**C# data types**

In this part of the C# tutorial, we will talk about data types.

Computer programs, including spreadsheets, text editors, calculators, or chat clients, work with data. Tools to work with various data types are essential part of a modern computer language. A*data type* is a set of values and the allowable operations on those values.

The two fundamental data types in C# are (i) value types and (ii) reference types.

**Value Types** – Include; Primitive types (except strings), enumerations, and structures.

**Reference Types** – Include; Classes, strings, interfaces, arrays, and delegates.

Every type has a default value. *Reference types* are created on the Heap. The lifetime of the reference type is managed by the .NET framework. The default value for reference types is null reference. Assignment to a variable of a reference type creates a copy of the reference rather than a copy of the referenced value. *Value types* are created on the stack. The lifetime is determined by the lifetime of the variable.

Assignment to a variable of a value type creates a copy of the value being assigned. **Value types** have different default values. For example, boolean default value is false, decimal 0, string an empty string "".

**Boolean values**

There is a duality built in our world. There is a Heaven and Earth, water and fire, jing and jang, man and woman, love and hatred. In C# the bool data type is a primitive data type having one of two values: true or false. This is a fundamental data type that is very common in computer programs.

Happy parents are waiting a child to be born. They have chosen a name for both possibilities. If it is going to be a boy, they have chosen John. If it is going to be a girl, they have chosen Victoria.

using System;

class BooleanType

{

static void Main()

{

bool male = false;

Random random = new Random();

male = Convert.ToBoolean(random.Next(0, 2));

if (male)

{

Console.WriteLine("We will use name John");

} else

{

Console.WriteLine("We will use name Victoria");

}

}

}

The program uses a random number generator to simulate our case.

bool male = false;

The male variable is our boolean variable, initiated at first to false.

Random random = new Random();

We create a Random object which is used to compute random numbers. It is part of the System namespace.

male = Convert.ToBoolean(random.Next(0, 2));

The Next() method returns a random number within a specified range. The lower bound is included, the upper bound is not. In other words, we receive either 0, or 1. Later the Convert() method converts these values to boolean ones, 0 to false, 1 to true.

if (male)

{

Console.WriteLine("We will use name John");

} else

{

Console.WriteLine("We will use name Victoria");

}

If the male variable is set to true, we choose the name John. Otherwise, we choose the name Victoria. Control structures like if/else statements work with boolean values.

$ ./booleantype.exe

We will use name John

$ ./booleantype.exe

We will use name Victoria

$ ./booleantype.exe

We will use name Victoria

$ ./booleantype.exe

We will use name John

Running the program several times gives this output.

**Integers**

Integers are a subset of the real numbers. They are written without a fraction or a decimal component. Integers fall within a set Z = {..., -2, -1, 0, 1, 2, ...}. Integers are infinite.

In computer languages, integers are primitive data types. Computers can practically work only with a subset of integer values, because computers have finite capacity. Integers are used to count discrete entities. We can have 3, 4, 6 humans, but we cannot have 3.33 humans. We can have 3.33 kilograms.

|  |  |  |  |
| --- | --- | --- | --- |
| **VB Alias** | **.NET Type** | **Size** | **Range** |
| sbyte | System.SByte | 1 byte | -128 to 127 |
| byte | System.Byte | 1 byte | 0 to 255 |
| short | System.Int16 | 2 bytes | -32,768 to 32,767 |
| ushort | System.UInt16 | 2 bytes | 0 to 65,535 |
| int | System.Int32 | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| uint | System.UInt32 | 4 bytes | 0 to 4,294,967,295 |
| long | System.Int64 | 8 bytes | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | System.UInt64 | 8 bytes | 0 to 18,446,744,073,709,551,615 |

These integer types may be used according to our needs. No one, (except perhaps for some biblical people), can be older than 120, 130 years. We can then use the byte type for age variable in a program. This will save some memory.

using System;

class Overflow

{

static void Main()

{

byte a = 254;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

a++;

Console.WriteLine(a);

}

}

In this example, we try to assign a value beyond the range of a data type. This leads to an arithmetic overflow. An *arithmetic overflow* is a condition that occurs when a calculation produces a result that is greater in magnitude than that which a given register or storage location can store or represent.

$ ./overflow.exe

254

255

0

1

In C#, when an overflow occurs, the variable is reset to the lower bound of the data type. (In case of a byte type it is zero.) In contrast, Visual Basic would throw an exception.

