and `E3`:

- If both are "not null" then the null state is "not null"
- Else if either is "maybe default" then the null state is "maybe default"
- Else the null state is "not null"

#### Query expressions

The null state of a query expression is the default null state of its type.

*Additional work needed here*

#### Assignment operators

`E1 = E2` and `E1 op= E2` have the same null state as `E2` after any implicit conversions have been applied.

#### Expressions that propagate null state

`(E)`, `checked(E)` and `unchecked(E)` all have the same null state as `E`.

#### Expressions that are never null

The null state of the following expression forms is always "not null":

- `this` access
- interpolated strings
- `new` expressions (object, delegate, anonymous object and array creation expressions)
- `typeof` expressions
- `nameof` expressions
- anonymous functions (anonymous methods and lambda expressions)
- null-forgiving expressions
- `is` expressions

#### Nested functions

Nested functions (lambdas and local functions) are treated like methods, except in regards to their captured variables. The initial state of a captured variable inside a lambda or local function is the intersection of the nullable state of the variable at all the "uses" of that nested function or lambda. A use of a local function is either a call to that function, or where it is converted to a delegate. A use of a lambda is the point at which it is defined in source.

## Type inference

#### nullable implicitly typed local variables

`var` infers an annotated type for reference types, and type parameters that aren't constrained to be a value type. For instance:

- in `var s = "";` the `var` is inferred as `string?`.
- in `var t = new T();` with an unconstrained `T` the `var` is inferred as `T?`.

#### Generic type inference

Generic type inference is enhanced to help decide whether inferred reference types should be nullable or not. This is a best effort. It may yield warnings regarding nullability constraints, and may lead to nullable warnings when the inferred types of the selected overload are applied to the arguments.

#### The first phase

Nullable reference types flow into the bounds from the initial expressions, as described below. In addition, two new kinds of bounds, namely `null` and `default` are introduced. Their purpose is to carry through occurrences of `null` or `default` in the input expressions, which may cause an inferred type to be nullable, even when it otherwise wouldn't. This works even for nullable *value* types, which are enhanced to pick up "nullness" in the inference process.

The determination of what bounds to add in the first phase are enhanced as follows:

If an argument `Ei` has a reference type, the type `U` used for inference depends on the null state of `Ei` as well as its declared type:

- If the declared type is a nonnullable reference type `U0` or a nullable reference type `U0?` then
  - if the null state of `Ei` is "not null" then `U` is `U0`
  - if the null state of `Ei` is "maybe null" then `U` is `U0?`
- Otherwise if `Ei` has a declared type, `U` is that type
- Otherwise if `Ei` is `null` then `U` is the special bound `null`
- Otherwise if `Ei` is `default` then `U` is the special bound `default`
- Otherwise no inference is made.

#### Exact, upper-bound and lower-bound inferences

In inferences *from* the type `U` *to* the type `V`, if `V` is a nullable reference type `V0?`, then `V0` is used instead of `V`

in the following clauses.

- If `V` is one of the unfixed type variables, `U` is added as an exact, upper or lower bound as before
- Otherwise, if `U` is `null` or `default`, no inference is made
- Otherwise, if `U` is a nullable reference type `U0?`, then `U0` is used instead of `U` in the subsequent clauses.

The essence is that nullability that pertains directly to one of the unfixed type variables is preserved into its bounds. For the inferences that recurse further into the source and target types, on the other hand, nullability is ignored. It may or may not match, but if it doesn't, a warning will be issued later if the overload is chosen and applied.

**Fixing**

The spec currently does not do a good job of describing what happens when multiple bounds are identity convertible to each other, but are different. This may happen between `object` and `dynamic`, between tuple types that differ only in element names, between types constructed thereof and now also between `C` and `C?` for reference types.

In addition we need to propagate "nullness" from the input expressions to the result type.

To handle these we add more phases to fixing, which is now:

1. Gather all the types in all the bounds as candidates, removing `?` from all that are nullable reference types
2. Eliminate candidates based on requirements of exact, lower and upper bounds (keeping `null` and `default` bounds)
3. Eliminate candidates that do not have an implicit conversion to all the other candidates
4. If the remaining candidates do not all have identity conversions to one another, then type inference fails
5. *Merge* the remaining candidates as described below
6. If the resulting candidate is a reference type or a nonnullable value type and *all* of the exact bounds or *any* of the lower bounds are nullable value types, nullable reference types, `null` or `default`, then `?` is added to the resulting candidate, making it a nullable value type or reference type.

*Merging* is described between two candidate types. It is transitive and commutative, so the candidates can be merged in any order with the same ultimate result. It is undefined if the two candidate types are not identity convertible to each other.

The *Merge* function takes two candidate types and a direction (*+* or *-*):

- *Merge*(`T`, `T`, *d*) = T
- *Merge*(`S`, `T?`, *+*) = *Merge*(`S?`, `T`, *+*) = *Merge*(`S`, `T`, *+*)`?`
- *Merge*(`S`, `T?`, *-*) = *Merge*(`S?`, `T`, *-*) = *Merge*(`S`, `T`, *-*)
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *+*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *+* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *-* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *-*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *-* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *+* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`(S1 s1,..., Sn sn)`, `(T1 t1,..., Tn tn)`, *d*) = `(` *Merge*(`S1`, `T1`, *d*)`n1,...,` *Merge*(`Sn`, `Tn`, *d*)`nn)`, *where*
  - `ni` is absent if `si` and `ti` differ, or if both are absent
  - `ni` is `si` if `si` and `ti` are the same
- *Merge*(`object`, `dynamic`) = *Merge*(`dynamic`, `object`) = `dynamic`

# Warnings

**Potential null assignment**

**Potential null dereference**

**Constraint nullability mismatch**

**Nullable types in disabled annotation context**

**Override and implementation nullability mismatch**

## Attributes for special null behavior

# Recursive Pattern Matching

11/2/2020 • 12 minutes to read • Edit Online

## Summary

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. Elements of this approach are inspired by related features in the programming languages F# and Scala.

## Detailed design

### Is Expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : is_pattern_expression
    ;
is_pattern_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

> Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do an expression in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

### Patterns

Patterns are used in the *is_pattern* operator, in a *switch_statement*, and in a *switch_expression* to express the shape of data against which incoming data (which we call the input value) is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    | positional_pattern
    | property_pattern
    | discard_pattern
    ;
declaration_pattern
    : type simple_designation
    ;
constant_pattern
    : constant_expression
    ;
var_pattern
    : 'var' designation
    ;
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
property_pattern
    : type? property_subpattern simple_designation?
    ;
simple_designation
    : single_variable_designation
    | discard_designation
    ;
discard_pattern
    : '_'
    ;
```

**Declaration Pattern**

```
declaration_pattern
    : type simple_designation
    ;
```

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. This may introduce a local variable of the given type named by the given identifier, if the designation is a *single_variable_designation*. That local variable is *definitely assigned* when the result of the pattern-matching operation is `true`.

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype) and not `null`, the result of the `is operator` is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern-compatible* with a type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, or if one of those types is an open type. It is a compile-time error if an input of

type `E` is not *pattern-compatible* with the *type* in a type pattern that it is matched with.

The type pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v
```

With the slightly more concise

```
if (expr is Type v) { // code using v
```

It is an error if *type* is a nullable value type.

The type pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block. After the block the variable `v` is in scope but not definitely assigned.

#### Constant Pattern

```
constant_pattern
    : constant_expression
    ;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant. When the input value is not an open type, the constant expression is implicitly converted to the type of the matched expression; if the type of the input value is not *pattern-compatible* with the type of the constant expression, the pattern-matching operation is an error.

The pattern *c* is considered matching the converted input value *e* if `object.Equals(c, e)` would return `true`.

We expect to see `e is null` as the most common way to test for `null` in newly written code, as it cannot invoke a user-defined `operator==`.

#### Var Pattern

```
var_pattern
    : 'var' designation
    ;
designation
    : simple_designation
    | tuple_designation
    ;
simple_designation
    : single_variable_designation
    | discard_designation
    ;
single_variable_designation
    : identifier
    ;
discard_designation
    : _
    ;
tuple_designation
    : '(' designations? ')'
    ;
designations
    : designation
    | designations ',' designation
    ;
```

If the *designation* is a *simple_designation*, an expression *e* matches the pattern. In other words, a match to a *var pattern* always succeeds with a *simple_designation*. If the *simple_designation* is a *single_variable_designation*, the value of *e* is bounds to a newly introduced local variable. The type of the local variable is the static type of *e*.

If the *designation* is a *tuple_designation*, then the pattern is equivalent to a *positional_pattern* of the form `(var` *designation*, ... `)` where the *designation*s are those found within the *tuple_designation*. For example, the pattern `var (x, (y, z))` is equivalent to `(var x, (var y, var z))`.

It is an error if the name `var` binds to a type.

#### Discard Pattern

```
discard_pattern
    : '_'
    ;
```

An expression *e* matches the pattern `_` always. In other words, every expression matches the discard pattern.

A discard pattern may not be used as the pattern of an *is_pattern_expression*.

#### Positional Pattern

A positional pattern checks that the input value is not `null`, invokes an appropriate `Deconstruct` method, and performs further pattern matching on the resulting values. It also supports a tuple-like pattern syntax (without the type being provided) when the type of the input value is the same as the type containing `Deconstruct`, or if the type of the input value is a tuple type, or if the type of the input value is `object` or `ITuple` and the runtime type of the expression implements `ITuple`.

```
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
```

If the *type* is omitted, we take it to be the static type of the input value.

Given a match of an input value to the pattern *type* `(` *subpattern_list* `)`, a method is selected by searching in *type* for accessible declarations of `Deconstruct` and selecting one among them using the same rules as for the deconstruction declaration.

It is an error if a *positional_pattern* omits the type, has a single *subpattern* without an *identifier*, has no *property_subpattern* and has no *simple_designation*. This disambiguates between a *constant_pattern* that is parenthesized and a *positional_pattern*.

In order to extract the values to match against the patterns in the list,

- If *type* was omitted and the input value's type is a tuple type, then the number of subpatterns is required to be the same as the cardinality of the tuple. Each tuple element is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a tuple element at the corresponding position in the tuple type.
- Otherwise, if a suitable `Deconstruct` exists as a member of *type*, it is a compile-time error if the type of the input value is not *pattern-compatible* with *type*. At runtime the input value is tested against *type*. If this fails then the positional pattern match fails. If it succeeds, the input value is converted to this type and `Deconstruct` is invoked with fresh compiler-generated variables to receive the `out` parameters. Each value that was received is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a parameter at the corresponding position of `Deconstruct`.
- Otherwise if *type* was omitted, and the input value is of type `object` or `ITuple` or some type that can be converted to `ITuple` by an implicit reference conversion, and no *identifier* appears among the subpatterns, then we match using `ITuple`.
- Otherwise the pattern is a compile-time error.

The order in which subpatterns are matched at runtime is unspecified, and a failed match may not attempt to match all subpatterns.

###### Example

This example uses many of the features described in this specification

```
    var newState = (GetState(), action, hasKey) switch {
        (DoorState.Closed, Action.Open, _) => DoorState.Opened,
        (DoorState.Opened, Action.Close, _) => DoorState.Closed,
        (DoorState.Closed, Action.Lock, true) => DoorState.Locked,
        (DoorState.Locked, Action.Unlock, true) => DoorState.Closed,
        (var state, _, _) => state };
```

**Property Pattern**

A property pattern checks that the input value is not `null` and recursively matches values extracted by the use of accessible properties or fields.

```
property_pattern
    : type? property_subpattern simple_designation?
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
```

It is an error if any *subpattern* of a *property_pattern* does not contain an *identifier* (it must be of the second form, which has an *identifier*). A trailing comma after the last subpattern is optional.

Note that a null-checking pattern falls out of a trivial property pattern. To check if the string `s` is non-null, you can write any of the following forms

```
if (s is object o) ... // o is of type object
if (s is string x) ... // x is of type string
if (s is {} x) ... // x is of type string
if (s is {}) ...
```

Given a match of an expression *e* to the pattern *type* `{` *property_pattern_list* `}`, it is a compile-time error if the expression *e* is not *pattern-compatible* with the type *T* designated by *type*. If the type is absent, we take it to be the static type of *e*. If the *identifier* is present, it declares a pattern variable of type *type*. Each of the identifiers appearing on the left-hand-side of its *property_pattern_list* must designate an accessible readable property or field of *T*. If the *simple_designation* of the *property_pattern* is present, it defines a pattern variable of type *T*.

At runtime, the expression is tested against *T*. If this fails then the property pattern match fails and the result is `false`. If it succeeds, then each *property_subpattern* field or property is read and its value matched against its corresponding pattern. The result of the whole match is `false` only if the result of any of these is `false`. The order in which subpatterns are matched is not specified, and a failed match may not match all subpatterns at runtime. If the match succeeds and the *simple_designation* of the *property_pattern* is a *single_variable_designation*, it defines a variable of type *T* that is assigned the matched value.

> Note: The property pattern can be used to pattern-match with anonymous types.

###### Example

```
if (o is string { Length: 5 } s)
```

**Switch Expression**

A *switch_expression* is added to support `switch`-like semantics for an expression context.

The C# language syntax is augmented with the following syntactic productions:

```
multiplicative_expression
    : switch_expression
    | multiplicative_expression '*' switch_expression
    | multiplicative_expression '/' switch_expression
    | multiplicative_expression '%' switch_expression
    ;
switch_expression
    : range_expression 'switch' '{' '}'
    | range_expression 'switch' '{' switch_expression_arms ','? '}'
    ;
switch_expression_arms
    : switch_expression_arm
    | switch_expression_arms ',' switch_expression_arm
    ;
switch_expression_arm
    : pattern case_guard? '=>' expression
    ;
case_guard
    : 'when' null_coalescing_expression
    ;
```

The *switch_expression* is not permitted as an *expression_statement*.

> We are looking at relaxing this in a future revision.

The type of the *switch_expression* is the *best common type* of the expressions appearing to the right of the `=>` tokens of the *switch_expression_arm*s if such a type exists and the expression in every arm of the switch expression can be implicitly converted to that type. In addition, we add a new *switch expression conversion*, which is a predefined implicit conversion from a switch expression to every type `T` for which there exists an implicit conversion from each arm's expression to `T`.

It is an error if some *switch_expression_arm*'s pattern cannot affect the result because some previous pattern and guard will always match.

A switch expression is said to be *exhaustive* if some arm of the switch expression handles every value of its input. The compiler shall produce a warning if a switch expression is not *exhaustive*.

At runtime, the result of the *switch_expression* is the value of the *expression* of the first *switch_expression_arm* for which the expression on the left-hand-side of the *switch_expression* matches the *switch_expression_arm*'s pattern, and for which the *case_guard* of the *switch_expression_arm*, if present, evaluates to `true`. If there is no such *switch_expression_arm*, the *switch_expression* throws an instance of the exception `System.Runtime.CompilerServices.SwitchExpressionException`.

**Optional parens when switching on a tuple literal**

In order to switch on a tuple literal using the *switch_statement*, you have to write what appear to be redundant parens

```
switch ((a, b))
{
```

To permit

```
switch (a, b)
{
```

the parentheses of the switch statement are optional when the expression being switched on is a tuple literal.

### Order of evaluation in pattern-matching

Giving the compiler flexibility in reordering the operations executed during pattern-matching can permit flexibility that can be used to improve the efficiency of pattern-matching. The (unenforced) requirement would be that properties accessed in a pattern, and the Deconstruct methods, are required to be "pure" (side-effect free, idempotent, etc). That doesn't mean that we would add purity as a language concept, only that we would allow the compiler flexibility in reordering operations.

**Resolution 2018-04-04 LDM**: confirmed: the compiler is permitted to reorder calls to `Deconstruct`, property accesses, and invocations of methods in `ITuple`, and may assume that returned values are the same from multiple calls. The compiler should not invoke functions that cannot affect the result, and we will be very careful before making any changes to the compiler-generated order of evaluation in the future.

### Some Possible Optimizations

The compilation of pattern matching can take advantage of common parts of patterns. For example, if the top-level type test of two successive patterns in a *switch_statement* is the same type, the generated code can skip the type test for the second pattern.

When some of the patterns are integers or strings, the compiler can generate the same kind of code it generates for a switch-statement in earlier versions of the language.

For more on these kinds of optimizations, see [Scott and Ramsey (2000)].

# default interface methods

11/2/2020 • 27 minutes to read • Edit Online

- [x] Proposed
- [ ] Prototype: In progress
- [ ] Implementation: None
- [ ] Specification: In progress, below

## Summary

Add support for *virtual extension methods* - methods in interfaces with concrete implementations. A class or struct that implements such an interface is required to have a single *most specific* implementation for the interface method, either implemented by the class or struct, or inherited from its base classes or interfaces. Virtual extension methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.

These are similar to Java's "Default Methods".

(Based on the likely implementation technique) this feature requires corresponding support in the CLI/CLR. Programs that take advantage of this feature cannot run on earlier versions of the platform.

## Motivation

The principal motivations for this feature are

- Default interface methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.
- The feature enables C# to interoperate with APIs targeting Android (Java) and iOS (Swift), which support similar features.
- As it turns out, adding default interface implementations provides the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming)). Traits have proven to be a powerful programming technique (http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf).

## Detailed design

The syntax for an interface is extended to permit

- member declarations that declare constants, operators, static constructors, and nested types;
- a *body* for a method or indexer, property, or event accessor (that is, a "default" implementation);
- member declarations that declare static fields, methods, properties, indexers, and events;
- member declarations using the explicit interface implementation syntax; and
- Explicit access modifiers (the default access is `public`).

Members with bodies permit the interface to provide a "default" implementation for the method in classes and structs that do not provide an overriding implementation.

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. Instance auto-properties are not supported in interfaces, as they would implicitly declare a hidden field.

Static and private methods permit useful refactoring and organization of code used to implement the interface's public API.

A method override in an interface must use the explicit interface implementation syntax.

It is an error to declare a class type, struct type, or enum type within the scope of a type parameter that was declared with a *variance_annotation*. For example, the declaration of `C` below is an error.

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type parameter 'T'
}
```

**Concrete methods in interfaces**

The simplest form of this feature is the ability to declare a *concrete method* in an interface, which is a method with a body.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

A class that implements this interface need not implement its concrete method.

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

The final override for `IA.M` in class `C` is the concrete method `M` declared in `IA`. Note that a class does not inherit members from its interfaces; that is not changed by this feature:

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Within an instance member of an interface, `this` has the type of the enclosing interface.

**Modifiers in interfaces**

The syntax for an interface is relaxed to permit modifiers on its members. The following are permitted: `private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `sealed`, `static`, `extern`, and `partial`.

> *TODO*: check what other modifiers exist.

An interface member whose declaration includes a body is a `virtual` member unless the `sealed` or `private` modifier is used. The `virtual` modifier may be used on a function member that would otherwise be implicitly `virtual`. Similarly, although `abstract` is the default on interface members without bodies, that modifier may be given explicitly. A non-virtual member may be declared using the `sealed` keyword.

It is an error for a `private` or `sealed` function member of an interface to have no body. A `private` function member may not have the modifier `sealed`.

Access modifiers may be used on interface members of all kinds of members that are permitted. The access level `public` is the default but it may be given explicitly.

> *Open Issue:* We need to specify the precise meaning of the access modifiers such as `protected` and `internal`, and which declarations do and do not override them (in a derived interface) or implement them (in a class that implements the interface).

Interfaces may declare `static` members, including nested types, methods, indexers, properties, events, and static constructors. The default access level for all interface members is `public`.

Interfaces may not declare instance constructors, destructors, or fields.

> ***Closed Issue:*** Should operator declarations be permitted in an interface? Probably not conversion operators, but what about others? ***Decision***: Operators are permitted *except* for conversion, equality, and inequality operators.

> ***Closed Issue:*** Should `new` be permitted on interface member declarations that hide members from base interfaces? ***Decision***: Yes.

> ***Closed Issue:*** We do not currently permit `partial` on an interface or its members. That would require a separate proposal. ***Decision***: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface

#### Overrides in interfaces

Override declarations (i.e. those containing the `override` modifier) allow the programmer to provide a most specific implementation of a virtual member in an interface where the compiler or runtime would not otherwise find one. It also allows turning an abstract member from a super-interface into a default member in a derived interface. An override declaration is permitted to *explicitly* override a particular base interface method by qualifying the declaration with the interface name (no access modifier is permitted in this case). Implicit overrides are not permitted.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}
interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

Override declarations in interfaces may not be declared `sealed`.

Public `virtual` function members in an interface may be overridden in a derived interface explicitly (by qualifying the name in the override declaration with the interface type that originally declared the method, and omitting an access modifier).

`virtual` function members in an interface may only be overridden explicitly (not implicitly) in derived interfaces, and members that are not `public` may only be implemented in a class or struct explicitly (not implicitly). In either case, the overridden or implemented member must be *accessible* where it is overridden.

#### Reabstraction

A virtual (concrete) method declared in an interface may be overridden to be abstract in a derived interface

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

The `abstract` modifier is not required in the declaration of `IB.M` (that is the default in interfaces), but it is probably good practice to be explicit in an override declaration.

This is useful in derived interfaces where the default implementation of a method is inappropriate and a more appropriate implementation should be provided by implementing classes.

> *Open Issue:* Should reabstraction be permitted?

#### The most specific override rule

We require that every interface and class have a *most specific override* for every virtual member among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the member itself is considered the most specific override.

One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

For example:

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
}
interface ID : IB, IC { } // error: no most specific override for 'IA.M'
abstract class C : IB, IC { } // error: no most specific override for 'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

The most specific override rule ensures that a conflict (i.e. an ambiguity arising from diamond inheritance) is resolved explicitly by the programmer at the point where the conflict arises.

Because we support explicit abstract overrides in interfaces, we could do so in classes as well

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

> *Open issue*: should we support explicit interface abstract overrides in classes?

In addition, it is an error if in a class declaration the most specific override of some interface method is an abstract override that was declared in an interface. This is an existing rule restated using the new terminology.

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

It is possible for a virtual property declared in an interface to have a most specific override for its `get` accessor in one interface and a most specific override for its `set` accessor in a different interface. This is considered a violation of the *most specific override* rule.

### `static` and `private` methods

Because interfaces may now contain executable code, it is useful to abstract common code into private and static methods. We now permit these in interfaces.

> *Closed issue*: Should we support private methods? Should we support static methods? **Decision: YES**

> *Open issue*: should we permit interface methods to be `protected` or `internal` or other access? If so, what are the semantics? Are they `virtual` by default? If so, is there a way to make them non-virtual?

> *Open issue*: If we support static methods, should we support (static) operators?

#### Base interface invocations

Code in a type that derives from an interface with a default method can explicitly invoke that interface's "base" implementation.

```
interface I0
{
   void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
   override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
   override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
   // an explicit override that invoke's a base interface's default method
   void I0.M() { I2.base.M(); }
}
```

An instance (nonstatic) method is permitted to invoke the implementation of an accessible instance method in a

direct base interface nonvirtually by naming it using the syntax `base(Type).M`. This is useful when an override that is required to be provided due to diamond inheritance is resolved by delegating to one particular base implementation.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
{
    void IA.M() { base(IB).M(); }
}
```

When a `virtual` or `abstract` member is accessed using the syntax `base(Type).M`, it is required that `Type` contains a unique *most specific override* for `M`.

**Binding base clauses**

Interfaces now contain types. These types may be used in the base clause as base interfaces. When binding a base clause, we may need to know the set of base interfaces to bind those types (e.g. to lookup in them and to resolve protected access). The meaning of an interface's base clause is thus circularly defined. To break the cycle, we add a new language rules corresponding to a similar rule already in place for classes.

While determining the meaning of the *interface_base* of an interface, the base interfaces are temporarily assumed to be empty. Intuitively this ensures that the meaning of a base clause cannot recursively depend on itself.

**We used to have the following rules:**

"When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ~~**class**~~ within which it is immediately nested (if any). Given this definition, the complete set of ~~**classes**~~ upon which a class depends is the reflexive and transitive closure of the **directly depends on** relationship."

It is a compile-time error for an interface to directly or indirectly inherit from itself. The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.

**We are adjusting them as follows:**

When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ***type*** within which it is immediately nested (if any).

When an interface IB extends an interface IA, it is a compile-time error for IA to depend on IB. An interface **directly depends on** its direct base interfaces (if any) and **directly depends on** the type within which it is immediately nested (if any).

Given these definitions, the complete set of **types** upon which a type depends is the reflexive and transitive closure of the **directly depends on** relationship.

**Effect on existing programs**

The rules presented here are intended to have no effect on the meaning of existing programs.

Example 1:

```
interface IA
{
    void M();
}
class C: IA // Error: IA.M has no concrete most specific override in C
{
    public static void M() { } // method unrelated to 'IA.M' because static
}
```

Example 2:

```
interface IA
{
    void M();
}
class Base: IA
{
    void IA.M() { }
}
class Derived: Base, IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

The same rules give similar results to the analogous situation involving default interface methods:

```
interface IA
{
    void M() { }
}
class Derived: IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

> *Closed issue*: confirm that this is an intended consequence of the specification. **Decision: YES**

#### Runtime method resolution

> *Closed Issue:* The spec should describe the runtime method resolution algorithm in the face of interface default methods. We need to ensure that the semantics are consistent with the language semantics, e.g. which declared methods do and do not override or implement an `internal` method.

#### CLR support API

In order for compilers to detect when they are compiling for a runtime that supports this feature, libraries for such runtimes are modified to advertise that fact through the API discussed in https://github.com/dotnet/corefx/issues/17116. We add

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation = nameof(DefaultInterfaceImplementation);
    }
}
```

> *Open issue*: Is that the best name for the *CLR* feature? The CLR feature does much more than just that (e.g. relaxes protection constraints, supports overrides in interfaces, etc). Perhaps it should be called something like "concrete methods in interfaces", or "traits"?

**Further areas to be specified**

- [ ] It would be useful to catalog the kinds of source and binary compatibility effects caused by adding default interface methods and overrides to existing interfaces.

# Drawbacks

This proposal requires a coordinated update to the CLR specification (to support concrete methods in interfaces and method resolution). It is therefore fairly "expensive" and it may be worth doing in combination with other features that we also anticipate would require CLR changes.

# Alternatives

None.

# Unresolved questions

- Open questions are called out throughout the proposal, above.
- See also https://github.com/dotnet/csharplang/issues/406 for a list of open questions.
- The detailed specification must describe the resolution mechanism used at runtime to select the precise method to be invoked.
- The interaction of metadata produced by new compilers and consumed by older compilers needs to be worked out in detail. For example, we need to ensure that the metadata representation that we use does not cause the addition of a default implementation in an interface to break an existing class that implements that interface when compiled by an older compiler. This may affect the metadata representation that we can use.
- The design must consider interoperation with other languages and existing compilers for other languages.

