Contents

>= Operator

+= Operator

-= Operator

*= Operator

/= Operator

%= Operator

&= Operator

|= Operator

^= Operator

<<= Operator

>>= Operator

-> Operator

?? Operator

=> Operator

C# operators

1/16/2019 • 7 minutes to read • Edit Online

C# provides many operators, which are symbols that specify which operations (math, indexing, function call, etc.) to perform in an expression. You can overload many operators to change their meaning when applied to a user-defined type.

Operations on integral types (such as `==`, `!=`, `<`, `>`, `&`, `|`) are generally allowed on enumeration (`enum`) types.

The sections below list the C# operators starting with the highest precedence to the lowest. The operators within each section share the same precedence level.

Primary operators

These are the highest precedence operators.

x.y – member access.

x?.y – null conditional member access. Returns `null` if the left-hand operand evaluates to `null`.

x?[y] - null conditional index access. Returns `null` if the left-hand operand evaluates to `null`.

f(x) – function invocation.

a[x] – aggregate object indexing.

x++ – postfix increment. Returns the value of x and then updates the storage location with the value of x that is one greater (typically adds the integer 1).

x-- – postfix decrement. Returns the value of x and then updates the storage location with the value of x that is one less (typically subtracts the integer 1).

new – type instantiation.

typeof – returns the Type object representing the operand.

checked – enables overflow checking for integer operations.

unchecked – disables overflow checking for integer operations. This is the default compiler behavior.

default(T) – produces the default value of type T.

delegate – declares and returns a delegate instance.

sizeof – returns the size in bytes of the type operand.

-> – pointer dereferencing combined with member access.

Unary operators

These operators have higher precedence than the next section and lower precedence than the previous section.

+x – returns the value of x.

-x – numeric negation.

!x – logical negation.

~x – bitwise complement.

++x – prefix increment. Returns the value of x after updating the storage location with the value of x that is one greater (typically adds the integer 1).

--x – prefix decrement. Returns the value of x after updating the storage location with the value of x that is one less (typically subtracts the integer 1).

(T)x – type casting.

await – awaits a `Task`.

&x – address of.

*x – dereferencing.

Multiplicative operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x * y – multiplication.

x / y – division. If the operands are integers, the result is an integer truncated toward zero (for example, `-7 / 2 is -3`).

x % y – remainder. If the operands are integers, this returns the remainder of dividing x by y. If `q = x / y` and `r = x % y`, then `x = q * y + r`.

Additive operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x + y – addition.

x – y – subtraction.

Shift operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x << y – shift bits left and fill with zero on the right.

x >> y – shift bits right. If the left operand is `int` or `long`, then left bits are filled with the sign bit. If the left operand is `uint` or `ulong`, then left bits are filled with zero.

Relational and type-testing operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x < y – less than (true if x is less than y).

x > y – greater than (true if x is greater than y).

x <= y – less than or equal to.

x >= y – greater than or equal to.

is – type compatibility. Returns true if the evaluated left operand can be cast to the type specified in the right operand (a static type).

as – type conversion. Returns the left operand cast to the type specified by the right operand (a static type), but `as` returns `null` where `(T)x` would throw an exception.

Equality operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x == y – equality. By default, for reference types other than `string`, this returns reference equality (identity test). However, types can overload `==`, so if your intent is to test identity, it is best to use the `ReferenceEquals` method on `object`.

x != y – not equal. See comment for `==`. If a type overloads `==`, then it must overload `!=`.

Logical AND operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x & y – logical or bitwise AND. You can generally use this with integer types and `enum` types.

Logical XOR operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x ^ y – logical or bitwise XOR. You can generally use this with integer types and `enum` types.

Logical OR operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x | y – logical or bitwise OR. You can generally use this with integer types and `enum` types.

Conditional AND operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x && y – logical AND. If the first operand evaluates to false, then C# does not evaluate the second operand.

Conditional OR operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x || y – logical OR. If the first operand evaluates to true, then C# does not evaluate the second operand.

Null-coalescing operator

This operator has higher precedence than the next section and lower precedence than the previous section.

x ?? y – returns `x` if it is non-`null`; otherwise, returns `y`.

Conditional operator

This operator has higher precedence than the next section and lower precedence than the previous section.

t ? x : y – if test `t` evaluates to true, then evaluate and return `x`; otherwise, evaluate and return `y`.

Assignment and Lambda operators

These operators have higher precedence than the next section and lower precedence than the previous section.

x = y – assignment.

x += y – increment. Add the value of `y` to the value of `x`, store the result in `x`, and return the new value. If `x` designates an `event`, then `y` must be an appropriate function that C# adds as an event handler.

x -= y – decrement. Subtract the value of `y` from the value of `x`, store the result in `x`, and return the new value. If `x` designates an `event`, then `y` must be an appropriate function that C# removes as an event handler

x *= y – multiplication assignment. Multiply the value of `y` to the value of `x`, store the result in `x`, and return the new value.

x /= y – division assignment. Divide the value of `x` by the value of `y`, store the result in `x`, and return the new value.

x %= y – remainder assignment. Divide the value of `x` by the value of `y`, store the remainder in `x`, and return the new value.

x &= y – AND assignment. AND the value of `y` with the value of `x`, store the result in `x`, and return the new value.

x |= y – OR assignment. OR the value of `y` with the value of `x`, store the result in `x`, and return the new value.

x ^= y – XOR assignment. XOR the value of `y` with the value of `x`, store the result in `x`, and return the new value.

x <<= y – left-shift assignment. Shift the value of `x` left by `y` places, store the result in `x`, and return the new value.

x >>= y – right-shift assignment. Shift the value of `x` right by `y` places, store the result in `x`, and return the new value.

=> – lambda declaration.

