C# Data Types

The data type tells the C# compiler what kind of value a variable can hold. C# includes many in-built data types for different kinds of data, e.g. String, number, float, decimal, etc.

Example: Data types

class Program

{

static void Main(string[] args)

{

string stringVar = "Hello World!!";

int intVar = 100;

float floatVar = 10.2f;

char charVar = 'A';

bool boolVar = true;

}

}

# Value Type and Reference Type

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

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

Here, we will learn about value types and reference types.

## **Value Type:**

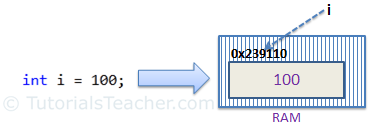
Value types are generally (not always) stored on the stack and are passed by copying.

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

[](https://www.tutorialsteacher.com/Content/images/csharp/value-type-memory-allocation.png)Memory allocation for Value Type

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 by Value:**

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: Value Type

static void ChangeValue(int x)

{

x = 200;

Console.WriteLine(x);

}

static void Main(string[] args)

{

int i = 100;

Console.WriteLine(i);

ChangeValue(i);

Console.WriteLine(i);

}

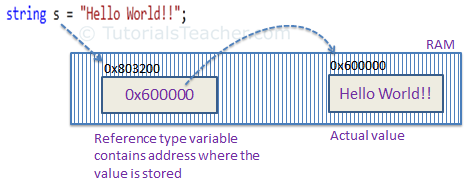
## **Reference Type**

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 following string variable:

string s = "Hello World!!";

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

[](https://www.tutorialsteacher.com/Content/images/csharp/raference-type-memory-allocation.png)Memory allocation for Reference type

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 value itself.

The following data types are of reference type:

* String
* All arrays, even if their elements are value types
* Class
* Delegates

**Boxing and Unboxing in C# .Net**  
C# provides us with Value types and Reference Types. Value Types are stored on the stack and Reference types are stored on the heap. The conversion of value type to reference type is known as boxing and converting reference type back to the value type is known as unboxing.

**Boxing**  
convert ValueTypes to Reference Types also known as boxing.

1. Int32 x = 10;
2. **object** o = x;  // Implicit boxing
3. Console.WriteLine("The Object o = {0}", o); // prints out 10
5. Int32 x = 10;
6. **object** o = (**object**)x; // Explicit Boxing
7. Console.WriteLine("The object o = {0}", o); // prints out 10

**Unboxing**

UnBoxing an object type back to value type.

1. Int32 x = 5;
2. **object** o = x; // Implicit Boxing
3. x = o; // Implicit UnBoxing
5. Int32 x = 5;
6. **object** o = x; // Implicit Boxing
7. x = (Int32)o; // Explicit UnBoxing

## **const in c#**

* You have to initialize const variables while declaration.
* You cannot reassign a const variable.
* Compiler evaluates the const variables.
* The static modifier is not allowed in a constant declaration.
* A const field of a reference type other than string can only be initialized with null.

### **Example for const in c#.net**

using System;

using System.Text;

namespace ProgramCAll

{

    class ProgramCall

    {

        /\*A const field of a reference type other than string

        can only be initialized with null\*/

        private const StringBuilder myName = null;

        public static void Main()

        {

            //You have to initilize Const varabiles while declartion

            const int myEmpId = 173524;

            //Valid scenario

            const int myNumber = 10 + 16;

            const string myName = "John";

            // Reassigning a const variable is not allowed.

            //Comment out below code to run the program

            myEmpId = 23456;

            Console.WriteLine(myEmpId);

            Console.Read();

        }

    }

}

## **readonly in c#**

* readonly fields can be initialized only while declaration or in the constructor.
* Once  you initialize a readonly field, you cannot reassign it.
* You can use static modifier for readonly fields
* readonly modifier can be used with reference types
* readonly modifier can be used only for instance or static fields, you cannot use readonly keyword for variables in the methods.

### **Example for readonly modifier in c#.net**

using System;

using System.Text;

namespace ProgramCall

{

    class MyClass

    {

        //readonly can only be used with instance or static variables

        public readonly int mynumber = 10;

        public readonly int addnumber = 10 + 15;

        //readonly modifier can be used with static varaibles

        public static string myEmployer = "ProgramCall.Com";

        //readonly can be used with reference types

        public readonly StringBuilder name = new StringBuilder("John");

        /\*  readonly varaible can be intilized in constructor but

             cannot be declared in constructor    \*/

        public readonly string myName;

        //readonly fields can be initlized in constructor

        public MyClass(string name)

        {

            myName = name;

        }

        private void myMethod()

        {

            /\* readonly modifier cannot be used in method level variables

           the below line throws a error. \*/

            //  readonly int num = 5;

        }

    }

    //main class

    class ProgramCall

    {

        public static void Main()

        {

            MyClass obj = new MyClass("Bill");

            Console.Read();

        }

    }

}

## **Difference between const and readonly**

* const fields has to be initialized while declaration only, while readonly fields can be initialized at declaration or in the constructor.
* const variables can declared in methods ,while readonly fields cannot be declared in methods.
* const fields cannot be used with static modifier, while readonly fields can be used with static modifier.
* A const field is a compile-time constant, the readonly field can be used for run time constants.

String

String is immutable, Immutable means if you create string object then you cannot modify it and It always create new object of string type in memory.

Example

string strMyValue = "Hello Visitor";

// create a new string instance instead of changing the old one

strMyValue += "How Are";

strMyValue += "You ??";

Stringbuilder

StringBuilder is mutable, means if create string builder object then you can perform any operation like insert, replace or append without creating new instance for every time.it will update string at one place in memory doesnt create new space in memory.

Example

StringBuilder sbMyValue = new StringBuilder("");

sbMyValue.Append("Hello Visitor");

sbMyValue.Append("How Are You ??");

string strMyValue = sbMyValue.ToString();