# Resolved Questions

**Abstract Override**

The earlier draft spec contained the ability to "reabstract" an inherited method:

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { }
}
interface IC : IB
{
    override void M(); // make it abstract again
}
```

My notes for 2017-03-20 showed that we decided not to allow this. However, there are at least two use cases for it:

1. The Java APIs, with which some users of this feature hope to interoperate, depend on this facility.
2. Programming with *traits* benefits from this. Reabstraction is one of the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming)). The following is permitted with classes:

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Unfortunately this code cannot be refactored as a set of interfaces (traits) unless this is permitted. By the *Jared principle of greed*, it should be permitted.

> ***Closed issue:*** Should reabstraction be permitted? [YES] My notes were wrong. The LDM notes say that reabstraction is permitted in an interface. Not in a class.

#### Virtual Modifier vs Sealed Modifier

From Aleksey Tsingauz:

> We decided to allow modifiers explicitly stated on interface members, unless there is a reason to disallow some of them. This brings an interesting question around virtual modifier. Should it be required on members with default implementation?
>
> We could say that:
>
> - if there is no implementation and neither virtual, nor sealed are specified, we assume the member is abstract.
> - if there is an implementation and neither abstract, nor sealed are specified, we assume the member is virtual.
> - sealed modifier is required to make a method neither virtual, nor abstract.
>
> Alternatively, we could say that virtual modifier is required for a virtual member. I.e, if there is a member with implementation not explicitly marked with virtual modifier, it is neither virtual, nor abstract. This approach might provide better experience when a method is moved from a class to an interface:
>
> - an abstract method stays abstract.

- a virtual method stays virtual.
- a method without any modifier stays neither virtual, nor abstract.
- sealed modifier cannot be applied to a method that is not an override.

What do you think?

> *Closed Issue:* Should a concrete method (with implementation) be implicitly `virtual`? [YES]

*Decisions:* Made in the LDM 2017-04-05:

1. non-virtual should be explicitly expressed through `sealed` or `private`.
2. `sealed` is the keyword to make interface instance members with bodies non-virtual
3. We want to allow all modifiers in interfaces
4. Default accessibility for interface members is public, including nested types
5. private function members in interfaces are implicitly sealed, and `sealed` is not permitted on them.
6. Private classes (in interfaces) are permitted and can be sealed, and that means sealed in the class sense of sealed.
7. Absent a good proposal, partial is still not allowed on interfaces or their members.

**Binary Compatibility 1**

When a library provides a default implementation

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{
}
```

We understand that the implementation of `I1.M` in `C` is `I1.M`. What if the assembly containing `I2` is changed as follows and recompiled

```
interface I2 : I1
{
    override void M() { Impl2 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Throws some kind of runtime error

*Decision:* Made 2017-04-11: Runs `I2.M`, which is the unambiguously most specific override at runtime.

**Event accessors (closed)**

> *Closed Issue:* Can an event be overridden "piecewise"?

Consider this case:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        // error: "remove" accessor missing
    }
}
```

This "partial" implementation of the event is not permitted because, as in a class, the syntax for an event declaration does not permit only one accessor; both (or neither) must be provided. You could accomplish the same thing by permitting the abstract remove accessor in the syntax to be implicitly abstract by the absence of a body:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        remove; // implicitly abstract
    }
}
```

Note that *this is a new (proposed) syntax*. In the current grammar, event accessors have a mandatory body.

> ***Closed Issue:*** Can an event accessor be (implicitly) abstract by the omission of a body, similarly to the way that methods in interfaces and property accessors are (implicitly) abstract by the omission of a body?

***Decision:*** (2017-04-18) No, event declarations require both concrete accessors (or neither).

**Reabstraction in a Class (closed)**

***Closed Issue:*** We should confirm that this is permitted (otherwise adding a default implementation would be a breaking change):

```
interface I1
{
    void M() { }
}
abstract class C : I1
{
    public abstract void M(); // implement I1.M with an abstract method in C
}
```

***Decision:*** (2017-04-18) Yes, adding a body to an interface member declaration shouldn't break C.

**Sealed Override (closed)**

The previous question implicitly assumes that the `sealed` modifier can be applied to an `override` in an interface. This contradicts the draft specification. Do we want to permit sealing an override? Source and binary compatibility effects of sealing should be considered.

> ***Closed Issue:*** Should we permit sealing an override?

***Decision:*** (2017-04-18) Let's not allowed `sealed` on overrides in interfaces. The only use of `sealed` on interface members is to make them non-virtual in their initial declaration.

**Diamond inheritance and classes (closed)**

The draft of the proposal prefers class overrides to interface overrides in diamond inheritance scenarios:

> We require that every interface and class have a *most specific override* for every interface method among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the method itself is considered the most specific override.
>
> One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either
>
> 1. `T1` contains `T2` among its direct or indirect interfaces, or
> 2. `T2` is an interface type but `T1` is not an interface type.

The scenario is this

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        Ia a = new Derived();
        a.M();           // what does it do?
    }
}
```

We should confirm this behavior (or decide otherwise)

> ***Closed Issue:*** Confirm the draft spec, above, for *most specific override* as it applies to mixed classes and interfaces (a class takes priority over an interface). See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes.

**Interface methods vs structs (closed)**

There are some unfortunate interactions between default interface methods and structs.

```
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Note that interface members are not inherited:

```
var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'
```

Consequently, the client must box the struct to invoke interface methods

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

Boxing in this way defeats the principal benefits of a `struct` type. Moreover, any mutation methods will have no apparent effect, because they are operating on a *boxed copy* of the struct:

```
interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}
struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0
```

> ***Closed Issue:*** What can we do about this:
>
> 1. Forbid a `struct` from inheriting a default implementation. All interface methods would be treated as abstract in a `struct`. Then we may take time later to decide how to make it work better.
> 2. Come up with some kind of code generation strategy that avoids boxing. Inside a method like `IB.Increment`, the type of `this` would perhaps be akin to a type parameter constrained to `IB`. In conjunction with that, to avoid boxing in the caller, non-abstract methods would be inherited from interfaces. This may increase compiler and CLR implementation work substantially.
> 3. Not worry about it and just leave it as a wart.
> 4. Other ideas?

***Decision:*** Not worry about it and just leave it as a wart. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations.

**Base interface invocations (closed)**

The draft spec suggests a syntax for base interface invocations inspired by Java: `Interface.base.M()`. We need to select a syntax, at least for the initial prototype. My favorite is `base<Interface>.M()`.

> ***Closed Issue:*** What is the syntax for a base member invocation?

***Decision:*** The syntax is `base(Interface).M()`. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation. The interface so named must be a base interface, but does not need to be a direct base interface.

> ***Open Issue:*** Should base interface invocations be permitted in class members?

*Decision*: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation

**Overriding non-public interface members (closed)**

In an interface, non-public members from base interfaces are overridden using the `override` modifier. If it is an "explicit" override that names the interface containing the member, the access modifier is omitted.

> *Closed Issue:* If it is an "implicit" override that does not name the interface, does the access modifier have to match?

*Decision:* Only public members may be implicitly overridden, and the access must match. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list.

> *Open Issue:* Is the access modifier required, optional, or omitted on an explicit override such as `override void IB.M() {}`?

> *Open Issue:* Is `override` required, optional, or omitted on an explicit override such as `void IB.M() {}`?

How does one implement a non-public interface member in a class? Perhaps it must be done explicitly?

```
interface IA
{
    internal void MI();
    protected void MP();
}
class C : IA
{
    // are these implementations?
    internal void MI() {}
    protected void MP() {}
}
```

> *Closed Issue:* How does one implement a non-public interface member in a class?

*Decision:* You can only implement non-public interface members explicitly. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list.

*Decision*: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member

**Binary Compatibility 2 (closed)**

Consider the following code in which each type is in a separate assembly

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
    override void M() { Impl2 }
}
interface I3 : I1
{
}
class C : I2, I3
{
}
```

We understand that the implementation of `I1.M` in `C` is `I2.M`. What if the assembly containing `I3` is changed as follows and recompiled

```
interface I3 : I1
{
    override void M() { Impl3 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Runs `I3.M`
4. Either 2 or 3, deterministically
5. Throws some kind of runtime exception

*Decision*: Throw an exception (5). See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods.

**Permit `partial` in interface? (closed)**

Given that interfaces may be used in ways analogous to the way abstract classes are used, it may be useful to declare them `partial`. This would be particularly useful in the face of generators.

> *Proposal:* Remove the language restriction that interfaces and members of interfaces may not be declared `partial`.

*Decision*: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface.

**`Main` in an interface? (closed)**

> *Open Issue:* Is a `static Main` method in an interface a candidate to be the program's entry point?

*Decision*: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface.

**Confirm intent to support public non-virtual methods (closed)**

Can we please confirm (or reverse) our decision to permit non-virtual public methods in an interface?

```
interface IA
{
    public sealed void M() { }
}
```

> *Semi-Closed Issue:* (2017-04-18) We think it is going to be useful, but will come back to it. This is a mental model tripping block.

*Decision*: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods.

**Does an `override` in an interface introduce a new member? (closed)**

There are a few ways to observe whether an override declaration introduces a new member or not.

```
interface IA
{
    void M(int x) { }
}
interface IB : IA
{
    override void M(int y) { }
}
interface IC : IB
{
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}
```

> *Open Issue:* Does an override declaration in an interface introduce a new member? (closed)

In a class, an overriding method is "visible" in some senses. For example, the names of its parameters take precedence over the names of parameters in the overridden method. It may be possible to duplicate that behavior in interfaces, as there is always a most specific override. But do we want to duplicate that behavior?

Also, it is possible to "override" an override method? [Moot]

*Decision*: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member.

**Properties with a private accessor (closed)**

We say that private members are not virtual, and the combination of virtual and private is disallowed. But what about a property with a private accessor?

```
interface IA
{
    public virtual int P
    {
        get => 3;
        private set => { }
    }
}
```

Is this allowed? Is the `set` accessor here `virtual` or not? Can it be overridden where it is accessible? Does the following implicitly implement only the `get` accessor?

```
class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}
```

Is the following presumably an error because IA.P.set isn't virtual and also because it isn't accessible?

```
class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}
```

*Decision*: The first example looks valid, while the last does not. This is resolved analogously to how it already works in C#. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor

**Base Interface Invocations, round 2 (closed)**

Our previous "resolution" to how to handle base invocations doesn't actually provide sufficient expressiveness. It turns out that in C# and the CLR, unlike Java, you need to specify both the interface containing the method declaration and the location of the implementation you want to invoke.

I propose the following syntax for base calls in interfaces. I'm not in love with it, but it illustrates what any syntax must be able to express:

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>(I1).M(); // calls I3's implementation of I1.M
        base<I4>(I1).M(); // calls I4's implementation of I1.M
    }
    void I2.M()
    {
        base<I3>(I2).M(); // calls I3's implementation of I2.M
        base<I4>(I2).M(); // calls I4's implementation of I2.M
    }
}
```

If there is no ambiguity, you can write it more simply

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}
```

Or

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
    }
}
```

Or

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}
```

*Decision*: Decided on `base(N.I1<T>).M(s)`, conceding that if we have an invocation binding there may be problem here later on. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations

**Warning for struct not implementing default method? (closed)**

@vancem asserts that we should seriously consider producing a warning if a value type declaration fails to override some interface method, even if it would inherit an implementation of that method from an interface. Because it causes boxing and undermines constrained calls.

*Decision*: This seems like something more suited for an analyzer. It also seems like this warning could be noisy, since it would fire even if the default interface method is never called and no boxing will ever occur. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method

**Interface static constructors (closed)**

When are interface static constructors run? The current CLI draft proposes that it occurs when the first static method or field is accessed. If there are neither of those then it might never be run??

[2018-10-09 The CLR team proposes "Going to mirror what we do for valuetypes (cctor check on access to each instance method)"]

***Decision***: Static constructors are also run on entry to instance methods, if the static constructor was not `beforefieldinit`, in which case static constructors are run before access to the first static field. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run

## Design meetings

2017-03-08 LDM Meeting Notes 2017-03-21 LDM Meeting Notes 2017-03-23 meeting "CLR Behavior for Default Interface Methods" 2017-04-05 LDM Meeting Notes 2017-04-11 LDM Meeting Notes 2017-04-18 LDM Meeting Notes 2017-04-19 LDM Meeting Notes 2017-05-17 LDM Meeting Notes 2017-05-31 LDM Meeting Notes 2017-06-14 LDM Meeting Notes 2018-10-17 LDM Meeting Notes 2018-11-14 LDM Meeting Notes

# Async Streams

11/2/2020 • 20 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype
- [ ] Implementation
- [ ] Specification

## Summary

C# has support for iterator methods and async methods, but no support for a method that is both an iterator and an async method. We should rectify this by allowing for `await` to be used in a new form of `async` iterator, one that returns an `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` rather than an `IEnumerable<T>` or `IEnumerator<T>`, with `IAsyncEnumerable<T>` consumable in a new `await foreach`. An `IAsyncDisposable` interface is also used to enable asynchronous cleanup.

## Related discussion

- https://github.com/dotnet/roslyn/issues/261
- https://github.com/dotnet/roslyn/issues/114

## Detailed design

## Interfaces

**IAsyncDisposable**

There has been much discussion of `IAsyncDisposable` (e.g. https://github.com/dotnet/roslyn/issues/114) and whether it's a good idea. However, it's a required concept to add in support of async iterators. Since `finally` blocks may contain `await`s, and since `finally` blocks need to be run as part of disposing of iterators, we need async disposal. It's also just generally useful any time cleaning up of resources might take any period of time, e.g. closing files (requiring flushes), deregistering callbacks and providing a way to know when deregistration has completed, etc.

The following interface is added to the core .NET libraries (e.g. System.Private.CoreLib / System.Runtime):

```
namespace System
{
    public interface IAsyncDisposable
    {
        ValueTask DisposeAsync();
    }
}
```

As with `Dispose`, invoking `DisposeAsync` multiple times is acceptable, and subsequent invocations after the first should be treated as nops, returning a synchronously completed successful task (`DisposeAsync` need not be thread-safe, though, and need not support concurrent invocation). Further, types may implement both `IDisposable` and `IAsyncDisposable`, and if they do, it's similarly acceptable to invoke `Dispose` and then `DisposeAsync` or vice versa, but only the first should be meaningful and subsequent invocations of either should be a nop. As such, if a type does implement both, consumers are encouraged to call once and only once the more relevant method based on

the context, `Dispose` in synchronous contexts and `DisposeAsync` in asynchronous ones.

(I'm leaving discussion of how `IAsyncDisposable` interacts with `using` to a separate discussion. And coverage of how it interacts with `foreach` is handled later in this proposal.)

Alternatives considered:

- `DisposeAsync` *accepting a* `CancellationToken`: while in theory it makes sense that anything async can be canceled, disposal is about cleanup, closing things out, free'ing resources, etc., which is generally not something that should be canceled; cleanup is still important for work that's canceled. The same `CancellationToken` that caused the actual work to be canceled would typically be the same token passed to `DisposeAsync`, making `DisposeAsync` worthless because cancellation of the work would cause `DisposeAsync` to be a nop. If someone wants to avoid being blocked waiting for disposal, they can avoid waiting on the resulting `ValueTask`, or wait on it only for some period of time.
- `DisposeAsync` *returning a* `Task`: Now that a non-generic `ValueTask` exists and can be constructed from an `IValueTaskSource`, returning `ValueTask` from `DisposeAsync` allows an existing object to be reused as the promise representing the eventual async completion of `DisposeAsync`, saving a `Task` allocation in the case where `DisposeAsync` completes asynchronously.
- *Configuring* `DisposeAsync` *with a* `bool continueOnCapturedContext` *(*`ConfigureAwait`*)*: While there may be issues related to how such a concept is exposed to `using`, `foreach`, and other language constructs that consume this, from an interface perspective it's not actually doing any `await`'ing and there's nothing to configure... consumers of the `ValueTask` can consume it however they wish.
- `IAsyncDisposable` *inheriting* `IDisposable`: Since only one or the other should be used, it doesn't make sense to force types to implement both.
- `IDisposableAsync` *instead of* `IAsyncDisposable`: We've been following the naming that things/types are an "async something" whereas operations are "done async", so types have "Async" as a prefix and methods have "Async" as a suffix.

**IAsyncEnumerable / IAsyncEnumerator**

Two interfaces are added to the core .NET libraries:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> MoveNextAsync();
        T Current { get; }
    }
}
```

Typical consumption (without additional language features) would look like:

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        Use(enumerator.Current);
    }
}
finally { await enumerator.DisposeAsync(); }
```

Discarded options considered:

- `Task<bool> MoveNextAsync(); T current { get; }`: Using `Task<bool>` would support using a cached task object to represent synchronous, successful `MoveNextAsync` calls, but an allocation would still be required for asynchronous completion. By returning `ValueTask<bool>`, we enable the enumerator object to itself implement `IValueTaskSource<bool>` and be used as the backing for the `ValueTask<bool>` returned from `MoveNextAsync`, which in turn allows for significantly reduced overheads.
- `ValueTask<(bool, T)> MoveNextAsync();`: It's not only harder to consume, but it means that `T` can no longer be covariant.
- `ValueTask<T?> TryMoveNextAsync();`: Not covariant.
- `Task<T?> TryMoveNextAsync();`: Not covariant, allocations on every call, etc.
- `ITask<T?> TryMoveNextAsync();`: Not covariant, allocations on every call, etc.
- `ITask<(bool,T)> TryMoveNextAsync();`: Not covariant, allocations on every call, etc.
- `Task<bool> TryMoveNextAsync(out T result);`: The `out` result would need to be set when the operation returns synchronously, not when it asynchronously completes the task potentially sometime long in the future, at which point there'd be no way to communicate the result.
- `IAsyncEnumerator<T>` *not implementing* `IAsyncDisposable`: We could choose to separate these. However, doing so complicates certain other areas of the proposal, as code must then be able to deal with the possibility that an enumerator doesn't provide disposal, which makes it difficult to write pattern-based helpers. Further, it will be common for enumerators to have a need for disposal (e.g. any C# async iterator that has a finally block, most things enumerating data from a network connection, etc.), and if one doesn't, it is simple to implement the method purely as `public ValueTask DisposeAsync() => default(ValueTask);` with minimal additional overhead.
- _ `IAsyncEnumerator<T> GetAsyncEnumerator()`: No cancellation token parameter.

**Viable alternative:**

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator();
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> WaitForNextAsync();
        T TryGetNext(out bool success);
    }
}
```

`TryGetNext` is used in an inner loop to consume items with a single interface call as long as they're available synchronously. When the next item can't be retrieved synchronously, it returns false, and any time it returns false, a caller must subsequently invoke `WaitForNextAsync` to either wait for the next item to be available or to determine that there will never be another item. Typical consumption (without additional language features) would look like:

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

The advantage of this is two-fold, one minor and one major:

- *Minor: Allows for an enumerator to support multiple consumers*. There may be scenarios where it's valuable for an enumerator to support multiple concurrent consumers. That can't be achieved when `MoveNextAsync` and `Current` are separate such that an implementation can't make their usage atomic. In contrast, this approach provides a single method `TryGetNext` that supports pushing the enumerator forward and getting the next item, so the enumerator can enable atomicity if desired. However, it's likely that such scenarios could also be enabled by giving each consumer its own enumerator from a shared enumerable. Further, we don't want to enforce that every enumerator support concurrent usage, as that would add non-trivial overheads to the majority case that doesn't require it, which means a consumer of the interface generally couldn't rely on this any way.
- *Major: Performance*. The `MoveNextAsync` / `Current` approach requires two interface calls per operation, whereas the best case for `WaitForNextAsync` / `TryGetNext` is that most iterations complete synchronously, enabling a tight inner loop with `TryGetNext`, such that we only have one interface call per operation. This can have a measurable impact in situations where the interface calls dominate the computation.

However, there are non-trivial downsides, including significantly increased complexity when consuming these manually, and an increased chance of introducing bugs when using them. And while the performance benefits show up in microbenchmarks, we don't believe they'll be impactful in the vast majority of real usage. If it turns out they are, we can introduce a second set of interfaces in a light-up fashion.

Discarded options considered:

- `ValueTask<bool> WaitForNextAsync(); bool TryGetNext(out T result);` : `out` parameters can't be covariant. There's also a small impact here (an issue with the try pattern in general) that this likely incurs a runtime write barrier for reference type results.

#### Cancellation

There are several possible approaches to supporting cancellation:

1. `IAsyncEnumerable<T>` / `IAsyncEnumerator<T>` are cancellation-agnostic: `CancellationToken` doesn't appear anywhere. Cancellation is achieved by logically baking the `CancellationToken` into the enumerable and/or enumerator in whatever manner is appropriate, e.g. when calling an iterator, passing the `CancellationToken` as an argument to the iterator method and using it in the body of the iterator, as is done with any other parameter.
2. `IAsyncEnumerator<T>.GetAsyncEnumerator(CancellationToken)` : You pass a `CancellationToken` to `GetAsyncEnumerator`, and subsequent `MoveNextAsync` operations respect it however it can.
3. `IAsyncEnumerator<T>.MoveNextAsync(CancellationToken)` : You pass a `CancellationToken` to each individual `MoveNextAsync` call.
4. 1 && 2: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s into `GetAsyncEnumerator`.
5. 1 && 3: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s

into `MoveNextAsync`.

From a purely theoretical perspective, (5) is the most robust, in that (a) `MoveNextAsync` accepting a `CancellationToken` enables the most fine-grained control over what's canceled, and (b) `CancellationToken` is just any other type that can passed as an argument into iterators, embedded in arbitrary types, etc.

However, there are multiple problems with that approach:

- How does a `CancellationToken` passed to `GetAsyncEnumerator` make it into the body of the iterator? We could expose a new `iterator` keyword that you could dot off of to get access to the `CancellationToken` passed to `GetEnumerator`, but a) that's a lot of additional machinery, b) we're making it a very first-class citizen, and c) the 99% case would seem to be the same code both calling an iterator and calling `GetAsyncEnumerator` on it, in which case it can just pass the `CancellationToken` as an argument into the method.
- How does a `CancellationToken` passed to `MoveNextAsync` get into the body of the method? This is even worse, as if it's exposed off of an `iterator` local object, its value could change across awaits, which means any code that registered with the token would need to unregister from it prior to awaits and then re-register after; it's also potentially quite expensive to need to do such registering and unregistering in every `MoveNextAsync` call, regardless of whether implemented by the compiler in an iterator or by a developer manually.
- How does a developer cancel a `foreach` loop? If it's done by giving a `CancellationToken` to an enumerable/enumerator, then either a) we need to support `foreach`'ing over enumerators, which raises them to being first-class citizens, and now you need to start thinking about an ecosystem built up around enumerators (e.g. LINQ methods) or b) we need to embed the `CancellationToken` in the enumerable anyway by having some `WithCancellation` extension method off of `IAsyncEnumerable<T>` that would store the provided token and then pass it into the wrapped enumerable's `GetAsyncEnumerator` when the `GetAsyncEnumerator` on the returned struct is invoked (ignoring that token). Or, you can just use the `CancellationToken` you have in the body of the foreach.
- If/when query comprehensions are supported, how would the `CancellationToken` supplied to `GetEnumerator` or `MoveNextAsync` be passed into each clause? The easiest way would simply be for the clause to capture it, at which point whatever token is passed to `GetAsyncEnumerator`/`MoveNextAsync` is ignored.

An earlier version of this document recommended (1), but we since switched to (4).

The two main problems with (1):

- producers of cancellable enumerables have to implement some boilerplate, and can only leverage the compiler's support for async-iterators to implement a `IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken)` method.
- it is likely that many producers would be tempted to just add a `CancellationToken` parameter to their async-enumerable signature instead, which will prevent consumers from passing the cancellation token they want when they are given an `IAsyncEnumerable` type.

There are two main consumption scenarios:

1. `await foreach (var i in GetData(token)) ...` where the consumer calls the async-iterator method,
2. `await foreach (var i in givenIAsyncEnumerable.WithCancellation(token)) ...` where the consumer deals with a given `IAsyncEnumerable` instance.

We find that a reasonable compromise to support both scenarios in a way that is convenient for both producers and consumers of async-streams is to use a specially annotated parameter in the async-iterator method. The `[EnumeratorCancellation]` attribute is used for this purpose. Placing this attribute on a parameter tells the compiler that if a token is passed to the `GetAsyncEnumerator` method, that token should be used instead of the value originally passed for the parameter.

Consider `IAsyncEnumerable<int> GetData([EnumeratorCancellation] CancellationToken token = default)`. The implementer of this method can simply use the parameter in the method body. The consumer can use either

consumption patterns above:

1. if you use `GetData(token)`, then the token is saved into the async-enumerable and will be used in iteration,
2. if you use `givenIAsyncEnumerable.WithCancellation(token)`, then the token passed to `GetAsyncEnumerator` will supersede any token saved in the async-enumerable.

# foreach

`foreach` will be augmented to support `IAsyncEnumerable<T>` in addition to its existing support for `IEnumerable<T>`. And it will support the equivalent of `IAsyncEnumerable<T>` as a pattern if the relevant members are exposed publicly, falling back to using the interface directly if not, in order to enable struct-based extensions that avoid allocating as well as using alternative awaitables as the return type of `MoveNextAsync` and `DisposeAsync`.

**Syntax**

Using the syntax:

```
foreach (var i in enumerable)
```

C# will continue to treat `enumerable` as a synchronous enumerable, such that even if it exposes the relevant APIs for async enumerables (exposing the pattern or implementing the interface), it will only consider the synchronous APIs.

To force `foreach` to instead only consider the asynchronous APIs, `await` is inserted as follows:

```
await foreach (var i in enumerable)
```

No syntax would be provided that would support using either the async or the sync APIs; the developer must choose based on the syntax used.

Discarded options considered:

- *`foreach (var i in await enumerable)`*: This is already valid syntax, and changing its meaning would be a breaking change. This means to `await` the `enumerable`, get back something synchronously iterable from it, and then synchronously iterate through that.
- *`foreach (var i await in enumerable)`*, *`foreach (var await i in enumerable)`*, *`foreach (await var i in enumerable)`*: These all suggest that we're awaiting the next item, but there are other awaits involved in foreach, in particular if the enumerable is an `IAsyncDisposable`, we will be `await`'ing its async disposal. That await is as the scope of the foreach rather than for each individual element, and thus the `await` keyword deserves to be at the `foreach` level. Further, having it associated with the `foreach` gives us a way to describe the `foreach` with a different term, e.g. a "await foreach". But more importantly, there's value in considering `foreach` syntax at the same time as `using` syntax, so that they remain consistent with each other, and `using (await ...)` is already valid syntax.
- *`foreach await (var i in enumerable)`*

Still to consider:

- `foreach` today does not support iterating through an enumerator. We expect it will be more common to have `IAsyncEnumerator<T>`s handed around, and thus it's tempting to support `await foreach` with both `IAsyncEnumerable<T>` and `IAsyncEnumerator<T>`. But once we add such support, it introduces the question of whether `IAsyncEnumerator<T>` is a first-class citizen, and whether we need to have overloads of combinators that operate on enumerators in addition to enumerables? Do we want to encourage methods to return enumerators rather than enumerables? We should continue to discuss this. If we decide we don't want to support it, we might want to introduce an extension method

`public static IAsyncEnumerable<T> AsEnumerable<T>(this IAsyncEnumerator<T> enumerator);` that would allow an enumerator to still be `foreach`'d. If we decide we do want to support it, we'll need to also decide on whether the `await foreach` would be responsible for calling `DisposeAsync` on the enumerator, and the answer is likely "no, control over disposal should be handled by whoever called `GetEnumerator`."

**Pattern-based Compilation**

The compiler will bind to the pattern-based APIs if they exist, preferring those over using the interface (the pattern may be satisfied with instance methods or extension methods). The requirements for the pattern are:

- The enumerable must expose a `GetAsyncEnumerator` method that may be called with no arguments and that returns an enumerator that meets the relevant pattern.
- The enumerator must expose a `MoveNextAsync` method that may be called with no arguments and that returns something which may be `await`ed and whose `GetResult()` returns a `bool`.
- The enumerator must also expose `Current` property whose getter returns a `T` representing the kind of data being enumerated.
- The enumerator may optionally expose a `DisposeAsync` method that may be invoked with no arguments and that returns something that can be `await`ed and whose `GetResult()` returns `void`.

