Table of Contents

+= Operator

-= Operator

*= Operator

/= Operator

%= Operator

&= Operator

|= Operator

^= Operator

<<= Operator

>>= Operator

-> Operator

?? Operator

=> Operator

Null-conditional Operators

C# Operators

1/5/2018 • 8 min 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 lists 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. NOTE, you can click on the operators to go the detailed pages with examples.

x.y – member access.

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

x?[y] - null conditional index access. Returns `null` if the left-hand operand is `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 System.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) – returns the default value of type T, `null` for reference types, zero for numeric types, and zero/`null` filled in members for struct types.

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. NOTE, you can click on the operators to go the detailed pages with examples.

+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 adds 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. NOTE, you can click on the operators to go the detailed pages with examples.

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 – modulus. 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. NOTE, you can click on the operators to go the detailed pages with examples.

x + y – addition.

x – y – subtraction.

Shift Operators

These operators have higher precedence than the next section and lower precedence than the previous section. NOTE, you can click on the operators to go the detailed pages with examples.

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. NOTE, you can click on the operators to go the detailed pages with examples.

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. NOTE, you can click on the operators to go the detailed pages with examples.

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. NOTE, you can click on the operator to go the details page with examples.

x & y – logical or bitwise AND. Use with integer types and `enum` types is generally allowed.

Logical XOR Operator

This operator has higher precedence than the next section and lower precedence than the previous section. NOTE, you can click on the operator to go the details page with examples.

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. NOTE, you can click on the operator to go the details page with examples.

x | y – logical or bitwise OR. Use with integer types and `enum` types is generally allowed.

Conditional AND Operator

This operator has higher precedence than the next section and lower precedence than the previous section. NOTE, you can click on the operator to go the details page with examples.

x && y – logical AND. If the first operand is 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. NOTE, you can click on the operator to go the details page with examples.

x || y – logical OR. If the first operand is 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. NOTE, you can click on the operator to go the details page with examples.

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. NOTE, you can click on the operator to go the details page with examples.

t ? x : y – if test `t` is 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. NOTE, you can click on the operators to go the detailed pages with examples.

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 – modulus 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)

1/5/2018 • 1 min to read • Edit Online

Square brackets (`[]`) are used for arrays, indexers, and attributes. They can also be used with pointers.

Remarks

An array type is a type followed by `[]`:

```
int[] fib; // fib is of type int[], "array of int".
fib = new int[100]; // Create a 100-element int array.
```

To access an element of an array, the index of the desired element is enclosed in brackets:

```
fib[0] = fib[1] = 1;
for (int i = 2; i < 100; ++i) fib[i] = fib[i - 1] + fib[i - 2];
```

An exception is thrown if an array index is out of range.

The array indexing operator cannot be overloaded; however, types can define indexers, and properties that take one or more parameters. Indexer parameters are enclosed in square brackets, just like array indexes, but indexer parameters can be declared to be of any type, unlike array indexes, which must be integral.

For example, the .NET Framework defines a `Hashtable` type that associates keys and values of arbitrary type:

```
System.Collections.Hashtable h = new System.Collections.Hashtable();
h["a"] = 123; // Note: using a string as the index.
```

Square brackets are also used to specify Attributes:

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

You can use square brackets to index off a pointer:

```
unsafe void M()
{
    int[] nums = {0,1,2,3,4,5};
    fixed ( int* p = nums )
    {
        p[0] = p[1] = 1;
        for( int i=2; i<100; ++i ) p[i] = p[i-1] + p[i-2];
    }
}
```

No bounds checking is performed.

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
Arrays
Indexers
unsafe
fixed Statement

() Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

In addition to being used to specify the order of operations in an expression, parentheses are used to perform the following tasks:

1. Specify casts, or type conversions.

```
double x = 1234.7;
int a;
a = (int)x; // Cast double to int
```

2. Invoke methods or delegates.

```
TestMethod();
```

Remarks

A cast explicitly invokes the conversion operator from one type to another; the cast fails if no such conversion operator is defined. To define a conversion operator, see explicit and implicit.

The `()` operator cannot be overloaded.

For more information, see Casting and Type Conversions.

