

BICTE. 465

Unit 2: Basic of C#

Prepared By: Bhesh Raj Pokhrel

1.1: Introduction (C#)

- C# is pronounced "C-Sharp".
- C# is a new computer-programming language developed by Microsoft Corporation lead by ***Anders Hejlsberg*** and his team.
- C# is a fully ***object-oriented*** language like JAVA.
- It has been designed to support the **key feature** of .NET Framework.
- It is **simple**, efficient, productive and type-safe language ***derived from the C and C++ languages.***
- The first version was released in ***year 2002***. The latest version, **C# 13**, was released in **November 2024**.

1.2. Advantages of C#:

- **Object-oriented programming:** C# is a *fully object-oriented* language, which allows developers to create reusable code and build complex applications with ease.
- **Large standard library:** C# has a *large standard library* that includes a wide range of *pre-built classes* and functions. This makes it easy for developers to perform common tasks without having to write a lot of custom code.
- **Cross-platform support:** C# can be used to develop applications for *Windows, Linux, and macOS*, and it can also be used to develop mobile and web applications.
- **Strongly typed:** C# is a strongly typed language, which means that *data types are checked at compile time*. This helps to reduce errors and improve the reliability of code.
- **Integration with Microsoft technologies:** C# is developed by Microsoft and integrates well with other Microsoft technologies, such as *ASP.NET, Azure, and Visual Studio*.

1.3. Applications: C # is used for

- C# is widely used for developing *desktop applications, web applications and web services*.
- C# is also used in *game* development.
- C# can be used for developing *machine learning applications* using frameworks such as **ML.NET**. ML.NET provides tools for training and deploying machine learning models in C# applications.
- C# can be used to *develop IoT* applications using .NET **IoT libraries**. These applications can run on devices such as *Raspberry Pi and Arduino*.
- C# can be used to create *database applications using ADO.NET or Entity Framework*. These applications can connect to various database systems, such as Microsoft SQL Server, Oracle, and MySQL.
- C# can be **used** to develop cross-platform mobile applications using frameworks such as *Xamarin*. These applications can run on Android, iOS, and Windows devices.

1.4. Characteristics/Feature of C#

1. Modern: C# is called a modern language due to a number of features it supports. It supports:

- Automatic garbage collection
- Rich intrinsic model for error handling
- Decimal data type for financial application
- Modern approach to debugging and
- Robust security model.

2. Object-Oriented : C# is truly object-oriented. It supports all the ***three tenets (principle) of object-oriented systems***, namely,

- Encapsulation
- Inheritance
- Polymorphism

In C#, ***everything is an object***. There are no more global functions, variables and constants.

1.4. Characteristics/Feature of C#

3. Type-safe: Type-safety promotes robust programs. C# incorporates a number of type-safe measures.

- All dynamically allocated object and arrays are ***initialized to zero***
- Use of any ***uninitialized variables*** produces an error message **by the compiler**
- Access to arrays are range-checked and warned if it ***goes out-of-bounds***
- C# does ***not permit unsafe casts***.
- C# enforces overflow checking in arithmetic operations.
- Reference parameters that are passed are type-safe
- C# supports automatic ***garbage collection***.

4. Compatible/ Inter-operability:

- C# enforces the .NET common language specifications and therefore ***allows inter-operation*** with other .NET languages.
- C# provides support for using COM(**Component Object Model**) objects, no matter what language was used to author them.
- C# also supports a special feature that enables a program to call out any native API.

1.5. Development Environment setup for C# programs (without IDE)

Setup the C# compiler (csc.exe):

Case 1:

- *In Windows 10 and newer version of Windows, the **built-in** .NET Framework developer pack is available .*
- *So, we just need to set the environment variable path of compiler (csc.exe) as follows.*

Open Environment Variable Windows : (My Pc -> Properties → Advanced System Setting → Environment Variables → System Variable → Path → Edit → New)

Set This Path: C:\Windows\Microsoft.NET\Framework64\v4.0.30319 \

- Then **csc (c sharp compiler)** will be executable. For confirmation just run the csc command in **DOS Prompt**.

Case 2:

- If we want to run .NET developer pack in older or newer version (except built in version) of windows then we should **Download** .NET framework and install developer packs.
- Download .NET Framework and install the **development packs**.

