Arithmetic operators (C# reference)

Article • 10/13/2022 • 11 minutes to read

The following operators perform arithmetic operations with operands of numeric types:

- Unary ++ (increment), -- (decrement), + (plus), and - (minus) operators
- Binary * (multiplication), / (division), % (remainder), + (addition), and - (subtraction) operators

Those operators are supported by all integral and floating-point numeric types.

In the case of integral types, those operators (except the `++` and `--` operators) are defined for the `int`, `uint`, `long`, and `ulong` types. When operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral or floating-point types, their values are converted to the closest containing type, if such a type exists. For more information, see the Numeric promotions section of the C# language specification. The `++` and `--` operators are defined for all integral and floating-point numeric types and the char type. The result type of a compound assignment expression is the type of the left-hand operand.

Increment operator ++

The unary increment operator `++` increments its operand by 1. The operand must be a variable, a property access, or an indexer access.

The increment operator is supported in two forms: the postfix increment operator, `x++`, and the prefix increment operator, `++x`.

Postfix increment operator

The result of `x++` is the value of `x` *before* the operation, as the following example shows:

C#

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Prefix increment operator

The result of `++x` is the value of `x` *after* the operation, as the following example shows:

```
C#

double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Decrement operator --

The unary decrement operator `--` decrements its operand by 1. The operand must be a variable, a property access, or an indexer access.

The decrement operator is supported in two forms: the postfix decrement operator, `x--`, and the prefix decrement operator, `--x`.

Postfix decrement operator

The result of `x--` is the value of `x` *before* the operation, as the following example shows:

```
C#

int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Prefix decrement operator

The result of `--x` is the value of `x` *after* the operation, as the following example shows:

```
C#

double a = 1.5;
Console.WriteLine(a);   // output: 1.5
```

```
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Unary plus and minus operators

The unary `+` operator returns the value of its operand. The unary `-` operator computes the numeric negation of its operand.

C#

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

The ulong type doesn't support the unary `-` operator.

Multiplication operator *

The multiplication operator `*` computes the product of its operands:

C#

```
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

The unary `*` operator is the pointer indirection operator.

Division operator /

The division operator `/` divides its left-hand operand by its right-hand operand.

Integer division

For the operands of integer types, the result of the `/` operator is of an integer type and equals the quotient of the two operands rounded towards zero:

```
C#

Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

To obtain the quotient of the two operands as a floating-point number, use the `float`, `double`, or `decimal` type:

```
C#

Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

Floating-point division

For the `float`, `double`, and `decimal` types, the result of the `/` operator is the quotient of the two operands:

```
C#

Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

If one of the operands is `decimal`, another operand can be neither `float` nor `double`, because neither `float` nor `double` is implicitly convertible to `decimal`. You must explicitly convert the `float` or `double` operand to the `decimal` type. For more information about conversions between numeric types, see Built-in numeric conversions.

Remainder operator %

The remainder operator `%` computes the remainder after dividing its left-hand operand by its right-hand operand.

Integer remainder

For the operands of integer types, the result of `a % b` is the value produced by `a - (a / b) * b`. The sign of the non-zero remainder is the same as the sign of the left-hand operand, as the following example shows:

C#

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Use the Math.DivRem method to compute both integer division and remainder results.

Floating-point remainder

For the `float` and `double` operands, the result of `x % y` for the finite `x` and `y` is the value `z` such that

- The sign of `z`, if non-zero, is the same as the sign of `x`.
- The absolute value of `z` is the value produced by `|x| - n * |y|` where `n` is the largest possible integer that is less than or equal to `|x| / |y|` and `|x|` and `|y|` are the absolute values of `x` and `y`, respectively.

> **Note**
>
> This method of computing the remainder is analogous to that used for integer operands, but different from the IEEE 754 specification. If you need the remainder operation that complies with the IEEE 754 specification, use the **Math.IEEERemainder** method.

For information about the behavior of the `%` operator with non-finite operands, see the Remainder operator section of the C# language specification.

For the `decimal` operands, the remainder operator `%` is equivalent to the remainder operator of the System.Decimal type.

The following example demonstrates the behavior of the remainder operator with floating-point operands:

```
C#

Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Addition operator +

The addition operator `+` computes the sum of its operands:

```
C#

Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

You can also use the `+` operator for string concatenation and delegate combination. For more information, see the + and += operators article.

Subtraction operator -

The subtraction operator `-` subtracts its right-hand operand from its left-hand operand:

```
C#

Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

You can also use the `-` operator for delegate removal. For more information, see the - and -= operators article.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
C#

x op= y
```

is equivalent to

```
C#

x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with arithmetic operators:

```
C#

int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

Because of numeric promotions, the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

```
C#
```

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

You also use the `+=` and `-=` operators to subscribe to and unsubscribe from an event, respectively. For more information, see How to subscribe to and unsubscribe from events.

Operator precedence and associativity

The following list orders arithmetic operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators
- Prefix increment `++x` and decrement `--x` and unary `+` and `-` operators
- Multiplicative `*`, `/`, and `%` operators
- Additive `+` and `-` operators

Binary arithmetic operators are left-associative. That is, operators with the same precedence level are evaluated from left to right.

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence and associativity.

C#

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

Arithmetic overflow and division by zero

When the result of an arithmetic operation is outside the range of possible finite values of the involved numeric type, the behavior of an arithmetic operator depends on the type of its operands.

Integer arithmetic overflow

Integer division by zero always throws a DivideByZeroException.

If integer arithmetic overflow occurs, the overflow-checking context, which can be checked or unchecked, controls the resulting behavior:

- In a checked context, if overflow happens in a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run time, an OverflowException is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that don't fit in the destination type.

Along with the checked and unchecked statements, you can use the `checked` and `unchecked` operators to control the overflow-checking context, in which an expression is evaluated:

C#

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

By default, arithmetic operations occur in an *unchecked* context.

Floating-point arithmetic overflow

Arithmetic operations with the `float` and `double` types never throw an exception. The result of arithmetic operations with those types can be one of special values that represent infinity and not-a-number:

C#

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

For the operands of the `decimal` type, arithmetic overflow always throws an OverflowException. Division by zero always throws a DivideByZeroException.

Round-off errors

Because of general limitations of the floating-point representation of real numbers and floating-point arithmetic, round-off errors might occur in calculations with floating-point types. That is, the produced result of an expression might differ from the expected mathematical result. The following example demonstrates several such cases:

C#

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.99999999999999999999999999
```

For more information, see remarks at the System.Double, System.Single, or System.Decimal reference pages.

Operator overloadability

A user-defined type can overload the unary (`++`, `--`, `+`, and `-`) and binary (`*`, `/`, `%`, `+`, and `-`) arithmetic operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type can't explicitly overload a compound assignment operator.

User-defined checked operators

Beginning with C# 11, when you overload an arithmetic operator, you can use the `checked` keyword to define the *checked* version of that operator. The following example shows how to do that:

C#

```
public record struct Point(int X, int Y)
{
    public static Point operator checked +(Point left, Point right)
    {
        checked
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

When you define a checked operator, you must also define the corresponding operator without the `checked` modifier. The checked operator is called in a checked context; the operator without the `checked` modifier is called in an unchecked context. If you only provide the operator without the `checked` modifier, it's called in both a `checked` and `unchecked` context.

When you define both versions of an operator, it's expected that their behavior differs only when the result of an operation is too large to represent in the result type as follows:

- A checked operator throws an OverflowException.

- An operator without the `checked` modifier returns an instance representing a *truncated* result.

For information about the difference in behavior of the built-in arithmetic operators, see the Arithmetic overflow and division by zero section.

You can use the `checked` modifier only when you overload any of the following operators:

- Unary `++`, `--`, and `-` operators
- Binary `*`, `/`, `+`, and `-` operators
- Explicit conversion operators

> **Note**
>
> The overflow-checking context within the body of a checked operator is not affected by the presence of the `checked` modifier. The default context is defined by the value of the **CheckForOverflowUnderflow** compiler option. Use the **checked and unchecked statements** to explicitly specify the overflow-checking context, as the example at the beginning of this section demonstrates.

C# language specification

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

- Postfix increment and decrement operators
- Prefix increment and decrement operators
- Unary plus operator
- Unary minus operator
- Multiplication operator
- Division operator
- Remainder operator
- Addition operator
- Subtraction operator
- Compound assignment
- The checked and unchecked operators
- Numeric promotions

See also

- C# reference
- C# operators and expressions
- System.Math
- System.MathF
- Numerics in .NET