A cast expression could lead to ambiguous syntax. For example, the expression `(x)-y` could be either interpreted as a cast expression (a cast of –y to type x) or as an additive expression combined with a parenthesized expression, which computes the value x – y.

For more information about method invocation, see Methods.

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

. Operator (C# Reference)

1/5/2018 • 1 min 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

:: Operator (C# Reference)

1/5/2018 • 1 min 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");
```

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)

1/5/2018 • 1 min 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 just the value of the operand.

Binary `+` operators are predefined for numeric and string types. For numeric types, + computes the sum of its two operands. When one or both operands are of type string, + concatenates the string representations of the operands.

Delegate types also provide a binary `+` operator, which performs delegate concatenation.

User-defined types can overload the unary `+` and binary `+` operators. Operations on integral types are generally allowed on enumeration. For more information, see operator (C# Reference).

Example

```
class Plus
{
    static void Main()
    {
        Console.WriteLine(+5);        // unary plus
        Console.WriteLine(5 + 5);     // addition
        Console.WriteLine(5 + .5);    // addition
        Console.WriteLine("5" + "5"); // string concatenation
        Console.WriteLine(5.0 + "5"); // string concatenation
        // note automatic conversion from double to string
    }
}
/*
Output:
5
10
5.5
55
55
*/
```

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
operator (C# Reference)

- Operator (C# Reference)

1/5/2018 • 1 min 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 (C# Reference).

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)

1/5/2018 • 1 min to read • Edit Online

The multiplication operator (`*`), which computes the product of its operands. Also, the dereference operator, which allows reading and writing to a pointer.

Remarks

All numeric types have predefined multiplication operators.

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

/ Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The division operator (`/`) divides its first operand by its second operand. All numeric types have predefined division operators.

Remarks

User-defined types can overload the `/` operator (see operator). An overload of the `/` operator implicitly overloads the /= operator.

When you divide two integers, the result is always an integer. For example, the result of 7 / 3 is 2. To determine the remainder of 7 / 3, use the remainder operator (%). To obtain a quotient as a rational number or fraction, give the dividend or divisor type `float` or type `double`. You can assign the type implicitly if you express the dividend or divisor as a decimal by putting a digit to the right side of the decimal point, as the following example shows.

Example

```
class Division
{
    static void Main()
    {
        Console.WriteLine("\nDividing 7 by 3.");
        // Integer quotient is 2, remainder is 1.
        Console.WriteLine("Integer quotient:           {0}", 7 / 3);
        Console.WriteLine("Negative integer quotient:  {0}", -7 / 3);
        Console.WriteLine("Remainder:                  {0}", 7 % 3);
        // Force a floating point quotient.
        float dividend = 7;
        Console.WriteLine("Floating point quotient:    {0}", dividend / 3);

        Console.WriteLine("\nDividing 8 by 5.");
        // Integer quotient is 1, remainder is 3.
        Console.WriteLine("Integer quotient:           {0}", 8 / 5);
        Console.WriteLine("Negative integer quotient:  {0}", 8 / -5);
        Console.WriteLine("Remainder:                  {0}", 8 % 5);
        // Force a floating point quotient.
        Console.WriteLine("Floating point quotient:    {0}", 8 / 5.0);
    }
}
// Output:
//Dividing 7 by 3.
//Integer quotient:           2
//Negative integer quotient:  -2
//Remainder:                  1
//Floating point quotient:    2.33333333333333

//Dividing 8 by 5.
//Integer quotient:           1
//Negative integer quotient:  -1
//Remainder:                  3
//Floating point quotient:    1.6
```

See Also

C# Reference

% Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The `%` operator computes the remainder after dividing its first operand by its second. All numeric types have predefined remainder operators.

Remarks

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

Example

```
class MainClass6
{
    static void Main()
    {
        Console.WriteLine(5 % 2);       // int
        Console.WriteLine(-5 % 2);      // int
        Console.WriteLine(5.0 % 2.2);   // double
        Console.WriteLine(5.0m % 2.2m); // decimal
        Console.WriteLine(-5.2 % 2.0);  // double
    }
}
/*
Output:
1
-1
0.6
0.6
-1.2
*/
```

Comments

Note the round-off errors associated with the double type.

See Also

C# Reference
C# Programming Guide
C# Operators

& Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The & operator can function as either a unary or a binary operator.

Remarks

The unary & operator returns the address of its operand (requires unsafe context).

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

The `&` operator evaluates both operators regardless of the first one's value. For example:

```
int i = 0;
if (false & ++i == 1)
{
    // i is incremented, but the conditional
    // expression evaluates to false, so
    // this block does not execute.
}
```

User-defined types can overload the binary `&` operator (see operator). Operations on integral types are generally allowed on enumeration. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
class BitwiseAnd
{
    static void Main()
    {
        // The following two statements perform logical ANDs.
        Console.WriteLine(true & false);
        Console.WriteLine(true & true);

        // The following line performs a bitwise AND of F8 (1111 1000) and
        // 3F (0011 1111).
        //    1111 1000
        //    0011 1111
        //    ---------
        //    0011 1000 or 38
        Console.WriteLine("0x{0:x}", 0xf8 & 0x3f);
    }
}
// Output:
// False
// True
// 0x38
```

See Also

C# Reference

| Operator (C# Reference)

1/5/2018 • 1 min 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

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)

1/5/2018 • 1 min 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)

1/5/2018 • 1 min to read • Edit Online

The logical negation operator (`!`) is a unary operator that negates its operand. It is defined for `bool` and returns `true` if and only if its operand is `false`.

Remarks

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

Example

```
class MainClass4
{
    static void Main()
    {
        Console.WriteLine(!true);
        Console.WriteLine(!false);
    }
}
/*
Output:
False
True
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

~ Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The `~` operator performs a bitwise complement operation on its operand, which has the effect of reversing each bit. Bitwise complement operators are predefined for int, uint, long, and ulong.

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

Remarks

User-defined types can overload the `~` operator. For more information, see operator. Operations on integral types are generally allowed on enumeration.

Example

```
class BWC
{
    static void Main()
    {
        int[] values = { 0, 0x111, 0xfffff, 0x8888, 0x22000022 };
        foreach (int v in values)
        {
            Console.WriteLine("~0x{0:x8} = 0x{1:x8}", v, ~v);
        }
    }
}
/*
Output:
~0x00000000 = 0xffffffff
~0x00000111 = 0xfffffeee
~0x000fffff = 0xfff00000
~0x00008888 = 0xffff7777
~0x22000022 = 0xddffffdd
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators
Finalizers

= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The assignment operator (`=`) stores the value of its right-hand operand in the storage location, property, or indexer denoted by its left-hand operand and returns the value as its result. The operands must be of the same type (or the right-hand operand must be implicitly convertible to the type of the left-hand operand).

Remarks

The assignment operator cannot be overloaded. However, you can define implicit conversion operators for a type, which enable you to use the assignment operator with those types. For more information, see Using Conversion Operators.

Example

```
class Assignment
{
    static void Main()
    {
        double x;
        int i;
        i = 5; // int to int assignment
        x = i; // implicit conversion from int to double
        i = (int)x; // needs cast
        Console.WriteLine("i is {0}, x is {1}", i, x);
        object obj = i;
        Console.WriteLine("boxed value = {0}, type is {1}",
                  obj, obj.GetType());
        i = (int)obj;
        Console.WriteLine("unboxed: {0}", i);
    }
}
/*
Output:
i is 5, x is 5
boxed value = 5, type is System.Int32
unboxed: 5
 */
```

See Also

C# Reference
C# Programming Guide
C# Operators

< Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

All numeric and enumeration types define a "less than" relational operator (`<`) that returns `true` if the first operand is less than the second, `false` otherwise.

Remarks

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

Example

```
class LT
{
    static void Main()
    {
        Console.WriteLine(1 < 1.1);
        Console.WriteLine(1.1 < 1.1);
    }
}
/*
Output:
True
False
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

> Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

All numeric and enumeration types define a "greater than" relational operator (`>`) that returns `true` if the first operand is greater than the second, `false` otherwise.

Remarks

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

Example

```
class GT
{
    static void Main()
    {
        Console.WriteLine(1.1 > 1);
        Console.WriteLine(1.1 > 1.1);
    }
}
/*
Output:
True
False
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators
explicit

?: Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The conditional operator (`?:`) returns one of two values depending on the value of a Boolean expression. Following is the syntax for the conditional operator.

```
condition ? first_expression : second_expression;
```

Remarks

The `condition` must evaluate to `true` or `false`. If `condition` is `true`, `first_expression` is evaluated and becomes the result. If `condition` is `false`, `second_expression` is evaluated and becomes the result. Only one of the two expressions is evaluated.

