

## 1.2 Operators and Expressions

### Operators

- An operators is a symbol that tells the computers to perform certain mathematical or logical manipulations.
- Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.

#### **Example**

2+3

+ is an operator that is used to carry out addition operation, while 2 and 3 are operands.

- Operators are used to manipulate variables and values in a program.

C# supports a number of operators that are classified based on the type of operations they perform.

1. Arithmetic operator
2. Relational operator
3. Logical operator
4. Assignment operator
5. Increment and decrement operator
6. Conditional operator
7. Bitwise operator
8. Special operator

### Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division and Modulus.

<b>Operator</b>	<b>Operator Name</b>	<b>Example</b>
+	Addition Operator	6 + 3 evaluates to 9
-	Subtraction Operator	10 - 4 evaluates to 9
*	Multiplication Operator	4 * 2 evaluates to 8
/	Division Operator	10 / 5 evaluates to 2
%	Modulus Operator (remainder)	16 % 3 evaluates to 1

## Example

```
using System;
namespace Operator
{
class ArithmeticOperator
{
public static void Main(string[] args)
{
double firstNumber = 14.40, secondNumber = 4.60, result;
int num1 = 26, num2 = 4, rem;

// Addition operator
result = firstNumber + secondNumber;
Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);

// Subtraction operator
result = firstNumber - secondNumber;
Console.WriteLine("{0} - {1} = {2}", firstNumber, secondNumber, result);

// Multiplication operator
result = firstNumber * secondNumber;
Console.WriteLine("{0} * {1} = {2}", firstNumber, secondNumber, result);

// Division operator
result = firstNumber / secondNumber;
Console.WriteLine("{0} / {1} = {2}", firstNumber, secondNumber, result);

// Modulo operator
rem = num1 % num2;
Console.WriteLine("{0} % {1} = {2}", num1, num2, rem);
}
}
}
```

## Output

```
14.4 + 4.6 = 19
14.4 - 4.6 = 9.8
14.4 * 4.6 = 66.24
14.4 / 4.6 = 3.1304347826087
26 % 4 = 2
```

## Relational Operators

Relational operators are used to check the relationship between two operands. If the relationship is true the result will be **true**, otherwise it will result in **false**.

Operator	Operator Name	Example
>	Greater than	3 > 1 evaluates to true
<	Less than	5 < 3 evaluates to false
>=	Greater than or equal to	4 >= 4 evaluates to true
<=	Division Operator	5 <= 3 evaluates to false
==	Less than or equal to	6 == 4 evaluates to false
!=	Not equal to	10 != 2 evaluates to true

### Example

```
using System;

namespace Operator
{
class RelationalOperator
    {
    public static void Main(string[] args)
    {
    bool result;
    int firstNumber = 10, secondNumber = 20;

result = (firstNumber==secondNumber);
Console.WriteLine("{0} == {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber > secondNumber);
Console.WriteLine("{0} > {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber < secondNumber);
Console.WriteLine("{0} < {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber >= secondNumber);
Console.WriteLine("{0} >= {1} returns {2}",firstNumber, secondNumber, result);
```

```

result = (firstNumber <= secondNumber);
Console.WriteLine("{0} <= {1} returns {2}",firstNumber, secondNumber, result);

result = (firstNumber != secondNumber);
Console.WriteLine("{0} != {1} returns {2}",firstNumber, secondNumber, result);
    }
}
}

```

### Output

```

10 == 20 returns false
10 > 20 returns false
10 < 20 returns true
10 >= 20 returns false
10 <= 20 returns true
10 != 20 returns true

```

### Logical Operators

- Logical operators are used to perform logical operation such as AND and OR. Logical operators operates on boolean expressions (true and false) and returns boolean values. Logical operators are used in decision making and loops.
- The logical operators && and || are used when we want to form compound conditions by combining two or more relation.

a>b && c>d

- An expression of this kind which combines two or more relational expressions in termed as logical expression or a compound relational expression

Operand 1	Operand 2	OR (  )	AND (&&)
true	true	true	true
false	true	true	false
true	false	true	false
false	false	false	false

Operand	NOT(!)
true	false
false	true

In simple words, the table can be summarized as:

- If one of the operand is true, the **OR** operator will evaluate it to **true**.
- If one of the operand is false, the **AND** operator will evaluate it to **false**

### Example

```
using System;
namespace Operator
{
class LogicalOperator
    {
    public static void Main(string[] args)
    {
        bool result;
        int firstNumber = 10, secondNumber = 20;

        // OR operator
        result = (firstNumber == secondNumber) || (firstNumber > 5);
        Console.WriteLine(result);

        // AND operator
        result = (firstNumber == secondNumber) && (firstNumber > 5);
        Console.WriteLine(result);

        // NOT operator
        result = !((firstNumber == secondNumber) || (firstNumber > 5));
        Console.WriteLine(result);

    }
}
```

```
}  
}
```

## Output

```
true  
false  
false
```

## Assignment Operator

- Assignment operator (=) is used to assign values to variables.
- Assignment operator (=) is used to assign values of an expression to variables.