1.6. Compile and Run C# programs (without IDE)

A. Compile and Run Class FirstProgram.cs:

1. Open text editor (Notepad++) and create a class and name it as **FirstProgram.cs** with code for printing "Hello World".

```
using System; //import System namespace
//namespace declaration
namespace FirstApp{
    //class declaration
    class FirstProgram{
        //Main method where the execution started
        public static void Main(String[] args){
            //Printing Hello World
            Console.WriteLine("Hello World");
        }
    }
}
```

2. Steps to compile and run the program:

- Open command prompt
- Navigate to path and then compile the program: ***csc FirstPrrogram.cs***
- Then run the program : ***FirstProgram***

2: C# Data Types : Concept

- **What is a Data Type in C#?**
- The Datatypes are something that gives information about
 - **Size** of the memory location.
 - The **Range of data** that can be stored inside that memory location
 - Possible **Legal Operations** that can be performed on that memory location.
 - What **Types of Results** come out from an expression when these types are used inside that expression?

The keyword which gives all the above information is called the data type in C#.

- Basically data types specify the **size and type** of values that can be stored.
- C# is a **strongly typed programming language** because in C# each type of data (such as integer, character, float etc) is predefined as part of the programming language and all constants or variables defined for a given program ***must be described with one of the data types.***

Data Types in C# are primarily divided into two categories:

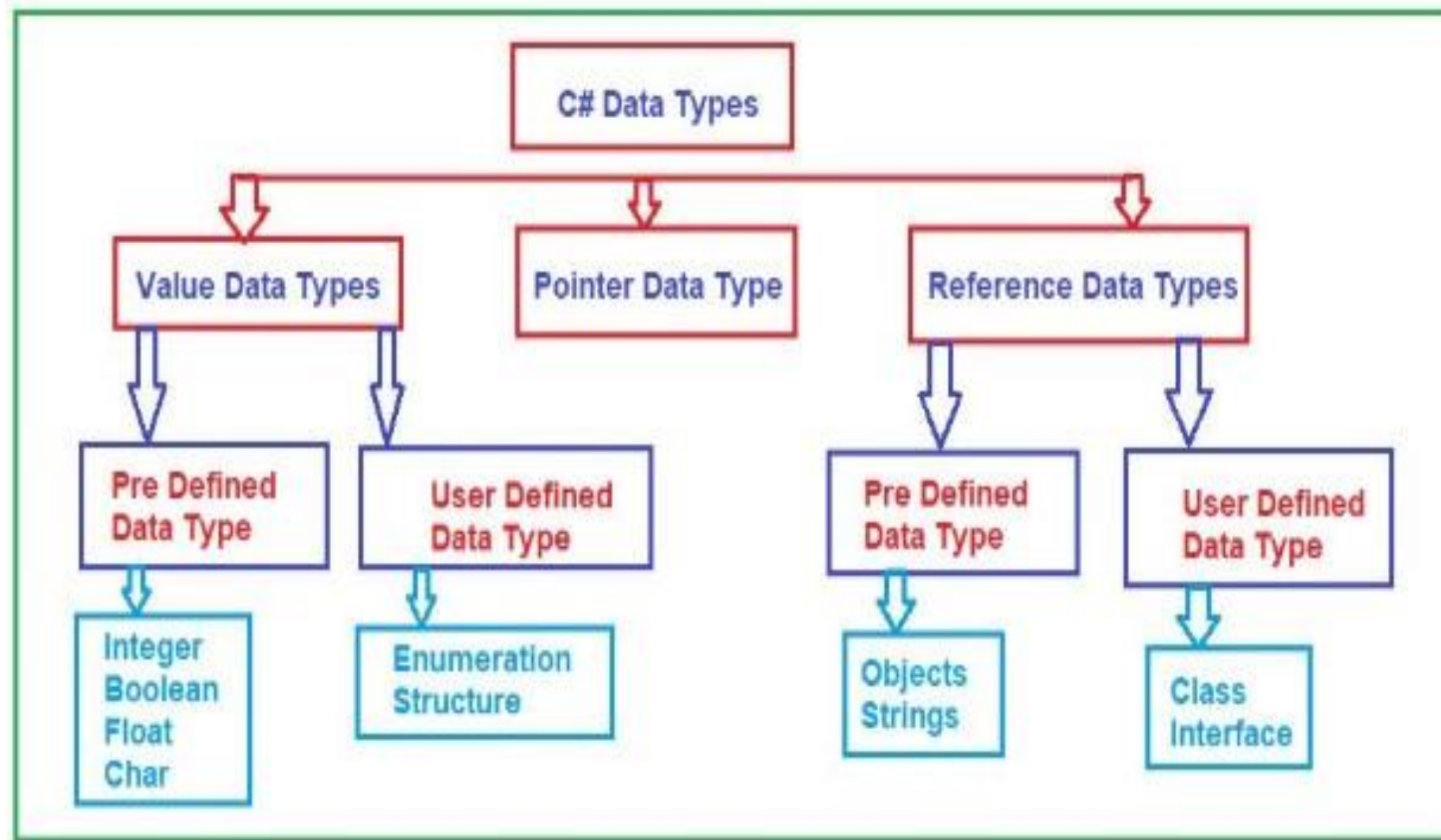
1. Value Data Types
2. Reference Data Types