Either the type of `first_expression` and `second_expression` must be the same, or an implicit conversion must exist from one type to the other.

You can express calculations that might otherwise require an `if-else` construction more concisely by using the conditional operator. For example, the following code uses first an `if` statement and then a conditional operator to classify an integer as positive or negative.

```
int input = Convert.ToInt32(Console.ReadLine());
string classify;

// if-else construction.
if (input > 0)
    classify = "positive";
else
    classify = "negative";

// ?: conditional operator.
classify = (input > 0) ? "positive" : "negative";
```

The conditional operator is right-associative. The expression `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`, not as `(a ? b : c) ? d : e`.

The conditional operator cannot be overloaded.

Example

```
class ConditionalOp
{
    static double sinc(double x)
    {
        return x != 0.0 ? Math.Sin(x) / x : 1.0;
    }

    static void Main()
    {
        Console.WriteLine(sinc(0.2));
        Console.WriteLine(sinc(0.1));
        Console.WriteLine(sinc(0.0));
    }
}
/*
Output:
0.993346653975306
0.998334166468282
1
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators
if-else
?. and ?Operators
?? Operator

++ Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The increment operator (`++`) increments its operand by 1. The increment operator can appear before or after its operand: `++variable` and `variable++` .

Remarks

The first form is a prefix increment operation. The result of the operation is the value of the operand after it has been incremented.

The second form is a postfix increment operation. The result of the operation is the value of the operand before it has been incremented.

Numeric and enumeration types have predefined increment operators. User-defined types can overload the `++` operator. Operations on integral types are generally allowed on enumeration.

Example

```
//++ operator
class MainClass
{
    static void Main()
    {
        double x;
        x = 1.5;
        Console.WriteLine(++x);
        x = 1.5;
        Console.WriteLine(x++);
        Console.WriteLine(x);
    }
}
/*
Output
2.5
1.5
2.5
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

-- Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The decrement operator (`--`) decrements its operand by 1. The decrement operator can appear before or after its operand: `--variable` and `variable--`. The first form is a prefix decrement operation. The result of the operation is the value of the operand "after" it has been decremented. The second form is a postfix decrement operation. The result of the operation is the value of the operand "before" it has been decremented.

Remarks

Numeric and enumeration types have predefined decrement operators.

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

Example

```
class MainClass5
{
    static void Main()
    {
        double x;
        x = 1.5;
        Console.WriteLine(--x);
        x = 1.5;
        Console.WriteLine(x--);
        Console.WriteLine(x);
    }
}
/*
Output:
0.5
1.5
0.5
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

&& Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The conditional-AND operator (`&&`) performs a logical-AND of its `bool` operands, but only evaluates its second operand if necessary.

Remarks

The operation

```
x && y
```

corresponds to the operation

```
x & y
```

except that if `x` is `false` , `y` is not evaluated, because the result of the AND operation is `false` no matter what the value of `y` is. This is known as "short-circuit" evaluation.

The conditional-AND operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators.

Example

In the following example, the conditional expression in the second `if` statement evaluates only the first operand because the operand returns `false` .

```
class LogicalAnd
{
    static void Main()
    {
        // Each method displays a message and returns a Boolean value.
        // Method1 returns false and Method2 returns true. When & is used,
        // both methods are called.
        Console.WriteLine("Regular AND:");
        if (Method1() & Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned false.");

        // When && is used, after Method1 returns false, Method2 is
        // not called.
        Console.WriteLine("\nShort-circuit AND:");
        if (Method1() && Method2())
            Console.WriteLine("Both methods returned true.");
        else
            Console.WriteLine("At least one of the methods returned false.");
    }

    static bool Method1()
    {
        Console.WriteLine("Method1 called.");
        return false;
    }

    static bool Method2()
    {
        Console.WriteLine("Method2 called.");
        return true;
    }
}
// Output:
// Regular AND:
// Method1 called.
// Method2 called.
// At least one of the methods returned false.