Arithmetic overflow

The arithmetic operators (+, -, *, /) can produce results that are outside the range of possible values for the numeric type involved. You should refer to the section on a particular operator for details, but in general:

- Integer arithmetic overflow either throws an OverflowException or discards the most significant bits of the result. Integer division by zero always throws a DivideByZeroException.

  When integer overflow occurs, what happens depends on the execution context, which can be checked or unchecked. In a checked context, an OverflowException is thrown. In an unchecked context, the most significant bits of the result are discarded and execution continues. Thus, C# gives you the choice of handling or ignoring overflow. By default, arithmetic operations occur in an *unchecked* context.

  In addition to the arithmetic operations, integral-type to integral-type casts can cause overflow (such as when you cast a long to an int), and are subject to checked or unchecked execution. However, bitwise operators and shift operators never cause overflow.

- Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).
- Decimal arithmetic overflow always throws an OverflowException. Decimal division by zero always throws a DivideByZeroException.

See also

- C# Reference
- C# Programming Guide
- C#
- Overloadable Operators
- C# Keywords

[] operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

Square brackets, `[]`, are typically used for array, indexer, or pointer element access.

For more information about pointer element access, see How to: access an array element with a pointer.

You also use square brackets to specify attributes:

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Array access

The following example demonstrates how to access array elements:

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

If an array index is outside the bounds of the corresponding dimension of an array, an IndexOutOfRangeException is thrown.

As the preceding example shows, you also use square brackets in declaration of an array type and instantiation of array instances.

For more information about arrays, see Arrays.

Indexer access

The following example uses .NET Dictionary<TKey,TValue> type to demonstrate indexer access:

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);
```

Indexers allow you to index instances of a user-defined type in the similar way as array indexing. Unlike array indices, which must be integer, the indexer arguments can be declared to be of any type.

For more information about indexers, see Indexers.

Operator overloadability

Element access `[]` is not considered an overloadable operator. Use indexers to support indexing with user-defined types.

C# language specification

For more information, see the Element access and Pointer element access sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Arrays
- Indexers
- Pointer types
- Attributes

() operator (C# Reference)

1/17/2019 • 2 minutes to read • Edit Online

Parentheses, `()`, are typically used for method or delegate invocation or in cast expressions.

You also use parentheses to specify the order in which to evaluate operations in an expression. For more information, see the Adding parentheses section of the Operators article. For the list of operators ordered by precedence level, see C# operators.

Method invocation

The following example demonstrates how to invoke a method, with or without arguments, and a delegate:

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

You also use parentheses when you invoke a constructor with a `new` operator.

For more information about methods, see Methods. For more information about delegates, see Delegates.

Cast expression

A cast expression of the form `(T)E` invokes a conversion operator to convert the value of expression `E` to type `T`. If no explicit conversion exists from the type of `E` to type `T`, a compile-time error occurs. For information about how to define a conversion operator, see the explicit and implicit keyword articles.

The following example demonstrates type conversion between numeric types:

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234
```

For more information about predefined explicit conversions between numeric types, see Explicit numeric conversions table.

For more information, see Casting and type conversions and Conversion operators.

Operator overloadability

The operator `()` cannot be overloaded.

C# language specification

For more information, see the Invocation expressions and Cast expressions sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators

. operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The dot operator (`.`) is used for member access. The dot operator specifies a member of a type or namespace. For example, the dot operator is used to access specific methods within the .NET Framework class libraries:

```
// The class Console in namespace System:
System.Console.WriteLine("hello");
```

For example, consider the following class:

```
class Simple
{
    public int a;
    public void b()
    {
    }
}
```

```
Simple s = new Simple();
```

The variable `s` has two members, `a` and `b`; to access them, use the dot operator:

```
s.a = 6;   // assign to field a;
s.b();     // invoke member function b;
```

The dot is also used to form qualified names, which are names that specify the namespace or interface, for example, to which they belong.

```
// The class Console in namespace System:
System.Console.WriteLine("hello");
```

The using directive makes some name qualification optional:

```
namespace ExampleNS
{
    using System;
    class C
    {
        void M()
        {
            System.Console.WriteLine("hello");
            Console.WriteLine("hello");   // Same as previous line.
        }
    }
}
```

But when an identifier is ambiguous, it must be qualified:

```
namespace Example2
{
    class Console
    {
        public static void WriteLine(string s){}
    }
}
namespace Example1
{
    using System;
    using Example2;
    class C
    {
        void M()
        {
            // Console.WriteLine("hello");   // Compiler error. Ambiguous reference.
            System.Console.WriteLine("hello"); //OK
            Example2.Console.WriteLine("hello"); //OK
        }
    }
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Operators

?. and ?[] null-conditional operators (C# and Visual Basic)

1/16/2019 • 2 minutes to read • Edit Online

Tests the value of the left-hand operand for null before performing a member access (`?.`) or index (`?[]`) operation; returns `null` if the left-hand operand evaluates to `null` .

These operators help you write less code to handle null checks, especially for descending into data structures.

```
int? length = customers?.Length; // null if customers is null
Customer first = customers?[0];  // null if customers is null
int? count = customers?[0]?.Orders?.Count();  // null if customers, the first customer, or Orders is null
```

The null-conditional operators are short-circuiting. If one operation in a chain of conditional member access and index operations returns null, the rest of the chain's execution stops. In the following example, `E` doesn't execute if `A` , `B` , or `C` evaluates to null.