This code:

```
var enumerable = ...;
await foreach (T item in enumerable)
{
   ...
}
```

is translated to the equivalent of:

```
var enumerable = ...;
var enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
       T item = enumerator.Current;
       ...
    }
}
finally
{
    await enumerator.DisposeAsync(); // omitted, along with the try/finally, if the enumerator doesn't expose
DisposeAsync
}
```

If the iterated type doesn't expose the right pattern, the interfaces will be used.

**ConfigureAwait**

This pattern-based compilation will allow `ConfigureAwait` to be used on all of the awaits, via a `ConfigureAwait` extension method:

```
await foreach (T item in enumerable.ConfigureAwait(false))
{
   ...
}
```

This will be based on types we'll add to .NET as well, likely to System.Threading.Tasks.Extensions.dll:

```
// Approximate implementation, omitting arg validation and the like
namespace System.Threading.Tasks
{
    public static class AsyncEnumerableExtensions
    {
        public static ConfiguredAsyncEnumerable<T> ConfigureAwait<T>(this IAsyncEnumerable<T> enumerable, bool
continueOnCapturedContext) =>
            new ConfiguredAsyncEnumerable<T>(enumerable, continueOnCapturedContext);

        public struct ConfiguredAsyncEnumerable<T>
        {
            private readonly IAsyncEnumerable<T> _enumerable;
            private readonly bool _continueOnCapturedContext;

            internal ConfiguredAsyncEnumerable(IAsyncEnumerable<T> enumerable, bool continueOnCapturedContext)
            {
                _enumerable = enumerable;
                _continueOnCapturedContext = continueOnCapturedContext;
            }

            public ConfiguredAsyncEnumerator<T> GetAsyncEnumerator() =>
                new ConfiguredAsyncEnumerator<T>(_enumerable.GetAsyncEnumerator(), _continueOnCapturedContext);

            public struct Enumerator
            {
                private readonly IAsyncEnumerator<T> _enumerator;
                private readonly bool _continueOnCapturedContext;

                internal Enumerator(IAsyncEnumerator<T> enumerator, bool continueOnCapturedContext)
                {
                    _enumerator = enumerator;
                    _continueOnCapturedContext = continueOnCapturedContext;
                }

                public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>
                    _enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

                public T Current => _enumerator.Current;

                public ConfiguredValueTaskAwaitable DisposeAsync() =>
                    _enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
            }
        }
    }
}
```

Note that this approach will not enable `ConfigureAwait` to be used with pattern-based enumerables, but then again it's already the case that the `ConfigureAwait` is only exposed as an extension on `Task` / `Task<T>` / `ValueTask` / `ValueTask<T>` and can't be applied to arbitrary awaitable things, as it only makes sense when applied to Tasks (it controls a behavior implemented in Task's continuation support), and thus doesn't make sense when using a pattern where the awaitable things may not be tasks. Anyone returning awaitable things can provide their own custom behavior in such advanced scenarios.

(If we can come up with some way to support a scope- or assembly-level `ConfigureAwait` solution, then this won't be necessary.)

## Async Iterators

The language / compiler will support producing `IAsyncEnumerable<T>`s and `IAsyncEnumerator<T>`s in addition to consuming them. Today the language supports writing an iterator like:

```
static IEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(1000);
            yield return i;
        }
    }
    finally
    {
        Thread.Sleep(200);
        Console.WriteLine("finally");
    }
}
```

but `await` can't be used in the body of these iterators. We will add that support.

**Syntax**

The existing language support for iterators infers the iterator nature of the method based on whether it contains any `yield` s. The same will be true for async iterators. Such async iterators will be demarcated and differentiated from synchronous iterators via adding `async` to the signature, and must then also have either `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` as its return type. For example, the above example could be written as an async iterator as follows:

```
static async IAsyncEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
    finally
    {
        await Task.Delay(200);
        Console.WriteLine("finally");
    }
}
```

Alternatives considered:

- *Not using `async` in the signature*: Using `async` is likely technically required by the compiler, as it uses it to determine whether `await` is valid in that context. But even if it's not required, we've established that `await` may only be used in methods marked as `async`, and it seems important to keep the consistency.
- *Enabling custom builders for `IAsyncEnumerable<T>`*: That's something we could look at for the future, but the machinery is complicated and we don't support that for the synchronous counterparts.
- *Having an `iterator` keyword in the signature*: Async iterators would use `async iterator` in the signature, and `yield` could only be used in `async` methods that included `iterator`; `iterator` would then be made optional on synchronous iterators. Depending on your perspective, this has the benefit of making it very clear by the signature of the method whether `yield` is allowed and whether the method is actually meant to return instances of type `IAsyncEnumerable<T>` rather than the compiler manufacturing one based on whether the code uses `yield` or not. But it is different from synchronous iterators, which don't and can't be made to require one. Plus some developers don't like the extra syntax. If we were designing it from scratch, we'd probably make this required, but at this point there's much more value in keeping async iterators close to sync iterators.

# LINQ

There are over ~200 overloads of methods on the `System.Linq.Enumerable` class, all of which work in terms of `IEnumerable<T>`; some of these accept `IEnumerable<T>`, some of them produce `IEnumerable<T>`, and many do both. Adding LINQ support for `IAsyncEnumerable<T>` would likely entail duplicating all of these overloads for it, for another ~200. And since `IAsyncEnumerator<T>` is likely to be more common as a standalone entity in the asynchronous world than `IEnumerator<T>` is in the synchronous world, we could potentially need another ~200 overloads that work with `IAsyncEnumerator<T>`. Plus, a large number of the overloads deal with predicates (e.g. `Where` that takes a `Func<T, bool>`), and it may be desirable to have `IAsyncEnumerable<T>`-based overloads that deal with both synchronous and asynchronous predicates (e.g. `Func<T, ValueTask<bool>>` in addition to `Func<T, bool>`). While this isn't applicable to all of the now ~400 new overloads, a rough calculation is that it'd be applicable to half, which means another ~200 overloads, for a total of ~600 new methods.

That is a staggering number of APIs, with the potential for even more when extension libraries like Interactive Extensions (Ix) are considered. But Ix already has an implementation of many of these, and there doesn't seem to be a great reason to duplicate that work; we should instead help the community improve Ix and recommend it for when developers want to use LINQ with `IAsyncEnumerable<T>`.

There is also the issue of query comprehension syntax. The pattern-based nature of query comprehensions would allow them to "just work" with some operators, e.g. if Ix provides the following methods:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
Func<TSource, TResult> func);
public static IAsyncEnumerable<T> Where(this IAsyncEnumerable<T> source, Func<T, bool> func);
```

then this C# code will "just work":

```
IAsyncEnumerable<int> enumerable = ...;
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select item * 2;
```

However, there is no query comprehension syntax that supports using `await` in the clauses, so if Ix added, for example:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
Func<TSource, ValueTask<TResult>> func);
```

then this would "just work":

```
IAsyncEnumerable<string> result = from url in urls
                                  where item % 2 == 0
                                  select SomeAsyncMethod(item);

async ValueTask<int> SomeAsyncMethod(int item)
{
    await Task.Yield();
    return item * 2;
}
```

but there'd be no way to write it with the `await` inline in the `select` clause. As a separate effort, we could look into adding `async { ... }` expressions to the language, at which point we could allow them to be used in query comprehensions and the above could instead be written as:

```
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select async
                               {
                                   await Task.Yield();
                                   return item * 2;
                               };
```

or to enabling `await` to be used directly in expressions, such as by supporting `async from`. However, it's unlikely a design here would impact the rest of the feature set one way or the other, and this isn't a particularly high-value thing to invest in right now, so the proposal is to do nothing additional here right now.

## Integration with other asynchronous frameworks

Integration with `IObservable<T>` and other asynchronous frameworks (e.g. reactive streams) would be done at the library level rather than at the language level. For example, all of the data from an `IAsyncEnumerator<T>` can be published to an `IObserver<T>` simply by `await foreach`'ing over the enumerator and `OnNext`'ing the data to the observer, so an `AsObservable<T>` extension method is possible. Consuming an `IObservable<T>` in a `await foreach` requires buffering the data (in case another item is pushed while the previous item is still being processing), but such a push-pull adapter can easily be implemented to enable an `IObservable<T>` to be pulled from with an `IAsyncEnumerator<T>`. Etc. Rx/Ix already provide prototypes of such implementations, and libraries like https://github.com/dotnet/corefx/tree/master/src/System.Threading.Channels provide various kinds of buffering data structures. The language need not be involved at this stage.

# Ranges

11/2/2020 • 12 minutes to read • Edit Online

## Summary

This feature is about delivering two new operators that allow constructing `System.Index` and `System.Range` objects, and using them to index/slice collections at runtime.

## Overview

**Well-known types and members**

To use the new syntactic forms for `System.Index` and `System.Range`, new well-known types and members may be necessary, depending on which syntactic forms are used.

To use the "hat" operator (`^`), the following is required

```
namespace System
{
    public readonly struct Index
    {
        public Index(int value, bool fromEnd);
    }
}
```

To use the `System.Index` type as an argument in an array element access, the following member is required:

```
int System.Index.GetOffset(int length);
```

The `..` syntax for `System.Range` will require the `System.Range` type, as well as one or more of the following members:

```
namespace System
{
    public readonly struct Range
    {
        public Range(System.Index start, System.Index end);
        public static Range StartAt(System.Index start);
        public static Range EndAt(System.Index end);
        public static Range All { get; }
    }
}
```

The `..` syntax allows for either, both, or none of its arguments to be absent. Regardless of the number of arguments, the `Range` constructor is always sufficient for using the `Range` syntax. However, if any of the other members are present and one or more of the `..` arguments are missing, the appropriate member may be substituted.

Finally, for a value of type `System.Range` to be used in an array element access expression, the following member must be present:

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeHelpers
    {
        public static T[] GetSubArray<T>(T[] array, System.Range range);
    }
}
```

### System.Index

C# has no way of indexing a collection from the end, but rather most indexers use the "from start" notion, or do a "length - i" expression. We introduce a new Index expression that means "from the end". The feature will introduce a new unary prefix "hat" operator. Its single operand must be convertible to `System.Int32`. It will be lowered into the appropriate `System.Index` factory method call.

We augment the grammar for *unary_expression* with the following additional syntax form:

```
unary_expression
    : '^' unary_expression
    ;
```

We call this the *index from end* operator. The predefined *index from end* operators are as follows:

```
System.Index operator ^(int fromEnd);
```

The behavior of this operator is only defined for input values greater than or equal to zero.

Examples:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var thirdItem = array[2];    // array[2]
var lastItem = array[^1];    // array[new Index(1, fromEnd: true)]
```

#### System.Range

C# has no syntactic way to access "ranges" or "slices" of collections. Usually users are forced to implement complex structures to filter/operate on slices of memory, or resort to LINQ methods like `list.Skip(5).Take(2)`. With the addition of `System.Span<T>` and other similar types, it becomes more important to have this kind of operation supported on a deeper level in the language/runtime, and have the interface unified.

The language will introduce a new range operator `x..y`. It is a binary infix operator that accepts two expressions. Either operand can be omitted (examples below), and they have to be convertible to `System.Index`. It will be lowered to the appropriate `System.Range` factory method call.

We replace the C# grammar rules for *multiplicative_expression* with the following (in order to introduce a new precedence level):

```
range_expression
    : unary_expression
    | range_expression? '..' range_expression?
    ;

multiplicative_expression
    : range_expression
    | multiplicative_expression '*' range_expression
    | multiplicative_expression '/' range_expression
    | multiplicative_expression '%' range_expression
    ;
```

All forms of the *range operator* have the same precedence. This new precedence group is lower than the *unary operators* and higher than the *multiplicative arithmetic operators*.

We call the `..` operator the *range operator*. The built-in range operator can roughly be understood to correspond to the invocation of a built-in operator of this form:

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Examples:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var slice1 = array[2..^3];    // array[new Range(2, new Index(3, fromEnd: true))]
var slice2 = array[..^3];     // array[Range.EndAt(new Index(3, fromEnd: true))]
var slice3 = array[2..];      // array[Range.StartAt(2)]
var slice4 = array[..];       // array[Range.All]
```

Moreover, `System.Index` should have an implicit conversion from `System.Int32`, in order to avoid the need to overload mixing integers and indexes over multi-dimensional signatures.

# Adding Index and Range support to existing library types

**Implicit Index support**

The language will provide an instance indexer member with a single parameter of type `Index` for types which meet the following criteria:

- The type is Countable.
- The type has an accessible instance indexer which takes a single `int` as the argument.
- The type does not have an accessible instance indexer which takes an `Index` as the first parameter. The `Index` must be the only parameter or the remaining parameters must be optional.

A type is ***Countable*** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. The language can make use of this property to convert an expression of type `Index` into an `int` at the point of the expression without the need to use the type `Index` at all. In case both `Length` and `Count` are present, `Length` will be preferred. For simplicity going forward, the proposal will use the name `Length` to represent `Count` or `Length`.

For such types, the language will act as if there is an indexer member of the form `T this[Index index]` where `T` is the return type of the `int` based indexer including any `ref` style annotations. The new member will have the same `get` and `set` members with matching accessibility as the `int` indexer.

The new indexer will be implemented by converting the argument of type `Index` into an `int` and emitting a call to the `int` based indexer. For discussion purposes, let's use the example of `receiver[expr]`. The conversion of `expr` to `int` will occur as follows:

- When the argument is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

This allows for developers to use the `Index` feature on existing types without the need for modification. For example:

```
List<char> list = ...;
var value = list[^1];

// Gets translated to
var value = list[list.Count - 1];
```

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    int this[int index] => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get()[^1];
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

**Implicit Range support**

The language will provide an instance indexer member with a single parameter of type `Range` for types which meet the following criteria:

- The type is Countable.
- The type has an accessible member named `Slice` which has two parameters of type `int`.
- The type does not have an instance indexer which takes a single `Range` as the first parameter. The `Range` must be the only parameter or the remaining parameters must be optional.

For such types, the language will bind as if there is an indexer member of the form `T this[Range range]` where `T` is the return type of the `Slice` method including any `ref` style annotations. The new member will also have matching accessibility with `Slice`.

When the `Range` based indexer is bound on an expression named `receiver`, it will be lowered by converting the `Range` expression into two values that are then passed to the `Slice` method. For discussion purposes, let's use the

example of `receiver[expr]`.

The first argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr2` can be omitted) and `expr1` has type `int`, then it will be emitted as `expr1`.
- When `expr` is of the form `^expr1..expr2` (where `expr2` can be omitted), then it will be emitted as `receiver.Length - expr1`.
- When `expr` is of the form `..expr2` (where `expr2` can be omitted), then it will be emitted as `0`.
- Otherwise, it will be emitted as `expr.Start.GetOffset(receiver.Length)`.

This value will be re-used in the calculation of the second `Slice` argument. When doing so it will be referred to as `start`. The second argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr1` can be omitted) and `expr2` has type `int`, then it will be emitted as `expr2 - start`.
- When `expr` is of the form `expr1..^expr2` (where `expr1` can be omitted), then it will be emitted as `(receiver.Length - expr2) - start`.
- When `expr` is of the form `expr1..` (where `expr1` can be omitted), then it will be emitted as `receiver.Length - start`.
- Otherwise, it will be emitted as `expr.End.GetOffset(receiver.Length) - start`.

The `receiver`, `Length`, and `expr` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    int[] Slice(int start, int length) {
        var slice = new int[length];
        Array.Copy(_array, start, slice, 0, length);
        return slice;
    }
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        var array = Get()[0..2];
        Console.WriteLine(array.length);
    }
}
```

This code will print "Get Length 2".

The language will special case the following known types:

- `string`: the method `Substring` will be used instead of `Slice`.
- `array`: the method `System.Reflection.CompilerServices.GetSubArray` will be used instead of `Slice`.

## Alternatives

The new operators (`^` and `..`) are syntactic sugar. The functionality can be implemented by explicit calls to `System.Index` and `System.Range` factory methods, but it will result in a lot more boilerplate code, and the experience will be unintuitive.

## IL Representation

These two operators will be lowered to regular indexer/method calls, with no change in subsequent compiler layers.

## Runtime behavior

- Compiler can optimize indexers for built-in types like arrays and strings, and lower the indexing to the appropriate existing methods.
- `System.Index` will throw if constructed with a negative value.
- `^0` does not throw, but it translates to the length of the collection/enumerable it is supplied to.
- `Range.All` is semantically equivalent to `0..^0`, and can be deconstructed to these indices.

## Considerations

**Detect Indexable based on ICollection**

The inspiration for this behavior was collection initializers. Using the structure of a type to convey that it had opted into a feature. In the case of collection initializers types can opt into the feature by implementing the interface `IEnumerable` (non generic).

This proposal initially required that types implement `ICollection` in order to qualify as Indexable. That required a number of special cases though:

- `ref struct`: these cannot implement interfaces yet types like `Span<T>` are ideal for index / range support.
- `string`: does not implement `ICollection` and adding that `interface` has a large cost.

This means to support key types special casing is already needed. The special casing of `string` is less interesting as the language does this in other areas (`foreach` lowering, constants, etc ...). The special casing of `ref struct` is more concerning as it's special casing an entire class of types. They get labeled as Indexable if they simply have a property named `Count` with a return type of `int`.

After consideration the design was normalized to say that any type which has a property `Count` / `Length` with a return type of `int` is Indexable. That removes all special casing, even for `string` and arrays.

**Detect just Count**

Detecting on the property names `Count` or `Length` does complicate the design a bit. Picking just one to standardize though is not sufficient as it ends up excluding a large number of types:

- Use `Length`: excludes pretty much every collection in System.Collections and sub-namespaces. Those tend to derive from `ICollection` and hence prefer `Count` over length.
- Use `Count`: excludes `string`, arrays, `Span<T>` and most `ref struct` based types

The extra complication on the initial detection of Indexable types is outweighed by its simplification in other aspects.

### Choice of Slice as a name

The name `Slice` was chosen as it's the de-facto standard name for slice style operations in .NET. Starting with netcoreapp2.1 all span style types use the name `Slice` for slicing operations. Prior to netcoreapp2.1 there really aren't any examples of slicing to look to for an example. Types like `List<T>`, `ArraySegment<T>`, `SortedList<T>` would've been ideal for slicing but the concept didn't exist when types were added.

Thus, `Slice` being the sole example, it was chosen as the name.

### Index target type conversion

Another way to view the `Index` transformation in an indexer expression is as a target type conversion. Instead of binding as if there is a member of the form `return_type this[Index]`, the language instead assigns a target typed conversion to `int`.

This concept could be generalized to all member access on Countable types. Whenever an expression with type `Index` is used as an argument to an instance member invocation and the receiver is Countable then the expression will have a target type conversion to `int`. The member invocations applicable for this conversion include methods, indexers, properties, extension methods, etc ... Only constructors are excluded as they have no receiver.

The target type conversion will be implemented as follows for any expression which has a type of `Index`. For discussion purposes lets use the example of `receiver[expr]`:

- When `expr` is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    int GetAt(int index) => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get().GetAt(^1);
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

This feature would be beneficial to any member which had a parameter that represented an index. For example `List<T>.InsertAt`. This also has the potential for confusion as the language can't give any guidance as to whether or not an expression is meant for indexing. All it can do is convert any `Index` expression to `int` when invoking a member on a Countable type.

Restrictions:

- This conversion is only applicable when the expression with type `Index` is directly an argument to the member. It would not apply to any nested expressions.

## Decisions made during implementation

- All members in the pattern must be instance members
- If a Length method is found but it has the wrong return type, continue looking for Count
- The indexer used for the Index pattern must have exactly one int parameter
- The Slice method used for the Range pattern must have exactly two int parameters
- When looking for the pattern members, we look for original definitions, not constructed members

## Design meetings

- Jan 10, 2018
- Jan 18, 2018
- Jan 22, 2018
- Dec 3, 2018
- Mar 25, 2019
- April 1st, 2019
- April 15, 2019

# "pattern-based using" and "using declarations"

11/2/2020 • 4 minutes to read • Edit Online

## Summary

The language will add two new capabilities around the `using` statement in order to make resource management simpler: `using` should recognize a disposable pattern in addition to `IDisposable` and add a `using` declaration to the language.

## Motivation

The `using` statement is an effective tool for resource management today but it requires quite a bit of ceremony. Methods that have a number of resources to manage can get syntactically bogged down with a series of `using` statements. This syntax burden is enough that most coding style guidelines explicitly have an exception around braces for this scenario.

The `using` declaration removes much of the ceremony here and gets C# on par with other languages that include resource management blocks. Additionally the pattern-based `using` lets developers expand the set of types that can participate here. In many cases removing the need to create wrapper types that only exist to allow for a values use in a `using` statement.

Together these features allow developers to simplify and expand the scenarios where `using` can be applied.

## Detailed Design

**using declaration**

The language will allow for `using` to be added to a local variable declaration. Such a declaration will have the same effect as declaring the variable in a `using` statement at the same location.

```
if (...)
{
   using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");
   // statements
}

// Equivalent to
if (...)
{
   using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))
   {
    // statements
   }
}
```

The lifetime of a `using` local will extend to the end of the scope in which it is declared. The `using` locals will then be disposed in the reverse order in which they are declared.

```
{
    using var f1 = new FileStream("...");
    using var f2 = new FileStream("..."), f3 = new FileStream("...");
    ...
    // Dispose f3
    // Dispose f2
    // Dispose f1
}
```

There are no restrictions around `goto`, or any other control flow construct in the face of a `using` declaration. Instead the code acts just as it would for the equivalent `using` statement:

```
{
    using var f1 = new FileStream("...");
  target:
    using var f2 = new FileStream("...");
    if (someCondition)
    {
        // Causes f2 to be disposed but has no effect on f1
        goto target;
    }
}
```

A local declared in a `using` local declaration will be implicitly read-only. This matches the behavior of locals declared in a `using` statement.

The language grammar for `using` declarations will be the following:

```
local-using-declaration:
  using type using-declarators

using-declarators:
  using-declarator
  using-declarators , using-declarator

using-declarator:
  identifier = expression
```

Restrictions around `using` declaration:

- May not appear directly inside a `case` label but instead must be within a block inside the `case` label.
- May not appear as part of an `out` variable declaration.
- Must have an initializer for each declarator.
- The local type must be implicitly convertible to `IDisposable` or fulfill the `using` pattern.

**pattern-based using**

The language will add the notion of a disposable pattern: that is a type which has an accessible `Dispose` instance method. Types which fit the disposable pattern can participate in a `using` statement or declaration without being required to implement `IDisposable`.

```
class Resource
{
    public void Dispose() { ... }
}

using (var r = new Resource())
{
    // statements
}
```

This will allow developers to leverage `using` in a number of new scenarios:

- `ref struct`: These types can't implement interfaces today and hence can't participate in `using` statements.
- Extension methods will allow developers to augment types in other assemblies to participate in `using` statements.

In the situation where a type can be implicitly converted to `IDisposable` and also fits the disposable pattern, then `IDisposable` will be preferred. While this takes the opposite approach of `foreach` (pattern preferred over interface) it is necessary for backwards compatibility.

The same restrictions from a traditional `using` statement apply here as well: local variables declared in the `using` are read-only, a `null` value will not cause an exception to be thrown, etc ... The code generation will be different only in that there will not be a cast to `IDisposable` before calling Dispose:

```
{
   Resource r = new Resource();
   try {
      // statements
   }
   finally {
      if (resource != null) resource.Dispose();
   }
}
```

In order to fit the disposable pattern the `Dispose` method must be accessible, parameterless and have a `void` return type. There are no other restrictions. This explicitly means that extension methods can be used here.

# Considerations

**case labels without blocks**

A `using declaration` is illegal directly inside a `case` label due to complications around its actual lifetime. One potential solution is to simply give it the same lifetime as an `out var` in the same location. It was deemed the extra complexity to the feature implementation and the ease of the work around (just add a block to the `case` label) didn't justify taking this route.

# Future Expansions

**fixed locals**

A `fixed` statement has all of the properties of `using` statements that motivated the ability to have `using` locals. Consideration should be given to extending this feature to `fixed` locals as well. The lifetime and ordering rules should apply equally well for `using` and `fixed` here.

# Static local functions

11/2/2020 • 2 minutes to read • Edit Online

## Summary

Support local functions that disallow capturing state from the enclosing scope.

## Motivation

Avoid unintentionally capturing state from the enclosing context. Allow local functions to be used in scenarios where a `static` method is required.

## Detailed design

A local function declared `static` cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a `static` local function.

A `static` local function cannot reference instance members from an implicit or explicit `this` or `base` reference.

A `static` local function may reference `static` members from the enclosing scope.

A `static` local function may reference `constant` definitions from the enclosing scope.

`nameof()` in a `static` local function may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` local functions.

A `static` local function definition is emitted as a `static` method in metadata, even if only used in a delegate.

A non-`static` local function or lambda can capture state from an enclosing `static` local function but cannot capture state outside the enclosing `static` local function.

A `static` local function cannot be invoked in an expression tree.

A call to a local function is emitted as `call` rather than `callvirt`, regardless of whether the local function is `static`.

Overload resolution of a call within a local function not affected by whether the local function is `static`.

Removing the `static` modifier from a local function in a valid program does not change the meaning of the program.

## Design meetings

https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions

# null coalescing assignment

11/2/2020 • 2 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype: Completed
- [x] Implementation: Completed
- [x] Specification: Below

## Summary

Simplifies a common coding pattern where a variable is assigned a value if it is null.

As part of this proposal, we will also loosen the type requirements on `??` to allow an expression whose type is an unconstrained type parameter to be used on the left-hand side.

## Motivation

It is common to see code of the form

```
if (variable == null)
{
    variable = expression;
}
```

This proposal adds a non-overloadable binary operator to the language that performs this function.

There have been at least eight separate community requests for this feature.

## Detailed design

We add a new form of assignment operator

```
assignment_operator
    : '??='
    ;
```

Which follows the existing semantic rules for compound assignment operators, except that we elide the assignment if the left-hand side is non-null. The rules for this feature are as follows.

Given `a ??= b`, where `A` is the type of `a`, `B` is the type of `b`, and `A0` is the underlying type of `A` if `A` is a nullable value type:

1. If `A` does not exist or is a non-nullable value type, a compile-time error occurs.
2. If `B` is not implicitly convertible to `A` or `A0` (if `A0` exists), a compile-time error occurs.
3. If `A0` exists and `B` is implicitly convertible to `A0`, and `B` is not dynamic, then the type of `a ??= b` is `A0`. `a ??= b` is evaluated at runtime as:

   ```
   var tmp = a.GetValueOrDefault();
   if (!a.HasValue) { tmp = b; a = tmp; }
   tmp
   ```

Except that `a` is only evaluated once.

4. Otherwise, the type of `a ??= b` is `A`. `a ??= b` is evaluated at runtime as `a ?? (a = b)`, except that `a` is only evaluated once.

For the relaxation of the type requirements of `??`, we update the spec where it currently states that, given `a ?? b`, where `A` is the type of `a`:

> 1. If A exists and is not a nullable type or a reference type, a compile-time error occurs.

We relax this requirement to:

1. If A exists and is a non-nullable value type, a compile-time error occurs.

This allows the null coalescing operator to work on unconstrained type parameters, as the unconstrained type parameter T exists, is not a nullable type, and is not a reference type.

# Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

# Alternatives

The programmer can write `(x = x ?? y)`, `if (x == null) x = y;`, or `x ?? (x = y)` by hand.