// Short-circuit AND:
// Method1 called.
// At least one of the methods returned false.
```

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

|| Operator (C# Reference)

1/5/2018 • 2 min to read • Edit Online

The conditional-OR operator (`||`) performs a logical-OR of its `bool` operands. If the first operand evaluates to `true` , the second operand isn't evaluated. If the first operand evaluates to `false` , the second operator determines whether the OR expression as a whole evaluates to `true` or `false` .

Remarks

The operation

```
x || y
```

corresponds to the operation

```
x | y
```

except that if `x` is `true` , `y` is not evaluated because the OR operation is `true` regardless of the value of `y` . This concept is known as "short-circuit" evaluation.

The conditional-OR operator cannot be overloaded, but overloads of the regular logical operators and the true and false operators are, with certain restrictions, also considered to be overloads of the conditional logical operators.

Example

In the following examples, the expression that uses `||` evaluates only the first operand. The expression that uses `|` evaluates both operands. In the second example, a run-time exception occurs if both operands are evaluated.

```
class ConditionalOr
{
    // Method1 returns true.
    static bool Method1()
    {
        Console.WriteLine("Method1 called.");
        return true;
    }

    // Method2 returns false.
    static bool Method2()
    {
        Console.WriteLine("Method2 called.");
        return false;
    }


    static bool Divisible(int number, int divisor)
    {
        // If the OR expression uses ||, the division is not attempted
        // when the divisor equals 0.
        return !(divisor == 0 || number % divisor != 0);

        // If the OR expression uses |, the division is attempted when
        // the divisor equals 0, and causes a divide-by-zero exception.
        // Replace the return statement with the following line to
        // see the exception.
```

```
        //return !(divisor == 0 | number % divisor != 0);
    }

    static void Main()
    {
        // Example #1 uses Method1 and Method2 to demonstrate
        // short-circuit evaluation.

        Console.WriteLine("Regular OR:");
        // The | operator evaluates both operands, even though after
        // Method1 returns true, you know that the OR expression is
        // true.
        Console.WriteLine("Result is {0}.\n", Method1() | Method2());

        Console.WriteLine("Short-circuit OR:");
        // Method2 is not called, because Method1 returns true.
        Console.WriteLine("Result is {0}.\n", Method1() || Method2());


        // In Example #2, method Divisible returns True if the
        // first argument is evenly divisible by the second, and False
        // otherwise. Using the | operator instead of the || operator
        // causes a divide-by-zero exception.

        // The following line displays True, because 42 is evenly
        // divisible by 7.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 7));

        // The following line displays False, because 42 is not evenly
        // divisible by 5.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 5));

        // The following line displays False when method Divisible
        // uses ||, because you cannot divide by 0.
        // If method Divisible uses | instead of ||, this line
        // causes an exception.
        Console.WriteLine("Divisible returns {0}.", Divisible(42, 0));
    }
}
/*
Output:
Regular OR:
Method1 called.
Method2 called.
Result is True.

Short-circuit OR:
Method1 called.
Result is True.

Divisible returns True.
Divisible returns False.
Divisible returns False.
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

<< Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The left-shift operator (`<<`) shifts its first operand left by the number of bits specified by its second operand. The type of the second operand must be an int or a type that has a predefined implicit numeric conversion to `int`.

Remarks

If the first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of the second operand. That is, the actual shift count is 0 to 31 bits.

If the first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of the second operand. That is, the actual shift count is 0 to 63 bits.

Any high-order bits that are not within the range of the type of the first operand after the shift are discarded, and the low-order empty bits are zero-filled. Shift operations never cause overflows.