Integers can be specified in two different *notations* in C#: decimal and hexadecimal. There are no notations for octal or binary values. Decimal numbers are used normally as we know them. Hexadecimal numbers are preceded with 0x characters.

using System;

class Notations

{

static void Main()

{

int num1 = 31;

int num2 = 0x31;

Console.WriteLine(num1);

Console.WriteLine(num2);

}

}

We assign 31 to two variables using two different notations. And we print them to the console.

$ ./intnotations.exe

31

49

The default notation is the decimal. The program shows these two numbers in decimal. In other words, hexadecimal 0x31 is 49 decimal.

If we work with integers, we deal with discrete entities. We would use integers to count apples.

using System;

public class Apples

{

static void Main()

{

int baskets = 16;

int applesInBasket = 24;

int total = baskets \* applesInBasket;

Console.WriteLine("There are total of {0} apples", total);

}

}

In our program, we count the total amount of apples. We use the multiplication operation.

int baskets = 16;

int applesInBasket = 24;

The number of baskets and the number of apples in each basket are integer values.

int total = baskets \* applesInBasket;

Multiplying those values we get an integer too.

$ ./apples.exe

There are total of 384 apples

This is the output of the program.

**Floating point numbers**

Floating point numbers represent real numbers in computing. Real numbers measure continuous quantities, like weight, height, or speed. In C# we have three floating point types: float, double, anddecimal.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C# Alias** | **.NET Type** | **Size** | **Precision** | **Range** |
| float | System.Single | 4 bytes | 7 digits | 1.5 x 10-45 to 3.4 x 1038 |
| double | System.Double | 8 bytes | 15-16 digits | 5.0 x 10-324 to 1.7 x 10308 |
| decimal | System.Decimal | 16 bytes | 28-29 decimal places | 1.0 x 10-28 to 7.9 x 1028 |

The above table gives the characteristics of the floating point types.

By default, real numbers are double in C# programs. To use a different type, we must use a suffix. The F/f for float numbers and M/m for decimal numbers.

using System;

class Floats

{

static void Main()

{

float n1 = 1.234f;

double n2 = 1.234;

decimal n3 = 1.234m;

Console.WriteLine(n1);

Console.WriteLine(n2);

Console.WriteLine(n3);

Console.WriteLine(n1.GetType());

Console.WriteLine(n2.GetType());

Console.WriteLine(n3.GetType());

}

}

In the above program, we use three different literal notations for floating point numbers.

float n1 = 1.234f;

The f suffix is used for a float number.

double n2 = 1.234;

If we do not use a suffix, then it is a double number.

Console.WriteLine(n1.GetType());

The GetType() method returns the type of the number.

$ ./floats.exe

1.234

1.234

1.234

System.Single

System.Double

System.Decimal

This is the output.

We can use various syntax to create floating point values.

using System;

class Notations

{

static void Main()

{

float n1 = 1.234f;

float n2 = 1.2e-3f;

float n3 = (float) 1 / 3;

Console.WriteLine(n1);

Console.WriteLine(n2);

Console.WriteLine(n3);

}

}

We have three ways to create floating point values. The first is the 'normal' way using a decimal point. The second uses a scientific notation. And the last one as a result of a numerical operation.

float n2 = 1.2e-3f;

This is the scientific notation for floating point numbers. Also known as exponential notation, it is a way of writing numbers too large or small to be conveniently written in standard decimal notation.

float n3 = (float) 1 / 3;

The (float) construct is called casting. The division operation returns integer numbers by default. By casting we get a float number.

$ ./fnotations.exe

1.234

0.0012

0.3333333

This is the output of the above program.

The float and double types are inexact.

using System;

class CSharpApp

{

static void Main()

{

float n1 = (float) 1 / 3;

double n2 = (double) 1 / 3;

if (n1 == n2)

{

Console.WriteLine("Numbers are equal");

} else {

Console.WriteLine("Numbers are not equal");

}

}

}

The float and double values are stored with different precision. Caution should be exercised when comparing floating point values.

$ ./fequal.exe

Numbers are not equal

And the numbers are not equal.

Let's say a sprinter for 100m ran 9.87s. What is his speed in km/h?

using System;

class Sprinter

{

static void Main()

{

float distance;

float time;

float speed;

distance = 0.1f;

time = 9.87f / 3600;

speed = distance / time;

Console.WriteLine("The average speed of a sprinter is {0} km/h", speed);

}

}

In this example, it is necessary to use floating point values.

distance = 0.1f;

100m is 0.1 km.

time = 9.87f / 3600;

9.87s is 9.87/60\*60 h.

speed = distance / time;

To get the speed, we divide the distance by the time.

$ ./sprinter.exe

The average speed of a sprinter is 36.47416 km/h

This is the output of the sprinter.exe program.