```
A?.B?.C?.Do(E);
A?.B?.C?[E];
```

Another use for the null-conditional member access is invoking delegates in a thread-safe way with much less code. The old way requires code like the following:

```
var handler = this.PropertyChanged;
if (handler != null)
    handler(…);
```

The new way is much simpler:

```
PropertyChanged?.Invoke(…)
```

The new way is thread-safe because the compiler generates code to evaluate `PropertyChanged` one time only, keeping the result in a temporary variable. You need to explicitly call the `Invoke` method because there is no null-conditional delegate invocation syntax `PropertyChanged?(e)` .

Language specifications

For more information, see Null-conditional operator in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- ?? (null-coalescing operator)
- C# Reference
- C# Programming Guide

:: operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The namespace alias qualifier (`::`) is used to look up identifiers. It is always positioned between two identifiers, as in this example:

```
global::System.Console.WriteLine("Hello World");
```

The `::` operator can also be used with a *using alias directive*:

```
// using Col=System.Collections.Generic;
var numbers = new Col::List<int> { 1, 2, 3 };
```

Remarks

The namespace alias qualifier can be `global` . This invokes a lookup in the global namespace, rather than an aliased namespace.

For more information

For an example of how to use the `::` operator, see the following section:

- How to: Use the Global Namespace Alias

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# operators
- Namespace Keywords
- . operator
- extern alias

+ Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The `+` operator is supported in two forms: a unary plus operator or a binary addition operator.

Unary plus operator

The unary `+` operator returns the value of its operand. It's supported by all numeric types.

Numeric addition

For numeric types, the `+` operator computes the sum of its operands:

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

String concatenation

When one or both operands are of type string, the `+` operator concatenates the string representations of its operands:

```
Console.WriteLine("Forgot " + " white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
// Output:
// Forgot white space
// Probably the oldest constant: 3.14159265358979
```

Starting with C# 6, string interpolation provides a more convenient way to format strings:

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Delegate combination

For delegate types, the `+` operator returns a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. If any of the operands is `null`, the `+` operator returns the value of another operand (which also might be `null`). The following example shows how delegates can be combined with the `+` operator:

```
Action<int> printDouble = (int s) => Console.WriteLine(2 * s);
Action<int> printTriple = (int s) => Console.WriteLine(3 * s);
Action<int> combined = printDouble + printTriple;
combined(5);
// Output:
// 10
// 15
```

For more information about delegate types, see Delegates.

Operator overloadability

User-defined types can overload the unary and binary `+` operators. When a binary `+` operator is overloaded, the addition assignment operator `+=` is also implicitly overloaded.

C# language specification

For more information, see the Unary plus operator and Addition operator sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- String interpolation
- How to: Concatenate Multiple Strings
- Delegates
- Checked and unchecked

- operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The `-` operator can function as either a unary or a binary operator.

Remarks

Unary `-` operators are predefined for all numeric types. The result of a unary `-` operation on a numeric type is the numeric negation of the operand.

Binary `-` operators are predefined for all numeric and enumeration types to subtract the second operand from the first.

Delegate types also provide a binary `-` operator, which performs delegate removal.

User-defined types can overload the unary `-` and binary `-` operators. For more information, see operator keyword.

Example

```
class MinusLinus
{
    static void Main()
    {
        int a = 5;
        Console.WriteLine(-a);
        Console.WriteLine(a - 1);
        Console.WriteLine(a - .5);
    }
}
/*
Output:
-5
4
4.5
*/
```

See also

- C# Reference
- C# Programming Guide
- C# operators

* operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The multiplication operator (`*`) computes the product of its operands. All numeric types have predefined multiplication operators.

`*` also serves as the dereference operator, which allows reading and writing to a pointer.

Remarks

The `*` operator is also used to declare pointer types and to dereference pointers. This operator can only be used in unsafe contexts, denoted by the use of the unsafe keyword, and requiring the /unsafe compiler option. The dereference operator is also known as the indirection operator.

User-defined types can overload the binary `*` operator (see operator). When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
class Multiply
{
    static void Main()
    {
        Console.WriteLine(5 * 2);
        Console.WriteLine(-.5 * .2);
        Console.WriteLine(-.5m * .2m); // decimal type
    }
}
    /*
Output
    10
    -0.1
    -0.10
*/
```

Example

```
public class Pointer
{
    unsafe static void Main()
    {
        int i = 5;
        int* j = &i;
        System.Console.WriteLine(*j);
    }
}
/*
Output:
5
*/
```

See also

- C# Reference
- C# Programming Guide
- Unsafe Code and Pointers
- C# Operators

/ Operator (C# Reference)

12/12/2018 • 2 minutes to read • Edit Online

The division operator `/` divides its first operand by its second operand. All numeric types support the division operator.

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:

```
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:

```
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:

```
Console.WriteLine(16.8f / 4.1f);
Console.WriteLine(16.8d / 4.1d);
Console.WriteLine(16.8m / 4.1m);
// Output:
// 4.097561
// 4.09756097560976
// 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 implicit conversions between numeric types, see Implicit numeric conversions table.

Operator overloadability

User-defined types can overload the `/` operator. When the `/` operator is overloaded, the division assignment operator `/=` is also implicitly overloaded.

C# language specification

For more information, see the Division operator section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- % Operator

2 minutes to read

& Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The `&` operator is supported in two forms: a unary address-of operator or a binary logical operator.

Unary address-of operator

The unary `&` operator returns the address of its operand. For more information, see How to: obtain the address of a variable.

The address-of operator `&` requires unsafe context.

Integer logical bitwise AND operator

For integer types, the `&` operator computes the logical bitwise AND of its operands:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1111;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

> **NOTE**
> The preceding example uses the binary literals introduced in C# 7.0 and enhanced in C# 7.2.

Because operations on integer types are generally allowed on enumeration types, the `&` operator also supports enum operands.

Boolean logical AND operator

For bool operands, the `&` operator computes the logical AND of its operands. The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The `&` operator evaluates both operands even if the first operand evaluates to `false`, so that the result must be `false` regardless of the value of the second operand. The following example demonstrates that behavior:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool test = false & SecondOperand();
Console.WriteLine(test);
// Output:
// Second operand is evaluated.
// False
```

The conditional AND operator `&&` also computes the logical AND of its operands, but evaluates the second operand only if the first operand evaluates to `true`.