For example,

```
double x;  
x = 50.05;  
Here, 50.05 is assigned to x.
```

## Example

```
using System;  
namespace Operator  
{  
class AssignmentOperator  
    {  
        public static void Main(string[] args)  
        {  
            int firstNumber, secondNumber, result;  
  
            // Assigning a constant to variable  
            firstNumber = 10;  
            Console.WriteLine("First Number = {0}", firstNumber);  
  
            // Assigning a variable to another variable  
            secondNumber = firstNumber;  
            Console.WriteLine("Second Number = {0}", secondNumber);  
  
            // Assigning a value of an expression to variable  
            result=secondNumber + firstNumber;  
            Console.WriteLine("Sum= {0}",result);
```

```
}  
}  
}
```

### **Output**

```
First Number = 10  
Second Number = 10  
Sum= 20
```

### **Shorthand assignment operator**

```
int a=10;  
a+=5;      // a=a+5;  
a-=5;      // a=a-5;  
a*=5;      // a=a*5;  
a/=5;      // a=a/5;
```

### **Increment and decrement operator**

Increment operator increases integer value by one i.e.

```
int a = 10;  
a++;  
++a;
```

**++a**

A prefix operator first adds 1 to the operand and then the result is assigned to the variable

**a++**

A postfix operator first assigns the value to the variable on the left and then increments the operand

Decrement operator decreases integer value by one i.e.

```
int a = 20;  
a--;  
--a;
```

### **Example**

```
using System;  
namespace MyApplication  
{  
    class Program
```

```
{
    static void Main(string[] args)
    {
        int x = 5;
        x++;
        Console.WriteLine(x);
        x--;
        Console.WriteLine(x);
    }
}
```

### Output

```
6
5
```

### Conditional operator

It is ternary operator which is a shorthand version of if-else statement. It has three operands and hence the name ternary. It will return one of two values depending on the value of a Boolean expression.

#### **The syntax of ternary operator is:**

```
Condition ? Expression1 : Expression2;
```

The ternary operator works as follows:

- If the expression stated by Condition is true, the result of Expression1 is returned by the ternary operator.
- If it is false, the result of Expression2 is returned.

### **Example**

```
using System;
namespace Conditional
{
    class Ternary
    {
        public static void Main(string[] args)
        {
            int number = 2;
            bool isEven;

            isEven = (number % 2 == 0) ? true : false ;
            Console.WriteLine(isEven);
        }
    }
}
```

```
}  
}  
}
```

## Output

```
true
```

## Bitwise operator

Bitwise and bit shift operators are used to perform bit level operations on integer (int, long, etc) and boolean data.

List of C# Bitwise Operators	
Operator	Operator Name
~	Bitwise Complement
&	Bitwise AND
	Bitwise OR
^	Bitwise Exclusive OR (XOR)
<<	Bitwise Left Shift
>>	Bitwise Right Shift

## Bitwise OR

- Bitwise OR operator is represented by |.
- It performs bitwise OR operation on the corresponding bits of two operands.
- If either of the bits is 1, the result is 1. Otherwise the result is 0.

Bitwise OR operation between 14 and 11:

```
14 - 00001110
```

```
11 - 00001011
```

```
00001111 = 15 (In Decimal)
```

## Bitwise AND

- Bitwise AND operator is represented by &.
- It performs bitwise AND operation on the corresponding bits of two operands.
- If either of the bits is 0, the result is 0. Otherwise the result is 1.

```
14 - 00001110
```

11 - 00001011

00001010 = 10 (In Decimal)

### Bitwise XOR

- Bitwise XOR operator is represented by  $\wedge$ .
- It performs bitwise XOR operation on the corresponding bits of two operands.
- If the corresponding bits are **same**, the result is  $0$ .
- If the corresponding bits are **different**, the result is  $1$ .

14 - 00001110

11 - 00001011

00000101 = 5 (In Decimal)

### Bitwise Complement

- Bitwise Complement operator is represented by  $\sim$ .
- It is a unary operator, i.e. operates on only one operand.
- The  $\sim$  operator inverts each bit i.e. changes 1 to 0 and 0 to 1.

