/\* arithmetic assignment operators \*/

using System;

namespace OperatorsAppl

{

class Program

{

static void Main(string[] args)

{

int a = 21;

int c;

c = a;

Console.WriteLine("Line 1 - = Value of c = {0}", c);

c += a;

Console.WriteLine("Line 2 - += Value of c = {0}", c);

c -= a;

Console.WriteLine("Line 3 - -= Value of c = {0}", c);

c \*= a;

Console.WriteLine("Line 4 - \*= Value of c = {0}", c);

c /= a;

Console.WriteLine("Line 5 - /= Value of c = {0}", c);

c = 200;

c %= a;

Console.WriteLine("Line 6 - %= Value of c = {0}", c);

c <<= 2;

Console.WriteLine("Line 7 - <<= Value of c = {0}", c);

c >>= 2;

Console.WriteLine("Line 8 - >>= Value of c = {0}", c);

c &= 2;

Console.WriteLine("Line 9 - &= Value of c = {0}", c);

c ^= 2;

Console.WriteLine("Line 10 - ^= Value of c = {0}", c);

c |= 2;

Console.WriteLine("Line 11 - |= Value of c = {0}", c);

Console.ReadLine();

}

}

}

Output:

Line 1 - = Value of c = 21

Line 2 - += Value of c = 42

Line 3 - -= Value of c = 21

Line 4 - \*= Value of c = 441

Line 5 - /= Value of c = 21

Line 6 - %= Value of c = 11

Line 7 - <<= Value of c = 44

Line 8 - >>= Value of c = 11

Line 9 - &= Value of c = 2

Line 10 - ^= Value of c = 0

Line 11 - |= Value of c = 2

Bitwise Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12, which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15, which is 0000 1111 |

Example

using System;

namespace OperatorsAppl

{

class Program

{

static void Main(string[] args)

{

int a = 60; /\* 60 = 0011 1100 \*/

int b = 13; /\* 13 = 0000 1101 \*/

int c = 0;

c = a & b; /\* 12 = 0000 1100 \*/

Console.WriteLine("Line 1 - Value of c is {0}", c );

c = a | b; /\* 61 = 0011 1101 \*/

Console.WriteLine("Line 2 - Value of c is {0}", c);

c = a ^ b; /\* 49 = 0011 0001 \*/

Console.WriteLine("Line 3 - Value of c is {0}", c);

c = ~a; /\*-61 = 1100 0011 \*/

Console.WriteLine("Line 4 - Value of c is {0}", c);

c = a << 2; /\* 240 = 1111 0000 \*/

Console.WriteLine("Line 5 - Value of c is {0}", c);

c = a >> 2; /\* 15 = 0000 1111 \*/

Console.WriteLine("Line 6 - Value of c is {0}", c);

Console.ReadLine();

}

}

}

/\* Output:

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15 \*/

Operators Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

using System;

namespace OperatorsAppl

{

class Program

{

static void Main(string[] args)

{

int a = 20;

int b = 10;

int c = 15;

int d = 5;

int e;

e = (a + b) \* c / d; // ( 30 \* 15 ) / 5

Console.WriteLine("Value of (a + b) \* c / d is : {0}", e);

e = ((a + b) \* c) / d; // (30 \* 15 ) / 5

Console.WriteLine("Value of ((a + b) \* c) / d is : {0}", e);

e = (a + b) \* (c / d); // (30) \* (15/5)

Console.WriteLine("Value of (a + b) \* (c / d) is : {0}", e);

e = a + (b \* c) / d; // 20 + (150/5)

Console.WriteLine("Value of a + (b \* c) / d is : {0}", e);

Console.ReadLine();

}

}

}

Output:

Value of (a + b) \* c / d is : 90

Value of ((a + b) \* c) / d is : 90

Value of (a + b) \* (c / d) is : 90

Value of a + (b \* c) / d is : 50

## The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

using System;

namespace Loops

{

class Program

{

static void Main(string[] args)

{

for (; ; )

{

Console.WriteLine("Hey! I am Trapped");

}

}

}

}

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

* break statement
* continue statement

programs based on the above statements in c#.

Type conversion is basically type casting or converting one type of data to another type. In C#, type casting has two forms:

* **Implicit type conversion** - these conversions are performed by C# in a type-safe manner. Examples are conversions from smaller to larger integral types and conversions from derived classes to base classes.
* **Explicit type conversion** - these conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion:

namespace TypeConversionApplication

{

class ExplicitConversion

{

static void Main(string[] args)

{

double d = 5673.74;

int i;

// cast double to int.

i = (int)d;

Console.WriteLine(i);

Console.ReadKey();

}

}

}