For nullable bool operands, the behavior of the `&` operator is consistent with SQL's three-valued logic. For more information, see the The bool? type section of the Using nullable types article.

Operator overloadability

User-defined types can overload the binary `&` operator. When a binary `&` operator is overloaded, the AND assignment operator `&=` is also implicitly overloaded.

C# language specification

For more information, see The address-of operator and Logical operators sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Pointer types
- | operator
- ^ operator
- ~ operator
- && operator

| operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

Binary `|` operators are predefined for the integral types and `bool`. For integral types, `|` computes the bitwise OR of its operands. For `bool` operands, `|` computes the logical OR of its operands; that is, the result is `false` if and only if both its operands are `false`.

Remarks

The binary `|` operator evaluates both operands regardless of the first one's value, in contrast to the conditional-OR operator `||`.

User-defined types can overload the `|` operator (see operator).

Example

```
class OR
{
    static void Main()
    {
        Console.WriteLine(true | false);  // logical or
        Console.WriteLine(false | false); // logical or
        Console.WriteLine("0x{0:x}", 0xf8 | 0x3f);   // bitwise or
    }
}
/*
Output:
True
False
0xff
*/
```

See also

- C# Reference
- C# Programming Guide
- C# operators

^ operator (C# Reference)

1/23/2019 • 2 minutes to read • Edit Online

Binary `^` operators are predefined for the integral types and `bool`. For integral types, `^` computes the bitwise exclusive-OR of its operands. For `bool` operands, `^` computes the logical exclusive-or of its operands; that is, the result is `true` if and only if exactly one of its operands is `true`.

Remarks

User-defined types can overload the `^` operator (see operator). Operations on integral types are generally allowed on enumeration.

Example

```
class XOR
{
    static void Main()
    {
        // Logical exclusive-OR

        // When one operand is true and the other is false, exclusive-OR
        // returns True.
        Console.WriteLine(true ^ false);
        // When both operands are false, exclusive-OR returns False.
        Console.WriteLine(false ^ false);
        // When both operands are true, exclusive-OR returns False.
        Console.WriteLine(true ^ true);


        // Bitwise exclusive-OR

        // Bitwise exclusive-OR of 0 and 1 returns 1.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0x0 ^ 0x1, 2));
        // Bitwise exclusive-OR of 0 and 0 returns 0.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0x0 ^ 0x0, 2));
        // Bitwise exclusive-OR of 1 and 1 returns 0.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0x1 ^ 0x1, 2));

        // With more than one digit, perform the exclusive-OR column by column.
        //    10
        //    11
        //    --
        //    01
        // Bitwise exclusive-OR of 10 (2) and 11 (3) returns 01 (1).
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0x2 ^ 0x3, 2));

        // Bitwise exclusive-OR of 101 (5) and 011 (3) returns 110 (6).
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0x5 ^ 0x3, 2));

        // Bitwise exclusive-OR of 1111 (decimal 15, hexadecimal F) and 0101 (5)
        // returns 1010 (decimal 10, hexadecimal A).
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0xf ^ 0x5, 2));

        // Finally, bitwise exclusive-OR of 11111000 (decimal 248, hexadecimal F8)
        // and 00111111 (decimal 63, hexadecimal 3F) returns 11000111, which is
        // 199 in decimal, C7 in hexadecimal.
        Console.WriteLine("Bitwise result: {0}", Convert.ToString(0xf8 ^ 0x3f, 2));
    }
}
/*
Output:
True
False
False
Bitwise result: 1
Bitwise result: 0
Bitwise result: 0
Bitwise result: 1
Bitwise result: 110
Bitwise result: 1010
Bitwise result: 11000111
*/
```

The computation of `0xf8 ^ 0x3f` in the previous example performs a bitwise exclusive-OR of the following two binary values, which correspond to the hexadecimal values F8 and 3F:

`1111 1000`

`0011 1111`

The result of the exclusive-OR is `1100 0111`, which is C7 in hexadecimal.

See also

- C# Reference
- C# Programming Guide
- C# operators

! operator (C# Reference)

2/15/2019 • 2 minutes to read • Edit Online

The logical negation operator `!` is a unary operator that computes logical negation of its bool operand. That is, it produces `true`, if the operand is `false`, and `false`, if the operand is `true`:

```
bool passed = false;
Console.WriteLine(!passed);   // output: True
Console.WriteLine(!true);     // output: False
```

Operator overloadability

User-defined types can overload the `!` operator.

C# language specification

For more information, see the Logical negation operator section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators

~ Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The bitwise complement operator `~` is a unary operator that produces a bitwise complement of its operand by reversing each bit. All integer types support the `~` operator.

> **NOTE**
> The `~` symbol is also used to declare finalizers. For more information, see Finalizers.

The following example demonstrates the usage of the `~` operator:

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

> **NOTE**
> The preceding example uses the binary literals introduced in C# 7.0 and enhanced in C# 7.2.

Operator overloadability

User-defined types can overload the `~` operator.

C# language specification

For more information, see the Bitwise complement operator section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Finalizers
- & operator
- | operator
- ^ operator

= Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The assignment operator `=` assigns the value of its right-hand operand to a variable, a property, or an indexer element given by its left-hand operand. The result of an assignment expression is the value assigned to the left-hand operand. The type of the right-hand operand must be the same as the type of the left-hand operand or implicitly convertible to it.

The assignment operator is right-associative, that is, an expression of the form

```
a = b = c
```

is evaluated as

```
a = (b = c)
```

The following example demonstrates the usage of the assignment operator to assign values to a local variable, a property, and an indexer element:

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

ref assignment operator

Beginning with C# 7.3, you can use the ref assignment operator `= ref` to reassign a ref local or ref readonly local variable. The following example demonstrates the usage of the ref assignment operator:

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

In the case of the ref assignment operator, the type of the left operand and the right operand must be the same.

