Data types:

<https://www.tutorialspoint.com/csharp/csharp_data_types>

On-line compiler:

<https://dotnetfiddle.net/>

The variables in C#, are categorized into the following types −

* Value types
* Reference types
* Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010 −

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine −

[Live Demo](http://tpcg.io/0migdg)

using System;

namespace DataTypeApplication {

class Program {

static void Main(string[] args) {

Console.WriteLine("Size of int: {0}", sizeof(int));

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Size of int: 4

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic,** and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

object obj;

obj = 100; // this is boxing

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is −

dynamic <variable\_name> = value;

For example,

dynamic d = 20;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

String str = "Tutorials Point";

A @quoted string literal looks as follows −

@"Tutorials Point";

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is −

type\* identifier;

For example,

char\* cptr;

int\* iptr;

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

* **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, 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 −

[Live Demo](http://tpcg.io/JzfLhC)

using System;

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();

}

}

}

When the above code is compiled and executed, it produces the following result −

5673

C# Type Conversion Methods

C# provides the following built-in type conversion methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | **ToBoolean**  Converts a type to a Boolean value, where possible. |
| 2 | **ToByte**  Converts a type to a byte. |
| 3 | **ToChar**  Converts a type to a single Unicode character, where possible. |
| 4 | **ToDateTime**  Converts a type (integer or string type) to date-time structures. |
| 5 | **ToDecimal**  Converts a floating point or integer type to a decimal type. |
| 6 | **ToDouble**  Converts a type to a double type. |
| 7 | **ToInt16**  Converts a type to a 16-bit integer. |
| 8 | **ToInt32**  Converts a type to a 32-bit integer. |
| 9 | **ToInt64**  Converts a type to a 64-bit integer. |
| 10 | **ToSbyte**  Converts a type to a signed byte type. |
| 11 | **ToSingle**  Converts a type to a small floating point number. |
| 12 | **ToString**  Converts a type to a string. |
| 13 | **ToType**  Converts a type to a specified type. |
| 14 | **ToUInt16**  Converts a type to an unsigned int type. |
| 15 | **ToUInt32**  Converts a type to an unsigned long type. |
| 16 | **ToUInt64**  Converts a type to an unsigned big integer. |

The following example converts various value types to string type −

[Live Demo](http://tpcg.io/Wjo68S)

using System;

namespace TypeConversionApplication {

class StringConversion {

static void Main(string[] args) {

int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;

Console.WriteLine(i.ToString());

Console.WriteLine(f.ToString());

Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result −

75

53.005

2345.7652

True

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as −

|  |  |
| --- | --- |
| **Type** | **Example** |
| Integral types | sbyte, byte, short, ushort, int, uint, long, ulong, and char |
| Floating point types | float and double |
| Decimal types | decimal |
| Boolean types | true or false values, as assigned |
| Nullable types | Nullable data types |

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**, which we will cover in subsequent chapters.

Defining Variables

Syntax for variable definition in C# is −

<data\_type> <variable\_list>;

Here, data\_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable\_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here −

int i, j, k;

char c, ch;

float f, salary;

double d;

You can initialize a variable at the time of definition as −

int i = 100;

Initializing Variables

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is −

variable\_name = value;

Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as −

<data\_type> <variable\_name> = value;

Some examples are −

int d = 3, f = 5; /\* initializing d and f. \*/

byte z = 22; /\* initializes z. \*/

double pi = 3.14159; /\* declares an approximation of pi. \*/

char x = 'x'; /\* the variable x has the value 'x'. \*/

It is a good programming practice to initialize variables properly, otherwise sometimes program may produce unexpected result.

The following example uses various types of variables −

[Live Demo](http://tpcg.io/u3on95)

using System;

namespace VariableDefinition {

class Program {

static void Main(string[] args) {

short a;

int b ;

double c;

/\* actual initialization \*/

a = 10;

b = 20;

c = a + b;

Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

a = 10, b = 20, c = 30

Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

int num;

num = Convert.ToInt32(Console.ReadLine());

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console.ReadLine()** accepts the data in string format.

Lvalue and Rvalue Expressions in C#

There are two kinds of expressions in C# −

* **lvalue** − An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
* **rvalue** − An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side. Following is a valid C# statement −

int g = 20;

But following is not a valid statement and would generate compile-time error −

10 = 20;

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, and there is no prefix id for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

212 /\* Legal \*/

215u /\* Legal \*/

0xFeeL /\* Legal \*/

Following are other examples of various types of Integer literals −

85 /\* decimal \*/

0x4b /\* hexadecimal \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Here are some examples of floating-point literals −

3.14159 /\* Legal \*/

314159E-5F /\* Legal \*/

510E /\* Illegal: incomplete exponent \*/

210f /\* Illegal: no decimal or exponent \*/

.e55 /\* Illegal: missing integer or fraction \*/

While representing in decimal form, you must include the decimal point, the exponent, or both; and while representing using exponential form you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Character Constants

Character literals are enclosed in single quotes. For example, 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (such as 'x'), an escape sequence (such as '\t'), or a universal character (such as '\u02C0').