# Unresolved questions

- [ ] Requires LDM review
- [ ] Should we also support `&&=` and `||=` operators?

# Design meetings

None.

# Readonly Instance Members

11/2/2020 • 4 minutes to read • Edit Online

Championed Issue: https://github.com/dotnet/csharplang/issues/1710

## Summary

Provide a way to specify individual instance members on a struct do not modify state, in the same way that `readonly struct` specifies no instance members modify state.

It is worth noting that `readonly instance member` != `pure instance member`. A `pure` instance member guarantees no state will be modified. A `readonly` instance member only guarantees that instance state will not be modified.

All instance members on a `readonly struct` could be considered implicitly `readonly instance members`. Explicit `readonly instance members` declared on non-readonly structs would behave in the same manner. For example, they would still create hidden copies if you called an instance member (on the current instance or on a field of the instance) which was itself not-readonly.

## Motivation

Today, users have the ability to create `readonly struct` types which the compiler enforces that all fields are readonly (and by extension, that no instance members modify the state). However, there are some scenarios where you have an existing API that exposes accessible fields or that has a mix of mutating and non-mutating members. Under these circumstances, you cannot mark the type as `readonly` (it would be a breaking change).

This normally doesn't have much impact, except in the case of `in` parameters. With `in` parameters for non-readonly structs, the compiler will make a copy of the parameter for each instance member invocation, since it cannot guarantee that the invocation does not modify internal state. This can lead to a multitude of copies and worse overall performance than if you had just passed the struct directly by value. For an example, see this code on sharplab

Some other scenarios where hidden copies can occur include `static readonly fields` and `literals`. If they are supported in the future, `blittable constants` would end up in the same boat; that is they all currently necessitate a full copy (on instance member invocation) if the struct is not marked `readonly`.

## Design

Allow a user to specify that an instance member is, itself, `readonly` and does not modify the state of the instance (with all the appropriate verification done by the compiler, of course). For example:

```
public struct Vector2
{
    public float x;
    public float y;

    public readonly float GetLengthReadonly()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public float GetLength()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public readonly float GetLengthIllegal()
    {
        var tmp = MathF.Sqrt(LengthSquared);

        x = tmp;    // Compiler error, cannot write x
        y = tmp;    // Compiler error, cannot write y

        return tmp;
    }

    public float LengthSquared
    {
        readonly get
        {
            return (x * x) +
                   (y * y);
        }
    }
}

public static class MyClass
{
    public static float ExistingBehavior(in Vector2 vector)
    {
        // This code causes a hidden copy, the compiler effectively emits:
        //    var tmpVector = vector;
        //    return tmpVector.GetLength();
        //
        // This is done because the compiler doesn't know that `GetLength()`
        // won't mutate `vector`.

        return vector.GetLength();
    }

    public static float ReadonlyBehavior(in Vector2 vector)
    {
        // This code is emitted exactly as listed. There are no hidden
        // copies as the `readonly` modifier indicates that the method
        // won't mutate `vector`.

        return vector.GetLengthReadonly();
    }
}
```

Readonly can be applied to property accessors to indicate that `this` will not be mutated in the accessor. The following examples have readonly setters because those accessors modify the state of member field, but do not modify the value of that member field.

```
public int Prop1
{
    readonly get
    {
        return this._store["Prop1"];
    }
    readonly set
    {
        this._store["Prop1"] = value;
    }
}
```

When `readonly` is applied to the property syntax, it means that all accessors are `readonly`.

```
public readonly int Prop2
{
    get
    {
        return this._store["Prop2"];
    }
    set
    {
        this._store["Prop2"] = value;
    }
}
```

Readonly can only be applied to accessors which do not mutate the containing type.

```
public int Prop3
{
    readonly get
    {
        return this._prop3;
    }
    set
    {
        this._prop3 = value;
    }
}
```

Readonly can be applied to some auto-implemented properties, but it won't have a meaningful effect. The compiler will treat all auto-implemented getters as readonly whether or not the `readonly` keyword is present.

```
// Allowed
public readonly int Prop4 { get; }
public int Prop5 { readonly get; }
public int Prop6 { readonly get; set; }

// Not allowed
public readonly int Prop7 { get; set; }
public int Prop8 { get; readonly set; }
```

Readonly can be applied to manually-implemented events, but not field-like events. Readonly cannot be applied to individual event accessors (add/remove).

```
// Allowed
public readonly event Action<EventArgs> Event1
{
    add { }
    remove { }
}

// Not allowed
public readonly event Action<EventArgs> Event2;
public event Action<EventArgs> Event3
{
    readonly add { }
    readonly remove { }
}
public static readonly event Event4
{
    add { }
    remove { }
}
```

Some other syntax examples:

- Expression bodied members: `public readonly float ExpressionBodiedMember => (x * x) + (y * y);`
- Generic constraints: `public static readonly void GenericMethod<T>(T value) where T : struct { }`

The compiler would emit the instance member, as usual, and would additionally emit a compiler recognized attribute indicating that the instance member does not modify state. This effectively causes the hidden `this` parameter to become `in T` instead of `ref T`.

This would allow the user to safely call said instance method without the compiler needing to make a copy.

The restrictions would include:

- The `readonly` modifier cannot be applied to static methods, constructors or destructors.
- The `readonly` modifier cannot be applied to delegates.
- The `readonly` modifier cannot be applied to members of class or interface.

## Drawbacks

Same drawbacks as exist with `readonly struct` methods today. Certain code may still cause hidden copies.

## Notes

Using an attribute or another keyword may also be possible.

This proposal is somewhat related to (but is more a subset of) `functional purity` and/or `constant expressions`, both of which have had some existing proposals.

# Permit `stackalloc` in nested contexts

11/2/2020 • 2 minutes to read • Edit Online

### Stack allocation

We modify the section *Stack allocation* of the C# language specification to relax the places when a `stackalloc` expression may appear. We delete

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

and replace them with

```
primary_no_array_creation_expression
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression? ']' array_initializer?
    | 'stackalloc' '[' expression? ']' array_initializer
    ;
```

Note that the addition of an *array_initializer* to *stackalloc_initializer* (and making the index expression optional) was an extension in C# 7.3 and is not described here.

The *element type* of the `stackalloc` expression is the *unmanaged_type* named in the stackalloc expression, if any, or the common type among the elements of the *array_initializer* otherwise.

The type of the *stackalloc_initializer* with *element type* `K` depends on its syntactic context:

- If the *stackalloc_initializer* appears directly as the *local_variable_initializer* of a *local_variable_declaration* statement or a *for_initializer*, then its type is `K*`.
- Otherwise its type is `System.Span<K>`.

### Stackalloc Conversion

The *stackalloc conversion* is a new built-in implicit conversion from expression. When the type of a *stackalloc_initializer* is `K*`, there is an implicit *stackalloc conversion* from the *stackalloc_initializer* to the type `System.Span<K>`.

# Records

11/2/2020 • 13 minutes to read • Edit Online

This proposal tracks the specification for the C# 9 records feature, as agreed to by the C# language design team.

The syntax for a record is as follows:

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' identifier type_parameter_list?
      parameter_list? record_base? type_parameter_constraints_clause* record_body
    ;

record_base
    : ':' class_type argument_list?
    | ':' interface_type_list
    | ':' class_type argument_list? ',' interface_type_list
    ;

record_body
    : '{' class_member_declaration* '}' ';'?
    | ';'
    ;
```

Record types are reference types, similar to a class declaration. It is an error for a record to provide a `record_base` `argument_list` if the `record_declaration` does not contain a `parameter_list`. At most one partial type declaration of a partial record may provide a `parameter_list`.

Record parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

## Inheritance

Records cannot inherit from classes, unless the class is `object`, and classes cannot inherit from records. Records can inherit from other records.

## Members of a record type

In addition to the members declared in the record body, a record type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record body or an accessible concrete non-virtual member with a "matching" signature is inherited. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. It is an error for a member of a record to be named "Clone". It is an error for an instance field of a record to have an unsafe type.

The synthesized members are as follows:

**Equality members**

If the record is derived from `object`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
Type EqualityContract { get; };
```

The property is `private` if the record type is `sealed`. Otherwise, the property is `virtual` and `protected`. The property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not

`sealed`.

If the record type is derived from a base record type `Base`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
protected override Type EqualityContract { get; };
```

The property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared property doesn't override a property with this signature in the record type `Base` (for example, if the property is missing in the `Base`, or sealed, or not virtual, etc.). The synthesized property returns `typeof(R)` where `R` is the record type.

The record type implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R? other)` where `R` is the record type. The method is `public`, and the method is `virtual` unless the record type is `sealed`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

If `Equals(R? other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

```
public virtual bool Equals(R? other);
```

The synthesized `Equals(R?)` returns `true` if and only if each of the following are `true`:

- `other` is not `null`, and
- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.Equals(other)` (a non-virtual call to `public virtual bool Equals(Base? other)`); otherwise the value of `EqualityContract == other.EqualityContract`.

The record type includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
public static bool operator==(R? r1, R? r2)
    => (object)r1 == r2 || (r1?.Equals(r2) ?? false);
public static bool operator!=(R? r1, R? r2)
    => !(r1 == r2);
```

The `Equals` method called by the `==` operator is the `Equals(R? other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

If the record type is derived from a base record type `Base`, the record type includes a synthesized override equivalent to a method declared as follows:

```
public sealed override bool Equals(Base? other);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override a method with same signature in record type `Base` (for example, if the method is missing in the `Base`, or sealed, or not virtual, etc.). The synthesized override returns `Equals((object?)other)`.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override bool Equals(object? obj);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override `object.Equals(object? obj)` (for example, due to shadowing in intermediate base types, etc.). The synthesized override returns `Equals(other as R)` where `R` is the record type.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override int GetHashCode();
```

The method can be declared explicitly. It is an error if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.GetHashCode()` (for example, due to shadowing in intermediate base types, etc.).

A warning is reported if one of `Equals(R?)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of a deterministic function combining the following values:

- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.GetHashCode()`; otherwise the value of `System.Collections.Generic.EqualityComparer<System.Type>.Default.GetHashCode(EqualityContract)`.

For example, consider the following record types:

```
record R1(T1 P1);
record R2(T1 P1, T2 P2) : R1(P1);
record R3(T1 P1, T2 P2, T3 P3) : R2(P1, P2);
```

For those record types, the synthesized equality members would be something like:

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }
    protected virtual Type EqualityContract => typeof(R1);
    public override bool Equals(object? obj) => Equals(obj as R1);
    public virtual bool Equals(R1? other)
    {
        return !(other is null) &&
            EqualityContract == other.EqualityContract &&
            EqualityComparer<T1>.Default.Equals(P1, other.P1);
    }
    public static bool operator==(R1? r1, R1? r2)
        => (object)r1 == r2 || (r1?.Equals(r2) ?? false);
    public static bool operator!=(R1? r1, R1? r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),
            EqualityComparer<T1>.Default.GetHashCode(P1));
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    protected override Type EqualityContract => typeof(R2);
    public override bool Equals(object? obj) => Equals(obj as R2);
    public sealed override bool Equals(R1? other) => Equals((object?)other);
    public virtual bool Equals(R2? other)
    {
        return base.Equals((R1?)other) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R2? r1, R2? r2)
        => (object)r1 == r2 || (r1?.Equals(r2) ?? false);
    public static bool operator!=(R2? r1, R2? r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

class R3 : R2, IEquatable<R3>
{
    public T3 P3 { get; init; }
    protected override Type EqualityContract => typeof(R3);
    public override bool Equals(object? obj) => Equals(obj as R3);
    public sealed override bool Equals(R2? other) => Equals((object?)other);
    public virtual bool Equals(R3? other)
    {
        return base.Equals((R2?)other) &&
            EqualityComparer<T3>.Default.Equals(P3, other.P3);
    }
    public static bool operator==(R3? r1, R3? r2)
        => (object)r1 == r2 || (r1?.Equals(r2) ?? false);
    public static bool operator!=(R3? r1, R3? r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T3>.Default.GetHashCode(P3));
    }
}
```

### Copy and Clone members

A record type contains two copying members:

- A constructor taking a single argument of the record type. It is referred to as a "copy constructor".
- A synthesized public parameterless instance "clone" method with a compiler-reserved name

The purpose of the copy constructor is to copy the state from the parameter to the new instance being created. This constructor doesn't run any instance field/property initializers present in the record declaration. If the constructor is not explicitly declared, a constructor will be synthesized by the compiler. If the record is sealed, the constructor will be private, otherwise it will be protected. An explicitly declared copy constructor must be either public or protected, unless the record is sealed. The first thing the constructor must do, is to call a copy constructor of the base, or a parameter-less object constructor if the record inherits from object. An error is reported if a user-defined copy constructor uses an implicit or explicit constructor initializer that doesn't fulfill this requirement. After a base copy constructor is invoked, a synthesized copy constructor copies values for all instance fields implicitly or explicitly declared within the record type. The sole presence of a copy constructor, whether explicit or implicit, doesn't prevent an automatic addition of a default instance constructor.

If a virtual "clone" method is present in the base record, the synthesized "clone" method overrides it and the return type of the method is the current containing type if the "covariant returns" feature is supported and the override return type otherwise. An error is produced if the base record clone method is sealed. If a virtual "clone" method is not present in the base record, the return type of the clone method is the containing type and the method is virtual, unless the record is sealed or abstract. If the containing record is abstract, the synthesized clone method is also abstract. If the "clone" method is not abstract, it returns the result of a call to a copy constructor.

### Printing members: PrintMembers and ToString methods

If the record is derived from `object`, the record includes a synthesized method equivalent to a method declared as follows:

```
bool PrintMembers(System.Text.StringBuilder builder);
```

The method is `private` if the record type is `sealed`. Otherwise, the method is `virtual` and `protected`.

The method:

1. for each of the record's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value: `this.member`, separated with ", ",
2. return true if the record has printable members.

If the record type is derived from a base record `Base`, the record includes a synthesized override equivalent to a method declared as follows:

```
protected override bool PrintMembers(StringBuilder builder);
```

If the record has no printable members, the method calls the base `PrintMembers` method with one argument (its `builder` parameter) and returns the result.

Otherwise, the method:

1. calls the base `PrintMembers` method with one argument (its `builder` parameter),
2. if the `PrintMembers` method returned true, append ", " to the builder,
3. for each of the record's printable members, appends that member's name followed by " = " followed by the member's value: `this.member` (or `this.member.ToString()` for value types), separated with ", ",
4. return true.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

The record includes a synthesized method equivalent to a method declared as follows:

```
public override string ToString();
```

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.ToString()` (for example, due to shadowing in intermediate base types, etc.).

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record name to the builder, followed by " { ",
3. invokes the record's `PrintMembers` method giving it the builder, followed by " " if it returned true,
4. appends "}",
5. returns the builder's contents with `builder.ToString()`.

For example, consider the following record types:

```
record R1(T1 P1);
record R2(T1 P1, T2 P2, T3 P3) : R1(P1);
```

For those record types, the synthesized printing members would be something like:

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1); if P1 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    public T3 P3 { get; init; }

    protected override void PrintMembers(StringBuilder builder)
    {
        if (base.PrintMembers(builder))
            builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2); if P2 has a value type

        builder.Append(", ");

        builder.Append(nameof(P3));
        builder.Append(" = ");
        builder.Append(this.P3); // or builder.Append(this.P3); if P3 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R2));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}
```

## Positional record members

In addition to the above members, records with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

**Primary Constructor**

A record type has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type, and causes the implicitly declared default class constructor, if present, to be suppressed. It is an error to have a primary constructor and a constructor with the same signature already present in the class.

At runtime the primary constructor

1. executes the instance initializers appearing in the class-body
2. invokes the base class constructor with the arguments provided in the `record_base` clause, if present

If a record has a primary constructor, any user-defined constructor, except "copy constructor" must have an explicit `this` constructor initializer.

Parameters of the primary constructor as well as members of the record are in scope within the `argument_list` of the `record_base` clause and within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable, similar to how base calls and initializers work in ordinary constructors today.

A warning is produced if a parameter of the primary constructor is not read.

Expression variables declared in the `argument_list` are in scope within the `argument_list`. The same shadowing rules as within an argument list of a regular constructor initializer apply.

**Properties**

For each record parameter of a record type declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record:

- A public `get` and `init` auto-property is created (see separate `init` accessor specification). An inherited `abstract` property with matching type is overridden. It is an error if the inherited property does not have `public` overridable `get` and `init` accessors. The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record parameter.

**Deconstruct**

A positional record with at least one parameter synthesizes a public void-returning instance method called Deconstruct with an out parameter declaration for each parameter of the primary constructor declaration. Each parameter of the Deconstruct method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the Deconstruct method to the value from an instance member access to a member of the same name. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

## `with` expression

A `with` expression is a new expression using the following syntax.

```
with_expression
    : switch_expression
    | switch_expression 'with' '{' member_initializer_list? '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : identifier '=' expression
    ;
```

A `with` expression is not permitted as a statement.

A `with` expression allows for "non-destructive mutation", designed to produce a copy of the receiver expression with modifications in assignments in the `member_initializer_list`.

A valid `with` expression has a receiver with a non-void type. The receiver type must be a record.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to *identifier*, which must be an accessible instance field or property of the receiver's type.

First, receiver's "clone" method (specified above) is invoked and its result is converted to the receiver's type. Then, each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

# Top-level statements

11/2/2020 • 4 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype: Started
- [x] Implementation: Started
- [ ] Specification: Not Started

## Summary

Allow a sequence of *statements* to occur right before the *namespace_member_declaration*s of a *compilation_unit* (i.e. source file).

The semantics are that if such a sequence of *statements* is present, the following type declaration, modulo the actual type name and the method name, would be emitted:

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

See also https://github.com/dotnet/csharplang/issues/3117.

## Motivation

There's a certain amount of boilerplate surrounding even the simplest of programs, because of the need for an explicit `Main` method. This seems to get in the way of language learning and program clarity. The primary goal of the feature therefore is to allow C# programs without unnecessary boilerplate around them, for the sake of learners and the clarity of code.

## Detailed design

**Syntax**

The only additional syntax is allowing a sequence of *statement*s in a compilation unit, just before the *namespace_member_declaration*s:

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes? statement* namespace_member_declaration*
    ;
```

Only one *compilation_unit* is allowed to have *statement*s.

Example:

```
if (args.Length == 0
    || !int.TryParse(args[0], out int n)
    || n < 0) return;
Console.WriteLine(Fib(n).curr);

(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}
```

**Semantics**

If any top-level statements are present in any compilation unit of the program, the meaning is as if they were combined in the block body of a `Main` method of a `Program` class in the global namespace, as follows:

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

Note that the names "Program" and "Main" are used only for illustrations purposes, actual names used by compiler are implementation dependent and neither the type, nor the method can be referenced by name from source code.

The method is designated as the entry point of the program. Explicitly declared methods that by convention could be considered as an entry point candidates are ignored. A warning is reported when that happens. It is an error to specify `-main:<type>` compiler switch when there are top-level statements.

The entry point method always has one formal parameter, `string[] args`. The execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified. The 'args' parameter is in scope within top-level statements and is not in scope outside of them. Regular name conflict/shadowing rules apply.

Async operations are allowed in top-level statements to the degree they are allowed in statements within a regular async entry point method. However, they are not required, if `await` expressions and other async operations are omitted, no warning is produced.

The signature of the generated entry point method is determined based on operations used by the top level statements as follows:

| ASYNC-OPERATIONS\RETURN-WITH-EXPRESSION | PRESENT | ABSENT |
|---|---|---|
| **Present** | `static Task<int> Main(string[] args)` | `static Task Main(string[] args)` |
| **Absent** | `static int Main(string[] args)` | `static void Main(string[] args)` |

The example above would yield the following `$Main` method declaration:

```
static class $Program
{
    static void $Main(string[] args)
    {
        if (args.Length == 0
            || !int.TryParse(args[0], out int n)
            || n < 0) return;
        Console.WriteLine(Fib(n).curr);

        (int curr, int prev) Fib(int i)
        {
            if (i == 0) return (1, 0);
            var (curr, prev) = Fib(i - 1);
            return (curr + prev, curr);
        }
    }
}
```

At the same time an example like this:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
```

would yield:

```
static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}
```

An example like this:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;
```

would yield:

```
static class $Program
{
    static async Task<int> $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
        return 0;
    }
}
```

And an example like this:

```
System.Console.WriteLine("Hi!");
return 2;
```

would yield:

```
static class $Program
{
    static int $Main(string[] args)
    {
        System.Console.WriteLine("Hi!");
        return 2;
    }
}
```

**Scope of top-level local variables and local functions**

Even though top-level local variables and functions are "wrapped" into the generated entry point method, they should still be in scope throughout the program in every compilation unit. For the purpose of simple-name evaluation, once the global namespace is reached:

- First, an attempt is made to evaluate the name within the generated entry point method and only if this attempt fails
- The "regular" evaluation within the global namespace declaration is performed.

This could lead to name shadowing of namespaces and types declared within the global namespace as well as to shadowing of imported names.

If the simple name evaluation occurs outside of the top-level statements and the evaluation yields a top-level local variable or function, that should lead to an error.

In this way we protect our future ability to better address "Top-level functions" (scenario 2 in https://github.com/dotnet/csharplang/issues/3117), and are able to give useful diagnostics to users who mistakenly believe them to be supported.

# Nullable Reference Types Specification

11/24/2020 • 18 minutes to read • Edit Online

*This is a work in progress - several parts are missing or incomplete.*

This feature adds two new kinds of nullable types (nullable reference types and nullable generic types) to the existing nullable value types, and introduces a static flow analysis for purpose of null-safety.

## Syntax

### Nullable reference types and nullable type parameters

Nullable reference types and nullable type parameters have the same syntax `T?` as the short form of nullable value types, but do not have a corresponding long form.

For the purposes of the specification, the current `nullable_type` production is renamed to `nullable_value_type`, and `nullable_reference_type` and `nullable_type_parameter` productions are added:

```
type
    : value_type
    | reference_type
    | nullable_type_parameter
    | type_parameter
    | type_unsafe
    ;

reference_type
    : ...
    | nullable_reference_type
    ;

nullable_reference_type
    : non_nullable_reference_type '?'
    ;

non_nullable_reference_type
    : reference_type
    ;

nullable_type_parameter
    : non_nullable_non_value_type_parameter '?'
    ;

non_nullable_non_value_type_parameter
    : type_parameter
    ;
```

The `non_nullable_reference_type` in a `nullable_reference_type` must be a nonnullable reference type (class, interface, delegate or array).

The `non_nullable_non_value_type_parameter` in `nullable_type_parameter` must be a type parameter that isn't constrained to be a value type.

Nullable reference types and nullable type parameters cannot occur in the following positions:

- as a base class or interface
- as the receiver of a `member_access`

- as the `type` in an `object_creation_expression`
- as the `delegate_type` in a `delegate_creation_expression`
- as the `type` in an `is_expression`, a `catch_clause` or a `type_pattern`
- as the `interface` in a fully qualified interface member name

A warning is given on a `nullable_reference_type` and `nullable_type_parameter` in a *disabled* nullable annotation context.

#### `class` and `class?` constraint

The `class` constraint has a nullable counterpart `class?`:

```
primary_constraint
    : ...
    | 'class' '?'
    ;
```

A type parameter constrained with `class` (in an *enabled* annotation context) must be instantiated with a nonnullable reference type.

A type parameter constrained with `class?` (or `class` in a *disabled* annotation context) may either be instantiated with a nullable or nonnullable reference type.

A warning is given on a `class?` constraint in a *disabled* annotation context.

#### `notnull` constraint

A type parameter constrained with `notnull` may not be a nullable type (nullable value type, nullable reference type or nullable type parameter).

```
primary_constraint
    : ...
    | 'notnull'
    ;
```

#### `default` constraint

The `default` constraint can be used on a method override or explicit implementation to disambiguate `T?` meaning "nullable type parameter" from "nullable value type" ( `Nullable<T>` ). Lacking the `default` constraint a `T?` syntax in an override or explicit implementation will be interpreted as `Nullable<T>`

See https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint

### The null-forgiving operator

The post-fix `!` operator is called the null-forgiving operator. It can be applied on a *primary_expression* or within a *null_conditional_expression*:

```
primary_expression
    : ...
    | null_forgiving_expression
    ;

null_forgiving_expression
    : primary_expression '!'
    ;

null_conditional_expression
    : primary_expression null_conditional_operations_no_suppression suppression?
    ;

null_conditional_operations_no_suppression
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;

null_conditional_operations
    : null_conditional_operations_no_suppression suppression?
    ;

suppression
    : '!'
    ;
```

For example:

```
var v = expr!;
expr!.M();
_ = a?.b!.c;
```

The `primary_expression` and `null_conditional_operations_no_suppression` must be of a nullable type.

The postfix `!` operator has no runtime effect - it evaluates to the result of the underlying expression. Its only role is to change the null state of the expression to "not null", and to limit warnings given on its use.

**Nullable compiler directives**

`#nullable` directives control the nullable annotation and warning contexts.