### For Example,

26 = 00011010 (In Binary)

Bitwise Complement operation on 26:

$\sim 00011010 = 11100101 = 229$  (In Decimal)

- We got -27 as output when we were expecting 229.
- It happens because the binary value 11100101 which we expect to be 229 is actually a 2's complement representation of -27.
- Negative numbers in computer are represented in 2's complement representation.

### 2's complement

Decimal	Binary	2's Complement
0	00000000	$-(11111111 + 1) = -00000000 = -0$ (In Decimal)
1	00000001	$-(11111110 + 1) = -11111111 = -256$ (In Decimal)

2's complement

Decimal	Binary	2's Complement
229	11100101	$-(00011010 + 1) = -00011011 = -27$

- The bitwise complement of 26 is 229 (in decimal) and the 2's complement of 229 is -27.
- Hence the output is -27 instead of 229.

### Bitwise Left Shift

- Bitwise left shift operator is represented by `<<`.
- The `<<` operator shifts a number to the left by a specified number of bits.
- Zeroes are added to the least significant bits.

```
42 << 1 = 84 (In binary 1010100)
42 << 2 = 168 (In binary 10101000)
42 << 4 = 672 (In binary 1010100000)
```

### Bitwise Right Shift

- Bitwise right shift operator is represented by `>>`.
- The `>>` operator shifts a number to the right by a specified number of bits.
- The first operand is shifted to right by the number of bits specified by second operand.

```
42 >> 1 = 21 (In binary 010101)
42 >> 2 = 10 (In binary 001010)
42 >> 4 = 2 (In binary 000010)
```

### Example

```
using System;
namespace Operator
{
class BitWiseOperater
    {
        public static void Main(string[] args)
        {
```

```

        int firstNumber = 14, secondNumber = 11, number =42, result;

// Bitwise OR
result = firstNumber | secondNumber;
Console.WriteLine("{0} | {1} = {2}", firstNumber, secondNumber, result);

// Bitwise AND
result = firstNumber & secondNumber;
Console.WriteLine("{0} & {1} = {2}", firstNumber, secondNumber, result);

// Bitwise XOR
result = firstNumber ^ secondNumber;
Console.WriteLine("{0} ^ {1} = {2}", firstNumber, secondNumber, result);

// Bitwise Complement
result = ~ firstNumber
Console.WriteLine("~{0} = {1}", firstNumber, result);

// Bitwise Left Shift
Console.WriteLine("{0}<<1 = {1}", number, number<<1);
Console.WriteLine("{0}<<2 = {1}", number, number<<2);
Console.WriteLine("{0}<<4 = {1}", number, number<<4);

// Bitwise Right Shift
Console.WriteLine("{0}>>1 = {1}", number, number>>1);
Console.WriteLine("{0}>>2 = {1}", number, number>>2);
Console.WriteLine("{0}>>4 = {1}", number, number>>4);
}
}
}

```

**Output**

14 | 11 = 15

14 & 11 = 10

14 ^ 11 = 5

~14 = -15

42<<1 = 84

42<<2 = 168

42<<4 = 672

42>>1 = 21

42>>2 = 10

42>>4 = 2

### **Special Operator**

C# supports the following special operators

- is (relational operators)
- as (relational operators)
- typeof (type operator)
- sizeof (size operator)
- new (object operator)
- .(dot) (member-access operator)

### **is (relational operators)**

- The **is** operator is used to check if the run-time type of an object is compatible with the given type or not.
- It returns *true* if the given object is of the same type otherwise, return *false*.
- It also returns *false* for *null* objects.

### **Syntax :**

expression is type

Here, the *expression* will be evaluated to an instance of some type. And *type* is the name of the type to that the result of the *expression* is to be converted. If the *expression* is not null and the object results from evaluating the *expression* can be converted to the specified *type* then is operator will return *true* otherwise it will return *false*.

### **Example**

```
using System;
```

```
public class HelloWorld  
{
```

```
public static void Main(string[] args)
{
    object x="Hello";
    object y=123;
    bool result;
    result= x is string;
    Console.WriteLine(result);
    result= y is string;
    Console.WriteLine(result);
}
}
```

### Output

```
True
False
```

### as (relational operators)

- The **as** operator is used to perform conversion between compatible reference types or Nullable types.
- The 'as' operator keyword in C# is used only for nullable, reference and boxing conversions. It can't perform user-defined conversions that can be only performed by using cast expression.