2: C# Data Types : Catagories

- A data type in C# **specifies** the **type of data that a variable can store** such as integer, floating, boolean, character, string, etc.
- The following diagram shows the different types of data types available in C#.

There are **3 types of data types** available in the C# language.

- 1. Value Data Types**
- 2. Reference Data Types**
- 3. Pointer Data Types**



2: C# Data Types : Value Types

- ***Value types*** and ***reference types*** differ in two characteristics:
 - Where they are stored in the memory
 - How they behave in the context of assignment statements.

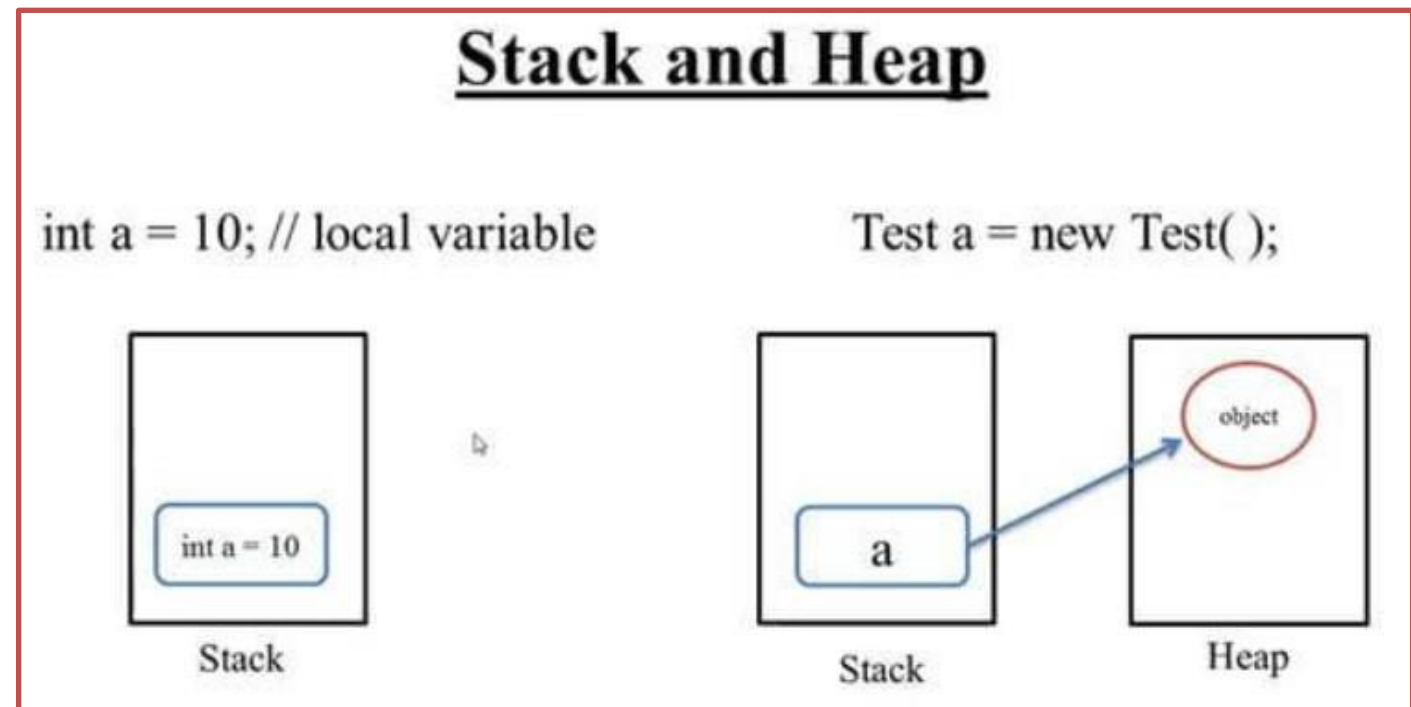
1. Value types (which are of fixed length)

- The data type which **stores the value directly in the memory** is called the Value Data Type in C#. The **examples** are *int*, *char*, *boolean*, and *float* which store numbers, alphabets, true/false, and floating-point numbers respectively.
- Value types are stored on the **stack**, and when a value of variable is assigned to another variable, the ***value is actually copied***.
- This means that ***two identical copies*** of the value are available in memory.
- The value data types in C# **again classified** into two types are as follows.
- **Predefined Data Types** – Example includes Integer, Boolean, Boolean, Long, Double, Float, etc.
- **User-defined Data Types** – Example includes Structure, Enumerations, etc.