For more information, see the feature proposal note.

Operator overloadability

A user-defined type cannot overload the assignment operator. However, a user-defined type can define an implicit conversion to another type. That way, the value of a user-defined type can be assigned to a variable, a property, or an indexer element of another type. For more information, see the implicit keyword article.

C# language specification

For more information, see the Simple assignment section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- ref keyword

< Operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The "less than" relational operator `<` returns `true` if its first operand is less than its second operand, `false` otherwise. All numeric and enumeration types support the `<` operator. For operands of the same enum type, the corresponding values of the underlying integral type are compared.

> **NOTE**
>
> For relational operators `==`, `>`, `<`, `>=`, and `<=`, if any of the operands is not a number (Double.NaN or Single.NaN) the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value. For more information and examples, see the Double.NaN or Single.NaN reference article.

The following example demonstrates the usage of the `<` operator:

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Operator overloadability

User-defined types can overload the `<` operator. If a type overloads the "less than" operator `<`, it must also overload the "greater than" operator `>`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- <= Operator
- System.IComparable<T>

> Operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The "greater than" relational operator `>` returns `true` if its first operand is greater than its second operand, `false` otherwise. All numeric and enumeration types support the `>` operator. For operands of the same enum type, the corresponding values of the underlying integral type are compared.

> **NOTE**
> For relational operators `==`, `>`, `<`, `>=`, and `<=`, if any of the operands is not a number (Double.NaN or Single.NaN) the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value. For more information and examples, see the Double.NaN or Single.NaN reference article.

The following example demonstrates the usage of the `>` operator:

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Operator overloadability

User-defined types can overload the `>` operator. If a type overloads the "greater than" operator `>`, it must also overload the "less than" operator `<`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- >= Operator
- System.IComparable<T>

?: Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The conditional operator `?:`, commonly known as the ternary conditional operator, evaluates a Boolean expression, and returns the result of evaluating one of two expressions, depending on whether the Boolean expression evaluates to `true` or `false`. Beginning with C# 7.2, the conditional ref expression returns the reference to the result of one of the two expressions.

The syntax for the conditional operator is as follows:

```
condition ? consequence : alternative
```

The `condition` expression must evaluate to `true` or `false`. If `condition` evaluates to `true`, the `consequence` expression is evaluated, and its result becomes the result of the operation. If `condition` evaluates to `false`, the `alternative` expression is evaluated, and its result becomes the result of the operation. Only `consequence` or `alternative` is evaluated.

The type of `consequence` and `alternative` must be the same, or there must be an implicit conversion from one type to the other.

The conditional operator is right-associative, that is, an expression of the form

```
a ? b : c ? d : e
```

is evaluated as

```
a ? b : (c ? d : e)
```

The following example demonstrates the usage of the conditional operator:

```
double sinc(double x) => x != 0.0 ? Math.Sin(x) / x : 1;

Console.WriteLine(sinc(0.1));
Console.WriteLine(sinc(0.0));
// Output:
// 0.998334166468282
// 1
```

Conditional ref expression

Beginning with C# 7.2, you can use the conditional ref expression to return the reference to the result of one of the two expressions. You can assign that reference to a ref local or ref readonly local variable, or use it as a reference return value or as a `ref` method parameter.

The syntax for the conditional ref expression is as follows:

```
condition ? ref consequence : ref alternative
```

Like the original conditional operator, the conditional ref expression evaluates only one of the two expressions:

either `consequence` or `alternative`.

In the case of the conditional ref expression, the type of `consequence` and `alternative` must be the same.

The following example demonstrates the usage of the conditional ref expression:

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

For more information, see the feature proposal note.

Conditional operator and an `if..else` statement

Use of the conditional operator over an if-else statement might result in more concise code in cases when you need conditionally to compute a value. The following example demonstrates two ways to classify an integer as negative or nonnegative:

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Operator overloadability

The conditional operator cannot be overloaded.

C# language specification

For more information, see the Conditional operator section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- if-else statement

- ?. and ?[] Operators
- ?? Operator
- ref keyword

++ Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The unary increment operator `++` increments its operand by 1. It's 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:

```
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:

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

Remarks

The increment operator is predefined for all integral types (including the char type), floating-point types, and any enum type.

An operand of the increment operator must be a variable, a property access, or an indexer access.

Operator overloadability

User-defined types can overload the `++` operator.

C# language specification

For more information, see the Postfix increment and decrement operators and Prefix increment and decrement operators sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- -- Operator
- How to: increment and decrement pointers

-- Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The unary decrement operator `--` decrements its operand by 1. It's 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:

```
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:

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Remarks

The decrement operator is predefined for all integral types (including the char type), floating-point types, and any enum type.

An operand of the decrement operator must be a variable, a property access, or an indexer access.

Operator overloadability

User-defined types can overload the `--` operator.

C# language specification

For more information, see the Postfix increment and decrement operators and Prefix increment and decrement operators sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- ++ Operator
- How to: increment and decrement pointers

&& Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The conditional logical AND operator `&&`, also known as the "short-circuiting" logical AND operator, computes the logical AND of its bool operands. The result of `x && y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`. If the first operand evaluates to `false`, the second operand is not evaluated and the result of operation is `false`. The following example demonstrates that behavior:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The logical AND operator `&` also computes the logical AND of its `bool` operands, but always evaluates both operands.

Operator overloadability

A user-defined type cannot overload the conditional logical AND operator. However, if a user-defined type overloads the logical AND and true and false operators in a certain way, the `&&` operation can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification.