#### Syntax :

```
MyClass myObject = obj as MyClass;
```

```
foreach (object item in list)
{
    // Try to convert item as a string
    string stype = item as string;
    if (stype != null)
        Console.WriteLine(item.ToString() + " converted successfully.");
    else
        Console.WriteLine(item.ToString() + " conversion failed.");
}
```

### typeof (type operator)

- The **typeof** is an operator keyword which is used to get a type at the compile-time.
- Or in other words, this operator is used to get the System.Type object for a type.

- This operator takes the *Type* itself as an argument and returns the marked type of the argument.

### Example :

```
using System;
class program
{
    public static void Main()
    {
        Console.WriteLine(typeof(int));
        Console.WriteLine(typeof(Array));
        Console.WriteLine(typeof(char));

    }
}
```

### Output:

```
System.Int32
System.Array
System.Char
```

### sizeof (size operator)

The sizeof() operator is used to obtain the size of a data type in bytes in bytes. It will not return the size of the variables or instances. Its return type is always int.

### Syntax:

```
int sizeof(type);
```

### Examples:

```
using System;

namespace Program
{
    class Test
    {
```

```
static void Main(string[] args)

{
    Console.WriteLine("sizeof(char) : {0}", sizeof(char));
    Console.WriteLine("sizeof(byte) : {0}", sizeof(byte));
    Console.WriteLine("sizeof(sbyte) : {0}", sizeof(sbyte));
    Console.WriteLine("sizeof(float) : {0}", sizeof(float));
    Console.WriteLine("sizeof(ushort) : {0}", sizeof(ushort));
    Console.WriteLine("sizeof(double) : {0}", sizeof(double));
    Console.WriteLine("sizeof(int) : {0}", sizeof(int));
    Console.WriteLine("sizeof(bool) : {0}", sizeof(bool));
    Console.WriteLine("sizeof(short) : {0}", sizeof(short));
}

}

}
```

### **Output:**

```
sizeof(char) : 2
sizeof(byte) : 1
sizeof(sbyte) : 1
sizeof(float) : 4
sizeof(ushort) : 2
sizeof(double) : 8
sizeof(int) : 4
sizeof(bool) : 1
sizeof(short) : 2
```

### **new (object operator)**

- Use the new keyword to create an instance of the array.
- The new operator is used to create an object or instantiate an object.

Example an object is created for the class using the new.

```
Calculate c = new Calculate();
```

You can also use the new keyword to create an instance of the array.

```
double[] points = new double[10];
```

### **.(dot) (member-access operator)**

The member access (dot) operator (“.”) is used frequently to access a field or to call a method on an object.

```
object a = new object();  
a.ToString();
```

The dot operator is also used to form qualified names: names that specify the namespace or interface

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

### **Arithmetic Expression**

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language.

C# can handle any complex mathematical expression.

<u>Mathematical Formula</u>	<u>C# Expression</u>
$b^2 - 4ac$	<code>b * b - 4 * a * c</code>
$a + bc$	<code>a + b * c</code>
$\frac{a + b}{c + d}$	<code>(a + b) / (c + d)</code>
$\frac{1}{1 + x^2}$	<code>1 / (1 + x * x)</code>
$ab - (b + c)$	<code>a * b - (b + c)</code>

## Evaluation of Expression

- Expressions are evaluated using an assignment operator

```
Variable=expression;
```

- Variable is any valid C# variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side.
- The variables in the expression must be declared and assigned values before they are used in the expression.

Example:

```
x=a+b*c;
```

In the above example variable a, b and c must be declared and assigned values before they are used in the expression.

## Precedence of Arithmetic Operators

An arithmetic expression without any parentheses will be evaluated from left to right using rules of precedence of operators.

There are two distinct priority levels of arithmetic operators in C#.

- High priority \* / %
- Low priority + -

There basic evaluation procedure includes two left-to-right passes through the expression.

- During first pass, the high priority operators (if any) are applied as they are encountered.
- During second pass, the low priority operators (if any) are applied as they are encountered.

Example:

```
x=a-b/3+c*2-1
```

When a=9 ,b=12 and c=3

```
x=9-12/3+3*2-1
```

*First pass*

Step1:  $x=9-4+3*2-1$

Step 2:  $x=9-4+6-1$

*Second pass*

Step 3 :  $x=5+6-1$

Step 4:  $x=11-1$

Step 5:  $x=10$

The order of the evaluation can be changed by introducing parentheses into the expression

Example:

$x=9-12/(3+3)*(2-1)$

Whenever the parentheses are used, the expression within parentheses assume highest priority. If two or more sets of parentheses appear one after another, the expression contained in the left-most set is evaluated first and the right-most set at last

*First pass:*

Step 1:  $x=9-12/6*(2-1)$

Step 2:  $x=9-12/6*1$

*Second pass:*

Step 3:  $x=9-2*1$

Step 4:  $x=9-2$

*Third pass:*

Step 5:  $x=7$

## **Type Conversion**

The process of converting the value of one type (int, float, double, etc.) to another type is known as type conversion.

