**Value Type and Reference Type**

In C#, these data types are categorized based on how they store their value in the memory. C# includes the following categories of data types:

1. Value type
2. Reference type
3. Pointer type

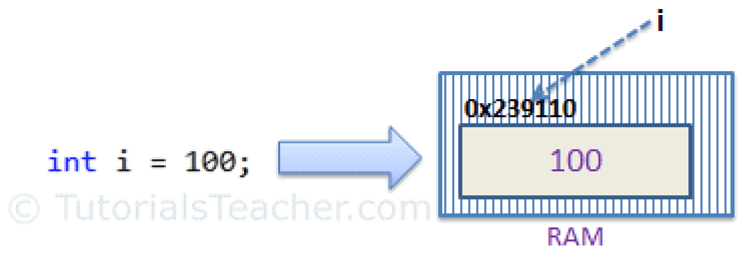
Value Type

A data type is a value type if it holds a data value within its own memory space. It means the variables of these data types directly contain values.

All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



Memory Allocation of Value Type Variable

The following data types are all of value type:

- bool
- byte
- char
- decimal
- double

- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort

Passing Value Type Variables

When you pass a value-type variable from one method to another, the system creates a separate copy of a variable in another method. If value got changed in the one method, it wouldn't affect the variable in another method.

Example: Passing Value Type Variables

```
static void ChangeValue(int x)
{
  x =  200;
  Console.WriteLine(x);
}
static void Main(string[] args)
{
  int i = 100;
  Console.WriteLine(i);
  ChangeValue(i);}
```

Try it

Output:

```
100
200
100
```

In the above example, variable i in the Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

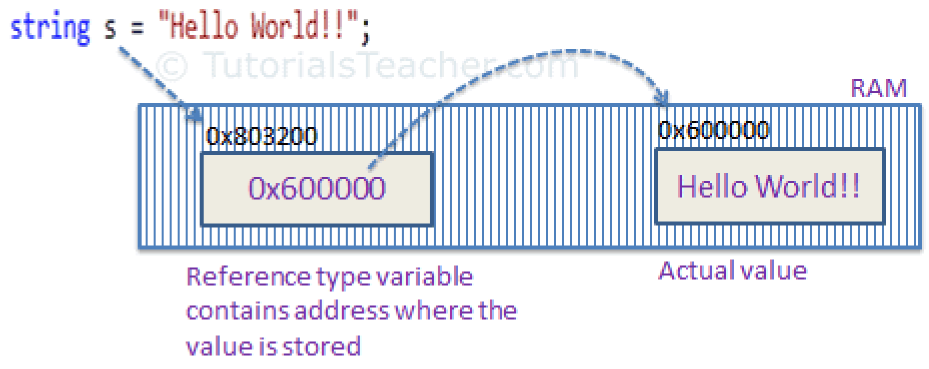
Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider the following string variable:

string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.


string s = "Hello World!!";
© TutorialsTeacher.com
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value

Memory Allocation of Reference Type Variable

As you can see in the above image, the system selects a random location in memory (0x803200) for the variable s. The value of a variable s is 0x600000, which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of the value itself.

The followings are reference type data types:

- String
- Arrays (even if their elements are value types)
- Class
- Delegate

Passing Reference Type Variables

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the variable's address. So, If we change the value of a variable in a method, it will also be reflected in the calling method.

Example: Passing Reference Type Variable

```
static void ChangeReferenceType(Student std2)
{
   std2.StudentName = "Steve";
}
static void Main(string[] args)
{
   Student std1 = new Student();
   std1.StudentName = "Bill";
   ChangeReferenceType(std1);
   Console.WriteLine(std1.StudentName);
}
```

Try it

Output:

```
Steve
```

In the above example, we pass the Student object std1 to the ChangeReferenceType() method. Here, it actually pass the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1 object, because std1 and std2 are both pointing to the same address in memory.

String is a reference type, but it is immutable. It means once we assigned a value, it cannot be changed. If we change a string value, then the compiler creates a new string object in the memory and point a variable to the new memory location. So, passing a string value to a function will create a new variable in the memory, and any change in the value in the function will not be reflected in the original value, as shown below.

