C# operators and expressions

Article • 12/02/2022 • 5 minutes to read

C# provides a number of operators. Many of them are supported by the built-in types and allow you to perform basic operations with values of those types. Those operators include the following groups:

- Arithmetic operators that perform arithmetic operations with numeric operands
- Comparison operators that compare numeric operands
- Boolean logical operators that perform logical operations with bool operands
- Bitwise and shift operators that perform bitwise or shift operations with operands of the integral types
- Equality operators that check if their operands are equal or not

Typically, you can overload those operators, that is, specify the operator behavior for the operands of a user-defined type.

The simplest C# expressions are literals (for example, integer and real numbers) and names of variables. You can combine them into complex expressions by using operators. Operator precedence and associativity determine the order in which the operations in an expression are performed. You can use parentheses to change the order of evaluation imposed by operator precedence and associativity.

In the following code, examples of expressions are at the right-hand side of assignments:

C#

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

Typically, an expression produces a result and can be included in another expression. A void method call is an example of an expression that doesn't produce a result. It can be used only as a statement, as the following example shows:

```
C#

Console.WriteLine("Hello, world!");
```

Here are some other kinds of expressions that C# provides:

- Interpolated string expressions that provide convenient syntax to create formatted strings:

```
C#

var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r *
r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- Lambda expressions that allow you to create anonymous functions:

```
C#

int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- Query expressions that allow you to use query capabilities directly in C#:

```
C#

var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

You can use an expression body definition to provide a concise definition for a method, constructor, property, indexer, or finalizer.

Operator precedence

In an expression with multiple operators, the operators with higher precedence are evaluated before the operators with lower precedence. In the following example, the multiplication is performed first because it has higher precedence than addition:

C#

```
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Use parentheses to change the order of evaluation imposed by operator precedence:

C#

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

The following table lists the C# operators starting with the highest precedence to the lowest. The operators within each row have the same precedence.

Operators	Category or name
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
switch, with	`switch` and `with` expressions
x * y, x / y, x % y	Multiplicative
x + y, x – y	Additive
x << y, x >> y, x >>> y	Shift

Operators	Category or name
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
`x & y`	Boolean logical AND or bitwise logical AND
`x ^ y`	Boolean logical XOR or bitwise logical XOR
`x | y`	Boolean logical OR or bitwise logical OR
x && y	Conditional AND
x || y	Conditional OR
x ?? y	Null-coalescing operator
c ? t : f	Conditional operator
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x >>>= y, x ??= y, =>	Assignment and lambda declaration

Operator associativity

When operators have the same precedence, associativity of the operators determines the order in which the operations are performed:

- *Left-associative* operators are evaluated in order from left to right. Except for the assignment operators and the null-coalescing operators, all binary operators are left-associative. For example, `a + b - c` is evaluated as `(a + b) - c`.
- *Right-associative* operators are evaluated in order from right to left. The assignment operators, the null-coalescing operators, lambdas, and the conditional operator ?: are right-associative. For example, `x = y = z` is evaluated as `x = (y = z)`.

> **Important**
>
> In an expression of the form `P?.A0?.A1`, if `P` is `null`, neither `A0` nor `A1` are evaluated. Similarly, in an expression of the form `P?.A0.A1`, because `A0` isn't evaluated when `P` is

null, neither is `A0.A1`. See the **C# language specification** for more details.

Use parentheses to change the order of evaluation imposed by operator associativity:

C#

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Operand evaluation

Unrelated to operator precedence and associativity, operands in an expression are evaluated from left to right. The following examples demonstrate the order in which operators and operands are evaluated:

Expression	Order of evaluation
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

Typically, all operator operands are evaluated. However, some operators evaluate operands conditionally. That is, the value of the leftmost operand of such an operator defines if (or which) other operands should be evaluated. These operators are the conditional logical AND (&&) and OR (||) operators, the null-coalescing operators ?? and ??=, the null-conditional operators ?. and ?[], and the conditional operator ?:. For more information, see the description of each operator.

C# language specification

For more information, see the following sections of the C# language specification:

- Expressions
- Operators

See also

- C# reference
- Operator overloading
- Expression trees