C# language specification

For more information, see the Conditional logical operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- || operator
- ! operator
- & operator

|| Operator (C# Reference)

1/19/2019 • 2 minutes to read • Edit Online

The conditional logical OR operator `||`, also known as the "short-circuiting" logical OR operator, computes the logical OR of its bool operands. The result of `x || y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`. If the first operand evaluates to `true`, the second operand is not evaluated and the result of operation is `true`. The following example demonstrates that behavior:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The logical OR operator `|` also computes the logical OR of its `bool` operands, but always evaluates both operands.

Operator overloadability

A user-defined type cannot overload the conditional logical OR operator. However, if a user-defined type overloads the logical OR and true and false operators in a certain way, the `||` operation can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification.

C# language specification

For more information, see the Conditional logical operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- && operator
- ! operator
- | operator

<< operator (C# Reference)

2/13/2019 • 2 minutes to read • Edit Online

The left-shift operator `<<` shifts its first operand left by the number of bits defined by its second operand. All integer types support the `<<` operator. However, the type of the second operand must be int or a type that has a predefined implicit numeric conversion to `int`.

The high-order bits that are outside the range of the result type are discarded, and the low-order empty bit positions are set to zero, as the following example shows:

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Shift count

For the expression `x << count`, the actual shift count depends on the type of `x` as follows:

- If the type of `x` is int or uint, the shift count is given by the low-order *five* bits of the second operand. That is, the shift count is computed from `count & 0x1F` (or `count & 0b_1_1111`).
- If the type of `x` is long or ulong, the shift count is given by the low-order *six* bits of the second operand. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

```
int a = 0b_0001;
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2
```

Remarks

Shift operations never cause overflows and produce the same results in checked and unchecked contexts.

Operator overloadability

User-defined types can overload the `<<` operator. If a user-defined type `T` overloads the `<<` operator, the type of the first operand must be `T` and the type of the second operand must be `int`. When the `<<` operator is overloaded, the left-shift assignment operator `<<=` is also implicitly overloaded.

C# language specification

For more information, see the Shift operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- >> operator

>> operator (C# Reference)

2/13/2019 • 2 minutes to read • Edit Online

The right-shift operator `>>` shifts its first operand right by the number of bits defined by its second operand. All integer types support the `>>` operator. However, the type of the second operand must be int or a type that has a predefined implicit numeric conversion to `int`.

The right-shift operation discards the low-order bits. The high-order empty bit positions are set based on the type of the first operand as follows:

- If the first operand is of type int or long, the right-shift operator performs an **arithmetic** shift: the value of the most significant bit (the sign bit) of the first operand is propagated to the high-order empty bit positions. That is, the high-order empty bit positions are set to zero if the first operand is non-negative and set to one if it's negative.
- If the first operand is of type uint or ulong, the right-shift operator performs a **logical** shift: the high-order empty bit positions are always set to zero.

The following example demonstrates that behavior:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10

int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:  11110000000000000000000000000000
```

Shift count

For the expression `x >> count`, the actual shift count depends on the type of `x` as follows:

- If the type of `x` is int or uint, the shift count is given by the low-order *five* bits of the second operand. That is, the shift count is computed from `count & 0x1F` (or `count & 0b_1_1111`).
- If the type of `x` is long or ulong, the shift count is given by the low-order *six* bits of the second operand. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

```
int a = 0b_0100;
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;
Console.WriteLine($"{a} >> {count1} is {a >> count1}; {a} >> {count2} is {a >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

Remarks

Shift operations never cause overflows and produce the same results in checked and unchecked contexts.

Operator overloadability

User-defined types can overload the `>>` operator. If a user-defined type `T` overloads the `>>` operator, the type of the first operand must be `T` and the type of the second operand must be `int`. When the `>>` operator is overloaded, the right-shift assignment operator `>>=` is also implicitly overloaded.

C# language specification

For more information, see the Shift operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# operators
- << operator

== Operator (C# Reference)

12/20/2018 • 2 minutes to read • Edit Online

The equality operator `==` returns `true` if its operands are equal, `false` otherwise.

Value types equality

Operands of the built-in value types are equal if their values are equal:

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

NOTE

For relational operators `==`, `>`, `<`, `>=`, and `<=`, if any of the operands is not a number (Double.NaN or Single.NaN) the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value. For more information and examples, see the Double.NaN or Single.NaN reference article.

Two operands of the same enum type are equal if the corresponding values of the underlying integral type are equal.

By default, the `==` operator is not defined for a user-defined struct type. A user-defined type can overload the `==` operator.

Beginning with C# 7.3, the `==` and `!=` operators are supported by C# tuples. For more information, see the Equality and tuples section of the C# tuple types article.

String equality

Two string operands are equal when both of them are `null` or both string instances are of the same length and have identical characters in each character position:

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

That is case-sensitive ordinal comparison. For more information about how to compare strings, see How to compare strings in C#.

Reference types equality

Two other than `string` reference type operands are equal when they refer to the same object:

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

The example shows that the `==` operator is supported by user-defined reference types. However, a user-defined reference type can overload the `==` operator. If a reference type overloads the `==` operator, use the Object.ReferenceEquals method to check if two references of that type refer to the same object.

Operator overloadability

User-defined types can overload the `==` operator. If a type overloads the equality operator `==`, it must also overload the inequality operator `!=`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Equality comparisons

!= Operator (C# Reference)

12/19/2018 • 2 minutes to read • Edit Online

The inequality operator `!=` returns `true` if its operands are not equal, `false` otherwise. For the operands of the built-in types, the expression `x != y` produces the same result as the expression `!(x == y)`. For more information, see the == Operator article.

The following example demonstrates the usage of the `!=` operator:

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Operator overloadability

User-defined types can overload the `!=` operator. If a type overloads the inequality operator `!=`, it must also overload the equality operator `==`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Equality comparisons

<= Operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The "less than or equal" relational operator `<=` returns `true` if its first operand is less than or equal to its second operand, `false` otherwise. All numeric and enumeration types support the `<=` operator. For operands of the same enum type, the corresponding values of the underlying integral type are compared.