```
pp_directive
    : ...
    | pp_nullable
    ;

pp_nullable
    : whitespace? '#' whitespace? 'nullable' whitespace nullable_action (whitespace nullable_target)?
pp_new_line
    ;

nullable_action
    : 'disable'
    | 'enable'
    | 'restore'
    ;

nullable_target
    : 'warnings'
    | 'annotations'
    ;
```

`#pragma warning` directives are expanded to allow changing the nullable warning context:

```
pragma_warning_body
    : ...
    | 'warning' whitespace warning_action whitespace 'nullable'
    ;
```

For example:

```
#pragma warning disable nullable
```

## Nullable contexts

Every line of source code has a *nullable annotation context* and a *nullable warning context*. These control whether nullable annotations have effect, and whether nullability warnings are given. The annotation context of a given line is either *disabled* or *enabled*. The warning context of a given line is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code), or anywhere within a source file via `#nullable` pre-processor directives. If no project level settings are provided the default is for both contexts to be *disabled*.

The `#nullable` directive controls the annotation and warning contexts within the source text, and take precedence over the project-level settings.

A directive sets the context(s) it controls for subsequent lines of code, until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable`: Sets the nullable annotation and warning contexts to *disabled*
- `#nullable enable`: Sets the nullable annotation and warning contexts to *enabled*
- `#nullable restore`: Restores the nullable annotation and warning contexts to project settings
- `#nullable disable annotations`: Sets the nullable annotation context to *disabled*
- `#nullable enable annotations`: Sets the nullable annotation context to *enabled*
- `#nullable restore annotations`: Restores the nullable annotation context to project settings

- `#nullable disable warnings`: Sets the nullable warning context to *disabled*
- `#nullable enable warnings`: Sets the nullable warning context to *enabled*
- `#nullable restore warnings`: Restores the nullable warning context to project settings

# Nullability of types

A given type can have one of three nullabilities: *oblivious*, *nonnullable*, and *nullable*.

*Nonnullable* types may cause warnings if a potential `null` value is assigned to them. *Oblivious* and *nullable* types, however, are "*null-assignable*" and can have `null` values assigned to them without warnings.

Values of *oblivious* and *nonnullable* types can be dereferenced or assigned without warnings. Values of *nullable* types, however, are "*null-yielding*" and may cause warnings when dereferenced or assigned without proper null checking.

The *default null state* of a null-yielding type is "maybe null" or "maybe default". The default null state of a non-null-yielding type is "not null".

The kind of type and the nullable annotation context it occurs in determine its nullability:

- A nonnullable value type `S` is always *nonnullable*
- A nullable value type `S?` is always *nullable*
- An unannotated reference type `C` in a *disabled* annotation context is *oblivious*
- An unannotated reference type `C` in an *enabled* annotation context is *nonnullable*
- A nullable reference type `C?` is *nullable* (but a warning may be yielded in a *disabled* annotation context)

Type parameters additionally take their constraints into account:

- A type parameter `T` where all constraints (if any) are either nullable types or the `class?` constraint is *nullable*
- A type parameter `T` where at least one constraint is either *oblivious* or *nonnullable* or one of the `struct` or `class` or `notnull` constraints is
  - *oblivious* in a *disabled* annotation context
  - *nonnullable* in an *enabled* annotation context
- A nullable type parameter `T?` is *nullable*, but a warning is yielded in a *disabled* annotation context if `T` isn't a value type

**Oblivious vs nonnullable**

A `type` is deemed to occur in a given annotation context when the last token of the type is within that context.

Whether a given reference type `C` in source code is interpreted as oblivious or nonnullable depends on the annotation context of that source code. But once established, it is considered part of that type, and "travels with it" e.g. during substitution of generic type arguments. It is as if there is an annotation like `?` on the type, but invisible.

# Constraints

Nullable reference types can be used as generic constraints.

`class?` is a new constraint denoting "possibly nullable reference type", whereas `class` in an *enabled* annotation context denotes "nonnullable reference type".

`default` is a new constraint denoting a type parameter that isn't known to be a reference or value type. It can only be used on overridden and explicitly implemented methods. With this constraint, `T?` means a nullable type parameter, as opposed to being a shorthand for `Nullable<T>`.

`notnull` is a new constraint denoting a type parameter that is nonnullable.

The nullability of a type argument or of a constraint does not impact whether the type satisfies the constraint, except where that is already the case today (nullable value types do not satisfy the `struct` constraint). However, if the type argument does not satisfy the nullability requirements of the constraint, a warning may be given.

# Null state and null tracking

Every expression in a given source location has a *null state*, which indicated whether it is believed to potentially evaluate to null. The null state is either "not null", "maybe null", or "maybe default". The null state is used to determine whether a warning should be given about null-unsafe conversions and dereferences.

The distinction between "maybe null" and "maybe default" is subtle and applies to type parameters. The distinction is that a type parameter `T` which has the state "maybe null" means the value is in the domain of legal values for `T` however that legal value may include `null`. Where as a "maybe default" means that the value may be outside the legal domain of values for `T`.

Example:

```
// The value `t` here has the state "maybe null". It's possible for `T` to be instantiated
// with `string?` in which case `null` would be within the domain of legal values here. The
// assumption though is the value provided here is within the legal values of `T`. Hence
// if `T` is `string` then `null` will not be a value, just as we assume that `null` is not
// provided for a normal `string` parameter
void M<T>(T t)
{
    // There is no guarantee that default(T) is within the legal values for T hence the
    // state *must* be "maybe-default" and hence `local` must be `T?`
    T? local = default(T);
}
```

### Null tracking for variables

For certain expressions denoting variables, fields or properties, the null state is tracked between occurrences, based on assignments to them, tests performed on them and the control flow between them. This is similar to how definite assignment is tracked for variables. The tracked expressions are the ones of the following form:

```
tracked_expression
    : simple_name
    | this
    | base
    | tracked_expression '.' identifier
    ;
```

Where the identifiers denote fields or properties.

The null state for tracked variables is "not null" in unreachable code. This follows other decisions around unreachable code like considering all locals to be definitely assigned.

*Describe null state transitions similar to definite assignment*

### Null state for expressions

The null state of an expression is derived from its form and type, and from the null state of variables involved in it.

### Literals

The null state of a `null` literal depends on the target type of the expression. If the target type is a type parameter constrained to a reference type then it's "maybe default". Otherwise it is "maybe null".

The null state of a `default` literal depends on the target type of the `default` literal. A `default` literal with target type `T` has the same null state as the `default(T)` expression.

The null state of any other literal is "not null".

### Simple names

If a `simple_name` is not classified as a value, its null state is "not null". Otherwise it is a tracked expression, and its null state is its tracked null state at this source location.

### Member access

If a `member_access` is not classified as a value, its null state is "not null". Otherwise, if it is a tracked expression, its null state is its tracked null state at this source location. Otherwise its null state is the default null state of its type.

```
var person = new Person();

// The receiver is a tracked expression hence the member_access of the property
// is tracked as well
if (person.FirstName is not null)
{
    Use(person.FirstName);
}

// The return of an invocation is not a tracked expression hence the member_access
// of the return is also not tracked
if (GetAnonymous().FirstName is not null)
{
    // Warning: Cannot convert null literal to non-nullable reference type.
    Use(GetAnonymous().FirstName);
}

void Use(string s)
{
    // ...
}

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    private static Person s_anonymous = new Person();
    public static Person GetAnonymous() => s_anonymous;
}
```

### Invocation expressions

If an `invocation_expression` invokes a member that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

The null state of an `invocation_expression` is not tracked by the compiler.

```
// The result of an invocation_expression is not tracked
if (GetText() is not null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    string s = GetText();
    // Warning: Dereference of a possibly null reference.
    Use(s);
}

// Nullable friendly pattern
if (GetText() is string s)
{
    Use(s);
}

string? GetText() => ...
Use(string s) {  }
```

#### Element access

If an `element_access` invokes an indexer that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

```
object?[] array = ...;
if (array[0] != null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    object o = array[0];
    // Warning: Dereference of a possibly null reference.
    Console.WriteLine(o.ToString());
}

// Nullable friendly pattern
if (array[0] is {} o)
{
    Console.WriteLine(o.ToString());
}
```

#### Base access

If `B` denotes the base type of the enclosing type, `base.I` has the same null state as `((B)this).I` and `base[E]` has the same null state as `((B)this)[E]`.

#### Default expressions

`default(T)` has the null state based on the properties of the type `T`:

- If the type is a *nonnullable* type then it has the null state "not null"
- Else if the type is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### Null-conditional expressions ?.

A `null_conditional_expression` has the null state based on the expression type. Note that this refers to the type of the `null_conditional_expression`, not the original type of the member being invoked:

- If the type is a *nullable* value type then it has the null state "maybe null"
- Else if the type is a *nullable* type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### Cast expressions

If a cast expression `(T)E` invokes a user-defined conversion, then the null state of the expression is the default null state for the type of the user-defined conversion. Otherwise:

- If `T` is a *nonnullable* value type then `T` has the null state "not null"
- Else if `T` is a *nullable* value type then `T` has the null state "maybe null"
- Else if `T` is a *nullable* type in the form `U?` where `U` is a type parameter then `T` has the null state "maybe default"
- Else if `T` is a *nullable* type, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe null"
- Else if `T` is a type parameter, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe default"
- Else `T` has the same null state as `E`

#### Unary and binary operators

If a unary or binary operator invokes an user-defined operator then the null state of the expression is the default null state for the type of the user-defined operator. Otherwise it is the null state of the expression.

*Something special to do for binary `+` over strings and delegates?*

#### Await expressions

The null state of `await E` is the default null state of its type.

#### The `as` operator

The null state of an `E as T` expression depends first on properties of the type `T`. If the type of `T` is *nonnullable* then the null state is "not null". Otherwise the null state depends on the conversion from the type of `E` to type `T`:

- If the conversion is an identity, boxing, implicit reference, or implicit nullable conversion, then the null state is the null state of `E`
- Else if `T` is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

#### The null-coalescing operator

The null state of `E1 ?? E2` is the null state of `E2`

#### The conditional operator

The null state of `E1 ? E2 : E3` is based on the null state of `E2` and `E3`:

- If both are "not null" then the null state is "not null"
- Else if either is "maybe default" then the null state is "maybe default"
- Else the null state is "not null"

#### Query expressions

The null state of a query expression is the default null state of its type.

*Additional work needed here*

#### Assignment operators

`E1 = E2` and `E1 op= E2` have the same null state as `E2` after any implicit conversions have been applied.

#### Expressions that propagate null state

`(E)`, `checked(E)` and `unchecked(E)` all have the same null state as `E`.

#### Expressions that are never null

The null state of the following expression forms is always "not null":

- `this` access
- interpolated strings
- `new` expressions (object, delegate, anonymous object and array creation expressions)
- `typeof` expressions
- `nameof` expressions
- anonymous functions (anonymous methods and lambda expressions)
- null-forgiving expressions
- `is` expressions

#### Nested functions

Nested functions (lambdas and local functions) are treated like methods, except in regards to their captured variables. The initial state of a captured variable inside a lambda or local function is the intersection of the nullable state of the variable at all the "uses" of that nested function or lambda. A use of a local function is either a call to that function, or where it is converted to a delegate. A use of a lambda is the point at which it is defined in source.

## Type inference

#### nullable implicitly typed local variables

`var` infers an annotated type for reference types, and type parameters that aren't constrained to be a value type. For instance:

- in `var s = "";` the `var` is inferred as `string?`.
- in `var t = new T();` with an unconstrained `T` the `var` is inferred as `T?`.

#### Generic type inference

Generic type inference is enhanced to help decide whether inferred reference types should be nullable or not. This is a best effort. It may yield warnings regarding nullability constraints, and may lead to nullable warnings when the inferred types of the selected overload are applied to the arguments.

#### The first phase

Nullable reference types flow into the bounds from the initial expressions, as described below. In addition, two new kinds of bounds, namely `null` and `default` are introduced. Their purpose is to carry through occurrences of `null` or `default` in the input expressions, which may cause an inferred type to be nullable, even when it otherwise wouldn't. This works even for nullable *value* types, which are enhanced to pick up "nullness" in the inference process.

The determination of what bounds to add in the first phase are enhanced as follows:

If an argument `Ei` has a reference type, the type `U` used for inference depends on the null state of `Ei` as well as its declared type:

- If the declared type is a nonnullable reference type `U0` or a nullable reference type `U0?` then
  - if the null state of `Ei` is "not null" then `U` is `U0`
  - if the null state of `Ei` is "maybe null" then `U` is `U0?`
- Otherwise if `Ei` has a declared type, `U` is that type
- Otherwise if `Ei` is `null` then `U` is the special bound `null`
- Otherwise if `Ei` is `default` then `U` is the special bound `default`
- Otherwise no inference is made.

#### Exact, upper-bound and lower-bound inferences

In inferences *from* the type `U` *to* the type `V`, if `V` is a nullable reference type `V0?`, then `V0` is used instead of `V`

in the following clauses.

- If `V` is one of the unfixed type variables, `U` is added as an exact, upper or lower bound as before
- Otherwise, if `U` is `null` or `default`, no inference is made
- Otherwise, if `U` is a nullable reference type `U0?`, then `U0` is used instead of `U` in the subsequent clauses.

The essence is that nullability that pertains directly to one of the unfixed type variables is preserved into its bounds. For the inferences that recurse further into the source and target types, on the other hand, nullability is ignored. It may or may not match, but if it doesn't, a warning will be issued later if the overload is chosen and applied.

**Fixing**

The spec currently does not do a good job of describing what happens when multiple bounds are identity convertible to each other, but are different. This may happen between `object` and `dynamic`, between tuple types that differ only in element names, between types constructed thereof and now also between `C` and `C?` for reference types.

In addition we need to propagate "nullness" from the input expressions to the result type.

To handle these we add more phases to fixing, which is now:

1. Gather all the types in all the bounds as candidates, removing `?` from all that are nullable reference types
2. Eliminate candidates based on requirements of exact, lower and upper bounds (keeping `null` and `default` bounds)
3. Eliminate candidates that do not have an implicit conversion to all the other candidates
4. If the remaining candidates do not all have identity conversions to one another, then type inference fails
5. *Merge* the remaining candidates as described below
6. If the resulting candidate is a reference type or a nonnullable value type and *all* of the exact bounds or *any* of the lower bounds are nullable value types, nullable reference types, `null` or `default`, then `?` is added to the resulting candidate, making it a nullable value type or reference type.

*Merging* is described between two candidate types. It is transitive and commutative, so the candidates can be merged in any order with the same ultimate result. It is undefined if the two candidate types are not identity convertible to each other.

The *Merge* function takes two candidate types and a direction (*+* or *-*):

- *Merge*(`T`, `T`, *d*) = T
- *Merge*(`S`, `T?`, *+*) = *Merge*(`S?`, `T`, *+*) = *Merge*(`S`, `T`, *+*)`?`
- *Merge*(`S`, `T?`, *-*) = *Merge*(`S?`, `T`, *-*) = *Merge*(`S`, `T`, *-*)
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *+*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *+* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *-* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *-*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *-* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *+* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`(S1 s1,..., Sn sn)`, `(T1 t1,..., Tn tn)`, *d*) = `(` *Merge*(`S1`, `T1`, *d*)`n1,...,` *Merge*(`Sn`, `Tn`, *d*)`nn)`, *where*
  - `ni` is absent if `si` and `ti` differ, or if both are absent
  - `ni` is `si` if `si` and `ti` are the same
- *Merge*(`object`, `dynamic`) = *Merge*(`dynamic`, `object`) = `dynamic`

# Warnings

**Potential null assignment**

**Potential null dereference**

**Constraint nullability mismatch**

**Nullable types in disabled annotation context**

**Override and implementation nullability mismatch**

## Attributes for special null behavior

# Pattern-matching changes for C# 9.0

11/2/2020 • 11 minutes to read • Edit Online

We are considering a small handful of enhancements to pattern-matching for C# 9.0 that have natural synergy and work well to address a number of common programming problems:

- https://github.com/dotnet/csharplang/issues/2925 Type patterns
- https://github.com/dotnet/csharplang/issues/1350 Parenthesized patterns to enforce or emphasize precedence of the new combinators
- https://github.com/dotnet/csharplang/issues/1350 Conjunctive `and` patterns that require both of two different patterns to match;
- https://github.com/dotnet/csharplang/issues/1350 Disjunctive `or` patterns that require either of two different patterns to match;
- https://github.com/dotnet/csharplang/issues/1350 Negated `not` patterns that require a given pattern *not* to match; and
- https://github.com/dotnet/csharplang/issues/812 Relational patterns that require the input value to be less than, less than or equal to, etc a given constant.

## Parenthesized Patterns

Parenthesized patterns permit the programmer to put parentheses around any pattern. This is not so useful with the existing patterns in C# 8.0, however the new pattern combinators introduce a precedence that the programmer may want to override.

```
primary_pattern
    : parenthesized_pattern
    | // all of the existing forms
    ;
parenthesized_pattern
    : '(' pattern ')'
    ;
```

## Type Patterns

We permit a type as a pattern:

```
primary_pattern
    : type-pattern
    | // all of the existing forms
    ;
type_pattern
    : type
    ;
```

This retcons the existing *is-type-expression* to be an *is-pattern-expression* in which the pattern is a *type-pattern*, though we would not change the syntax tree produced by the compiler.

One subtle implementation issue is that this grammar is ambiguous. A string such as `a.b` can be parsed either as a qualified name (in a type context) or a dotted expression (in an expression context). The compiler is already capable of treating a qualified name the same as a dotted expression in order to handle something like `e is Color.Red`. The

compiler's semantic analysis would be further extended to be capable of binding a (syntactic) constant pattern (e.g. a dotted expression) as a type in order to treat it as a bound type pattern in order to support this construct.

After this change, you would be able to write

```
void M(object o1, object o2)
{
    var t = (o1, o2);
    if (t is (int, string)) {} // test if o1 is an int and o2 is a string
    switch (o1) {
        case int: break; // test if o1 is an int
        case System.String: break; // test if o1 is a string
    }
}
```

## Relational Patterns

Relational patterns permit the programmer to express that an input value must satisfy a relational constraint when compared to a constant value:

```
    public static LifeStage LifeStageAtAge(int age) => age switch
    {
        < 0 =>  LifeStage.Prenatal,
        < 2 =>  LifeStage.Infant,
        < 4 =>  LifeStage.Toddler,
        < 6 =>  LifeStage.EarlyChild,
        < 12 => LifeStage.MiddleChild,
        < 20 => LifeStage.Adolescent,
        < 40 => LifeStage.EarlyAdult,
        < 65 => LifeStage.MiddleAdult,
        _ =>    LifeStage.LateAdult,
    };
```

Relational patterns support the relational operators `<`, `<=`, `>`, and `>=` on all of the built-in types that support such binary relational operators with two operands of the same type in an expression. Specifically, we support all of these relational patterns for `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `nint`, and `nuint`.

```
primary_pattern
    : relational_pattern
    ;
relational_pattern
    : '<' relational_expression
    | '<=' relational_expression
    | '>' relational_expression
    | '>=' relational_expression
    ;
```

The expression is required to evaluate to a constant value. It is an error if that constant value is `double.NaN` or `float.NaN`. It is an error if the expression is a null constant.

When the input is a type for which a suitable built-in binary relational operator is defined that is applicable with the input as its left operand and the given constant as its right operand, the evaluation of that operator is taken as the meaning of the relational pattern. Otherwise we convert the input to the type of the expression using an explicit nullable or unboxing conversion. It is a compile-time error if no such conversion exists. The pattern is considered not to match if the conversion fails. If the conversion succeeds then the result of the pattern-matching operation is the result of evaluating the expression `e OP v` where `e` is the converted input, `OP` is the relational operator, and

`v` is the constant expression.

## Pattern Combinators

Pattern *combinators* permit matching both of two different patterns using `and` (this can be extended to any number of patterns by the repeated use of `and`), either of two different patterns using `or` (ditto), or the *negation* of a pattern using `not`.

A common use of a combinator will be the idiom

```
if (e is not null) ...
```

More readable than the current idiom `e is object`, this pattern clearly expresses that one is checking for a non-null value.

The `and` and `or` combinators will be useful for testing ranges of values

```
bool IsLetter(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

This example illustrates that `and` will have a higher parsing priority (i.e. will bind more closely) than `or`. The programmer can use the *parenthesized pattern* to make the precedence explicit:

```
bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

Like all patterns, these combinators can be used in any context in which a pattern is expected, including nested patterns, the *is-pattern-expression*, the *switch-expression*, and the pattern of a switch statement's case label.

```
pattern
    : disjunctive_pattern
    ;
disjunctive_pattern
    : disjunctive_pattern 'or' conjunctive_pattern
    | conjunctive_pattern
    ;
conjunctive_pattern
    : conjunctive_pattern 'and' negated_pattern
    | negated_pattern
    ;
negated_pattern
    : 'not' negated_pattern
    | primary_pattern
    ;
primary_pattern
    : // all of the patterns forms previously defined
    ;
```

## Change to 7.5.4.2 Grammar Ambiguities

Due to the introduction of the *type pattern*, it is possible for a generic type to appear before the token `=>`. We therefore add `=>` to the set of tokens listed in *7.5.4.2 Grammar Ambiguities* to permit disambiguation of the `<` that begins the type argument list. See also https://github.com/dotnet/roslyn/issues/47614.

## Open Issues with Proposed Changes

**Syntax for relational operators**

Are `and`, `or`, and `not` some kind of contextual keyword? If so, is there a breaking change (e.g. compared to their use as a designator in a *declaration-pattern*).

**Semantics (e.g. type) for relational operators**

We expect to support all of the primitive types that can be compared in an expression using a relational operator. The meaning in simple cases is clear

```
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

But when the input is not such a primitive type, what type do we attempt to convert it to?

```
bool IsValidPercentage(object x) => x is >= 0 and <= 100;
```

We have proposed that when the input type is already a comparable primitive, that is the type of the comparison. However, when the input is not a comparable primitive, we treat the relational as including an implicit type test to the type of the constant on the right-hand-side of the relational. If the programmer intends to support more than one input type, that must be done explicitly:

```
bool IsValidPercentage(object x) => x is
    >= 0 and <= 100 or    // integer tests
    >= 0F and <= 100F or  // float tests
    >= 0D and <= 100D;    // double tests
```

**Flowing type information from the left to the right of `and`**

It has been suggested that when you write an `and` combinator, type information learned on the left about the top-level type could flow to the right. For example

```
bool isSmallByte(object o) => o is byte and < 100;
```

Here, the *input type* to the second pattern is narrowed by the *type narrowing* requirements of left of the `and`. We would define type narrowing semantics for all patterns as follows. The *narrowed type* of a pattern `P` is defined as follows:

1. If `P` is a type pattern, the *narrowed type* is the type of the type pattern's type.
2. If `P` is a declaration pattern, the *narrowed type* is the type of the declaration pattern's type.
3. If `P` is a recursive pattern that gives an explicit type, the *narrowed type* is that type.
4. If `P` is matched via the rules for `ITuple`, the *narrowed type* is the type `System.Runtime.CompilerServices.ITuple`.
5. If `P` is a constant pattern where the constant is not the null constant and where the expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
6. If `P` is a relational pattern where the constant expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
7. If `P` is an `or` pattern, the *narrowed type* is the common type of the *narrowed type* of the subpatterns if such a common type exists. For this purpose, the common type algorithm considers only identity, boxing, and implicit reference conversions, and it considers all subpatterns of a sequence of `or` patterns (ignoring parenthesized patterns).
8. If `P` is an `and` pattern, the *narrowed type* is the *narrowed type* of the right pattern. Moreover, the *narrowed type* of the left pattern is the *input type* of the right pattern.
9. Otherwise the *narrowed type* of `P` is `P`'s input type.

**Variable definitions and definite assignment**

The addition of `or` and `not` patterns creates some interesting new problems around pattern variables and definite assignment. Since variables can normally be declared at most once, it would seem any pattern variable declared on one side of an `or` pattern would not be definitely assigned when the pattern matches. Similarly, a variable declared inside a `not` pattern would not be expected to be definitely assigned when the pattern matches. The simplest way to address this is to forbid declaring pattern variables in these contexts. However, this may be too restrictive. There are other approaches to consider.

One scenario that is worth considering is this

```
if (e is not int i) return;
M(i); // is i definitely assigned here?
```

This does not work today because, for an *is-pattern-expression*, the pattern variables are considered *definitely assigned* only where the *is-pattern-expression* is true ("definitely assigned when true").

Supporting this would be simpler (from the programmer's perspective) than also adding support for a negated-condition `if` statement. Even if we add such support, programmers would wonder why the above snippet does not work. On the other hand, the same scenario in a `switch` makes less sense, as there is no corresponding point in the program where *definitely assigned when false* would be meaningful. Would we permit this in an *is-pattern-expression* but not in other contexts where patterns are permitted? That seems irregular.

Related to this is the problem of definite assignment in a *disjunctive-pattern*.

```
if (e is 0 or int i)
{
    M(i); // is i definitely assigned here?
}
```

We would only expect `i` to be definitely assigned when the input is not zero. But since we don't know whether the input is zero or not inside the block, `i` is not definitely assigned. However, what if we permit `i` to be declared in different mutually exclusive patterns?

```
if ((e1, e2) is (0, int i) or (int i, 0))
{
    M(i);
}
```

Here, the variable `i` is definitely assigned inside the block, and takes it value from the other element of the tuple when a zero element is found.

It has also been suggested to permit variables to be (multiply) defined in every case of a case block:

```
    case (0, int x):
    case (int x, 0):
        Console.WriteLine(x);
```

To make any of this work, we would have to carefully define where such multiple definitions are permitted and under what conditions such a variable is considered definitely assigned.

Should we elect to defer such work until later (which I advise), we could say in C# 9

- beneath a `not` or `or`, pattern variables may not be declared.

Then, we would have time to develop some experience that would provide insight into the possible value of relaxing that later.

**Diagnostics, subsumption, and exhaustiveness**

These new pattern forms introduce many new opportunities for diagnosable programmer error. We will need to decide what kinds of errors we will diagnose, and how to do so. Here are some examples:

```
case >= 0 and <= 100D:
```

This case can never match (because the input cannot be both an `int` and a `double`). We already have an error when we detect a case that can never match, but its wording ("The switch case has already been handled by a previous case" and "The pattern has already been handled by a previous arm of the switch expression") may be misleading in new scenarios. We may have to modify the wording to just say that the pattern will never match the input.

```
case 1 and 2:
```

Similarly, this would be an error because a value cannot be both `1` and `2`.

```
case 1 or 2 or 3 or 1:
```

This case is possible to match, but the `or 1` at the end adds no meaning to the pattern. I suggest we should aim to produce an error whenever some conjunct or disjunct of a compound pattern does not either define a pattern variable or affect the set of matched values.

```
case < 2: break;
case 0 or 1 or 2 or 3 or 4 or 5: break;
```

Here, `0 or 1 or` adds nothing to the second case, as those values would have been handled by the first case. This too deserves an error.

```
byte b = ...;
int x = b switch { <100 => 0, 100 => 1, 101 => 2, >101 => 3 };
```

A switch expression such as this should be considered *exhaustive* (it handles all possible input values).

In C# 8.0, a switch expression with an input of type `byte` is only considered exhaustive if it contains a final arm whose pattern matches everything (a *discard-pattern* or *var-pattern*). Even a switch expression that has an arm for every distinct `byte` value is not considered exhaustive in C# 8. In order to properly handle exhaustiveness of relational patterns, we will have to handle this case too. This will technically be a breaking change, but no user is likely to notice.

# Init Only Setters

11/24/2020 • 15 minutes to read • Edit Online

## Summary

This proposal adds the concept of init only properties and indexers to C#. These properties and indexers can be set at the point of object creation but become effectively `get` only once object creation has completed. This allows for a much more flexible immutable model in C#.

## Motivation

The underlying mechanisms for building immutable data in C# haven't changed since 1.0. They remain:

1. Declaring fields as `readonly`.
2. Declaring properties that contain only a `get` accessor.

These mechanisms are effective at allowing the construction of immutable data but they do so by adding cost to the boilerplate code of types and opting such types out of features like object and collection initializers. This means developers must choose between ease of use and immutability.

A simple immutable object like `Point` requires twice as much boiler plate code to support construction as it does to declare the type. The bigger the type the bigger the cost of this boiler plate:

```
struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

The `init` accessor makes immutable objects more flexible by allowing the caller to mutate the members during the act of construction. That means the object's immutable properties can participate in object initializers and thus removes the need for all constructor boilerplate in the type. The `Point` type is now simply:

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}
```

The consumer can then use object initializers to create the object

```
var p = new Point() { X = 42, Y = 13 };
```

## Detailed Design

**init accessors**

An init only property (or indexer) is declared by using the `init` accessor in place of the `set` accessor:

```
class Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

An instance property containing an `init` accessor is considered settable in the following circumstances, except when in a local function or lambda:

- During an object initializer
- During a `with` expression initializer
- Inside an instance constructor of the containing or derived type, on `this` or `base`
- Inside the `init` accessor of any property, on `this` or `base`
- Inside attribute usages with named parameters

The times above in which the `init` accessors are settable are collectively referred to in this document as the construction phase of the object.

This means the `Student` class can be used in the following ways:

```
var s = new Student()
{
    FirstName = "Jared",
    LastName = "Parosns",
};
s.LastName = "Parsons"; // Error: LastName is not settable
```

The rules around when `init` accessors are settable extend across type hierarchies. If the member is accessible and the object is known to be in the construction phase then the member is settable. That specifically allows for the following:

```
class Base
{
    public bool Value { get; init; }
}

class Derived : Base
{
    Derived()
    {
        // Not allowed with get only properties but allowed with init
        Value = true;
    }
}

class Consumption
{
    void Example()
    {
        var d = new Derived() { Value = true; };
    }
}
```

At the point an `init` accessor is invoked, the instance is known to be in the open construction phase. Hence an

`init` accessor is allowed to take the following actions in addition to what a normal `set` accessor can do:

1. Call other `init` accessors available through `this` or `base`
2. Assign `readonly` fields declared on the same type through `this`

```
class Complex
{
    readonly int Field1;
    int Field2;
    int Prop1 { get; init ; }
    int Prop2
    {
        get => 42;
        init
        {
            Field1 = 13; // okay
            Field2 = 13; // okay
            Prop1 = 13; // okay
        }
    }
}
```

The ability to assign `readonly` fields from an `init` accessor is limited to those fields declared on the same type as the accessor. It cannot be used to assign `readonly` fields in a base type. This rule ensures that type authors remain in control over the mutability behavior of their type. Developers who do not wish to utilize `init` cannot be impacted from other types choosing to do so:

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
    internal readonly int DerivedField;
    internal int DerivedProperty
    {
        get => DerivedField;
        init
        {
            DerivedField = 42;  // Okay
            Property = 0;       // Okay
            Field = 13;         // Error Field is readonly
        }
    }

    public Derived()
    {
        Property = 42;  // Okay
        Field = 13;     // Error Field is readonly
    }
}
```

When `init` is used in a virtual property then all the overrides must also be marked as `init`. Likewise it is not possible to override a simple `set` with `init`.

```
class Base
{
    public virtual int Property { get; init; }
}

class C1 : Base
{
    public override int Property { get; init; }
}

class C2 : Base
{
    // Error: Property must have init to override Base.Property
    public override int Property { get; set; }
}
```

An `interface` declaration can also participate in `init` style initialization via the following pattern:

```
interface IPerson
{
    string Name { get; init; }
}

class Init
{
    void M<T>() where T : IPerson, new()
    {
        var local = new T()
        {
            Name = "Jared"
        };
        local.Name = "Jraed"; // Error
    }
}
```

Restrictions of this feature:

- The `init` accessor can only be used on instance properties
- A property cannot contain both an `init` and `set` accessor
- All overrides of a property must have `init` if the base had `init`. This rule also applies to interface implementation.