User-defined types can overload the `<<` operator (see operator); the type of the first operand must be the user-defined type, and the type of the second operand must be `int`. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
class MainClass11
{
    static void Main()
    {
        int i = 1;
        long lg = 1;
        // Shift i one bit to the left. The result is 2.
        Console.WriteLine("0x{0:x}", i << 1);
        // In binary, 33 is 100001. Because the value of the five low-order
        // bits is 1, the result of the shift is again 2.
        Console.WriteLine("0x{0:x}", i << 33);
        // Because the type of lg is long, the shift is the value of the six
        // low-order bits. In this example, the shift is 33, and the value of
        // lg is shifted 33 bits to the left.
        //     In binary:     10 0000 0000 0000 0000 0000 0000 0000 0000
        //     In hexadecimal: 2    0    0    0    0    0    0    0    0
        Console.WriteLine("0x{0:x}", lg << 33);
    }
}
/*
Output:
0x2
0x2
0x200000000
*/
```

Comments

Note that `i<<1` and `i<<33` give the same result, because 1 and 33 have the same low-order five bits.

See Also

>> Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The right-shift operator (`>>`) shifts its first operand right by the number of bits specified by its second operand.

Remarks

If the first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of the second operand (second operand & 0x1f).

If the first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of the second operand (second operand & 0x3f).

If the first operand is an int or long, the right-shift is an arithmetic shift (high-order empty bits are set to the sign bit). If the first operand is of type uint or ulong, the right-shift is a logical shift (high-order bits are zero-filled).

User-defined types can overload the `>>` operator; the type of the first operand must be the user-defined type, and the type of the second operand must be int. For more information, see operator. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
class RightShift
{
    static void Main()
    {
        int i = -1000;
        Console.WriteLine(i >> 3);
    }
}
/*
Output:
-125
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

== Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

For predefined value types, the equality operator (`==`) returns true if the values of its operands are equal, `false` otherwise. For reference types other than string, `==` returns `true` if its two operands refer to the same object. For the `string` type, `==` compares the values of the strings.

Remarks

User-defined value types can overload the `==` operator (see operator). So can user-defined reference types, although by default `==` behaves as described above for both predefined and user-defined reference types. If `==` is overloaded, `!=` must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
class Equality
{
    static void Main()
    {
        // Numeric equality: True
        Console.WriteLine((2 + 2) == 4);

        // Reference equality: different objects,
        // same boxed value: False.
        object s = 1;
        object t = 1;
        Console.WriteLine(s == t);

        // Define some strings:
        string a = "hello";
        string b = String.Copy(a);
        string c = "hello";

        // Compare string values of a constant and an instance: True
        Console.WriteLine(a == b);

        // Compare string references;
        // a is a constant but b is an instance: False.
        Console.WriteLine((object)a == (object)b);

        // Compare string references, both constants
        // have the same value, so string interning
        // points to same reference: True.
        Console.WriteLine((object)a == (object)c);
    }
}
/*
Output:
True
False
True
False
True
*/
```

See Also

!= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The inequality operator (`!=`) returns false if its operands are equal, true otherwise. Inequality operators are predefined for all types, including string and object. User-defined types can overload the `!=` operator.

Remarks

For predefined value types, the inequality operator (`!=`) returns true if the values of its operands are different, false otherwise. For reference types other than `string` , `!=` returns true if its two operands refer to different objects. For the `string` type, `!=` compares the values of the strings.

User-defined value types can overload the `!=` operator (see operator). So can user-defined reference types, although by default `!=` behaves as described above for both predefined and user-defined reference types. If `!=` is overloaded, `==` must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
class InequalityTest
{
    static void Main()
    {
        // Numeric inequality:
        Console.WriteLine((2 + 2) != 4);

        // Reference equality: two objects, same boxed value
        object s = 1;
        object t = 1;
        Console.WriteLine(s != t);

        // String equality: same string value, same string objects
        string a = "hello";
        string b = "hello";

        // compare string values
        Console.WriteLine(a != b);

        // compare string references
        Console.WriteLine((object)a != (object)b);
    }
}
/*
Output:
False
True
False
False
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

<= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

All numeric and enumeration types define a "less than or equal" relational operator (`<=`) that returns `true` if the first operand is less than or equal to the second, `false` otherwise.

Remarks

User-defined types can overload the `<=` operator. For more information, see operator. If `<=` is overloaded, `>=` must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
class LTE
{
    static void Main()
    {
        Console.WriteLine(1 <= 1.1);
        Console.WriteLine(1.1 <= 1.1);
    }
}
/*
Output:
True
True
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators
explicit

>= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

All numeric and enumeration types define a "greater than or equal" relational operator, `>=` that returns `true` if the first operand is greater than or equal to the second, `false` otherwise.