Example: Passing String

```
static void ChangeReferenceType(string name)

{

   name = "Steve";

}


static void Main(string[] args)

{

   string name = "Bill";

   ChangeReferenceType(name);

   Console.WriteLine(name);

}
```

Try it

Output:

```
Bill
```

ℹ️ **Information**
*The code in the following article was tested with C# 8 or above.*

In C#, we have various techniques available for controlling the passing of arguments to methods. Among these, the keywords ref, out, and in play a particularly important role. These parameter modifiers allow you to specify the way data is transferred between methods, although they have some limitations.

**Ref**

The ref keyword is used to pass an argument to a method by reference, rather than by value. This means that if a method changes the value of a ref parameter, this change will also be visible to the variable passed by the caller.

🔎 **Insight**
*In pass by value, a copy of the actual parameter value is created and passed to the function. Any modifications made to the parameter inside the function do not affect the original value outside the function.*
*In pass by reference, the reference or memory address of the parameter is passed. Any modifications made to the parameter inside the function are reflected in the original value outside the function.*

```
public void SampleMethod(ref int x)

{

  x = 10;

}

int num = 1;

SampleMethod(ref num);

Console.WriteLine(num); // Prints 10
```

It's important to highlight that to use a ref parameter, both the method definition and the calling method must explicitly use the ref keyword. Also, the variable passed as a ref parameter must be initialized before being passed to the method.

⚠ **Warning**
*Do not confuse the concept of passing by reference with the concept of reference types. The two concepts are not the same. A method parameter can be modified by ref regardless of whether it is a value type or a reference type. There is no boxing of a value type when it is passed by reference.*

**Out**

The out keyword is used to pass arguments by reference. Unlike ref, however, out does not require that the input variable be initialized before being passed. Also, the method must necessarily assign a value to the out parameter before it concludes its execution.

```
public void SampleMethod(out int x)

{

   x = 10;

}


int num;

SampleMethod(out num);

Console.WriteLine(num); // Prints 10
```

⚠ **Warning**
*The out keyword can also be used with a generic type parameter to specify that the type parameter is covariant.*

**In**

The in keyword, introduced in C# 7.2, is used to pass an argument to a method by reference but prevents the value from being modified within the method. This is particularly useful when working with large structures, because it avoids the overhead of copying, while at the same time ensuring the integrity of the data.

```
public void SampleMethod(in int x)
{
   // x = 10; // This will generate an error
   Console.WriteLine(x);
}


int num = 1;
SampleMethod(in num); // Prints 1
```

⚠ **Warning**
*The in keyword can also be used with a generic type parameter to specify that the type parameter is contravariant, as part of a foreach statement, or as part of a join clause in a LINQ query.*

**Limitations**

Despite their usefulness, the keywords ref, out, and in have some limitations:

- They cannot be used in async methods defined with the async modifier.
- They cannot be used in iterator methods that include a yield return or yield break statement.
- The first argument of an extension method cannot have the in modifier unless that argument is a struct.
- They cannot be used in the first argument of an extension method in which that argument is a generic type, even when that type is constrained to be a struct.

The restrictions also extend to extension methods:

- The out keyword cannot be used in the first argument of an extension method.
- The ref keyword cannot be used in the first argument of an extension method, unless the argument is a struct or an unconstrained generic type.
- The in keyword cannot be used unless the first argument is a struct. Furthermore, it cannot be used in any generic type, even when it is constrained to be a struct.

Additionally, it's important to remember that members of a class cannot have signatures that differ only by ref, in, or out. In practice, a compiler error will occur if the only difference between two members of a type is that one of them has a ref parameter and the other has an out or in parameter.

⛔ **Compiler Error CS0663**
*Cannot define overloaded methods that differ only on ref and out.*

```
public class TestClass

{

  public void SampleMethod(out int i) { }

  public void SampleMethod(ref int i) { }

}
```

Overloading is legal, however, if one method takes a ref, in, or out argument and the other has none of those modifiers, like this:

```
public class TestClass1

{

   public void SampleMethod(int i) { }

   public void SampleMethod(ref int i) { }

}
```

```
public class TestClass2
{
   public void SampleMethod(int i) { }
   public void SampleMethod(out int i) => i = 5;
}


public class TestClass3
{
   public void SampleMethod(int i) { }
   public void SampleMethod(in int i) { }
}
```

**Conclusion**

The ref, out, and in keywords provide detailed control over how arguments are passed to methods. Understanding and using them correctly can enhance code efficiency and safety. However, it's crucial to be aware of their limitations and use them cautiously, as improper use can render the code complex and hard to manage.