> **NOTE**
>
> For relational operators `==`, `>`, `<`, `>=`, and `<=`, if any of the operands is not a number (Double.NaN or Single.NaN) the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value. For more information and examples, see the Double.NaN or Single.NaN reference article.

The following example demonstrates the usage of the `<=` operator:

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Operator overloadability

User-defined types can overload the `<=` operator. If a type overloads the "less than or equal" operator `<=`, it must also overload the "greater than or equal" operator `>=`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- < Operator
- == Operator
- System.IComparable<T>

>= Operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The "greater than or equal" relational operator `>=` returns `true` if its first operand is greater than or equal to its second operand, `false` otherwise. All numeric and enumeration types support the `>=` operator. For operands of the same enum type, the corresponding values of the underlying integral type are compared.

> **NOTE**
>
> For relational operators `==`, `>`, `<`, `>=`, and `<=`, if any of the operands is not a number (Double.NaN or Single.NaN) the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value. For more information and examples, see the Double.NaN or Single.NaN reference article.

The following example demonstrates the usage of the `>=` operator:

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Operator overloadability

User-defined types can overload the `>=` operator. If a type overloads the "greater than or equal" operator `>=`, it must also overload the "less than or equal" operator `<=`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- > Operator
- == Operator
- System.IComparable<T>

+= Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The addition assignment operator.

An expression using the `+=` operator, such as

```
x += y
```

is equivalent to

```
x = x + y
```

except that `x` is only evaluated once.

For numeric types, the addition operator `+` computes the sum of its operands. If one or both operands is of type string, it concatenates the string representations of its operands. For delegate types, the `+` operator returns a new delegate instance that is combination of its operands.

You also use the `+=` operator to specify an event handler method when you subscribe to an event. For more information, see How to: Subscribe to and Unsubscribe from Events.

The following example demonstrates the usage of the `+=` operator:

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

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action<int> printer = (int s) => Console.WriteLine(s);
printer += (int s) => Console.WriteLine(2 * s);
printer(3);
// Output:
// 3
// 6
```

Operator overloadability

If a user-defined type overloads the addition operator `+`, the addition assignment operator `+=` is implicitly overloaded. A user-defined type cannot explicitly overload the addition assignment operator.

C# language specification

For more information, see the Compound assignment and Event assignment sections of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Events
- Delegates

-= operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The subtraction assignment operator.

Remarks

An expression using the `-=` assignment operator, such as

```
x -= y
```

is equivalent to

```
x = x - y
```

except that `x` is only evaluated once. The meaning of the - operator is dependent on the types of `x` and `y` (subtraction for numeric operands, delegate removal for delegate operands, and so forth).

The `-=` operator cannot be overloaded directly, but user-defined types can overload the - operator (see operator).

The -= operator is also used in C# to unsubscribe from an event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Example

```
class MainClass3
{
    static void Main()
    {
        int a = 5;
        a -= 6;
        Console.WriteLine(a);
    }
}
/*
Output:
-1
*/
```

See also

- C# Reference
- C# Programming Guide
- C# operators

*= Operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The binary multiplication assignment operator.

Remarks

An expression using the `*=` assignment operator, such as

```
x *= y
```

is equivalent to

```
x = x * y
```

except that `x` is only evaluated once. The * operator is predefined for numeric types to perform multiplication.

The `*=` operator cannot be overloaded directly, but user-defined types can overload the * operator (see operator).

Example

```
class MainClass10
{
    static void Main()
    {
        int a = 5;
        a *= 6;
        Console.WriteLine(a);
    }
}
/*
Output:
30
*/
```

See also

- C# Reference
- C# Programming Guide
- C# Operators

/= Operator (C# Reference)

12/12/2018 • 2 minutes to read • Edit Online

The division assignment operator.

An expression using the `/=` operator, such as

```
x /= y
```

is equivalent to

```
x = x / y
```

except that `x` is only evaluated once.

The division operator `/` divides its first operand by its second operand. It's supported by all numeric types.

The following example demonstrates the usage of the `/=` operator:

```
int a = 4;
int b = 5;
a /= b;
Console.WriteLine(a);   // output: 0

double x = 4;
double y = 5;
x /= y;
Console.WriteLine(x);   // output: 0.8
```

Operator overloadability

If a user-defined type overloads the division operator `/`, the division assignment operator `/=` is implicitly overloaded. A user-defined type cannot explicitly overload the division assignment operator.

C# language specification

For more information, see the Compound assignment section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators

2 minutes to read

&= Operator (C# Reference)

12/11/2018 • 2 minutes to read • Edit Online

The AND assignment operator.

An expression using the `&=` operator, such as

```
x &= y
```

is equivalent to

```
x = x & y
```

except that `x` is only evaluated once.

For integer operands, the `&` operator computes the bitwise logical AND of its operands; for bool operands, it computes the logical AND of its operands.

The following example demonstrates the usage of the `&=` operator:

```
byte a = 0b_1111_1000;
a &= 0b_1001_1111;
Console.WriteLine(Convert.ToString(a, toBase: 2));
// Output:
// 10011000

bool b = true;
b &= false;
Console.WriteLine(b);
// Output:
// False
```

Operator overloadability

If a user-defined type overloads the `&` operator, the AND assignment operator `&=` is implicitly overloaded. A user-defined type cannot explicitly overload the AND assignment operator.

C# language specification

For more information, see the Compound assignment section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators

|= operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The OR assignment operator.

Remarks

An expression using the `|=` assignment operator, such as

```
x |= y
```

is equivalent to

```
x = x | y
```

except that `x` is only evaluated once. The | operator performs a bitwise logical OR operation on integral operands and logical OR on bool operands.

The `|=` operator cannot be overloaded directly, but user-defined types can overload the | operator (see operator).

Example