2: C# Data Types : Reference Types

2. Reference types (Which are of variable length):

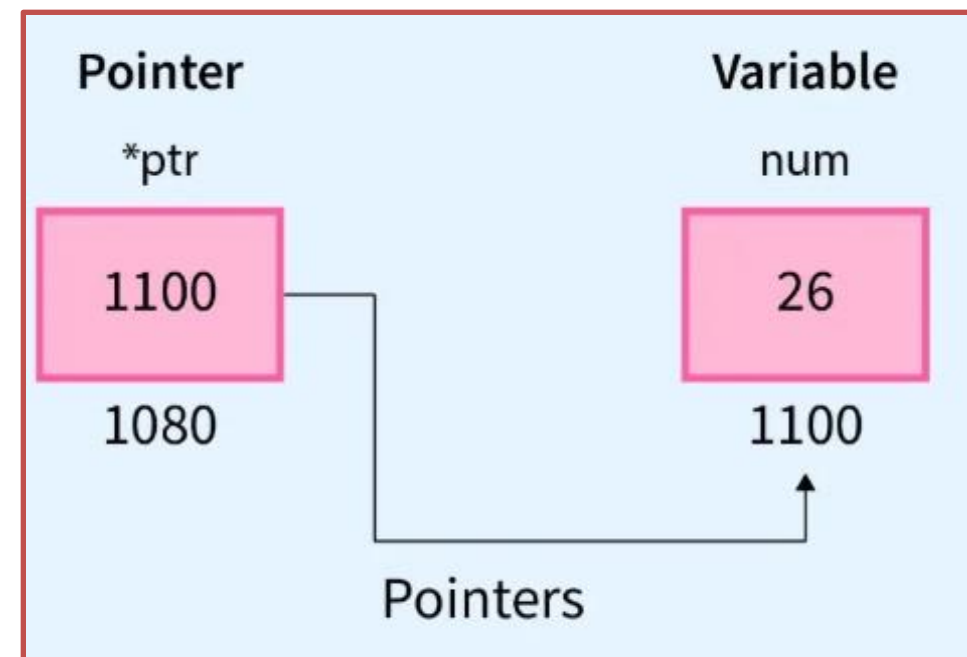
- The data type which is **used to store the reference of a variable** is called Reference Data Type.
- The data types which are stored on the **heap**, and when an assignment between two **reference variables** occurs, only the **reference is copied**; the actual value remains in the same memory location.
- In other words, we can say that the reference types **do not store the actual data** stored in a variable, rather they store the reference to the variables.
- This means that there are two references to single value.
- Again, the Reference Data Types are categorized into 2 types. They are as follows.
- **Predefined Types** – Examples include Objects, Strings, and dynamics.
- **User-defined Types** – Examples include Classes and Interfaces.



2: C# Data Types : Pointer Types

3. What is Pointer Type in C#?

- The pointer in C# language is a **variable**, it is also known as a **locator** or **indicator** that **points to an address of the value(another variable)**.
- Which means pointer-type variables **stores the memory address of another type**. To get the pointer details we have two symbols ampersand (&) and asterisk (*).
- **ampersand (&)**: It is known as **Address Operator**. It is used to determine the address of a variable.
- **asterisk (*)**: It is also known as **Indirection Operator**. It is **used to access the value** of an address.
- For a better understanding, **look at the below example** which shows the use of the Pointer data Type in C#.
- **In order to run** the below program, we need to use **unsafe mode**.