**Readonly structs**

`init` accessors (both auto-implemented accessors and manually-implemented accessors) are permitted on properties of `readonly struct`s, as well as `readonly` properties. `init` accessors are not permitted to be marked `readonly` themselves, in both `readonly` and non-`readonly` `struct` types.

```
readonly struct ReadonlyStruct1
{
    public int Prop1 { get; init; } // Allowed
}

struct ReadonlyStruct2
{
    public readonly int Prop2 { get; init; } // Allowed

    public int Prop3 { get; readonly init; } // Error
}
```

**Metadata encoding**

Property `init` accessors will be emitted as a standard `set` accessor with the return type marked with a modreq of `IsExternalInit`. This is a new type which will have the following definition:

```
namespace System.Runtime.CompilerServices
{
    public sealed class IsExternalInit
    {
    }
}
```

The compiler will match the type by full name. There is no requirement that it appear in the core library. If there are multiple types by this name then the compiler will tie break in the following order:

1. The one defined in the project being compiled
2. The one defined in corelib

If neither of these exist then a type ambiguity error will be issued.

The design for `IsExternalInit` is futher covered in this issue

# Questions

**Breaking changes**

One of the main pivot points in how this feature is encoded will come down to the following question:

> Is it a binary breaking change to replace `init` with `set`?

Replacing `init` with `set` and thus making a property fully writable is never a source breaking change on a non-virtual property. It simply expands the set of scenarios where the property can be written. The only behavior in question is whether or not this remains a binary breaking change.

If we want to make the change of `init` to `set` a source and binary compatible change then it will force our hand on the modreq vs. attributes decision below because it will rule out modreqs as a soulution. If on the other hand this is seen as a non-interesting then this will make the modreq vs. attribute decision less impactful.

**Resolution** This scenario is not seen as compelling by LDM.

**Modreqs vs. attributes**

The emit strategy for `init` property accessors must choose between using attributes or modreqs when emitting during metadata. These have different trade offs that need to be considered.

Annotating a property set accessor with a modreq declaration means CLI compliant compilers will ignore the accessor unless it understands the modreq. That means only compilers aware of `init` will read the member. Compilers unaware of `init` will ignore the `set` accessor and hence will not accidentally treat the property as read / write.

The downside of modreq is `init` becomes a part of the binary signature of the `set` accessor. Adding or removing `init` will break binary compatbility of the application.

Using attributes to annotate the `set` accessor means that only compilers which understand the attribute will know to limit access to it. A compiler unaware of `init` will see it as a simple read / write property and allow access.

This would seemingly mean this decision is a choice between extra safety at the expense of binary compatibility. Digging in a bit the extra safety is not exactly what it seems. It will not protect against the following circumstances:

1. Reflection over `public` members

2. The use of `dynamic`
3. Compilers that don't recognize modreqs

It should also be considered that, when we complete the IL verification rules for .NET 5, `init` will be one of those rules. That means extra enforcement will be gained from simply verifying compilers emitting verifiable IL.

The primary languages for .NET (C#, F# and VB) will all be updated to recognize these `init` accessors. Hence the only realistic scenario here is when a C# 9 compiler emits `init` properties and they are seen by an older toolset such as C# 8, VB 15, etc ... C# 8. That is the trade off to consider and weigh against binary compatibility.

**Note** This discussion primarily applies to members only, not to fields. While `init` fields were rejected by LDM they are still interesting to consider for the modreq vs. attribute discussion. The `init` feature for fields is a relaxation of the existing restriction of `readonly`. That means if we emit the fields as `readonly` + an attribute there is no risk of older compilers mis-using the field because they would already recognize `readonly`. Hence using a modreq here doesn't add any extra protection.

**Resolution** The feature will use a modreq to encode the property `init` setter. The compelling factors were (in no particular order):

- Desire to discourage older compilers from violating `init` semantics
- Desire to make adding or removing `init` in a `virtual` declaration or `interface` both a source and binary breaking change.

Given there was also no significant support for removing `init` to be a binary compatible change it made the choice of using modreq straight forward.

**init vs. initonly**

There were three syntax forms which got significant consideration during our LDM meeting:

```
// 1. Use init
int Option1 { get; init; }
// 2. Use init set
int Option2 { get; init set; }
// 3. Use initonly
int Option3 { get; initonly; }
```

**Resolution** There was no syntax which was overwhelmingly favored in LDM.

One point which got significant attention was how the choice of syntax would impact our ability to do `init` members as a general feature in the future. Choosing option 1 would mean that it would be difficult to define a property which had an `init` style `get` method in the future. Eventually it was decided that if we decided to go forward with general `init` members in future, we could allow `init` to be a modifier in the property accessor list as well as a short hand for `init set`. Essentially the following two declarations would be identical.

```
int Property1 { get; init; }
int Property1 { get; init set; }
```

The decision was made to move forward with `init` as a standalone accessor in the property accessor list.

**Warn on failed init**

Consider the following scenario. A type declares an `init` only member which is not set in the constructor. Should the code which constructs the object get a warning if they failed to initialize the value?

At that point it is clear the field will never be set and hence has a lot of similarities with the warning around failing to initialize `private` data. Hence a warning would seemingly have some value here?

There are significant downsides to this warning though:

1. It complicates the compatibility story of changing `readonly` to `init`.
2. It requires carrying additional metadata around to denote the members which are required to be initialized by the caller.

Further if we believe there is value here in the overall scenario of forcing object creators to be warned / error'd about specific fields then this likely makes sense as a general feature. There is no reason it should be limited to just `init` members.

**Resolution** There will be no warning on consumption of `init` fields and properties.

LDM wants to have a broader discussion on the idea of required fields and properties. That may cause us to come back and reconsider our position on `init` members and validation.

## Allow init as a field modifier

In the same way `init` can serve as a property accessor it could also serve as a designation on fields to give them similar behaviors as `init` properties. That would allow for the field to be assigned before construction was complete by the type, derived types, or object initializers.

```
class Student
{
    public init string FirstName;
    public init string LastName;
}

var s = new Student()
{
    FirstName = "Jarde",
    LastName = "Parsons",
}

s.FirstName = "Jared"; // Error FirstName is readonly
```

In metadata these fields would be marked in the same way as `readonly` fields but with an additional attribute or modreq to indicate they are `init` style fields.

**Resolution** LDM agrees this proposal is sound but overall the scenario felt disjoint from properties. The decision was to proceed only with `init` properties for now. This has a suitable level of flexibility as an `init` property can mutate a `readonly` field on the declaring type of the property. This will be reconsidered if there is significant customer feedback that justifies the scenario.

#### Allow init as a type modifier

In the same way the `readonly` modifier can be applied to a `struct` to automatically declare all fields as `readonly`, the `init` only modifier can be declared on a `struct` or `class` to automatically mark all fields as `init`. This means the following two type declarations are equivalent:

```
struct Point
{
    public init int X;
    public init int Y;
}

// vs.

init struct Point
{
    public int X;
    public int Y;
}
```

**Resolution** This feature is too *cute* here and conflicts with the `readonly struct` feature on which it is based. The `readonly struct` feature is simple in that it applies `readonly` to all members: fields, methods, etc ... The `init struct` feature would only apply to properties. This actually ends up making it confusing for users.

Given that `init` is only valid on certain aspects of a type, we rejected the idea of having it as a type modifier.

# Considerations

### Compatibility

The `init` feature is designed to be compatible with existing `get` only properties. Specifically it is meant to be a completely additive change for a property which is `get` only today but desires more flexbile object creation semantics.

For example consider the following type:

```
class Name
{
    public string First { get; }
    public string Last { get; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}
```

Adding `init` to these properties is a non-breaking change:

```
class Name
{
    public string First { get; init; }
    public string Last { get; init; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}
```

### IL verification

When .NET Core decides to re-implement IL verification, the rules will need to be adjusted to account for `init` members. This will need to be included in the rule changes for non-mutating acess to `readonly` data.

The IL verification rules will need to be broken into two parts:

1. Allowing `init` members to set a `readonly` field.
2. Determining when an `init` member can be legally called.

The first is a simple adjustment to the existing rules. The IL verifier can be taught to recognize `init` members and from there it just needs to consider a `readonly` field to be settable on `this` in such a member.

The second rule is more complicated. In the simple case of object initializers the rule is straight forward. It should be legal to call `init` members when the result of a `new` expression is still on the stack. That is until the value has been stored in a local, array element or field or passed as an argument to another method it will still be legal to call `init` members. This ensures that once the result of the `new` expression is published to a named identifier (other than `this`) then it will no longer be legal to call `init` members.

The more complicated case though is when we mix `init` members, object initializers and `await`. That can cause the newly created object to be temporarily hoisted into a state machine and hence put into a field.

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Here the result of `new Student()` will be hoised into a state machine as a field before the set of `Name` occurs. The compiler will need to mark such hoisted fields in a way that the IL verifier understands they're not user accessible and hence doesn't violate the intended semantics of `init`.

**init members**

The `init` modifier could be extended to apply to all instance members. This would generalize the concept of `init` during object construction and allow types to declare helper methods that could partipate in the construction process to initialize `init` fields and properties.

Such members would have all the restricions that an `init` accessor does in this design. The need is questionable though and this can be safely added in a future version of the language in a compatible manner.

**Generate three accessors**

One potential implementation of `init` properties is to make `init` completely separate from `set`. That means that a property can potentially have three different accessors: `get`, `set`, and `init`.

This has the potential advantage of allowing the use of modreq to enforce correctness while maintaining binary compatibility. The implementation would roughly be the following:

1. An `init` accessor is always emitted if there is a `set`. When not defined by the developer it is simply a reference to `set`.
2. The set of a property in an object initializer will always use `init` if present but fall back to `set` if it's missing.

This means that a developer can always safely delete `init` from a property.

The downside of this design is that is only useful if `init` is **always** emitted when there is a `set`. The language can't know if `init` was deleted in the past, it has to assume it was and hence the `init` must always be emitted. That would cause a significant metadata expansion and is simply not worth the cost of the compatibility here.

# Target-typed `new` expressions

11/2/2020 • 3 minutes to read • Edit Online

- [x] Proposed
- [x] Prototype
- [ ] Implementation
- [ ] Specification

## Summary

Do not require type specification for constructors when the type is known.

## Motivation

Allow field initialization without duplicating the type.

```
Dictionary<string, List<int>> field = new() {
    { "item1", new() { 1, 2, 3 } }
};
```

Allow omitting the type when it can be inferred from usage.

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Instantiate an object without spelling out the type.

```
private readonly static object s_syncObj = new();
```

## Specification

A new syntactic form, *target_typed_new* of the *object_creation_expression* is accepted in which the *type* is optional.

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    | target_typed_new
    ;
target_typed_new
    : 'new' '(' argument_list? ')' object_or_collection_initializer?
    ;
```

A *target_typed_new* expression does not have a type. However, there is a new *object creation conversion* that is an implicit conversion from expression, that exists from a *target_typed_new* to every type.

Given a target type `T`, the type `T0` is `T`'s underlying type if `T` is an instance of `System.Nullable`. Otherwise `T0` is `T`. The meaning of a *target_typed_new* expression that is converted to the type `T` is the same as the meaning of a corresponding *object_creation_expression* that specifies `T0` as the type.

It is a compile-time error if a *target_typed_new* is used as an operand of a unary or binary operator, or if it is used

where it is not subject to an *object creation conversion*.

> **Open Issue:** should we allow delegates and tuples as the target-type?

The above rules include delegates (a reference type) and tuples (a struct type). Although both types are constructible, if the type is inferable, an anonymous function or a tuple literal can already be used.

```
(int a, int b) t = new(1, 2); // "new" is redundant
Action a = new(() => {}); // "new" is redundant

(int a, int b) t = new(); // OK; same as (0, 0)
Action a = new(); // no constructor found
```

**Miscellaneous**

The following are consequences of the specification:

- `throw new()` is allowed (the target type is `System.Exception`)
- Target-typed `new` is not allowed with binary operators.
- It is disallowed when there is no type to target: unary operators, collection of a `foreach`, in a `using`, in a deconstruction, in an `await` expression, as an anonymous type property (`new { Prop = new() }`), in a `lock` statement, in a `sizeof`, in a `fixed` statement, in a member access (`new().field`), in a dynamically dispatched operation (`someDynamic.Method(new())`), in a LINQ query, as the operand of the `is` operator, as the left operand of the `??` operator, ...
- It is also disallowed as a `ref`.
- The following kinds of types are not permitted as targets of the conversion
  - **Enum types:** `new()` will work (as `new Enum()` works to give the default value), but `new(1)` will not work as enum types do not have a constructor.
  - **Interface types:** This would work the same as the corresponding creation expression for COM types.
  - **Array types:** arrays need a special syntax to provide the length.
  - **dynamic:** we don't allow `new dynamic()`, so we don't allow `new()` with `dynamic` as a target type.
  - **tuples:** These have the same meaning as an object creation using the underlying type.
  - All the other types that are not permitted in the *object_creation_expression* are excluded as well, for instance, pointer types.

# Drawbacks

There were some concerns with target-typed `new` creating new categories of breaking changes, but we already have that with `null` and `default`, and that has not been a significant problem.

# Alternatives

Most of complaints about types being too long to duplicate in field initialization is about *type arguments* not the type itself, we could infer only type arguments like `new Dictionary(...)` (or similar) and infer type arguments locally from arguments or the collection initializer.

# Questions

- Should we forbid usages in expression trees? (no)
- How the feature interacts with `dynamic` arguments? (no special treatment)
- How IntelliSense should work with `new()`? (only when there is a single target-type)

## Design meetings

- LDM-2017-10-18
- LDM-2018-05-21
- LDM-2018-06-25
- LDM-2018-08-22
- LDM-2018-10-17
- LDM-2020-03-25

# Module Initializers

11/2/2020 • 2 minutes to read • Edit Online

- [x] Proposed
- [ ] Prototype: In Progress
- [ ] Implementation: In Progress
- [ ] Specification:

## Summary

Although the .NET platform has a feature that directly supports writing initialization code for the assembly (technically, the module), it is not exposed in C#. This is a rather niche scenario, but once you run into it the solutions appear to be pretty painful. There are reports of a number of customers (inside and outside Microsoft) struggling with the problem, and there are no doubt more undocumented cases.

## Motivation

- Enable libraries to do eager, one-time initialization when loaded, with minimal overhead and without the user needing to explicitly call anything
- One particular pain point of current `static` constructor approaches is that the runtime must do additional checks on usage of a type with a static constructor, in order to decide whether the static constructor needs to be run or not. This adds measurable overhead.
- Enable source generators to run some global initialization logic without the user needing to explicitly call anything

## Detailed design

A method can be designated as a module initializer by decorating it with a `[ModuleInitializer]` attribute.

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public sealed class ModuleInitializerAttribute : Attribute { }
}
```

The attribute can be used like this:

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

Some requirements are imposed on the method targeted with this attribute:

1. The method must be `static`.

2. The method must be parameterless.
3. The method must return `void`.
4. The method must not be generic or be contained in a generic type.
5. The method must be accessible from the containing module.
    - This means the method's effective accessibility must be `internal` or `public`.
    - This also means the method cannot be a local function.

When one or more valid methods with this attribute are found in a compilation, the compiler will emit a module initializer which calls each of the attributed methods. The calls will be emitted in a reserved, but deterministic order.

## Drawbacks

Why should we *not* do this?

- Perhaps the existing third-party tooling for "injecting" module initializers is sufficient for users who have been asking for this feature.

## Design meetings

**April 8th, 2020**

# Extending Partial Methods

11/2/2020 • 5 minutes to read • Edit Online

## Summary

This proposal aims to remove all restrictions around the signatures of `partial` methods in C#. The goal being to expand the set of scenarios in which these methods can work with source generators as well as being a more general declaration form for C# methods.

See also the original partial methods specification.

## Motivation

C# has limited support for developers splitting methods into declarations and definitions / implementations.

```
partial class C
{
    // The declaration of C.M
    partial void M(string message);
}

partial class C
{
    // The definition of C.M
    partial void M(string message) => Console.WriteLine(message);
}
```

One behavior of `partial` methods is that when the definition is absent then the language will simply erase any calls to the `partial` method. Essentially it behaves like a call to a `[Conditional]` method where the condition was evaluated to false.

```
partial class D
{
    partial void M(string message);

    void Example()
    {
        M(GetIt()); // Call to M and GetIt erased at compile time
    }

    string GetIt() => "Hello World";
}
```

The original motivation for this feature was source generation in the form of designer generated code. Users were constantly editing the generated code because they wanted to hook some aspect of the generated code. Most notably parts of the Windows Forms startup process, after components were initialized.

Editing the generated code was error prone because any action which caused the designer to regenerate the code would cause the user edit to be erased. The `partial` method feature eased this tension because it allowed designers to emit hooks in the form of `partial` methods.

Designers could emit hooks like `partial void OnComponentInit()` and developers could define declarations for them or not define them. In either case though the generated code would compile and developers who were interested in the process could hook in as needed.

This does mean that partial methods have several restrictions:

1. Must have a `void` return type.
2. Cannot have `out` parameters.
3. Cannot have any accessibility (implicitly `private`).

These restrictions exist because the language must be able to emit code when the call site is erased. Given they can be erased `private` is the only possible accessibility because the member can't be exposed in assembly metadata. These restrictions also serve to limit the set of scenarios in which `partial` methods can be applied.

The proposal here is to remove all of the existing restrictions around `partial` methods. Essentially let them have `out`, non-void return types or any type of accessibility. Such `partial` declarations would then have the added requirement that a definition must exist. That means the language does not have to consider the impact of erasing the call sites.

This would expand the set of generator scenarios that `partial` methods could participate in and hence link in nicely with our source generators feature. For example a regex could be defined using the following pattern:

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

This gives both the developer a simple declarative way of opting into generators as well as giving generators a very easy set of declarations to look through in the source code to drive their generated output.

Compare that with the difficulty that a generator would have hooking up the following snippet of code.

```
var regex = new RegularExpression("(dog|cat|fish)");
if (regex.IsMatch(someInput))
{

}
```

Given that the compiler doesn't allow generators to modify code hooking up this pattern would be pretty much impossible for generators. They would need to resort to reflection in the `IsMatch` implementation, or asking users to change their call sites to a new method + refactor the regex to pass the string literal as an argument. It's pretty messy.

## Detailed Design

The language will change to allow `partial` methods to be annotated with an explicit accessibility modifier. This means they can be labeled as `private`, `public`, etc ...

When a `partial` method has an explicit accessibility modifier though the language will require that the declaration has a matching definition even when the accessibility is `private`:

```
partial class C
{
    // Okay because no definition is required here
    partial void M1();

    // Okay because M2 has a definition
    private partial void M2();

    // Error: partial method M3 must have a definition
    private partial void M3();
}

partial class C
{
    private partial void M2() { }
}
```

Further the language will remove all restrictions on what can appear on a `partial` method which has an explicit accessibility. Such declarations can contain non-void return types, `out` parameters, `extern` modifier, etc ... These signatures will have the full expressivity of the C# language.

```
partial class D
{
    // Okay
    internal partial bool TryParse(string s, out int i);
}

partial class D
{
    internal partial bool TryParse(string s, out int i) { }
}
```

This explicitly allows for `partial` methods to participate in `overrides` and `interface` implementations:

```
interface IStudent
{
    string GetName();
}

partial class C : IStudent
{
    public virtual partial string GetName();
}

partial class C
{
    public virtual partial string GetName() => "Jarde";
}
```

The compiler will change the error it emits when a `partial` method contains an illegal element to essentially say:

> Cannot use `ref` on a `partial` method that lacks explicit accessibility

This will help point developers in the right direction when using this feature.

Restrictions:

- `partial` declarations with explicit accessibility must have a definition
- `partial` declarations and definition signatures must match on all method and parameter modifiers. The only aspects which can differ are parameter names and attribute lists (this is not new but rather an existing

requirement of `partial` methods).

# Questions

**partial on all members**

Given that we're expanding `partial` to be more friendly to source generators should we also expand it to work on all class members? For example should we be able to declare `partial` constructors, operators, etc ...

**Resolution** The idea is sound but at this point in the C# 9 schedule we're trying to avoid unnecessary feature creep. Want to solve the immediate problem of expanding the feature to work with modern source generators.

Extending `partial` to support other members will be considered for the C# 10 release. Seems likely that we will consider this extension.

**Use abstract instead of partial**

The crux of this proposal is essentially ensuring that a declaration has a corresponding definition / implementation. Given that should we use `abstract` since it's already a language keyword that forces the developer to think about having an implementation?

**Resolution** There was a healthy discussion about this but eventually it was decided against. Yes the requirements are familiar but the concepts are significantly different. Could easily lead the developer to believe they were creating virtual slots when they were not doing so.

# Static anonymous functions

11/2/2020 • 2 minutes to read • Edit Online

## Summary

Allow a 'static' modifier on lambdas and anonymous methods, which disallows capture of locals or instance state from containing scopes.

## Motivation

Avoid unintentionally capturing state from the enclosing context, which can result in unexpected retention of captured objects or unexpected additional allocations.

## Detailed design

A lambda or anonymous method may have a `static` modifier. The `static` modifier indicates that the lambda or anonymous method is a *static anonymous function*.

A *static anonymous function* cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a *static anonymous function*.

A *static anonymous function* cannot reference instance members from an implicit or explicit `this` or `base` reference.

A *static anonymous function* may reference `static` members from the enclosing scope.

A *static anonymous function* may reference `constant` definitions from the enclosing scope.

`nameof()` in a *static anonymous function* may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` anonymous functions.

No guarantee is made as to whether a *static anonymous function* definition is emitted as a `static` method in metadata. This is left up to the compiler implementation to optimize.

A non-`static` local function or anonymous function can capture state from an enclosing *static anonymous function* but cannot capture state outside the enclosing *static anonymous function*.

Removing the `static` modifier from an anonymous function in a valid program does not change the meaning of the program.

# Target-Typed Conditional Expression

11/2/2020 • 3 minutes to read • Edit Online

## Conditional Expression Conversion

For a conditional expression `c ? e1 : e2`, when

1. there is no common type for `e1` and `e2`, or
2. for which a common type exists but one of the expressions `e1` or `e2` has no implicit conversion to that type

we define a new implicit *conditional expression conversion* that permits an implicit conversion from the conditional expression to any type `T` for which there is a conversion-from-expression from `e1` to `T` and also from `e2` to `T`. It is an error if a conditional expression neither has a common type between `e1` and `e2` nor is subject to a *conditional expression conversion*.

## Better Conversion from Expression

We change

> **Better conversion from expression**
> Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not exactly match `T2` and at least one of the following holds:
>
> - `E` exactly matches `T1` (Exactly matching Expression)
> - `T1` is a better conversion target than `T2` (Better conversion target)

to

> **Better conversion from expression**
> Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not exactly match `T2` and at least one of the following holds:
>
> - `E` exactly matches `T1` (Exactly matching Expression)
> - `C1` **is not a *conditional expression conversion* and** `C2` **is a *conditional expression conversion*.**
> - `T1` is a better conversion target than `T2` (Better conversion target) **and either** `C1` **and** `C2` **are both *conditional expression conversions* or neither is a *conditional expression conversion*.**

## Cast Expression

The current C# language specification says

> A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion (Explicit conversions) of the value of `E` to type `T`.

In the presence of the *conditional expression conversion* there may be more than one possible conversion from `E` to `T`. With the addition of *conditional expression conversion*, we prefer any other conversion to a *conditional expression conversion*, and use the *conditional expression conversion* only as a last resort.

# Design Notes

The reason for the change to *Better conversion from expression* is to handle a case such as this:

```
M(b ? 1 : 2);

void M(short);
void M(long);
```

This approach does have two small downsides. First, it is not quite the same as the switch expression:

```
M(b ? 1 : 2); // calls M(long)
M(b switch { true => 1, false => 2 }); // calls M(short)
```

This is still a breaking change, but its scope is less likely to affect real programs:

```
M(b ? 1 : 2, 1); // calls M(long, long) without this feature; ambiguous with this feature.

M(short, short);
M(long, long);
```

This becomes ambiguous because the conversion to `long` is better for the first argument (because it does not use the *conditional expression conversion*), but the conversion to `short` is better for the second argument (because `short` is a *better conversion target* than `long`). This breaking change seems less serious because it does not silently change the behavior of an existing program.

The reason for the notes on the cast expression is to handle a case such as this:

```
_ = (short)(b ? 1 : 2);
```

This program currently uses the explicit conversion from `int` to `short`, and we want to preserve the current language meaning of this program. The change would be unobservable at runtime, but with the following program the change would be observable:

```
_ = (A)(b ? c : d);
```

where `c` is of type `C`, `d` is of type `D`, and there is an implicit user-defined conversion from `C` to `D`, and an implicit user-defined conversion from `D` to `A`, and an implicit user-defined conversion from `C` to `A`. If this code is compiled before C# 9.0, when `b` is true we convert from `c` to `D` then to `A`. If we use the *conditional expression conversion*, then when `b` is true we convert from `c` to `A` directly, which executes a different sequence of user code. Therefore we treat the *conditional expression conversion* as a last resort in a cast, to preserve existing behavior.

# Covariant returns

11/2/2020 • 10 minutes to read • Edit Online

## Summary

Support *covariant return types*. Specifically, permit the override of a method to declare a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to declare a more derived type. Override declarations appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types. Callers of the method or property would statically receive the more refined return type from an invocation.

## Motivation

It is a common pattern in code that different method names have to be invented to work around the language constraint that overrides must return the same type as the overridden method.

This would be useful in the factory pattern. For example, in the Roslyn code base we would have

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

```
class CSharpCompilation : Compilation
{
    public override CSharpCompilation WithOptions(Options options)...
}
```

## Detailed design

This is a specification for covariant return types in C#. Our intent is to permit the override of a method to return a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to return a more derived return type. Callers of the method or property would statically receive the more refined return type from an invocation, and overrides appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types.