In C#, there are two basic types of type conversion:

- Implicit Type Conversions
- Explicit Type Conversions

### **Implicit Type Conversion in C#**

- In implicit type conversion, the C# compiler automatically converts one type to another.
- Generally, smaller types like int (having less memory size) are automatically converted to larger types like double (having larger memory size).

## Byte->short->int->long->float->double

Following table shows the implicit types of conversion that is supported by C# :

Convert from Data Type	Convert to Data Type
byte	short, int, long, float, double
short	int, long, float, double
int	long, float, double
long	float, double
float	double

```
using System;
namespace Casting
{
    class Program
    {
        public static void Main(String []args)
        {
            int i = 57;
            // automatic type conversion
            long l = i;
            // automatic type conversion
            float f = l;
            Console.WriteLine("Int value " +i);
            Console.WriteLine("Long value " +l);
            Console.WriteLine("Float value " +f);
        }
    }
}
```

**Output:**

```
Int value 57  
Long value 57  
Float value 57
```

**C# Explicit Type Conversion**

- In explicit type conversion, we explicitly convert one type to another.
- Generally, larger types like double (having large memory size) are converted to smaller types like int (having small memory size).

```
using System;  
namespace Casting  
{  
class Program  
{  
    public static void Main(String []args)  
    {  
        double d = 765.12;  
        // Explicit Type Casting  
        int i = (int)d;  
        Console.WriteLine("Value of i is " +i);  
    }  
}  
}
```

**Output:**

```
Value of i is 765
```

**C# provides built-in methods for Type-Conversions as follows :**

Method	Description
ToBoolean	It will converts a type to Boolean value
ToChar	It will converts a type to a character value
ToByte	It will converts a value to Byte Value
ToDecimal	It will converts a value to Decimal point value
ToDouble	It will converts a type to double data type
ToInt16	It will converts a type to 16-bit integer
ToInt32	It will converts a type to 32 bit integer
ToInt64	It will converts a type to 64 bit integer
ToString	It will converts a given type to string
ToUInt16	It will converts a type to unsigned 16 bit integer
ToUInt32	It will converts a type to unsigned 32 bit integer
ToUInt64	It will converts a type to unsigned 64 bit integer

Example :

```
using System;
namespace Casting
{
class Program
{
    public static void Main(String []args)
    {
        int i = 12;
        double d = 765.12;
        float f = 56.123F;
        // Using Built- In Type Conversion
        // Methods & Displaying Result
        Console.WriteLine(Convert.ToString(f));
        Console.WriteLine(Convert.ToInt32(d));
        Console.WriteLine(Convert.ToUInt32(f));
        Console.WriteLine(Convert.ToDouble(i));
        Console.WriteLine("GeeksforGeeks");
    }
}
}
```

**Output:**

56.123

765

56

12

GeeksforGeeks

### **Mathematical Functions in C#**

The System.Math class in C# provides methods are properties to perform mathematical operations, trigonometric, logarithmic calculations, etc.

Sr.No	Method & Description
1	Abs(Decimal) Returns the absolute value of a Decimal number.
2	Abs(Double) Returns the absolute value of a double-precision floating-point number.
3	Abs(Int16) Returns the absolute value of a 16-bit signed integer.
4	Abs(Int32) Returns the absolute value of a 32-bit signed integer.
5	Abs(Int64) Returns the absolute value of a 64-bit signed integer.
6	Abs(SByte) Returns the absolute value of an 8-bit signed integer.
7	Abs(Single) Returns the absolute value of a single-precision floating-point number.
8	Acos(Double) Returns the angle whose cosine is the specified number.
9	Asin(Double) Returns the angle whose sine is the specified number.
10	Atan(Double) Returns the angle whose tangent is the specified number.

using System;

```
class Program
{
    static void Main()
    {
        int val1 = 250;
        int val2 = -150;

        Console.WriteLine("Before...");
        Console.WriteLine(val1);
        Console.WriteLine(val2);
    }
}
```

```
int abs1 = Math.Abs(val1);
int abs2 = Math.Abs(val2);

Console.WriteLine("After...");
Console.WriteLine(abs1);
Console.WriteLine(abs2);
}
}
```

**Output:**

```
Before...
250
-150
After...
250
150
```