2: C# Data Types : Pointer Types (Example)

```
using System;

class PointerTypeExample
{
    static void Main(string[] args)
    {
        unsafe
        {
            // declare a variable
            int number = 10;
            // store variable number address location in pointer variable ptr
            int* ptr = &number;

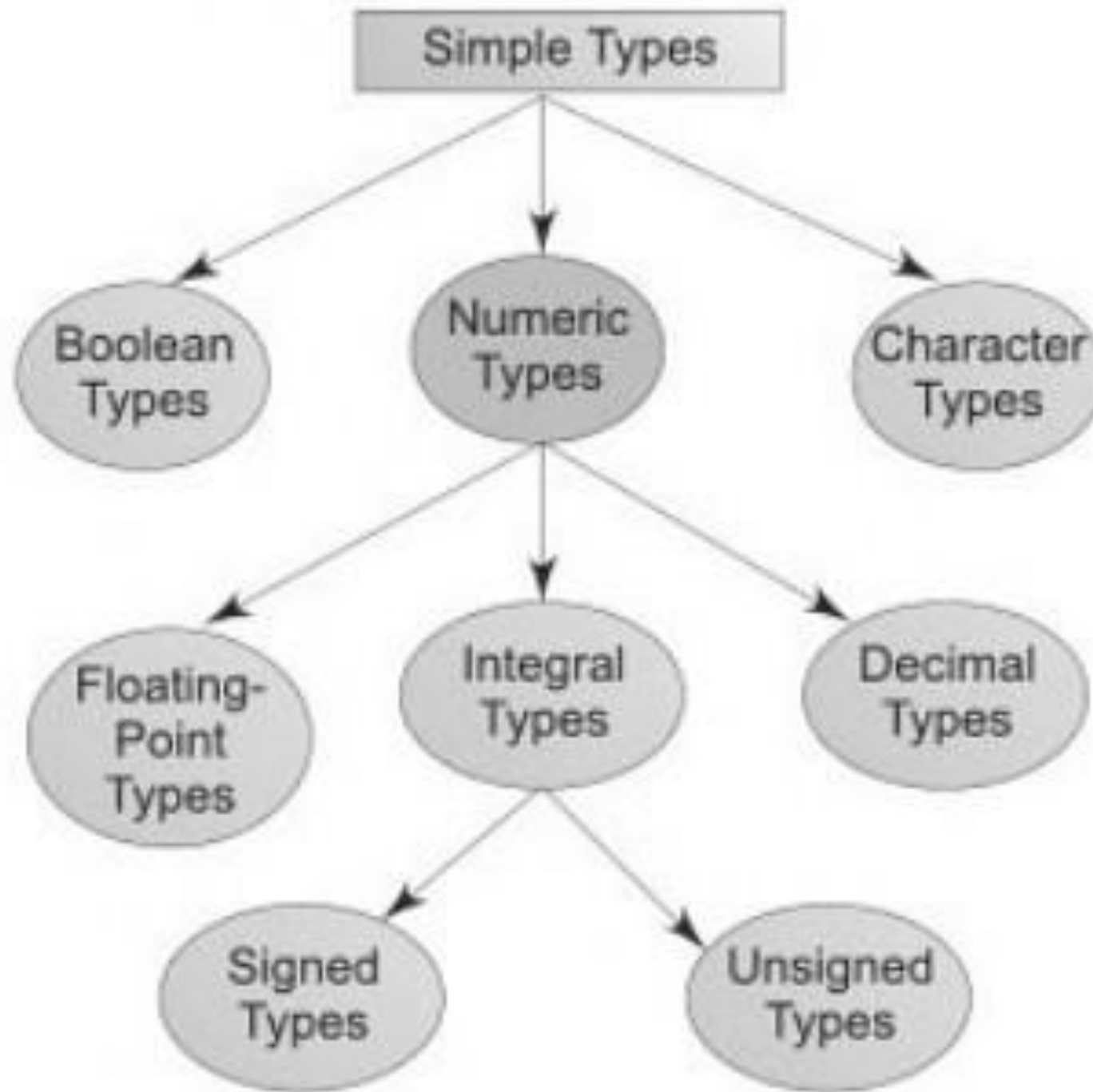
            Console.WriteLine("simple/direct value (number) =" + number);
            Console.WriteLine("Address of pointer variable hold (ptr) =" + ((int)ptr));
            Console.WriteLine("Pointed Value (*ptr) =" + (*ptr));
        }
    }
}
```

```
C:\C#_Projects\batch2\dataTypes>csc /unsafe PointerTypeExample.cs
```

//compile with **unsafe**
mode

```
C:\C#_Projects\batch2\dataTypes>PointerTypeExample
simple/direct value (number) =10
Address of pointer variable hold (ptr)=7334204
Pointed Value (*ptr)=10
```

Categories of primitive/predefined types



2.1.1: Integral Types

- Integral types can hold ***whole numbers*** such as 123, -34.
- The size of the values that can be stored depends on the integral data type we choose.
- C# supports the concept of ***unsigned types*** and therefore it supports eight types of integers as shown in following figure.