**Enumerations**

Enumerated type (also called enumeration or enum) is a data type consisting of a set of named values. A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. Enumerations make the code more readable.

using System;

class Enumerations

{

enum Days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

static void Main()

{

Days day = Days.Monday;

if (day == Days.Monday)

{

Console.WriteLine("It is Monday");

}

Console.WriteLine(day);

foreach(int i in Enum.GetValues(typeof(Days)))

Console.WriteLine(i);

}

}

In our code example, we create an enumeration for week days.

enum Days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

The enumeration is created with a enum keyword. The Monday, Tuesday ... barewords store in fact numbers 0..6.

Days day = Days.Monday;

We have a variable called day which is of the enumerated type Days. It is initialized to Monday.

if (day == Days.Monday)

{

Console.WriteLine("It is Monday");

}

This code is more readable than comparing a day variable to some number.

Console.WriteLine(day);

This line prints Monday to the console.

foreach(int i in Enum.GetValues(typeof(Days)))

Console.WriteLine(i);

This loop prints 0..6 to the console. We get underlying types of the enum values. For a computer, anenum is just a number. The typeof is an operator used to obtain the System.Type object for a type. It is needed by the GetValues() method. This method returns an array of the values of a specified enumeration. And the foreach keyword goes through the array, element by element and prints them to the terminal.

We further work with enumerations.

using System;

class Enumerations2

{

public enum Seasons : byte

{

Spring = 1,

Summer = 2,

Autumn = 3,

Winter = 4

}

static void Main()

{

Seasons s1 = Seasons.Spring;

Seasons s2 = Seasons.Autumn;

Console.WriteLine(s1);

Console.WriteLine(s2);

}

}

Seasons can be easily used as enums. We can specify the underlying type for the enum and we can give exact values for them.

public enum Seasons : byte

{

Spring = 1,

Summer = 2,

Autumn = 3,

Winter = 4

}

With a colon and a data type we specify the underlying type for the enum. We also give each member a specific number.

Console.WriteLine(s1);

Console.WriteLine(s2);

These two lines print the enum values to the console.

$ ./seasons.exe

Spring

Autumn

This is the output of the seasons.exe program.

**Strings and chars**

The string is a data type representing textual data in computer programs. A string in C# is a sequence of Unicode characters. A char is a single Unicode character. Strings are enclosed by double quotes.

Since strings are very important in every programming language, we will dedicate a whole chapter to them. Here we only present a small example.

using System;

class Strings

{

static void Main()

{

string word = "ZetCode";

char c = word[0];

Console.WriteLine(c);

}

}

The program prints 'Z' character to the terminal.

string word = "ZetCode";

Here we create a string variable and assign it the "ZetCode" value.

char c = word[0];

A string is an array of Unicode characters. We can use the array access notation to get a specific character from the string. The number inside the square brackets is the index into the array of characters. The index is counted from zero. It means that the first character has index 0.

$ ./char.exe

Z

The program prints the first character of the "ZetCode" string to the console.

**Arrays**

The array is a complex data type which handles a collection of elements. Each of the elements can be accessed by an index. All the elements of an array must be of the same data type.

We dedicate a whole chapter to arrays; here we show only a small example.

using System;

class ArrayExample

{

static void Main()

{

int[] numbers = new int[5];

numbers[0] = 3;

numbers[1] = 2;

numbers[2] = 1;

numbers[3] = 5;

numbers[4] = 6;

int len = numbers.Length;

for (int i=0; i<len; i++)

{

Console.WriteLine(numbers[i]);

}

}

}

In this example, we declare an array, fill it with data and then print the contents of the array to the console.

int[] numbers = new int[5];

We declare an integer array which can store up to 5 integers. So we have an array of five elements, with indexes 0..4.

numbers[0] = 3;

numbers[1] = 2;

numbers[2] = 1;

numbers[3] = 5;

numbers[4] = 6;

Here we assign values to the created array. We can access the elements of an array by the array access notation. It consists of the array name followed by square brackets. Inside the brackets we specify the index to the element that we want.

int len = numbers.Length;

Each array has a Length property which returns the number of elements in the array.

for (int i=0; i<len; i++)

{

Console.WriteLine(numbers[i]);

}

We traverse the array and print the data to the console.

**DateTime**

The DateTime is a value type. It represents an instant in time, typically expressed as a date and time of day.

using System;

class DateTimeExample

{

static void Main()

{

DateTime today;

today = DateTime.Now;

System.Console.WriteLine(today);

System.Console.WriteLine(today.ToShortDateString());

System.Console.WriteLine(today.ToShortTimeString());

}

}

We show today's date in three different formats: date & time, date, and time.

