C#

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Right-shift operator >>

The `>>` operator shifts its left-hand operand right by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the Shift count of the shift operators section.

The right-shift operation discards the low-order bits, as the following example shows:

C#

```
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).PadLeft(4, '0'), 4}");
// Output:
// Before: 1001
// After:  0010
```

The high-order empty bit positions are set based on the type of the left-hand operand as follows:

- If the left-hand 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 left-hand operand is propagated to the high-order empty bit positions. That is, the high-order empty bit positions are set to zero if the left-hand operand is non-negative and set to one if it's negative.

C#

```
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
```

- If the left-hand 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.

C#

```csharp
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:  00010000000000000000000000000000
```

ⓘ **Note**

Use the **unsigned right-shift operator** to perform a *logical* shift on operands of signed integer types. This is preferred to casting a left-hand operand to an unsigned type and then casting the result of a shift operation back to a signed type.

Unsigned right-shift operator >>>

Available in C# 11 and later, the `>>>` operator shifts its left-hand operand right by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the Shift count of the shift operators section.

The `>>>` operator always performs a *logical* shift. That is, the high-order empty bit positions are always set to zero, regardless of the type of the left-hand operand. The >> operator performs an *arithmetic* shift (that is, the value of the most significant bit is propagated to the high-order empty bit positions) if the left-hand operand is of a signed type. The following example demonstrates the difference between `>>` and `>>>` operators for a negative left-hand operand:

C#

```csharp
int x = -8;
Console.WriteLine($"Before:    {x,11}, hex: {x,8:x}, binary: {Con-
```

```
vert.ToString(x, toBase: 2), 32}");

int y = x >> 2;
Console.WriteLine($"After  >>: {y,11}, hex: {y,8:x}, binary: {Con-
vert.ToString(y, toBase: 2), 32}");

int z = x >>> 2;
Console.WriteLine($"After >>>: {z,11}, hex: {z,8:x}, binary: {Con-
vert.ToString(z, toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before:             -8, hex: fffffff8, binary:
11111111111111111111111111111000
// After  >>:          -2, hex: fffffffe, binary:
11111111111111111111111111111110
// After >>>:  1073741822, hex: 3ffffffe, binary:
00111111111111111111111111111110
```

Logical AND operator &

The `&` operator computes the bitwise logical AND of its integral operands:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

For `bool` operands, the `&` operator computes the logical AND of its operands. The unary `&` operator is the address-of operator.

Logical exclusive OR operator ^

The `^` operator computes the bitwise logical exclusive OR, also known as the bitwise logical XOR, of its integral operands:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

For `bool` operands, the `^` operator computes the logical exclusive OR of its operands.

Logical OR operator |

The `|` operator computes the bitwise logical OR of its integral operands:

C#

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

For `bool` operands, the `|` operator computes the logical OR of its operands.

Compound assignment

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

C#

```
x op= y
```

is equivalent to

C#

```
x = x op y
```

except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with bitwise and shift operators:

C#

```
uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a);  // output: 10011000
```

```
a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a);  // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
Display(a);  // output: 01111000

a = INITIAL_VALUE;
a <<= 2;
Display(a);  // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a);  // output: 00001111

a = INITIAL_VALUE;
a >>>= 4;
Display(a);  // output: 00001111



void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, to-
Base: 2).PadLeft(8, '0'), 8}");
```

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

C#

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output:
1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Operator precedence

The following list orders bitwise and shift operators starting from the highest precedence to the lowest:

- Bitwise complement operator `~`

- Shift operators `<<`, `>>`, and `>>>`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`

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

```C#
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, to-
Base: 2), 4}");
```

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

Shift count of the shift operators

For the built-in shift operators `<<`, `>>`, and `>>>`, the type of the right-hand operand must be `int` or a type that has a predefined implicit numeric conversion to `int`.

For the `x << count`, `x >> count`, and `x >>> count` expressions, 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 defined by the low-order *five* bits of the right-hand 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 defined by the low-order *six* bits of the right-hand operand. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

C#

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {coun-
t2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {coun-
t2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

int count = -31;
int c = 0b_0001;
Console.WriteLine($"{c} << {count} is {c << count}");



// Output:
// 1 << -31 is 2
```

> **Note**
>
> As the preceding example shows, the result of a shift operation can be non-zero even if the value of the right-hand operand is greater than the number of bits in the left-hand operand.

Enumeration logical operators

The `~`, `&`, `|`, and `^` operators are also supported by any enumeration type. For operands of the same enumeration type, a logical operation is performed on the corresponding values of the underlying integral type. For example, for any `x` and `y` of an enumeration type `T` with an underlying type `U`, the `x & y` expression produces the same result as the `(T)((U)x & (U)y)` expression.

You typically use bitwise logical operators with an enumeration type that is defined with the Flags attribute. For more information, see the Enumeration types as bit flags section of the Enumeration types article.

Operator overloadability

A user-defined type can overload the `~`, `<<`, `>>`, `>>>`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. A user-defined type can't explicitly overload a compound assignment operator.

If a user-defined type `T` overloads the `<<`, `>>`, or `>>>` operator, the type of the left-hand operand must be `T`. In C# 10 and earlier, the type of the right-hand operand must be `int`; beginning with C# 11, the type of the right-hand operand of an overloaded shift operator can be any.

C# language specification

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

- Bitwise complement operator
- Shift operators

- Logical operators
- Compound assignment
- Numeric promotions
- C# 11 - Relaxed shift requirements
- C# 11 - Logical right-shift operator

See also

- C# reference
- C# operators and expressions
- Boolean logical operators