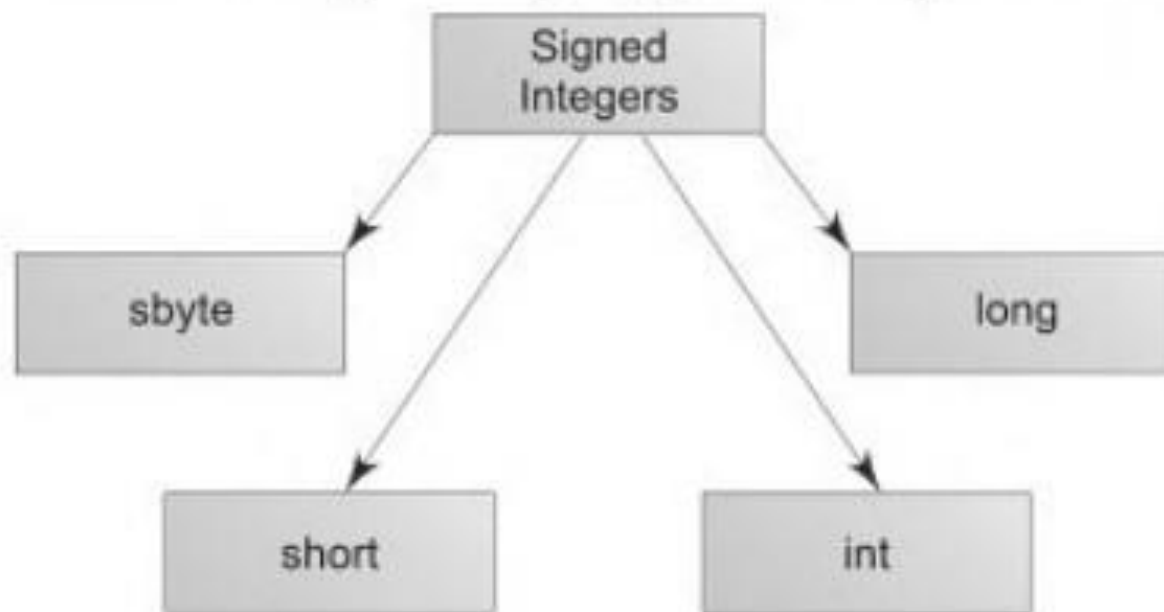


Fig. 4.4 Signed integer types

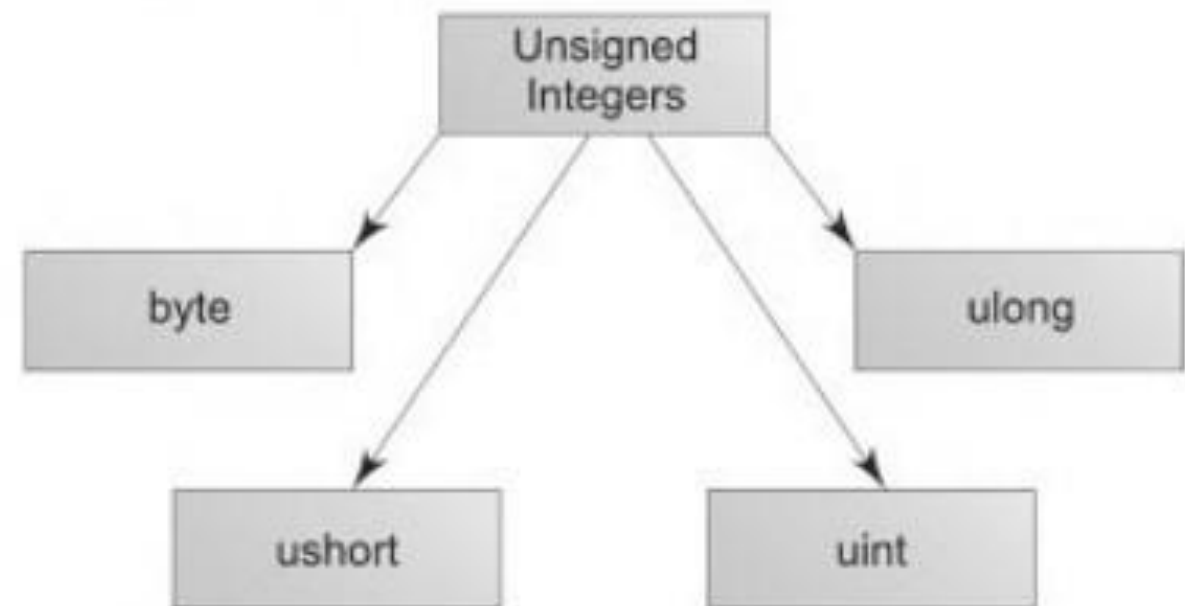


Fig. 4.5 Unsigned integer types

Signed Integers

- Signed integer types can hold both positive and negative numbers. The memory size and range of all the four signed integer data types are as follows .