Remarks

User-defined types can overload the `>=` operator. For more information, see operator. If `>=` is overloaded, `<=` must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
class GTE
{
    static void Main()
    {
        Console.WriteLine(1.1 >= 1);
        Console.WriteLine(1.1 >= 1.1);
    }
}
/*
Output:
True
True
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

+= Operator (C# Reference)

1/19/2018 • 1 min to read • Edit Online

The addition 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 depends on the types of `x` and `y` (addition for numeric operands, concatenation for string 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 to specify a method that will be called in response to an event; such methods are called event handlers. The use of the `+=` operator in this context is referred to as *subscribing to an event*. For more information, see How to: Subscribe to and Unsubscribe from Events and Delegates.

Example

```
class AddAssigment
{
    static void Main()
    {
        //addition
        int a = 5;
        a += 6;
        Console.WriteLine(a);

        //string concatenation
        string s = "Hello";
        s += " world.";
        Console.WriteLine(s);
    }
}
/*
Output:
11
Hello world
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

-= Operator (C# Reference)

1/5/2018 • 1 min 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)

1/5/2018 • 1 min 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)

1/5/2018 • 1 min to read • Edit Online

The division 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 division.

The `/=` operator cannot be overloaded directly, but user-defined types can overload the / operator (see operator). On all compound assignment operators, overloading the binary operator implicitly overloads the equivalent compound assignment.

Example

```
class MainClass2
{
    static void Main()
    {
        int a = 5;
        a /= 6;
        Console.WriteLine(a);
        double b = 5;
        b /= 6;
        Console.WriteLine(b);
    }
}
/*
Output:
0
0.833333333333333
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

%= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The remainder 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 compute the remainder after division.

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

Example

```
class Test2
{
    static void Main()
    {
        int a = 5;
        a %= 3;
        Console.WriteLine(a);
    }
}
// Output: 2
```

See Also

C# Reference
C# Programming Guide
C# Operators

&= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The AND 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 AND operation on integral operands and logical AND on `bool` operands.

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

Example

```
class AndAssignment
{
    static void Main()
    {
        int a = 0x0c;
        a &= 0x06;
        Console.WriteLine("0x{0:x8}", a);
        bool b = true;
        b &= false;
        Console.WriteLine(b);
    }
}
/*
Output:
0x00000004
False
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

|= Operator (C# Reference)

1/5/2018 • 1 min 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)

1/5/2018 • 1 min 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)

1/5/2018 • 1 min to read • Edit Online

The left-shift 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 shifts `x` left by the number of bits specified by `y`.

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

Example

```
class MainClass9
{
    static void Main()
    {
        int a = 1000;
        a <<= 4;
        Console.WriteLine(a);
    }
}
/*
Output:
16000
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

>>= Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

The right-shift 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 shifts `x` right by an amount specified by `y`.

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

Example

```
class MainClass8
{
    static void Main()
    {
        int a = 1000;
        a >>= 4;
        Console.WriteLine(a);
    }
}
/*
Output:
62
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

-> Operator (C# Reference)

1/5/2018 • 1 min to read • Edit Online

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

Remarks

An expression of the form,

```
x->y
```

(where `x` is a pointer of type `T*` and `y` is a member of `T`) is equivalent to,

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

The `->` operator can be used only in code that is marked as unsafe.

The `->` operator cannot be overloaded.

Example

```
// compile with: /unsafe

struct Point
{
    public int x, y;
}

class MainClass12
{
    unsafe static void Main()
    {
        Point pt = new Point();
        Point* pp = &pt;
        pp->x = 123;
        pp->y = 456;
        Console.WriteLine("{0} {1}", pt.x, pt.y);
    }
}
/*
Output:
123 456
*/
```

See Also

C# Reference
C# Programming Guide
C# Operators

?? Operator (C# Reference)

1/5/2018 • 1 min 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");
    }
}
```

See Also

C# Reference

C# Programming Guide
C# Operators
Nullable Types
What Exactly Does 'Lifted' mean?

=> Operator (C# Reference)

1/5/2018 • 3 min to read • Edit Online

The `=>` operator can be used in two ways in C#:

- As the lambda operator in a lambda expression, it separates the input variables from the lambda body.
- In an expression body definition, it separates a member name from the member implementation.