---

**Class Method Override**

The existing constraint on class override methods

> - The override method and the overridden base method have the same return type.

is modified to

> - The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a ref return) implicit reference conversion to the return type of the overridden base method.

And the following additional requirements are appended to that list:

> - The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a ref return) implicit reference conversion to the return type of every override of the overridden base method that is declared in a (direct or indirect) base type of the override method.
> - The override method's return type must be at least as accessible as the override method (Accessibility domains).

This constraint permits an override method in a `private` class to have a `private` return type. However it requires a `public` override method in a `public` type to have a `public` return type.

**Class Property and Indexer Override**

The existing constraint on class override properties

> An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion ~~between the type of the overriding and the inherited property~~. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

is modified to

> An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion **or (if the inherited property is read-only and has a value return - not a ref return) implicit reference conversion from the type of the overriding property to the type of the inherited property**. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors. **The overriding property's type must be at least as accessible as the overriding property (Accessibility domains).**

---

***The remainder of the draft specification below proposes a further extension to covariant returns of interface methods to be considered later.***

**Interface Method, Property, and Indexer Override**

Adding to the kinds of members that are permitted in an interface with the addition of the DIM feature in C# 8.0, we further add support for `override` members along with covariant returns. These follow the rules of `override` members as specified for classes, with the following differences:

The following text in classes:

> The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class of `C`, starting with the direct base class of `C` and continuing with each successive direct base class, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments.

is given the corresponding specification for interfaces:

> The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in an interface `I`, the overridden base method is determined by examining each direct or indirect base interface of `I`, collecting the set of interfaces declaring an accessible method which has the same signature as `M` after substitution of type arguments. If this set of interfaces has a *most derived type*, to which

> there is an identity or implicit reference conversion from every type in this set, and that type contains a unique such method declaration, then that is the *overridden base method*.

We similarly permit `override` properties and indexers in interfaces as specified for classes in *15.7.6 Virtual, sealed, override, and abstract accessors*.

**Name Lookup**

Name lookup in the presence of class `override` declarations currently modify the result of name lookup by imposing on the found member details from the most derived `override` declaration in the class hierarchy starting from the type of the identifier's qualifier (or `this` when there is no qualifier). For example, in *12.6.2.2 Corresponding parameters* we have

> For virtual methods and indexers defined in classes, the parameter list is picked from the first declaration or override of the function member found when starting with the static type of the receiver, and searching through its base classes.

to this we add

> For virtual methods and indexers defined in interfaces, the parameter list is picked from the declaration or override of the function member found in the most derived type among those types containing the declaration of override of the function member. It is a compile-time error if no unique such type exists.

For the result type of a property or indexer access, the existing text

> - If I identifies an instance property, then the result is a property access with an associated instance expression of E and an associated type that is the type of the property. If T is a class type, the associated type is picked from the first declaration or override of the property found when starting with T, and searching through its base classes.

is augmented with

> If T is an interface type, the associated type is picked from the declaration or override of the property found in the most derived of T or its direct or indirect base interfaces. It is a compile-time error if no unique such type exists.

A similar change should be made in *12.7.7.3 Indexer access*

In *12.7.6 Invocation expressions* we augment the existing text

> - Otherwise, the result is a value, with an associated type of the return type of the method or delegate. If the invocation is of an instance method, and the receiver is of a class type T, the associated type is picked from the first declaration or override of the method found when starting with T and searching through its base classes.

with

> If the invocation is of an instance method, and the receiver is of an interface type T, the associated type is picked from the declaration or override of the method found in the most derived interface from among T and its direct and indirect base interfaces. It is a compile-time error if no unique such type exists.

**Implicit Interface Implementations**

This section of the specification

> For purposes of interface mapping, a class member `A` matches an interface member `B` when:
>
> - `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
> - `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
> - `A` and `B` are events, and the name and type of `A` and `B` are identical.
> - `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

is modified as follows:

> For purposes of interface mapping, a class member `A` matches an interface member `B` when:
>
> - `A` and `B` are methods, and the name and formal parameter lists of `A` and `B` are identical, and the return type of `A` is convertible to the return type of `B` via an identity of implicit reference convertion to the return type of `B`.
> - `A` and `B` are properties, the name of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly property, an implicit reference conversion.
> - `A` and `B` are events, and the name and type of `A` and `B` are identical.
> - `A` and `B` are indexers, the formal parameter lists of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly indexer, an implicit reference conversion.

This is technically a breaking change, as the program below prints "C1.M" today, but would print "C2.M" under the proposed revision.

```
using System;

interface I1 { object M(); }
class C1 : I1 { public object M() { return "C1.M"; } }
class C2 : C1, I1 { public new string M() { return "C2.M"; } }
class Program
{
    static void Main()
    {
        I1 i = new C2();
        Console.WriteLine(i.M());
    }
}
```

Due to this breaking change, we might consider not supporting covariant return types on implicit implementations.

**Constraints on Interface Implementation**

**We will need a rule that an explicit interface implementation must declare a return type no less derived than the return type declared in any override in its base interfaces.**

**API Compatibility Implications**

*TBD*

**Open Issues**

The specification does not say how the caller gets the more refined return type. Presumably that would be done in a way similar to the way that callers get the most derived override's parameter specifications.

---

If we have the following interfaces:

```
interface I1 { I1 M(); }
interface I2 { I2 M(); }
interface I3: I1, I2 { override I3 M(); }
```

Note that in `I3`, the methods `I1.M()` and `I2.M()` have been "merged". When implementing `I3`, it is necessary to implement them both together.

Generally, we require an explicit implementation to refer to the original method. The question is, in a class

```
class C : I1, I2, I3
{
    C IN.M();
}
```

What does that mean here? What should *N* be?

I suggest that we permit implementing either `I1.M` or `I2.M` (but not both), and treat that as an implementation of both.

## Drawbacks

- [ ] Every language change must pay for itself.
- [ ] We should ensure that the performance is reasonable, even in the case of deep inheritance hierarchies
- [ ] We should ensure that artifacts of the translation strategy do not affect language semantics, even when consuming new IL from old compilers.

## Alternatives

We could relax the language rules slightly to allow, in source,

```
abstract class Cloneable
{
    public abstract Cloneable Clone();
}

class Digit : Cloneable
{
    public override Cloneable Clone()
    {
        return this.Clone();
    }

    public new Digit Clone() // Error: 'Digit' already defines a member called 'Clone' with the same parameter
types
    {
        return this;
    }
}
```

## Unresolved questions

- [ ] How will APIs that have been compiled to use this feature work in older versions of the language?

# Design meetings

- some discussion at https://github.com/dotnet/roslyn/issues/357.
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-01-08.md
- Offline discussion toward a decision to support overriding of class methods only in C# 9.0.

# Extension `GetEnumerator` support for `foreach` loops.

11/2/2020 • 5 minutes to read • Edit Online

## Summary

Allow foreach loops to recognize an extension method GetEnumerator method that otherwise satisfies the foreach pattern, and loop over the expression when it would otherwise be an error.

## Motivation

This will bring foreach inline with how other features in C# are implemented, including async and pattern-based deconstruction.

## Detailed design

The spec change is relatively straightforward. We modify `The foreach statement` section to this text:

> The compile-time processing of a foreach statement first determines the ***collection type***, ***enumerator type*** and ***element type*** of the expression. This determination proceeds as follows:
>
> - If the type `X` of *expression* is an array type then there is an implicit reference conversion from `X` to the `IEnumerable` interface (since `System.Array` implements this interface). The ***collection type*** is the `IEnumerable` interface, the ***enumerator type*** is the `IEnumerator` interface and the ***element type*** is the element type of the array type `X`.
> - If the type `X` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface (Implicit dynamic conversions). The ***collection type*** is the `IEnumerable` interface and the ***enumerator type*** is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the ***element type*** is `dynamic`, otherwise it is `object`.
> - Otherwise, determine whether the type `X` has an appropriate `GetEnumerator` method:
>   - Perform member lookup on the type `X` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
>   - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.
>   - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
>   - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
>   - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
>   - Overload resolution is performed on the method group with an empty argument list. If overload

resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
    - If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the ***collection type*** is the interface `IEnumerable<T>`, the ***enumerator type*** is the interface `IEnumerator<T>`, and the ***element type*** is `T`.
    - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
    - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.IEnumerator`, and the ***element type*** is `object`.
- Otherwise, determine whether the type 'X' has an appropriate `GetEnumerator` extension method:
    - Perform extension method lookup on the type `X` with identifier `GetEnumerator`. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match which is not a method group, an error is produced and no further steps are taken. It is recommended that a warning be issues if member lookup produces anything except a method group or no match.
    - Perform overload resolution using the resulting method group and a single argument of type `X`. If overload resolution produces no applicable methods, results in an ambiguity, or results in a single best method but that method is not accessible, an error is produced an no further steps are taken.
        - This resolution permits the first argument to be passed by ref if `X` is a struct type, and the ref kind is `in`.
    - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
    - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, an error is produced and no further steps are taken.

For `await foreach`, the rules are similarly modified. The only change that is required to that spec is removing the `Extension methods do not contribute.` line from the description, as the rest of that spec is based on the above rules with different names substituted for the pattern methods.

## Drawbacks

Every change adds additional complexity to the language, and this potentially allows things that weren't designed

to be `foreach` ed to be `foreach` ed, like `Range` .

## Alternatives

Doing nothing.

## Unresolved questions

None at this point.

# Lambda discard parameters

11/2/2020 • 2 minutes to read • Edit Online

## Summary

Allow discards (`_`) to be used as parameters of lambdas and anonymous methods. For example:

- lambdas: `(_, _) => 0`, `(int _, int _) => 0`
- anonymous methods: `delegate(int _, int _) { return 0; }`

## Motivation

Unused parameters do not need to be named. The intent of discards is clear, i.e. they are unused/discarded.

## Detailed design

Method parameters In the parameter list of a lambda or anonymous method with more than one parameter named `_`, such parameters are discard parameters. Note: if a single parameter is named `_` then it is a regular parameter for backwards compatibility reasons.

Discard parameters do not introduce any names to any scopes. Note this implies they do not cause any `_` (underscore) names to be hidden.

Simple names If `K` is zero and the *simple_name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (Declarations) contains a local variable, parameter (with the exception of discard parameters) or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.

Scopes With the exception of discard parameters, the scope of a parameter declared in a *lambda_expression* (Anonymous function expressions) is the *anonymous_function_body* of that *lambda_expression* With the exception of discard parameters, the scope of a parameter declared in an *anonymous_method_expression* (Anonymous function expressions) is the *block* of that *anonymous_method_expression*.

## Related spec sections

- Corresponding parameters

# Attributes on local functions

11/2/2020 • 2 minutes to read • Edit Online

## Attributes

Local function declarations are now permitted to have attributes. Parameters and type parameters on local functions are also allowed to have attributes.

Attributes with a specified meaning when applied to a method, its parameters, or its type parameters will have the same meaning when applied to a local function, its parameters, or its type parameters, respectively.

A local function can be made conditional in the same sense as a conditional method by decorating it with a `[ConditionalAttribute]`. A conditional local function must also be `static`. All restrictions on conditional methods also apply to conditional local functions, including that the return type must be `void`.

## Extern

The `extern` modifier is now permitted on local functions. This makes the local function external in the same sense as an external method.

Similarly to an external method, the *local-function-body* of an external local function must be a semicolon. A semicolon *local-function-body* is only permitted on an external local function.

An external local function must also be `static`.

## Syntax

The local functions grammar is modified as follows:

```
local-function-header
    : attributes? local-function-modifiers? return-type identifier type-parameter-list?
        ( formal-parameter-list? ) type-parameter-constraints-clauses
    ;

local-function-modifiers
    : (async | unsafe | static | extern)*
    ;

local-function-body
    : block
    | arrow-expression-body
    | ';'
    ;
```

# Native-sized integers

11/2/2020 • 10 minutes to read • Edit Online

## Summary

Language support for a native-sized signed and unsigned integer types.

The motivation is for interop scenarios and for low-level libraries.

## Design

The identifiers `nint` and `nuint` are new contextual keywords that represent native signed and unsigned integer types. The identifiers are only treated as keywords when name lookup does not find a viable result at that program location.

```
nint x = 3;
string y = nameof(nuint);
_ = nint.Equals(x, 3);
```

The types `nint` and `nuint` are represented by the underlying types `System.IntPtr` and `System.UIntPtr` with compiler surfacing additional conversions and operations for those types as native ints.

### Constants

Constant expressions may be of type `nint` or `nuint`. There is no direct syntax for native int literals. Implicit or explicit casts of other integral constant values can be used instead: `const nint i = (nint)42;`.

`nint` constants are in the range [ `int.MinValue`, `int.MaxValue` ].

`nuint` constants are in the range [ `uint.MinValue`, `uint.MaxValue` ].

There are no `MinValue` or `MaxValue` fields on `nint` or `nuint` because, other than `nuint.MinValue`, those values cannot be emitted as constants.

Constant folding is supported for all unary operators { `+`, `-`, `~` } and binary operators { `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&`, `|`, `^`, `<<`, `>>` }. Constant folding operations are evaluated with `Int32` and `UInt32` operands rather than native ints for consistent behavior regardless of compiler platform. If the operation results in a constant value in 32-bits, constant folding is performed at compile-time. Otherwise the operation is executed at runtime and not considered a constant.

### Conversions

There is an identity conversion between `nint` and `IntPtr`, and between `nuint` and `UIntPtr`. There is an identity conversion between compound types that differ by native ints and underlying types only: arrays, `Nullable<>`, constructed types, and tuples.

The tables below cover the conversions between special types. (The IL for each conversion includes the variants for `unchecked` and `checked` contexts if different.)

| OPERAND | TARGET | CONVERSION | IL |
|---|---|---|---|
| `object` | `nint` | Unboxing | `unbox` |

| OPERAND | TARGET | CONVERSION | IL |
| --- | --- | --- | --- |
| `void*` | `nint` | PointerToVoid | `conv.i` |
| `sbyte` | `nint` | ImplicitNumeric | `conv.i` |
| `byte` | `nint` | ImplicitNumeric | `conv.u` |
| `short` | `nint` | ImplicitNumeric | `conv.i` |
| `ushort` | `nint` | ImplicitNumeric | `conv.u` |
| `int` | `nint` | ImplicitNumeric | `conv.i` |
| `uint` | `nint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `long` | `nint` | ExplicitNumeric | `conv.i` / `conv.ovf.i` |
| `ulong` | `nint` | ExplicitNumeric | `conv.i` / `conv.ovf.i` |
| `char` | `nint` | ImplicitNumeric | `conv.i` |
| `float` | `nint` | ExplicitNumeric | `conv.i` / `conv.ovf.i` |
| `double` | `nint` | ExplicitNumeric | `conv.i` / `conv.ovf.i` |
| `decimal` | `nint` | ExplicitNumeric | `long decimal.op_Explicit(decimal) conv.i` / `... conv.ovf.i` |
| `IntPtr` | `nint` | Identity | |
| `UIntPtr` | `nint` | None | |
| `object` | `nuint` | Unboxing | `unbox` |
| `void*` | `nuint` | PointerToVoid | `conv.u` |
| `sbyte` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `byte` | `nuint` | ImplicitNumeric | `conv.u` |
| `short` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `ushort` | `nuint` | ImplicitNumeric | `conv.u` |
| `int` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `uint` | `nuint` | ImplicitNumeric | `conv.u` |

| OPERAND | TARGET | CONVERSION | IL |
| --- | --- | --- | --- |
| `long` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `ulong` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `char` | `nuint` | ImplicitNumeric | `conv.u` |
| `float` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `double` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `decimal` | `nuint` | ExplicitNumeric | `ulong decimal.op_Explicit(decimal) conv.u` / `... conv.ovf.u.un` |
| `IntPtr` | `nuint` | None | |
| `UIntPtr` | `nuint` | Identity | |
| Enumeration | `nint` | ExplicitEnumeration | |
| Enumeration | `nuint` | ExplicitEnumeration | |

| OPERAND | TARGET | CONVERSION | IL |
| --- | --- | --- | --- |
| `nint` | `object` | Boxing | `box` |
| `nint` | `void*` | PointerToVoid | `conv.i` |
| `nint` | `nuint` | ExplicitNumeric | `conv.u` / `conv.ovf.u` |
| `nint` | `sbyte` | ExplicitNumeric | `conv.i1` / `conv.ovf.i1` |
| `nint` | `byte` | ExplicitNumeric | `conv.u1` / `conv.ovf.u1` |
| `nint` | `short` | ExplicitNumeric | `conv.i2` / `conv.ovf.i2` |
| `nint` | `ushort` | ExplicitNumeric | `conv.u2` / `conv.ovf.u2` |
| `nint` | `int` | ExplicitNumeric | `conv.i4` / `conv.ovf.i4` |
| `nint` | `uint` | ExplicitNumeric | `conv.u4` / `conv.ovf.u4` |
| `nint` | `long` | ImplicitNumeric | `conv.i8` / `conv.ovf.i8` |
| `nint` | `ulong` | ExplicitNumeric | `conv.i8` / `conv.ovf.i8` |
| `nint` | `char` | ExplicitNumeric | `conv.u2` / `conv.ovf.u2` |

| OPERAND | TARGET | CONVERSION | IL |
|---|---|---|---|
| `nint` | `float` | ImplicitNumeric | `conv.r4` |
| `nint` | `double` | ImplicitNumeric | `conv.r8` |
| `nint` | `decimal` | ImplicitNumeric | `conv.i8 decimal decimal.op_Implicit(long)` |
| `nint` | `IntPtr` | Identity | |
| `nint` | `UIntPtr` | None | |
| `nint` | Enumeration | ExplicitEnumeration | |
| `nuint` | `object` | Boxing | `box` |
| `nuint` | `void*` | PointerToVoid | `conv.u` |
| `nuint` | `nint` | ExplicitNumeric | `conv.i` / `conv.ovf.i` |
| `nuint` | `sbyte` | ExplicitNumeric | `conv.i1` / `conv.ovf.i1` |
| `nuint` | `byte` | ExplicitNumeric | `conv.u1` / `conv.ovf.u1` |
| `nuint` | `short` | ExplicitNumeric | `conv.i2` / `conv.ovf.i2` |
| `nuint` | `ushort` | ExplicitNumeric | `conv.u2` / `conv.ovf.u2` |
| `nuint` | `int` | ExplicitNumeric | `conv.i4` / `conv.ovf.i4` |
| `nuint` | `uint` | ExplicitNumeric | `conv.u4` / `conv.ovf.u4` |
| `nuint` | `long` | ExplicitNumeric | `conv.i8` / `conv.ovf.i8` |
| `nuint` | `ulong` | ImplicitNumeric | `conv.u8` / `conv.ovf.u8` |
| `nuint` | `char` | ExplicitNumeric | `conv.u2` / `conv.ovf.u2.un` |
| `nuint` | `float` | ImplicitNumeric | `conv.r.un conv.r4` |
| `nuint` | `double` | ImplicitNumeric | `conv.r.un conv.r8` |
| `nuint` | `decimal` | ImplicitNumeric | `conv.u8 decimal decimal.op_Implicit(ulong)` |
| `nuint` | `IntPtr` | None | |
| `nuint` | `UIntPtr` | Identity | |

| OPERAND | TARGET | CONVERSION | IL |
|---|---|---|---|
| `nuint` | Enumeration | ExplicitEnumeration | |

Conversion from `A` to `Nullable<B>` is:

- an implicit nullable conversion if there is an identity conversion or implicit conversion from `A` to `B`;
- an explicit nullable conversion if there is an explicit conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `B` is:

- an explicit nullable conversion if there is an identity conversion or implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `Nullable<B>` is:

- an identity conversion if there is an identity conversion from `A` to `B`;
- an explicit nullable conversion if there is an implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

**Operators**

The predefined operators are as follows. These operators are considered during overload resolution based on normal rules for implicit conversions *if at least one of the operands is of type* `nint` *or* `nuint`.

(The IL for each operator includes the variants for `unchecked` and `checked` contexts if different.)

| UNARY | OPERATOR SIGNATURE | IL |
|---|---|---|
| `+` | `nint operator +(nint value)` | `nop` |
| `+` | `nuint operator +(nuint value)` | `nop` |
| `-` | `nint operator -(nint value)` | `neg` |
| `~` | `nint operator ~(nint value)` | `not` |
| `~` | `nuint operator ~(nuint value)` | `not` |

| BINARY | OPERATOR SIGNATURE | IL |
|---|---|---|
| `+` | `nint operator +(nint left, nint right)` | `add` / `add.ovf` |
| `+` | `nuint operator +(nuint left, nuint right)` | `add` / `add.ovf.un` |
| `-` | `nint operator -(nint left, nint right)` | `sub` / `sub.ovf` |
| `-` | `nuint operator -(nuint left, nuint right)` | `sub` / `sub.ovf.un` |

| BINARY | OPERATOR SIGNATURE | IL |
| --- | --- | --- |
| `*` | `nint operator *(nint left, nint right)` | `mul` / `mul.ovf` |
| `*` | `nuint operator *(nuint left, nuint right)` | `mul` / `mul.ovf.un` |
| `/` | `nint operator /(nint left, nint right)` | `div` |
| `/` | `nuint operator /(nuint left, nuint right)` | `div.un` |
| `%` | `nint operator %(nint left, nint right)` | `rem` |
| `%` | `nuint operator %(nuint left, nuint right)` | `rem.un` |
| `==` | `bool operator ==(nint left, nint right)` | `beq` / `ceq` |
| `==` | `bool operator ==(nuint left, nuint right)` | `beq` / `ceq` |
| `!=` | `bool operator !=(nint left, nint right)` | `bne` |
| `!=` | `bool operator !=(nuint left, nuint right)` | `bne` |
| `<` | `bool operator <(nint left, nint right)` | `blt` / `clt` |
| `<` | `bool operator <(nuint left, nuint right)` | `blt.un` / `clt.un` |
| `<=` | `bool operator <=(nint left, nint right)` | `ble` |
| `<=` | `bool operator <=(nuint left, nuint right)` | `ble.un` |
| `>` | `bool operator >(nint left, nint right)` | `bgt` / `cgt` |
| `>` | `bool operator >(nuint left, nuint right)` | `bgt.un` / `cgt.un` |
| `>=` | `bool operator >=(nint left, nint right)` | `bge` |
| `>=` | `bool operator >=(nuint left, nuint right)` | `bge.un` |
| `&` | `nint operator &(nint left, nint right)` | `and` |

| BINARY | OPERATOR SIGNATURE | IL |
|---|---|---|
| `&` | `nuint operator &(nuint left, nuint right)` | `and` |
| `\|` | `nint operator \|(nint left, nint right)` | `or` |
| `\|` | `nuint operator \|(nuint left, nuint right)` | `or` |
| `^` | `nint operator ^(nint left, nint right)` | `xor` |
| `^` | `nuint operator ^(nuint left, nuint right)` | `xor` |
| `<<` | `nint operator <<(nint left, int right)` | `shl` |
| `<<` | `nuint operator <<(nuint left, int right)` | `shl` |
| `>>` | `nint operator >>(nint left, int right)` | `shr` |
| `>>` | `nuint operator >>(nuint left, int right)` | `shr.un` |

For some binary operators, the IL operators support additional operand types (see ECMA-335 III.1.5 Operand type table). But the set of operand types supported by C# is limited for simplicity and for consistency with existing operators in the language.

Lifted versions of the operators, where the arguments and return types are `nint?` and `nuint?`, are supported.

Compound assignment operations `x op= y` where `x` or `y` are native ints follow the same rules as with other primitive types with pre-defined operators. Specifically the expression is bound as `x = (T)(x op y)` where `T` is the type of `x` and where `x` is only evaluated once.

The shift operators should mask the number of bits to shift - to 5 bits if `sizeof(nint)` is 4, and to 6 bits if `sizeof(nint)` is 8. (see shift operators in C# spec).

The C#9 compiler will report errors binding to predefined native integer operators when compiling with an earlier language version, but will allow use of predefined conversions to and from native integers.

`csc -langversion:9 -t:library A.cs`

```
public class A
{
    public static nint F;
}
```

`csc -langversion:8 -r:A.dll B.cs`

```
class B : A
{
    static void Main()
    {
        F = F + 1; // error: nint operator+ not available with -langversion:8
        F = (System.IntPtr)F + 1; // ok
    }
}
```

#### Dynamic

The conversions and operators are synthesized by the compiler and are not part of the underlying `IntPtr` and `UIntPtr` types. As a result those conversions and operators *are not available* from the runtime binder for `dynamic`.

```
nint x = 2;
nint y = x + x; // ok
dynamic d = x;
nint z = d + x; // RuntimeBinderException: '+' cannot be applied 'System.IntPtr' and 'System.IntPtr'
```

#### Type members

The only constructor for `nint` or `nuint` is the parameter-less constructor.

The following members of `System.IntPtr` and `System.UIntPtr` *are explicitly excluded* from `nint` or `nuint`:

```
// constructors
// arithmetic operators
// implicit and explicit conversions
public static readonly IntPtr Zero; // use 0 instead
public static int Size { get; }     // use sizeof() instead
public static IntPtr Add(IntPtr pointer, int offset);
public static IntPtr Subtract(IntPtr pointer, int offset);
public int ToInt32();
public long ToInt64();
public void* ToPointer();
```

The remaining members of `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `nuint`. For .NET Framework 4.7.2:

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
public string ToString(string format);
```

Interfaces implemented by `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `nuint`, with occurrences of the underlying types replaced by the corresponding native integer types. For instance if `IntPtr` implements `ISerializable, IEquatable<IntPtr>, IComparable<IntPtr>`, then `nint` implements `ISerializable, IEquatable<nint>, IComparable<nint>`.

#### Overriding, hiding, and implementing

`nint` and `System.IntPtr`, and `nuint` and `System.UIntPtr`, are considered equivalent for overriding, hiding, and implementing.

Overloads cannot differ by `nint` and `System.IntPtr`, and `nuint` and `System.UIntPtr`, alone. Overrides and implementations may differ by `nint` and `System.IntPtr`, or `nuint` and `System.UIntPtr`, alone. Methods hide other methods that differ by `nint` and `System.IntPtr`, or `nuint` and `System.UIntPtr`, alone.

#### Miscellaneous

`nint` and `nuint` expressions used as array indices are emitted without conversion.

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` and `nuint` can be used as an `enum` base type.

```
enum E : nint // ok
{
}
```

Reads and writes are atomic for types `nint`, `nuint`, and `enum` with base type `nint` or `nuint`.

Fields may be marked `volatile` for types `nint` and `nuint`. ECMA-334 15.5.4 does not include `enum` with base type `System.IntPtr` or `System.UIntPtr` however.

`default(nint)` and `new nint()` are equivalent to `(nint)0`.

`typeof(nint)` is `typeof(IntPtr)`.

`sizeof(nint)` is supported but requires compiling in an unsafe context (as does `sizeof(IntPtr)`). The value is not a compile-time constant. `sizeof(nint)` is implemented as `sizeof(IntPtr)` rather than `IntPtr.Size`.

Compiler diagnostics for type references involving `nint` or `nuint` report `nint` or `nuint` rather than `IntPtr` or `UIntPtr`.

**Metadata**

`nint` and `nuint` are represented in metadata as `System.IntPtr` and `System.UIntPtr`.

Type references that include `nint` or `nuint` are emitted with a `System.Runtime.CompilerServices.NativeIntegerAttribute` to indicate which parts of the type reference are native ints.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Event |
        AttributeTargets.Field |
        AttributeTargets.GenericParameter |
        AttributeTargets.Parameter |
        AttributeTargets.Property |
        AttributeTargets.ReturnValue,
        AllowMultiple = false,
        Inherited = false)]
    public sealed class NativeIntegerAttribute : Attribute
    {
        public NativeIntegerAttribute()
        {
            TransformFlags = new[] { true };
        }
        public NativeIntegerAttribute(bool[] flags)
        {
            TransformFlags = flags;
        }
        public readonly bool[] TransformFlags;
    }
}
```

The encoding of type references with `NativeIntegerAttribute` is covered in NativeIntegerAttribute.md.