Table 4.3 *Size and range of signed integer types*

<i>TYPE</i>	<i>SIZE</i>	<i>MINIMUM VALUE</i>	<i>MAXIMUM VALUE</i>
sbyte	One byte	-128	127
short	Two bytes	-32,768	32,767
int	Four bytes	-2,147,483,648	2,147,483,647
long	Eight bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Unsigned Integers

- We can increase the size of positive value stored in an integer type by making it '*unsigned*'.
- The memory size and range of all four unsigned integer data types are as follows:

Table 4.4 *Size and range of unsigned integer types*

<i>TYPE</i>	<i>SIZE</i>	<i>MINIMUM VALUE</i>	<i>MAXIMUM VALUE</i>
byte	One byte	0	255
ushort	Two bytes	0	65,535
uint	Four bytes	0	4,294,967,295
ulong	Eight bytes	0	18,446,744,073,709,551,615

Unsigned Integers

- All integers are by default **int** type. In order to specify which of the other integer types the value must take, we must ***append*** the characters ***U, L or UL*** as shown below:
 - 246U (for uint type)
 - 246L (for long type)
 - 246UL (for ulong type).

2.1.2: Floating-Point Types

- Integer types can hold ***only whole numbers*** and therefore we use another type known as floating-point type to hold numbers containing ***fractional parts*** such as 25.98 and -1.375.
- There are two kinds of floating point storage in C# as shown in following figure:

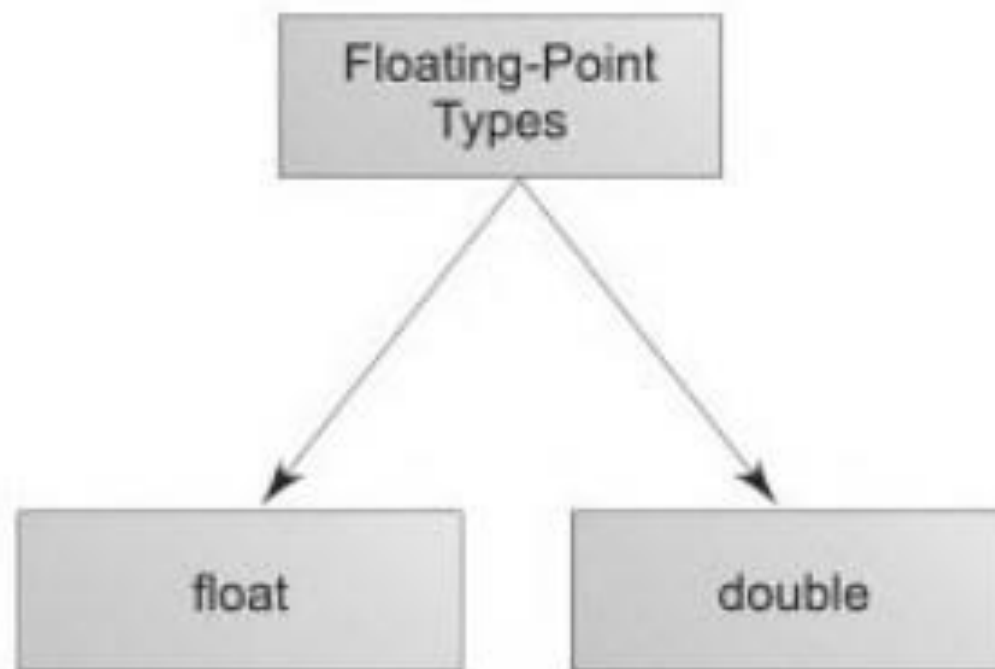


Fig. 4.6 *Floating-point data types*

Floating-Point Types

- The float type value are *single-precision* numbers with a precision of *seven digits*. The double types represent double-precision numbers with a precision of *15/16 digits*.
- **The size and range of floating-point types:**

Table 4.5 *Size and range of floating-point types*

<i>TYPE</i>	<i>SIZE</i>	<i>MINIMUM VALUE</i>	<i>MAXIMUM VALUE</i>
float	4 bytes	1.5×10^{-45}	3.4×10^{38}
double	8 bytes	5.0×10^{-324}	1.7×10^{308}

Floating-Point Types

- **Floating-point** numbers, *by default, as double-precision* quantities. To force them to be in single precision mode, we must *append f or F* to the numbers.
- For example: 1.23f or 1.23F
- **Double-precision** types are used when we need greater precision in storage of floating-point numbers.
- Floating-point data types support a special value known as ***Not-a-Number (NaN)***. NaN is used to represent the result of operations such as *dividing zero by zero*, where actual number is not produced.

