**Local functions (C# Programming Guide)**

* 9 minutes to read

Starting with C# 7.0, C# supports *local functions*. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

* Methods, especially iterator methods and async methods
* Constructors
* Property accessors
* Event accessors
* Anonymous methods
* Lambda expressions
* Finalizers
* Other local functions

However, local functions can't be declared inside an expression-bodied member.

**Note**

In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see [**Local functions vs. lambda expressions**](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions#local-functions-vs-lambda-expressions).

Local functions make the intent of your code clear. Anyone reading your code can see that the method is not callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

**Local function syntax**

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

C#Copy

<modifiers> <return-type> <method-name> <parameter-list>

You can use the following modifiers with a local function:

* [async](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/async)
* [unsafe](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe)
* [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) (in C# 8.0 and later). A static local function can't capture local variables or instance state.
* [extern](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/extern) (in C# 9.0 and later). An external local function must be static.

All local variables that are defined in the containing member, including its method parameters, are accessible in a non-static local function.

Unlike a method definition, a local function definition cannot include the member access modifier. Because all local functions are private, including an access modifier, such as the private keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."

The following example defines a local function named AppendPathSeparator that is private to a method named GetText:

C#Copy

private static string GetText(string path, string filename)

{

var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");

var text = reader.ReadToEnd();

return text;

string AppendPathSeparator(string filepath)

{

return filepath.EndsWith(@"\") ? filepath : filepath + @"\";

}

}

Beginning with C# 9.0, you can apply attributes to a local function, its parameters and type parameters, as the following example shows:

C#Copy

#nullable enable

private static void Process(string?[] lines, string mark)

{

foreach (var line in lines)

{

if (IsValid(line))

{

// Processing logic...

}

}

bool IsValid([NotNullWhen(true)] string? line)

{

return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;

}

}

The preceding example uses a [special attribute](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis) to assist the compiler in static analysis in a nullable context.

**Local functions and exceptions**

One of the useful features of local functions is that they can allow exceptions to surface immediately. For method iterators, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an OddSequence method that enumerates odd numbers in a specified range. Because it passes a number greater than 100 to the OddSequence enumerator method, the method throws an [ArgumentOutOfRangeException](https://docs.microsoft.com/en-us/dotnet/api/system.argumentoutofrangeexception). As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

C#Copy

using System;

using System.Collections.Generic;

public class IteratorWithoutLocalExample

{

public static void Main()

{

IEnumerable<int> xs = OddSequence(50, 110);

Console.WriteLine("Retrieved enumerator...");

foreach (var x in xs) // line 11

{

Console.Write($"{x} ");

}

}

public static IEnumerable<int> OddSequence(int start, int end)

{

if (start < 0 || start > 99)

throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");

if (end > 100)

throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");

if (start >= end)

throw new ArgumentException("start must be less than end.");

for (int i = start; i <= end; i++)

{

if (i % 2 == 1)

yield return i;

}

}

}

// The example displays the output like this:

//

// Retrieved enumerator...

// Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100. (Parameter 'end')

// at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in IteratorWithoutLocal.cs:line 22

// at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11

If you put iterator logic into a local function, argument validation exceptions are thrown when you retrieve the enumerator, as the following example shows:

C#Copy

using System;

using System.Collections.Generic;

public class IteratorWithLocalExample

{

public static void Main()

{

IEnumerable<int> xs = OddSequence(50, 110); // line 8

Console.WriteLine("Retrieved enumerator...");

foreach (var x in xs)

{

Console.Write($"{x} ");

}

}

public static IEnumerable<int> OddSequence(int start, int end)

{

if (start < 0 || start > 99)

throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");

if (end > 100)

throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");

if (start >= end)

throw new ArgumentException("start must be less than end.");

return GetOddSequenceEnumerator();

IEnumerable<int> GetOddSequenceEnumerator()

{

for (int i = start; i <= end; i++)

{

if (i % 2 == 1)

yield return i;

}

}

}

}

// The example displays the output like this:

//

// Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100. (Parameter 'end')

// at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in IteratorWithLocal.cs:line 22

// at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8

**Local functions vs. lambda expressions**

At first glance, local functions and [lambda expressions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions) are very similar. In many cases, the choice between using lambda expressions and local functions is a matter of style and personal preference. However, there are real differences in where you can use one or the other that you should be aware of.

Let's examine the differences between the local function and lambda expression implementations of the factorial algorithm. Here's the version using a local function:

C#Copy

public static int LocalFunctionFactorial(int n)

{

return nthFactorial(n);

int nthFactorial(int number) => number < 2

? 1

: number \* nthFactorial(number - 1);

}

This version uses lambda expressions:

C#Copy

public static int LambdaFactorial(int n)

{

Func<int, int> nthFactorial = default(Func<int, int>);

nthFactorial = number => number < 2

? 1

: number \* nthFactorial(number - 1);

return nthFactorial(n);

}

**Naming**

Local functions are explicitly named like methods. Lambda expressions are anonymous methods and need to be assigned to variables of a delegate type, typically either Action or Func types. When you declare a local function, the process is like writing a normal method; you declare a return type and a function signature.

**Function signatures and lambda expression types**

Lambda expressions rely on the type of the Action/Func variable that they're assigned to determine the argument and return types. In local functions, since the syntax is much like writing a normal method, argument types and return type are already part of the function declaration.

