**Basics of C#.NET Programming**

**(Part no. 5)**

# **More about Variables**

In our previous tutorials, we have talked a lot about the variables but let's take things to some advanced level regarding variables. Today we will talk about these topics at a theoretical level. Following are those topics we will discuss today.

* Type Conversions – In this one, you convert values from one type/datatype to another one.
* Enumerations – Variable types that have a user-defined discrete set of possible values that can be used in a human-readable way.
* Structs – Composite type of variables made up of a user-defined set of other variable types.
* Arrays – It holds multiples variables of one single type, allowing the index access to the individual values.

# **Type Conversions**

The meaning of the variable is determined by how this data is interpreted. The simplest example of this is the char type. This type represents a character in the Unicode character set using a number. The number is stored in the same way as a ushort — both of them store a number between 0 and 65535. This implies that even if it were possible to place the sequence of bits from one variable into a variable of a different type (perhaps they use the same amount of storage, or perhaps the target type has enough storage space to include all the source bits), the results might not be what you expect.

Instead of this one-to-one mapping of bits from one variable into another, you need to use type conversion on the data. Type conversion takes two forms:

## **Implicit Conversions**

Conversion from type A to type B is possible in all circumstances, and the rules for performing the conversion are simple enough for you to trust in the compiler. Implicit conversion requires no work on your part and no additional code. Consider the code shown here:

var1 = var2;

This assignment may involve an implicit conversion if the type of var2 can be implicitly converted into the type of var1. However, it could just as easily involve two variables with the same type, in which case no implicit conversion is necessary. For example, the values of ushort and char are effectively interchangeable, because both store a number between 0 and 65535. You can convert values between these types implicitly, as demonstrated by the following code:

ushort VarOne;

char VarTwo = 'a';

VarOne = VarTwo;

Console.WriteLine($"Variable Two value: {VarTwo}");

Console.WriteLine($"Variable One value: {VarOne}");

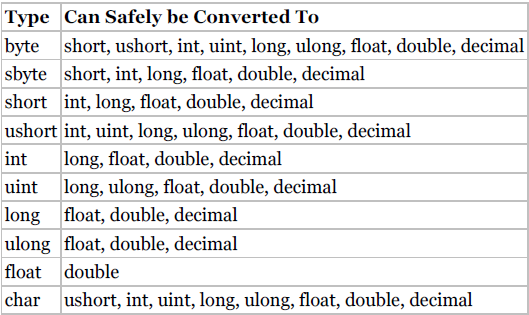
Here, the value stored in VarTwo is placed in VarOne. When you output the variables with the two Console.WriteLine() commands, you get the following output:

Variable Two value: a

Variable One value: 97

Even though the two variables store the same information, they are interpreted in different ways using their type.

There is a table for the types that can be converted implicitly and safely without facing any errors or omissions.



Don't worry, you don't need to learn this table by heart, because it's quite easy to work out which conversions the compiler can do implicitly.

## **Explicit Conversions**

As the name suggests, an explicit conversion occurs when you explicitly ask the compiler to convert a value from one data type to another. These conversions require extra code, and the format of this code may vary, depending on the exact conversion method. For example, the following modification to the code from the last section attempts to convert a short value into a byte:

byte VarOne;

short VarTwo = 7;

VarOne = VarTwo;

Console.WriteLine($"Variable Two value: {VarTwo}");

Console.WriteLine($"Variable One value: {VarOne}");

If you attempt to compile the preceding code, you will receive the following error:

Cannot implicitly convert type 'short' to 'byte'. An explicit conversion

exists

(are you missing a cast?)

You can, therefore, modify your example using this syntax to force the conversion from a short to a byte:

VarTwo = 7;

VarOne = (byte)VarTwo;

Console.WriteLine($"Variable Two value: {VarTwo}");

Console.WriteLine($"Variable One value: {VarOne}");

This results in the following output:

Variable Two value: 7

Variable One value: 7

# **Complex Variables Types**

In addition to all the simple variable types, C# also offers three slightly more complex sorts of variables:

* Enumerations (often referred to as enums)
* Structs (occasionally referred to as structures)
* Arrays

## **Enumerations**

Each of the types you've seen so far (except for string) has a clearly defined set of allowed values. Admittedly, this set is so large in types such as double that it can practically be considered a continuum, but it is a fixed set nevertheless. In situations like this, enumerations can be very useful. Enumerations do exactly what you want in this orientation type: They allow the definition of a type that can take one of a finite set of values that you supply. What you need to do, is create your enumeration type called orientation that can take one of the four possible values.

You can use the enum keyword to define enumerations as follows:

enum <typeName>

{

<value1>,

<value2>,

<value3>,

…

<valueN>

}

Next, you can declare variables of this new type as follows:

<typeName> <varName>;

You can assign values using the following:

<varName> = <typeName>.<value>;

Enumerations have an underlying type used for storage. Each of the values that an enumeration type can take is stored as a value of this underlying type, which by default is int.