```
class MainClass7
{
    static void Main()
    {
        int a = 0x0c;
        a |= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b |= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x0000000e
True
*/
```

See also

- C# Reference
- C# Programming Guide
- C# operators

^= operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The exclusive-OR assignment operator.

Remarks

An expression of the form

```
x ^= y
```

is evaluated as

```
x = x ^ y
```

except that `x` is only evaluated once. The ^ operator performs a bitwise exclusive-OR operation on integral operands and logical exclusive-OR on bool operands.

The ^= operator cannot be overloaded directly, but user-defined types can overload the ^ operator (see operator).

Example

```
class XORAssignment
{
    static void Main()
    {
        int a = 0x0c;
        a ^= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b ^= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x0000000a
True
*/
```

See also

- C# Reference
- C# Programming Guide
- C# operators

<<= operator (C# Reference)

2/13/2019 • 2 minutes to read • Edit Online

The left-shift assignment operator.

An expression using the `<<=` operator, such as

```
x <<= y
```

is equivalent to

```
x = x << y
```

except that `x` is only evaluated once.

The `<<` operator shifts its first operand left by the number of bits defined by its second operand.

The following example demonstrates the usage of the `<<=` operator:

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

x <<= 4;
Console.WriteLine($"After:  {Convert.ToString(x, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Operator overloadability

If a user-defined type overloads the `<<` operator, the left-shift assignment operator `<<=` is implicitly overloaded. A user-defined type cannot explicitly overload the left-shift assignment operator.

C# language specification

For more information, see the Compound assignment section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators

>>= operator (C# Reference)

2/13/2019 • 2 minutes to read • Edit Online

The right-shift assignment operator.

An expression using the `>>=` operator, such as

```
x >>= y
```

is equivalent to

```
x = x >> y
```

except that `x` is only evaluated once.

The `>>` operator shifts its first operand right by the number of bits defined by its second operand.

The following example demonstrates the usage of the `>>=` operator:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

x >>= 2;
Console.WriteLine($"After:  {Convert.ToString(x, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

Operator overloadability

If a user-defined type overloads the `>>` operator, the right-shift assignment operator `>>=` is implicitly overloaded. A user-defined type cannot explicitly overload the right-shift assignment operator.

C# language specification

For more information, see the Compound assignment section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# operators

-> Operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The pointer member access operator `->` combines pointer indirection and member access.

If `x` is a pointer of the type `T*` and `y` is an accessible member of `T`, an expression of the form

```
x->y
```

is equivalent to

```
(*x).y
```

The `->` operator requires unsafe context.

For more information, see How to: access a member with a pointer.

Operator overloadability

The `->` operator cannot be overloaded.

C# language specification

For more information, see the Pointer member access section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Pointer types

?? operator (C# Reference)

1/16/2019 • 2 minutes to read • Edit Online

The `??` operator is called the null-coalescing operator. It returns the left-hand operand if the operand is not null; otherwise it returns the right hand operand.

Remarks

A nullable type can represent a value from the type's domain, or the value can be undefined (in which case the value is null). You can use the `??` operator's syntactic expressiveness to return an appropriate value (the right hand operand) when the left operand has a nullable type whose value is null. If you try to assign a nullable value type to a non-nullable value type without using the `??` operator, you will generate a compile-time error. If you use a cast, and the nullable value type is currently undefined, an `InvalidOperationException` exception will be thrown.

For more information, see Nullable Types.

The result of a ?? operator is not considered to be a constant even if both its arguments are constants.

Example

```
class NullCoalesce
{
    static int? GetNullableInt()
    {
        return null;
    }

    static string GetStringValue()
    {
        return null;
    }

    static void Main()
    {
        int? x = null;

        // Set y to the value of x if x is NOT null; otherwise,
        // if x == null, set y to -1.
        int y = x ?? -1;

        // Assign i to return value of the method if the method's result
        // is NOT null; otherwise, if the result is null, set i to the
        // default value of int.
        int i = GetNullableInt() ?? default(int);

        string s = GetStringValue();
        // Display the value of s if s is NOT null; otherwise,
        // display the string "Unspecified".
        Console.WriteLine(s ?? "Unspecified");
    }
}
```

C# language specification

For more information, see The null coalescing operator in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# operators
- Nullable Types
- What Exactly Does 'Lifted' mean?

=> operator (C# Reference)

1/30/2019 • 2 minutes to read • Edit Online

The `=>` token is supported in two forms: as the lambda operator and as a separator of a member name and the member implementation in an expression body definition.

Lambda operator

In lambda expressions, the lambda operator `=>` separates the input variables on the left side from the lambda body on the right side.

The following example uses the LINQ feature with method syntax to demonstrate the usage of lambda expressions:

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 1, 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Input variables of lambda expressions are strongly typed at compile time. When the compiler can infer the types of input variables, like in the preceding example, you may omit type declarations. If you need to specify the type of input variables, you must do that for each variable, as the following example shows:

```
int[] numbers = { 1, 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);   // output: 280
```

The following example shows how to define a lambda expression without input variables:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

For more information, see Lambda expressions.

Expression body definition

An expression body definition has the following general syntax:

```
member => expression;
```

where *expression* is a valid expression. Note that *expression* can be a *statement expression* only if the member's return type is `void`, or if the member is a constructor, a finalizer, or a property `set` accessor.

The following example shows an expression body definition for a `Person.ToString` method:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

It's a shorthand version of the following method definition:

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

Expression body definitions for methods and read-only properties are supported starting with C# 6. Expression body definitions for constructors, finalizers, property accessors, and indexers are supported starting with C# 7.0.

For more information, see Expression-bodied members.

Operator overloadability

The `=>` operator cannot be overloaded.

C# language specification

For more information, see the Anonymous function expressions section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- C# Operators
- Lambda expressions
- Expression-bodied members