There are certain characters in C# when they are preceded by a backslash. They have special meaning and they are used to represent like newline (\n) or tab (\t). Here, is a list of some of such escape sequence codes −

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \xhh . . . | Hexadecimal number of one or more digits |

Following is the example to show few escape sequence characters −

[Live Demo](http://tpcg.io/dF4h0V)

using System;

namespace EscapeChar {

class Program {

static void Main(string[] args) {

Console.WriteLine("Hello\tWorld\n\n");

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Hello World

String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

@"hello dear"

Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is −

const <data\_type> <constant\_name> = value;

The following program demonstrates defining and using a constant in your program −

[Live Demo](http://tpcg.io/C4QjxW)

using System;

namespace DeclaringConstants {

class Program {

static void Main(string[] args) {

const double pi = 3.14159;

// constant declaration

double r;

Console.WriteLine("Enter Radius: ");

r = Convert.ToDouble(Console.ReadLine());

double areaCircle = pi \* r \* r;

Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result −

Enter Radius:

3

Radius: 3, Area: 28.27431

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

This tutorial explains the arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then −

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B = 30 |
| - | Subtracts second operand from the first | A - B = -10 |
| \* | Multiplies both operands | A \* B = 200 |
| / | Divides numerator by de-numerator | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division | B % A = 0 |
| ++ | Increment operator increases integer value by one | A++ = 11 |
| -- | Decrement operator decreases integer value by one | A-- = 9 |

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then −

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non zero then condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non zero then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; then in the binary format they are as follows −

A = 0011 1100

B = 0000 1101

-------------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_bitwise_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -61, which is 1100 0011 in 2's complement 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 = 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 = 15, which is 0000 1111 |

Assignment Operators

There are following assignment operators supported by C# −

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B assigns value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

Miscellaneous Operators

There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_misc_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a data type. | sizeof(int), returns 4. |
| typeof() | Returns the type of a class. | typeof(StreamReader); |
| & | Returns the address of an variable. | &a; returns actual address of the variable. |
| \* | Pointer to a variable. | \*a; creates pointer named 'a' to a variable. |
| ? : | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |
| is | Determines whether an object is of a certain type. | If( Ford is Car) // checks if Ford is an object of the Car class. |
| as | Cast without raising an exception if the cast fails. | Object obj = new StringReader("Hello");  StringReader r = obj as StringReader; |

Operator Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. 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 the first evaluation takes place for 3\*2 and then 7 is added into it.

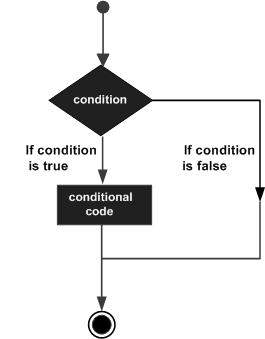
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C# provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/csharp/if_statement_in_csharp.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statement](https://www.tutorialspoint.com/csharp/if_else_statement_in_csharp.htm)  An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| 3 | [nested if statements](https://www.tutorialspoint.com/csharp/nested_if_statements_in_csharp.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| 4 | [switch statement](https://www.tutorialspoint.com/csharp/switch_statement_in_csharp.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |
| 5 | [nested switch statements](https://www.tutorialspoint.com/csharp/nested_switch_statements_in_csharp.htm)  You can use one **switch** statement inside another **switch**statement(s). |

The ? : Operator

We have covered **conditional operator ? :** in previous chapter which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

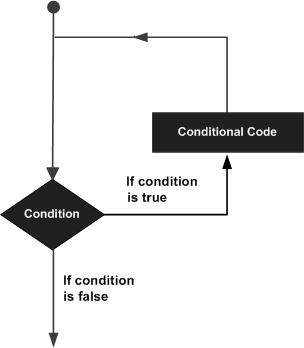
Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined as follows: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C# provides following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/csharp/csharp_while_loop.htm)  It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/csharp/csharp_for_loop.htm)  It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/csharp/csharp_do_while_loop.htm)  It is similar to a while statement, except that it tests the condition at the end of the loop body |
| 4 | [nested loops](https://www.tutorialspoint.com/csharp/csharp_nested_loops.htm)  You can use one or more loop inside any another while, for or do..while loop. |

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.

C# provides the following control statements. Click the following links to check their details.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/csharp/csharp_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/csharp/csharp_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

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.

Example

using System;

namespace Loops {

class Program {

static void Main(string[] args) {

for (; ; ) {

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

}

}

}

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.