## Alternatives

An alternative to the "type erasure" approach above is to introduce new types: `System.NativeInt` and `System.NativeUInt`.

```
public readonly struct NativeInt
{
    public IntPtr Value;
}
```

Distinct types would allow overloading distinct from `IntPtr` and would allow distinct parsing and `ToString()`. But there would be more work for the CLR to handle these types efficiently which defeats the primary purpose of the feature - efficiency. And interop with existing native int code that uses `IntPtr` would be more difficult.

Another alternative is to add more native int support for `IntPtr` in the framework but without any specific compiler support. Any new conversions and arithmetic operations would be supported by the compiler automatically. But the language would not provide keywords, constants, or `checked` operations.

## Design meetings

- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-05-26.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-06-13.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators
- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-10-23.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md

# Function Pointers

11/2/2020 • 19 minutes to read • Edit Online

## Summary

This proposal provides language constructs that expose IL opcodes that cannot currently be accessed efficiently, or at all, in C# today: `ldftn` and `calli`. These IL opcodes can be important in high performance code and developers need an efficient way to access them.

## Motivation

The motivations and background for this feature are described in the following issue (as is a potential implementation of the feature):

https://github.com/dotnet/csharplang/issues/191

This is an alternate design proposal to compiler intrinsics

## Detailed Design

**Function pointers**

The language will allow for the declaration of function pointers using the `delegate*` syntax. The full syntax is described in detail in the next section but it is meant to resemble the syntax used by `Func` and `Action` type declarations.

```
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

These types are represented using the function pointer type as outlined in ECMA-335. This means invocation of a `delegate*` will use `calli` where invocation of a `delegate` will use `callvirt` on the `Invoke` method. Syntactically though invocation is identical for both constructs.

The ECMA-335 definition of method pointers includes the calling convention as part of the type signature (section 7.1). The default calling convention will be `managed`. Unmanaged calling conventions can by specified by putting an `unmanaged` keyword afer the `delegate*` syntax, which will use the runtime platform default. Specific unmanaged conventions can then be specified in brackets to the `unmanaged` keyword by specifying any type starting with `CallConv` in the `System.Runtime.CompilerServices` namespace, leaving off the `CallConv` prefix. These types must come from the program's core library, and the set of valid combinations is platform-dependent.

```
//This method has a managed calling convention. This is the same as leaving the managed keyword off.
delegate* managed<int, int>;

// This method will be invoked using whatever the default unmanaged calling convention on the runtime
// platform is. This is platform and architecture dependent and is determined by the CLR at runtime.
delegate* unmanaged<int, int>;

// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;

// This method will be invoked using the stdcall calling convention, and suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

Conversions between `delegate*` types is done based on their signature including the calling convention.

```
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;

        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

A `delegate*` type is a pointer type which means it has all of the capabilities and restrictions of a standard pointer type:

- Only valid in an `unsafe` context.
- Methods which contain a `delegate*` parameter or return type can only be called from an `unsafe` context.
- Cannot be converted to `object`.
- Cannot be used as a generic argument.
- Can implicitly convert `delegate*` to `void*`.
- Can explicitly convert from `void*` to `delegate*`.

Restrictions:

- Custom attributes cannot be applied to a `delegate*` or any of its elements.
- A `delegate*` parameter cannot be marked as `params`
- A `delegate*` type has all of the restrictions of a normal pointer type.
- Pointer arithmetic cannot be performed directly on function pointer types.

**Function pointer syntax**

The full function pointer syntax is represented by the following grammar:

```
pointer_type
    : ...
    | funcptr_type
    ;

funcptr_type
    : 'delegate' '*' calling_convention_specifier? '<' funcptr_parameter_list funcptr_return_type '>'
    ;

calling_convention_specifier
    : 'managed'
    | 'unmanaged' ('[' unmanaged_calling_convention ']')?
    ;

unmanaged_calling_convention
    : 'Cdecl'
    | 'Stdcall'
    | 'Thiscall'
    | 'Fastcall'
    | identifier (',' identifier)*
    ;

funptr_parameter_list
    : (funcptr_parameter ',')*
    ;

funcptr_parameter
    : funcptr_parameter_modifier? type
    ;

funcptr_return_type
    : funcptr_return_modifier? return_type
    ;

funcptr_parameter_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

funcptr_return_modifier
    : 'ref'
    | 'ref readonly'
    ;
```

If no `calling_convention_specifier` is provided, the default is `managed`. The precise metadata encoding of the `calling_convention_specifier` and what `identifier`s are valid in the `unmanaged_calling_convention` is covered in Metadata Representation of Calling Conventions.

```
delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate*<string, int>;
delegate*<delegate*<string, int>, delegate*<string, int>>;

// Function pointer equivalent with calling convention
delegate* managed<string, int>;
delegate*<delegate* managed<string, int>, delegate*<string, int>>;
```

**Function pointer conversions**

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- *Existing conversions*
- From *funcptr_type* `F0` to another *funcptr_type* `F1`, provided all of the following are true:
  - `F0` and `F1` have the same number of parameters, and each parameter `D0n` in `F0` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter `D1n` in `F1`.
  - For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `F0` to the corresponding parameter type in `F1`.
  - For each `ref`, `out`, or `in` parameter, the parameter type in `F0` is the same as the corresponding parameter type in `F1`.
  - If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F1` to the return type of `F0`.
  - If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F1` are the same as the return type and `ref` modifiers of `F0`.
  - The calling convention of `F0` is the same as the calling convention of `F1`.

**Allow address-of to target methods**

Method groups will now be allowed as arguments to an address-of expression. The type of such an expression will be a `delegate*` which has the equivalent signature of the target method and a managed calling convention:

```
unsafe class Util {
    public static void Log() { }

    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*<int>";
        delegate*<int> ptr2 = &Util.Log;
   }
}
```

In an unsafe context, a method `M` is compatible with a function pointer type `F` if all of the following are true:

- `M` and `F` have the same number of parameters, and each parameter in `M` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter in `F`.
- For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `M` to the corresponding parameter type in `F`.
- For each `ref`, `out`, or `in` parameter, the parameter type in `M` is the same as the corresponding parameter type in `F`.
- If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F` to the return type of `M`.
- If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F` are the same as the return type and `ref` modifiers of `M`.
- The calling convention of `M` is the same as the calling convention of `F`. This includes both the calling convention bit, as well as any calling convention flags specified in the unmanaged identifier.
- `M` is a static method.

In an unsafe context, an implicit conversion exists from an address-of expression whose target is a method group `E` to a compatible function pointer type `F` if `E` contains at least one method that is applicable in its normal form to an argument list constructed by use of the parameter types and modifiers of `F`, as described in the following.

- A single method `M` is selected corresponding to a method invocation of the form `E(A)` with the following

modifications:

  - The arguments list `A` is a list of expressions, each classified as a variable and with the type and modifier ( `ref` , `out` , or `in` ) of the corresponding *funcptr_parameter_list* of `F` .
  - The candidate methods are only those methods that are applicable in their normal form, not those applicable in their expanded form.
  - The candidate methods are only those methods that are static.
- If the algorithm of overload resolution produces an error, then a compile-time error occurs. Otherwise, the algorithm produces a single best method `M` having the same number of parameters as `F` and the conversion is considered to exist.
- The selected method `M` must be compatible (as defined above) with the function pointer type `F` . Otherwise, a compile-time error occurs.
- The result of the conversion is a function pointer of type `F` .

This means developers can depend on overload resolution rules to work in conjunction with the address-of operator:

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
```

The address-of operator will be implemented using the `ldftn` instruction.

Restrictions of this feature:

- Only applies to methods marked as `static` .
- Non- `static` local functions cannot be used in `&` . The implementation details of these methods are deliberately not specified by the language. This includes whether they are static vs. instance or exactly what signature they are emitted with.

**Operators on Function Pointer Types**

The section in unsafe code on operators is modified as such:

> In an unsafe context, several constructs are available for operating on all _pointer_type_s that are not _funcptr_type_s:
>
> - The `*` operator may be used to perform pointer indirection (Pointer indirection).
> - The `->` operator may be used to access a member of a struct through a pointer (Pointer member access).
> - The `[]` operator may be used to index a pointer (Pointer element access).
> - The `&` operator may be used to obtain the address of a variable (The address-of operator).
> - The `++` and `--` operators may be used to increment and decrement pointers (Pointer increment and decrement).
> - The `+` and `-` operators may be used to perform pointer arithmetic (Pointer arithmetic).
> - The `==` , `!=` , `<` , `>` , `<=` , and `=>` operators may be used to compare pointers (Pointer comparison).
> - The `stackalloc` operator may be used to allocate memory from the call stack (Fixed size buffers).
> - The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (The fixed

> statement).
>
> In an unsafe context, several constructs are available for operating on all _funcptr_type_s:
>
> - The `&` operator may be used to obtain the address of static methods (Allow address-of to target methods)
> - The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (Pointer comparison).

Additionally, we modify all the sections in `Pointers in expressions` to forbid function pointer types, except `Pointer comparison` and `The sizeof operator`.

**Better function member**

The better function member specification will be changed to include the following line:

> A `delegate*` is more specific than `void*`

This means that it is possible to overload on `void*` and a `delegate*` and still sensibly use the address-of operator.

## Metadata representation of `in`, `out`, and `ref readonly` parameters and return types

Function pointer signatures have no parameter flags location, so we must encode whether parameters and the return type are `in`, `out`, or `ref readonly` by using modreqs.

`in`

We reuse `System.Runtime.InteropServices.InAttribute`, applied as a `modreq` to the ref specifier on a parameter or return type, to mean the following:

- If applied to a parameter ref specifier, this parameter is treated as `in`.
- If applied to the return type ref specifier, the return type is treated as `ref readonly`.

`out`

We use `System.Runtime.InteropServices.OutAttribute`, applied as a `modreq` to the ref specifier on a parameter type, to mean that the parameter is an `out` parameter.

**Errors**

- It is an error to apply `OutAttribute` as a modreq to a return type.
- It is an error to apply both `InAttribute` and `OutAttribute` as a modreq to a parameter type.
- If either are specified via modopt, they are ignored.

**Metadata Representation of Calling Conventions**

Calling conventions are encoded in a method signature in metadata by a combination of the `CallKind` flag in the signature and zero or more `modopt`s at the start of the signature. ECMA-335 currently declares the following elements in the `CallKind` flag:

```
CallKind
   : default
   | unmanaged cdecl
   | unmanaged fastcall
   | unmanaged thiscall
   | unmanaged stdcall
   | varargs
   ;
```

Of these, function pointers in C# will support all but `varargs`.

In addition, the runtime (and eventually 335) will be updated to include a new `CallKind` on new platforms. This does not have a formal name currently, but this document will use `unmanaged ext` as a placeholder to stand for the new extensible calling convention format. With no `modopt`s, `unmanaged ext` is the platform default calling convention, `unmanaged` without the square brackets.

#### Mapping the `calling_convention_specifier` to a `CallKind`

A `calling_convention_specifier` that is omitted, or specified as `managed`, maps to the `default` `CallKind`. This is default `CallKind` of any method not attributed with `UnmanagedCallersOnly`.

C# recognizes 4 special identifiers that map to specific existing unmanaged `CallKind`s from ECMA 335. In order for this mapping to occur, these identifiers must be specified on their own, with no other identifiers, and this requirement is encoded into the spec for `unmanaged_calling_convention`s. These identifiers are `Cdecl`, `Thiscall`, `Stdcall`, and `Fastcall`, which correspond to `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, and `unmanaged fastcall`, respectively. If more than one `identifer` is specified, or the single `identifier` is not of the specially recognized identifiers, we perform special name lookup on the identifier with the following rules:

- We prepend the `identifier` with the string `CallConv`
- We look only at types defined in the `System.Runtime.CompilerServices` namespace.
- We look only at types defined in the core library of the application, which is the library that defines `System.Object` and has no dependencies.
- We look only at public types.

If lookup succeeds on all of the `identifier`s specified in an `unmanaged_calling_convention`, we encode the `CallKind` as `unmanaged ext`, and encode each of the resolved types in the set of `modopt`s at the beginning of the function pointer signature. As a note, these rules mean that users cannot prefix these `identifier`s with `CallConv`, as that will result in looking up `CallConvCallConvVectorCall`.

When interpreting metadata, we first look at the `CallKind`. If it is anything other than `unmanaged ext`, we ignore all `modopt`s on the return type for the purposes of determining the calling convention, and use only the `CallKind`. If the `CallKind` is `unmanaged ext`, we look at the modopts at the start of the function pointer type, taking the union of all types that meet the following requirements:

- The is defined in the core library, which is the library that references no other libraries and defines `System.Object`.
- The type is defined in the `System.Runtime.CompilerServices` namespace.
- The type starts with the prefix `CallConv`.
- The type is public.

These represent the types that must be found when performing lookup on the `identifier`s in an `unmanaged_calling_convention` when defining a function pointer type in source.

It is an error to attempt to use a function pointer with a `CallKind` of `unmanaged ext` if the target runtime does not support the feature. This will be determined by looking for the presence of the `System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constant. If this constant is present, the runtime is considered to support the feature.

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` is an attribute used by the CLR to indicate that a method should be called with a specific calling convention. Because of this, we introduce the following support for working with the attribute:

- It is an error to directly call a method annotated with this attribute from C#. Users must obtain a function pointer

to the method and then invoke that pointer.

- It is an error to apply the attribute to anything other than an ordinary static method or ordinary static local function. The C# compiler will mark any non-static or static non-ordinary methods imported from metadata with this attribute as unsupported by the language.
- It is an error for a method marked with the attribute to have a parameter or return type that is not an `unmanaged_type`.
- It is an error for a method marked with the attribute to have type parameters, even if those type parameters are constrained to `unmanaged`.
- It is an error for a method in a generic type to be marked with the attribute.
- It is an error to convert a method marked with the attribute to a delegate type.
- It is an error to specify any types for `UnmanagedCallersOnly.CallConvs` that do not meet the requirements for calling convention `modopt`s in metadata.

When determining the calling convention of a method marked with a valid `UnmanagedCallersOnly` attribute, the compiler performs the following checks on the types specified in the `CallConvs` property to determine the effective `CallKind` and `modopt`s that should be used to determine the calling convention:

- If no types are specified, the `CallKind` is treated as `unmanaged ext`, with no calling convention `modopt`s at the start of the function pointer type.
- If there is one type specified, and that type is named `CallConvCdecl`, `CallConvThiscall`, `CallConvStdcall`, or `CallConvFastcall`, the `CallKind` is treated as `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, or `unmanaged fastcall`, respectively, with no calling convention `modopt`s at the start of the function pointer type.
- If multiple types are specified or the single type is not named one of the specially called out types above, the `CallKind` is treated as `unmanaged ext`, with the union of the types specified treated as `modopt`s at the start of the function pointer type.

The compiler then looks at this effective `CallKind` and `modopt` collection and uses normal metadata rules to determine the final calling convention of the function pointer type.

# Open Questions

### Detecting runtime support for `unmanaged ext`

https://github.com/dotnet/runtime/issues/38135 tracks adding this flag. Depending on the feedback from review, we will either use the property specified in the issue, or use the presence of `UnmanagedCallersOnlyAttribute` as the flag that determines whether the runtimes supports `unmanaged ext`.

# Considerations

### Allow instance methods

The proposal could be extended to support instance methods by taking advantage of the `EXPLICITTHIS` CLI calling convention (named `instance` in C# code). This form of CLI function pointers puts the `this` parameter as an explicit first parameter of the function pointer syntax.

```
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
    }
}
```

This is sound but adds some complication to the proposal. Particularly because function pointers which differed by the calling convention `instance` and `managed` would be incompatible even though both cases are used to invoke

managed methods with the same C# signature. Also in every case considered where this would be valuable to have there was a simple work around: use a `static` local function.

```
unsafe class Instance {
    void Use() {
        static string toString(Instance i) = i.ToString();
        delgate*<Instance, string> f = &toString;
        f(this);
    }
}
```

**Don't require unsafe at declaration**

Instead of requiring `unsafe` at every use of a `delegate*`, only require it at the point where a method group is converted to a `delegate*`. This is where the core safety issues come into play (knowing that the containing assembly cannot be unloaded while the value is alive). Requiring `unsafe` on the other locations can be seen as excessive.

This is how the design was originally intended. But the resulting language rules felt very awkward. It's impossible to hide the fact that this is a pointer value and it kept peeking through even without the `unsafe` keyword. For example the conversion to `object` can't be allowed, it can't be a member of a `class`, etc ... The C# design is to require `unsafe` for all pointer uses and hence this design follows that.

Developers will still be capable of presenting a *safe* wrapper on top of `delegate*` values the same way that they do for normal pointer types today. Consider:

```
unsafe struct Action {
    delegate*<void> _ptr;

    Action(delegate*<void> ptr) => _ptr = ptr;
    public void Invoke() => _ptr();
}
```

**Using delegates**

Instead of using a new syntax element, `delegate*`, simply use existing `delegate` types with a `*` following the type:

```
Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

Handling calling convention can be done by annotating the `delegate` types with an attribute that specifies a `CallingConvention` value. The lack of an attribute would signify the managed calling convention.

Encoding this in IL is problematic. The underlying value needs to be represented as a pointer yet it also must:

1. Have a unique type to allow for overloads with different function pointer types.
2. Be equivalent for OHI purposes across assembly boundaries.

The last point is particularly problematic. This mean that every assembly which uses `Func<int>*` must encode an equivalent type in metadata even though `Func<int>*` is defined in an assembly though don't control. Additionally any other type which is defined with the name `System.Func<T>` in an assembly that is not mscorlib must be different than the version defined in mscorlib.

One option that was explored was emitting such a pointer as `mod_req(Func<int>) void*`. This doesn't work though as a `mod_req` cannot bind to a `TypeSpec` and hence cannot target generic instantiations.

**Named function pointers**

The function pointer syntax can be cumbersome, particularly in complex cases like nested function pointers. Rather

than have developers type out the signature every time the language could allow for named declarations of function pointers as is done with `delegate`.

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();
    }
}
```

Part of the problem here is the underlying CLI primitive doesn't have names hence this would be purely a C# invention and require a bit of metadata work to enable. That is doable but is a significant about of work. It essentially requires C# to have a companion to the type def table purely for these names.

Also when the arguments for named function pointers were examined we found they could apply equally well to a number of other scenarios. For example it would be just as convenient to declare named tuples to reduce the need to type out the full signature in all cases.

```
(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

After discussion we decided to not allow named declaration of `delegate*` types. If we find there is significant need for this based on customer usage feedback then we will investigate a naming solution that works for function pointers, tuples, generics, etc ... This is likely to be similar in form to other suggestions like full `typedef` support in the language.

# Future Considerations

**static delegates**

This refers to the proposal to allow for the declaration of `delegate` types which can only refer to `static` members. The advantage being that such `delegate` instances can be allocation free and better in performance sensitive scenarios.

If the function pointer feature is implemented the `static delegate` proposal will likely be closed out. The proposed advantage of that feature is the allocation free nature. However recent investigations have found that is not possible to achieve due to assembly unloading. There must be a strong handle from the `static delegate` to the method it refers to in order to keep the assembly from being unloaded out from under it.

To maintain every `static delegate` instance would be required to allocate a new handle which runs counter to the goals of the proposal. There were some designs where the allocation could be amortized to a single allocation per call-site but that was a bit complex and didn't seem worth the trade off.

That means developers essentially have to decide between the following trade offs:

1. Safety in the face of assembly unloading: this requires allocations and hence `delegate` is already a sufficient option.
2. No safety in face of assembly unloading: use a `delegate*`. This can be wrapped in a `struct` to allow usage outside an `unsafe` context in the rest of the code.

# Suppress emitting of `localsinit` flag.

11/2/2020 • 3 minutes to read • Edit Online

- [x] Proposed
- [ ] Prototype: Not Started
- [ ] Implementation: Not Started
- [ ] Specification: Not Started

## Summary

Allow suppressing emit of `localsinit` flag via `SkipLocalsInitAttribute` attribute.

## Motivation

**Background**

Per CLR spec local variables that do not contain references are not initialized to a particular value by the VM/JIT. Reading from such variables without initialization is type-safe, but otherwise the behavior is undefined and implementation specific. Typically uninitialized locals contain whatever values were left in the memory that is now occupied by the stack frame. That could lead to nondeterministic behavior and hard to reproduce bugs.

There are two ways to "assign" a local variable:

- by storing a value or
- by specifying `localsinit` flag which forces everything that is allocated form the local memory pool to be zero-initialized NOTE: this includes both local variables and `stackalloc` data.

Use of uninitialized data is discouraged and is not allowed in verifiable code. While it might be possible to prove that by the means of flow analysis, it is permitted for the verification algorithm to be conservative and simply require that `localsinit` is set.

Historically C# compiler emits `localsinit` flag on all methods that declare locals.

While C# employs definite-assignment analysis which is more strict than what CLR spec would require (C# also needs to consider scoping of locals), it is not strictly guaranteed that the resulting code would be formally verifiable:

- CLR and C# rules may not agree on whether passing a local as `out` argument is a `use`.
- CLR and C# rules may not agree on treatment of conditional branches when conditions are known (constant propagation).
- CLR could as well simply require `localinits`, since that is permitted.

**Problem**

In high-performance application the cost of forced zero-initialization could be noticeable. It is particularly noticeable when `stackalloc` is used.

In some cases JIT can elide initial zero-initialization of individual locals when such initialization is "killed" by subsequent assignments. Not all JITs do this and such optimization has limits. It does not help with `stackalloc`.

To illustrate that the problem is real - there is a known bug where a method not containing any `IL` locals would not have `localsinit` flag. The bug is already being exploited by users by putting `stackalloc` into such methods - intentionally to avoid initialization costs. That is despite the fact that absence of `IL` locals is an unstable metric and may vary depending on changes in codegen strategy. The bug should be fixed and users should get a more

documented and reliable way of suppressing the flag.

## Detailed design

Allow specifying `System.Runtime.CompilerServices.SkipLocalsInitAttribute` as a way to tell the compiler to not emit `localsinit` flag.

The end result of this will be that the locals may not be zero-initialized by the JIT, which is in most cases unobservable in C#.
In addition to that `stackalloc` data will not be zero-initialized. That is definitely observable, but also is the most motivating scenario.

Permitted and recognized attribute targets are: `Method`, `Property`, `Module`, `Class`, `Struct`, `Interface`, `Constructor`. However compiler will not require that attribute is defined with the listed targets nor it will care in which assembly the attribute is defined.

When attribute is specified on a container (`class`, `module`, containing method for a nested method, ...), the flag affects all methods contained within the container.

Synthesized methods "inherit" the flag from the logical container/owner.

The flag affects only codegen strategy for actual method bodies. I.E. the flag has no effect on abstract methods and is not propagated to overriding/implementing methods.

This is explicitly a ***compiler feature*** and ***not a language feature***.
Similarly to compiler command line switches the feature controls implementation details of a particular codegen strategy and does not need to be required by the C# spec.

## Drawbacks

- Old/other compilers may not honor the attribute. Ignoring the attribute is compatible behavior. Only may result in a slight perf hit.
- The code without `localinits` flag may trigger verification failures. Users that ask for this feature are generally unconcerned with verifiability.
- Applying the attribute at higher levels than an individual method has nonlocal effect, which is observable when `stackalloc` is used. Yet, this is the most requested scenario.

## Alternatives

- omit `localinits` flag when method is declared in `unsafe` context. That could cause silent and dangerous behavior change from deterministic to nondeterministic in a case of `stackalloc`.
- omit `localinits` flag always. Even worse than above.
- omit `localinits` flag unless `stackalloc` is used in the method body. Does not address the most requested scenario and may turn code unverifiable with no option to revert that back.

## Unresolved questions

- Should the attribute be actually emitted to metadata?

## Design meetings

None yet.

# Unconstrained type parameter annotations

11/2/2020 • 2 minutes to read • Edit Online

## Summary

Allow nullable annotations for type parameters that are not constrained to value types or reference types: `T?`.

```
static T? FirstOrDefault<T>(this IEnumerable<T> collection) { ... }
```

## `?` annotation

In C#8, `?` annotations could only be applied to type parameters that were explicitly constrained to value types or reference types. In C#9, `?` annotations can be applied to any type parameter, regardless of constraints.

Unless a type parameter is explicitly constrained to value types, annotations can only be applied within a `#nullable enable` context.

If a type parameter `T` is substituted with a reference type, then `T?` represents a nullable instance of that reference type.

```
var s1 = new string[0].FirstOrDefault();  // string? s1
var s2 = new string?[0].FirstOrDefault(); // string? s2
```

If `T` is substituted with a value type, then `T?` represents an instance of `T`.

```
var i1 = new int[0].FirstOrDefault();  // int i1
var i2 = new int?[0].FirstOrDefault(); // int? i2
```

If `T` is substituted with an annotated type `U?`, then `T?` represents the annotated type `U?` rather than `U??`.

```
var u1 = new U[0].FirstOrDefault();  // U? u1
var u2 = new U?[0].FirstOrDefault(); // U? u2
```

If `T` is substituted with a type `U`, then `T?` represents `U?`, even within a `#nullable disable` context.

```
#nullable disable
var u3 = new U[0].FirstOrDefault();  // U? u3
```

For return values, `T?` is equivalent to `[MaybeNull]T`; for argument values, `T?` is equivalent to `[AllowNull]T`. The equivalence is important when overriding or implementing interfaces from an assembly compiled with C#8.

```
public abstract class A
{
    [return: MaybeNull] public abstract T F1<T>();
    public abstract void F2<T>([AllowNull] T t);
}

public class B : A
{
    public override T? F1<T>() where T : default { ... }       // matches A.F1<T>()
    public override void F2<T>(T? t) where T : default { ... } // matches A.F2<T>()
}
```

## `default` constraint

For compatibility with existing code where overridden and explicitly implemented generic methods could not include explicit constraint clauses, `T?` in an overridden or explicitly implemented method is treated as `Nullable<T>` where `T` is a value type.

To allow annotations for type parameters constrained to reference types, C#8 allowed explicit `where T : class` and `where T : struct` constraints on the overridden or explicitly implemented method.

```
class A1
{
    public virtual void F1<T>(T? t) where T : struct { }
    public virtual void F1<T>(T? t) where T : class { }
}

class B1 : A1
{
    public override void F1<T>(T? t) /*where T : struct*/ { }
    public override void F1<T>(T? t) where T : class { }
}
```

To allow annotations for type parameters that are not constrained to reference types or value types, C#9 allows a new `where T : default` constraint.

```
class A2
{
    public virtual void F2<T>(T? t) where T : struct { }
    public virtual void F2<T>(T? t) { }
}

class B2 : A2
{
    public override void F2<T>(T? t) /*where T : struct*/ { }
    public override void F2<T>(T? t) where T : default { }
}
```

It is an error to use a `default` constraint other than on a method override or explicit implementation. It is an error to use a `default` constraint when the corresponding type parameter in the overridden or interface method is constrained to a reference type or value type.

## Design meetings

- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-11-25.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md#t