**Definite assignment**

Lambda expressions are objects that are declared and assigned at runtime. In order for a lambda expression to be used, it needs to be definitely assigned: the Action/Func variable that it will be assigned to must be declared and the lambda expression assigned to it. Notice that LambdaFactorial must declare and initialize the lambda expression nthFactorial before defining it. Not doing so results in a compile time error for referencing nthFactorial before assigning it.

Local functions are defined at compile time. As they're not assigned to variables, they can be referenced from any code location **where it is in scope**; in our first example LocalFunctionFactorial, we could declare our local function either above or below the return statement and not trigger any compiler errors.

These differences mean that recursive algorithms are easier to create using local functions. You can declare and define a local function that calls itself. Lambda expressions must be declared, and assigned a default value before they can be re-assigned to a body that references the same lambda expression.

**Implementation as a delegate**

Lambda expressions are converted to delegates when they're declared. Local functions are more flexible in that they can be written like a traditional method *or* as a delegate. Local functions are only converted to delegates when ***used*** as a delegate.

If you declare a local function and only reference it by calling it like a method, it will not be converted to a delegate.

**Variable capture**

The rules of [definite assignment](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/variables#definite-assignment) also affect any variables that are captured by the local function or lambda expression. The compiler can perform static analysis that enables local functions to definitely assign captured variables in the enclosing scope. Consider this example:

C#Copy

int M()

{

int y;

LocalFunction();

return y;

void LocalFunction() => y = 0;

}

The compiler can determine that LocalFunction definitely assigns y when called. Because LocalFunction is called before the return statement, y is definitely assigned at the return statement.

Note that when a local function captures variables in the enclosing scope, the local function is implemented as a delegate type.

**Heap allocations**

Depending on their use, local functions can avoid heap allocations that are always necessary for lambda expressions. If a local function is never converted to a delegate, and none of the variables captured by the local function are captured by other lambdas or local functions that are converted to delegates, the compiler can avoid heap allocations.

Consider this async example:

C#Copy

public async Task<string> PerformLongRunningWorkLambda(string address, int index, string name)

{

if (string.IsNullOrWhiteSpace(address))

throw new ArgumentException(message: "An address is required", paramName: nameof(address));

if (index < 0)

throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");

if (string.IsNullOrWhiteSpace(name))

throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

Func<Task<string>> longRunningWorkImplementation = async () =>

{

var interimResult = await FirstWork(address);

var secondResult = await SecondStep(index, name);

return $"The results are {interimResult} and {secondResult}. Enjoy.";

};

return await longRunningWorkImplementation();

}

The closure for this lambda expression contains the address, index and name variables. In the case of local functions, the object that implements the closure may be a struct type. That struct type would be passed by reference to the local function. This difference in implementation would save on an allocation.

The instantiation necessary for lambda expressions means extra memory allocations, which may be a performance factor in time-critical code paths. Local functions do not incur this overhead. In the example above, the local functions version has two fewer allocations than the lambda expression version.

If you know that your local function won't be converted to a delegate and none of the variables captured by it are captured by other lambdas or local functions that are converted to delegates, you can guarantee that your local function avoids being allocated on the heap by declaring it as a static local function. Note that this feature is available in C# 8.0 and newer.

**Note**

The local function equivalent of this method also uses a class for the closure. Whether the closure for a local function is implemented as a class or a struct is an implementation detail. A local function may use a struct whereas a lambda will always use a class.

C#Copy

public async Task<string> PerformLongRunningWork(string address, int index, string name)

{

if (string.IsNullOrWhiteSpace(address))

throw new ArgumentException(message: "An address is required", paramName: nameof(address));

if (index < 0)

throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-negative");

if (string.IsNullOrWhiteSpace(name))

throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

return await longRunningWorkImplementation();

async Task<string> longRunningWorkImplementation()

{

var interimResult = await FirstWork(address);

var secondResult = await SecondStep(index, name);

return $"The results are {interimResult} and {secondResult}. Enjoy.";

}

}

**Usage of the yield keyword**

One final advantage not demonstrated in this sample is that local functions can be implemented as iterators, using the yield return syntax to produce a sequence of values.

C#Copy

public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)

{

if (!input.Any())

{

throw new ArgumentException("There are no items to convert to lowercase.");

}

return LowercaseIterator();

IEnumerable<string> LowercaseIterator()

{

foreach (var output in input.Select(item => item.ToLower()))

{

yield return output;

}

}

}

The yield return statement is not allowed in lambda expressions, see [compiler error CS1621](https://docs.microsoft.com/en-us/dotnet/csharp/misc/cs1621).

While local functions may seem redundant to lambda expressions, they actually serve different purposes and have different uses. Local functions are more efficient for the case when you want to write a function that is called only from the context of another method.

**See also**

* [Methods](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods)

**Recommended content**

**[C# operators and expressions - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/)**

Learn about C# operators and expressions, operator precedence, and operator associativity.

**[Arithmetic operators - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators)**

Learn about C# operators that perform multiplication, division, remainder, addition, and subtraction operations with numeric types.

**[Floating-point numeric types - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types)**

Learn about the built-in C# floating-point types: float, double, and decimal

**[Boolean logical operators - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/boolean-logical-operators)**

Learn about C# operators that perform logical negation, conjunction (AND), and inclusive and exclusive disjunction (OR) operations with Boolean operands.

**[goto statement - C# Reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto)**

goto statement - C# Reference

**[Casting and type conversions - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions)**

Learn about casting and type conversions, such as implicit, explicit (casts), and user-defined conversions.

**[Integral numeric types - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types)**

Learn the range, storage size, and uses for each of the integral numeric types.

**[ref keyword - C# Reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref)**

ref keyword - C# Reference

Show less