# Indices and ranges

Article • 11/14/2023

Ranges and indices provide a succinct syntax for accessing single elements or ranges in a sequence.

In this tutorial, you'll learn how to:

- ✓ Use the syntax for ranges in a sequence.
- ✓ Implicitly define a Range.
- ✓ Understand the design decisions for the start and end of each sequence.
- ✓ Learn scenarios for the Index and Range types.

## Language support for indices and ranges

Indices and ranges provide a succinct syntax for accessing single elements or ranges in a sequence.

This language support relies on two new types and two new operators:

- System.Index represents an index into a sequence.
- The index from end operator ^, which specifies that an index is relative to the end of a sequence.
- System.Range represents a sub range of a sequence.
- The range operator .., which specifies the start and end of a range as its operands.

Let's start with the rules for indices. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. The expression `sequence[^0]` throws an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

C#

```csharp
private string[] words = [
                // index from start     index from end
    "first",    // 0                    ^10
    "second",   // 1                    ^9
    "third",    // 2                    ^8
    "fourth",   // 3                    ^7
    "fifth",    // 4                    ^6
    "sixth",    // 5                    ^5
    "seventh",  // 6                    ^4
    "eighth",   // 7                    ^3
    "ninth",    // 8                    ^2
    "tenth"      // 9                    ^1
];               // 10 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index. Add the following code below the initialization:

C#

```csharp
Console.WriteLine($"The last word is < {words[^1]} >."); // The last word is
< tenth >.
```

A range specifies the *start* and *end* of a range. The start of the range is inclusive, but the end of the range is exclusive, meaning the *start* is included in the range but the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

The following code creates a subrange with the words "second", "third", and "fourth". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range.

C#

```csharp
string[] secondThirdFourth = words[1..4]; // contains "second", "third" and
"fourth"

// < second >< third >< fourth >
foreach (var word in secondThirdFourth)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following code returns the range with "ninth" and "tenth". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included.

C#

```csharp
string[] lastTwo = words[^2..^0]; // contains "ninth" and "tenth"

// < ninth >< tenth >
foreach (var word in lastTwo)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following examples create ranges that are open ended for the start, end, or both:

C#

```
string[] allWords = words[..]; // contains "first" through "tenth".
string[] firstPhrase = words[..4]; // contains "first" through "fourth"
string[] lastPhrase = words[6..]; // contains "seventh", "eight", "ninth"
and "tenth"

// < first >< second >< third >< fourth >< fifth >< sixth >< seventh ><
eighth >< ninth >< tenth >
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();

// < first >< second >< third >< fourth >
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();

// < seventh >< eighth >< ninth >< tenth >
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

You can also declare ranges or indices as variables. The variable can then be used inside the `[` and `]` characters:

C#

```
Index thirdFromEnd = ^3;
Console.WriteLine($"< {words[thirdFromEnd]} > "); // < eighth >
Range phrase = 1..4;
string[] text = words[phrase];

// < second >< third >< fourth >
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following sample shows many of the reasons for those choices. Modify `x`, `y`, and `z` to try different combinations. When you experiment, use values where `x` is less than `y`, and `y` is less than `z` for valid combinations. Add the following code in a new method. Try different combinations:

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

Not only arrays support indices and ranges. You can also use indices and ranges with string, Span<T>, or ReadOnlySpan<T>.

## Implicit range operator expression conversions

When using the range operator expression syntax, the compiler implicitly converts the start and end values to an Index and from them, creates a new Range instance. The following code shows an example implicit conversion from the range operator expression syntax, and its corresponding explicit alternative:

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
```

```
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

ⓘ **Important**

Implicit conversions from Int32 to Index throw an ArgumentOutOfRangeException when the value is negative. Likewise, the `Index` constructor throws an `ArgumentOutOfRangeException` when the `value` parameter is negative.

## Type support for indices and ranges

Indexes and ranges provide clear, concise syntax to access a single element or a range of elements in a sequence. An index expression typically returns the type of the elements of a sequence. A range expression typically returns the same sequence type as the source sequence.

Any type that provides an indexer with an Index or Range parameter explicitly supports indices or ranges respectively. An indexer that takes a single Range parameter may return a different sequence type, such as System.Span<T>.

ⓘ **Important**

The performance of code using the range operator depends on the type of the sequence operand.

The time complexity of the range operator depends on the sequence type. For example, if the sequence is a `string` or an array, then the result is a copy of the specified section of the input, so the time complexity is *O(N)* (where N is the length of the range). On the other hand, if it's a System.Span<T> or a System.Memory<T>, the result references the same backing store, which means there is no copy and the operation is *O(1)*.

In addition to the time complexity, this causes extra allocations and copies, impacting performance. In performance sensitive code, consider using `Span<T>` or `Memory<T>` as the sequence type, since the range operator does not allocate for them.

A type is **countable** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. A countable type that doesn't explicitly support indices or ranges may provide an implicit support for them. For more information, see the Implicit

Index support and Implicit Range support sections of the feature proposal note. Ranges using implicit range support return the same sequence type as the source sequence.

For example, the following .NET types support both indices and ranges: String, Span<T>, and ReadOnlySpan<T>. The List<T> supports indices but doesn't support ranges.

Array has more nuanced behavior. Single dimension arrays support both indices and ranges. Multi-dimensional arrays don't support indexers or ranges. The indexer for a multi-dimensional array has multiple parameters, not a single parameter. Jagged arrays, also referred to as an array of arrays, support both ranges and indexers. The following example shows how to iterate a rectangular subsection of a jagged array. It iterates the section in the center, excluding the first and last three rows, and the first and last two columns from each selected row:

C#

```
int[][] jagged =
[
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
   [10,11,12,13,14,15,16,17,18,19],
   [20,21,22,23,24,25,26,27,28,29],
   [30,31,32,33,34,35,36,37,38,39],
   [40,41,42,43,44,45,46,47,48,49],
   [50,51,52,53,54,55,56,57,58,59],
   [60,61,62,63,64,65,66,67,68,69],
   [70,71,72,73,74,75,76,77,78,79],
   [80,81,82,83,84,85,86,87,88,89],
   [90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

In all cases, the range operator for Array allocates an array to store the elements returned.

## Scenarios for indices and ranges

You'll often use ranges and indices when you want to analyze a portion of a larger sequence. The new syntax is clearer in reading exactly what portion of the sequence is involved. The local function `MovingAverage` takes a Range as its argument. The method then enumerates just that range when calculating the min, max, and average. Try the following code in your project:

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: 
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: 
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range 
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

## A Note on Range Indices and Arrays

When taking a range from an array, the result is an array that is copied from the initial array, rather than referenced. Modifying values in the resulting array will not change values in the initial array.

For example:

C#

```csharp
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

## See also

- Member access operators and expressions