## **Structs**

The struct (short for structure) is just that. That is, structs are data structures composed of several pieces of data, possibly of different types. They enable you to define your types of variables based on this structure. For example, suppose that you want to store the route to a location from a starting point, where the route consists of a direction and a distance in miles. For simplicity, you can assume that the direction is one of the compass points (such that it can be represented using the orientation enumeration from the last section), and that distance in miles can be represented as a double type. You could use two separate variables for this using code you've seen already:

orientation myVal1;

double myVal2;

There is nothing wrong with using two variables like this, but it is far simpler (especially where multiple routes are required) to store this information in one place.

To allow the code that calls the struct to access the struct's data members, you use the keyword public for <accessibility>. For example:

struct route

{

public orientation myVal1;

public double myVal2;

}

Once you have a struct type defined, you use it by defining variables of the new type:

route myRoute;

That was it for the Structs lets talk about some arrays.

## **Arrays**

All the types you've seen so far have one thing in common, each of them stores a single value (or a single set of values in the case of structs). Sometimes, in situations where you want to store a lot of data, this isn't very convenient. You may want to store several values of the same type at the same time, without having to use a different variable for each value.

For example, suppose you want to perform some processing that involves the names of all your friends. You could use simple string variables as follows:

string friendName1 = "Todd Anthony";

string friendName2 = "Kevin Holton";

string friendName3 = "Shane Laigle";

But this looks like it will require a lot of effort, especially because you need to write different code to process each variable. You couldn't for example, iterate through this list of strings in a loop. The alternative is to use an array. Arrays are indexed lists of variables stored in a single array type variable. For example, you might have an array called friendNames that stores the three names shown in the preceding string variables. You can access individual members of the array by specifying their index in square brackets, as shown here:

friendNames[<index>]

The index is simply an integer, starting with 0 for the first entry, using 1 for the second, and so on. This means that you can go through the entries using a loop:

int i;

for (i = 0; i < 3; i++)

{

Console.WriteLine($"Name with index of {i}: {friendNames[i]}");

}

Arrays have a single base type — that is, individual entries in an array are all of the same types. This friendNames array has a base type of string because it is intended for storing string variables. Array entries are often referred to as elements.

Arrays must be initialized before you have access to them. You can't just access or assign values to the array elements like this:

int[] myIntArray;

myIntArray[10] = 5;

Arrays can be initialized in two ways. You can either specify the complete contents of the array in a literal form or specify the size of the array and use the new keyword to initialize all array elements.

int[] myIntArray = { 5, 9, 10, 2, 99 }; and

int[] myIntArray = new int[5];

Here, you use the new keyword to explicitly initialize the array, and a constant value to define the size. This method results in all the array members being assigned a default value, which is 0 for numeric types. You can also use nonconstant variables for this initialization:

int[] myIntArray = new int[arraySize];

Besides, you can combine these two methods of initialization if you want:

int[] myIntArray = new int[5] { 5, 9, 10, 2, 99 };

With this method, the sizes must match. You can't, for example, write the following:

int[] myIntArray = new int[10] { 5, 9, 10, 2, 99 };

Let's talk about some other types of Arrays too.

## **Multidimensional Arrays**

A multidimensional array is simply one that uses multiple indices to access its elements. For example, suppose you want to plot the height of a hill against the position measured. You might specify a position using two coordinates, x, and y. You want to use these two coordinates as indices, such that an array called roomHeight would store the height at each pair of coordinates. This involves using multidimensional arrays.

A two-dimensional array such as this is declared as follows:

<baseType>[,] <name>;

Arrays of more dimensions simply require more commas:

<baseType>[,,,] <name>;

This would declare a four-dimensional array. Assigning values also uses similar syntax, with commas separating sizes. Declaring and initializing the two-dimensional array roomHeight, with a base type of double, an x size of 3, and a y

size of 4 requires the following:

double[,] roomHeight = new double[3,4];

Alternatively, you can use literal values for the initial assignment. Here, you use nested blocks of curly braces, separated by commas:

double[,] roomHeight = { { 1, 2, 3, 4 },{ 2, 3, 4, 5 }, { 3, 4, 5, 6 } };

This array has the same dimensions as the previous one that is, three rows and four columns. By providing literal values, these dimensions are defined implicitly.

To access individual elements of a multidimensional array, you simply specify the indices separated by commas:

roomHeight[2,1]

That’s it for today guys, I hope you have enjoyed a lot today learning about enumerations, structs, and arrays we will code all of these in our very second tutorial because today’s target was only to inculcate all these concepts into your minds we will do some hardcore practice in our next tutorials and will dive deep into this amazing world. 😉