Lambda operator

The `=>` token is called the lambda operator. It is used in *lambda expressions* to separate the input variables on the left side from the lambda body on the right side. Lambda expressions are inline expressions similar to anonymous methods but more flexible; they are used extensively in LINQ queries that are expressed in method syntax. For more information, see Lambda Expressions.

The following example shows two ways to find and display the length of the shortest string in an array of strings. The first part of the example applies a lambda expression (`w => w.Length`) to each element of the `words` array and then uses the Min method to find the smallest length. For comparison, the second part of the example shows a longer solution that uses query syntax to do the same thing.

```
string[] words = { "cherry", "apple", "blueberry" };

// Use method syntax to apply a lambda expression to each element
// of the words array.
int shortestWordLength = words.Min(w => w.Length);
Console.WriteLine(shortestWordLength);

// Compare the following code that uses query syntax.
// Get the lengths of each word in the words array.
var query = from w in words
            select w.Length;
// Apply the Min method to execute the query and get the shortest length.
int shortestWordLength2 = query.Min();
Console.WriteLine(shortestWordLength2);

// Output:
// 5
// 5
```

Remarks

The `=>` operator has the same precedence as the assignment operator (`=`) and is right-associative.

You can specify the type of the input variable explicitly or let the compiler infer it; in either case, the variable is strongly typed at compile time. When you specify a type, you must enclose the type name and the variable name in parentheses, as the following example shows.

```
int shortestWordLength = words.Min((string w) => w.Length);
```

Example

The following example shows how to write a lambda expression for the overload of the standard query operator Enumerable.Where that takes two arguments. Because the lambda expression has more than one parameter, the parameters must be enclosed in parentheses. The second parameter, `index`, represents the index of the current

element in the collection. The `Where` expression returns all the strings whose lengths are less than their index positions in the array.

```
static void Main(string[] args)
{
    string[] digits = { "zero", "one", "two", "three", "four", "five",
            "six", "seven", "eight", "nine" };

    Console.WriteLine("Example that uses a lambda expression:");
    var shortDigits = digits.Where((digit, index) => digit.Length < index);
    foreach (var sD in shortDigits)
    {
        Console.WriteLine(sD);
    }

    // Output:
    // Example that uses a lambda expression:
    // five
    // six
    // seven
    // eight
    // nine
}
```

Expression body definition

An expression body definition provides a member's implementation in a highly condensed, readable form. It 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 or a finalizer.

Expression body definitions for methods and property get statements are supported starting with C# 6. Expression body definitions for constructors, finalizers, property set statements, and indexers are supported starting with C# 7.

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

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

It is a shorthand version of the following method definition:

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

For more detailed information on expression body definitions, see Expression-bodied members.

See Also

C# Reference
C# Programming Guide
Lambda Expressions
Expression-bodied members.

Null-conditional Operators (C# and Visual Basic)

1/5/2018 • 1 min to read • Edit Online

Used to test for null before performing a member access (`?.`) or index (`?[`) operation. 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
```

```
Dim length = customers?.Length  ' null if customers is null
Dim first as Customer = customers?(0)  ' null if customers is null
Dim count as Integer? = customers?(0)?.Orders?.Count()  ' null if customers, the first customer, or Orders is
null
```

The last example demonstrates that the null-condition operators are short-circuiting. If one operation in a chain of conditional member access and index operation returns null, then the rest of the chain's execution stops. Other operations with lower precedence in the expression continue. For example, `E` in the following executes in the second line, and the `??` and `==` operations execute. In the first line, the `??` short circuits and `E` does not execute when the left side evaluates to non-null.

```
A?.B?.C?[0] ?? E
A?.B?.C?[0] == E
```

```
A?.B?.C?(0) ?? E
A?.B?.C?(0) == E
```

Another use for the null-condition 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(…);
```

```
Dim handler = AddressOf(Me.PropertyChanged)
If handler IsNot Nothing
    Call handler(…)
```

The new way is much simpler:

```
PropertyChanged?.Invoke(e)
```

```
PropertyChanged?.Invoke(e)
```

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)`. There were too many ambiguous parsing situations to allow it.

Language Specifications

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

For more information, see the Visual Basic Language Reference.

See Also

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