DateTime today;

We declare a variable of DateTime data type.

today = DateTime.Now;

Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.

System.Console.WriteLine(today);

This line prints the date in full format.

System.Console.WriteLine(today.ToShortDateString());

System.Console.WriteLine(today.ToShortTimeString());

The ToShortDateString() returns a short date string format, the ToShortTimeString() returns a short time string format.

$ ./date.exe

10/15/2010 10:56:37 AM

10/15/2010

10:56 AM

We see the output of the example.

**Type casting**

We often work with multiple data types at once. Converting one data type to another one is a common job in programming. *Type conversion* or *typecasting* refers to changing an entity of one data type into another. There are two types of conversion: implicit and explicit. Implicit type conversion, also known as coercion, is an automatic type conversion by the compiler.

using System;

class ImplicitTypeConversion

{

static void Main()

{

int val1 = 0;

byte val2 = 15;

val1 = val2;

Console.WriteLine(val1.GetType());

Console.WriteLine(val2.GetType());

Console.WriteLine(12 + 12.5);

Console.WriteLine("12" + 12);

}

}

In this example, we have several implicit conversions.

val1 = val2;

Here we work with two different types: int and byte. We assign a byte value to an int value. It is a widening operation. The int values have four bytes; byte values have only one byte. *Widening*conversions are allowed. If we wanted to assign a int to a byte, this would be a *shortening*conversion. Implicit shortening conversions are not allowed by C# compiler. This is because in implicit shortening conversion we could unintentionally loose precision. We can do shortening conversions, but we must inform the compiler about it. That we know what we are doing. It can be done with explicit conversion.

Console.WriteLine(12 + 12.5);

We add two values, one integer and one floating point value. The result is a floating point value. It is a widening implicit conversion.

Console.WriteLine("12" + 12);

The result is 1212. An integer is converted to a string and the two strings are concatenated.

Next we will show some explicit conversions in C#.

using System;

class ExplicitTypeConversion

{

static void Main()

{

float a;

double b = 13.5;

int c;

a = (float) b;

c = (int) a;

Console.WriteLine(a);

Console.WriteLine(b);

Console.WriteLine(c);

}

}

We have three values. We do some explicit conversions with these values.

float a;

double b = 13.5;

int c;

We have a float value, a double value and an int value.

a = (float) b;

We convert a double value to a float value. Explicit conversion is done by specifying the intended type between two square brackets. In this case, no precision is lost. 13.5 can be safely assigned to both types.

c = (int) a;

We convert a float value to int value. In this statement, we loose some precision. 13.5 becomes 13.

$ ./explicit.exe

13.5

13.5

13

We see the output of the explicit.exe program.

**Nullable types**

Value types cannot be assigned a null literal, reference types can. Applications that work with databases deal with the null value. Because of this, special nullable types were introduced into the C# language. Nullable types are instances of the System.Nullable<T> struct.

using System;

class NullableType

{

static void Main()

{

Nullable<bool> male = null;

int? age = null;

Console.WriteLine(male.HasValue);

Console.WriteLine(age.HasValue);

}

}

A simple example demonstrating nullable types.

Nullable<bool> male = null;

int? age = null;

There are two ways how to declare a nullable type. Either with the Nullable<T> generic structure in which the type is specified between the angle brackets. Or we can use a question mark after the type. The latter is in fact a shorthand for the first notation.

$ ./nullabletypes.exe

False

False

This is the output of the example.

**Convert & Parse methods**

There are two groups of methods which are used to convert values.

using System;

class CSharpApp

{

static void Main()

{

Console.WriteLine(Convert.ToBoolean(0.3));

Console.WriteLine(Convert.ToBoolean(3));

Console.WriteLine(Convert.ToBoolean(0));

Console.WriteLine(Convert.ToBoolean(-1));

Console.WriteLine(Convert.ToInt32("452"));

Console.WriteLine(Convert.ToInt32(34.5));

}

}

The Convert class has many methods for converting values. We use two of them.

Console.WriteLine(Convert.ToBoolean(0.3));

We convert a double value to a bool value.

Console.WriteLine(Convert.ToInt32("452"));

And here we convert a string to an int.

using System;

class CSharpApp

{

static void Main()

{

Console.WriteLine(Int32.Parse("34"));

Console.WriteLine(Int32.Parse("-34"));

Console.WriteLine(Int32.Parse("+34"));

}

}

Converting strings to integers is a very common task. We often do such conversions when we fetch values from databases or GUI widgets.

Console.WriteLine(Int32.Parse("34"));

We use the Parse() method of the Int32 class to convert a string to int value.