2.1.4:Character Type

- In order to ***store single characters*** in memory, C# provides a character data type called char.
- The char type assumes a size of ***two bytes*** but, in fact it can hold only a single character.
- It has been designed to hold a ***16-bit Unicode character***, in which the 8-bit ASCII code is a subset.

2.1.5: Boolean Type

- Boolean type is used when ***we want to test*** a particular condition during the execution of the program.
- There are only two values that a Boolean type can take: ***true or false***.
- Boolean type is denoted by the keyword ***bool*** and uses only one bit of storage.
- In contrast to C and C++, in C#, we ***cannot use zero*** for false and non-zero for true.
- No conversion between bool type and other integer types is possible.

2.2: Operators and Expressions

- Operators in C# are **symbols** that are used to perform **operations** on **operands**. The Operators in C# are used to manipulate the variables and values in a program.
- For example:
 - `int x = 10, y = 20;`
 - `int result1 = x + y;` //Operator Manipulating Variables, where x and y are variables and + is operator
 - `int result2 = 10 + 20;` //Operator Manipulating Values, where 10 and 20 are value and + is operator
- **Note:** In the above example, x, y, 10, and 20 are called Operands. So, the operand may be variables or values.
- C# supports a **rich set of operators**. C# operators can be classified into a number of related categories as below:

- | | |
|--------------------------------------|--------------------------|
| 1. Arithmetic Operators | 6. Conditional operators |
| 2. Relational Operators | 7. Bitwise Operators |
| 3. Logical Operators | 8. Special operators |
| 4. Assignment Operators | |
| 5. Increment and decrement operators | |

2.2: Operators and Expressions

- In C#, the Operators can also be categorized based on the Number of Operands:

Unary Operator: The Operator that requires **one operand** (variable or value) to perform the operation is called Unary Operator.

Binary Operator: Then Operator that requires **two operands** (variables or values) to perform the operation is called Binary Operator.

Ternary Operator: The Operator that requires **three operands** (variables or values) to perform the operation is called Ternary Operator. The Ternary Operator is also called **Conditional Operator**.

	Operator	Type
Binary Operator →	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator →	++, --	Unary Operators
Ternary Operator →	?:	Ternary Operator or Conditional Operator

2.2.1:Arithmetic Operators

- The Arithmetic Operators in C# are used to **perform arithmetic/mathematical operations** like addition, subtraction, multiplication, division, etc. on operands.
- These can operate on any ***built-in numeric data type(int,long,float,double etc)***. We **cannot use** these operators on **Boolean** type.
- The following Operators are falling into this category.

Operator	Meaning	Example	Result
+	Adds two operands	int c = 10 + 5;	c = 15
-	Subtracts right operand from left operand	int c = 10 - 5;	c = 5
*	Multiplies two operands	int c = 10 * 5;	c = 50
/	Divides left operand by right operand	int c = 10 / 5;	c = 2
%	Returns the remainder after division	int c = 10 % 5;	c = 0

Example to Understand Arithmetic Operators in C#:

```
using System;
class ArithmeticOperators
{
    static void Main(string[] args)
    {
        int Result;
        int Num1 = 20, Num2 = 10;

        // Addition Operation
        Result = (Num1 + Num2);
        Console.WriteLine("Addition Operator:"+Result );

        // Subtraction Operation
        Result = (Num1 - Num2);
        Console.WriteLine("Subtraction Operator:"+Result);

        // Multiplication Operation
        Result = (Num1 * Num2);
        Console.WriteLine("Multiplication Operator:"+Result);

        // Division Operation
        Result = (Num1 / Num2);
        Console.WriteLine("Division Operator: "+Result);

        // Modulo Operation
        Result = (Num1 % Num2);
        Console.WriteLine("Modulo Operator: "+Result);
        // Console.WriteLine($"Modulo Operator: {Result}");
    }
}
```

2.2:Relational Operators

- The Relational Operators in C# are **also known** as Comparison Operators.
- It determines the **relationship between two operands** and returns the **Boolean results**, i.e. **true** or **false** after the comparison.
- The Different Types of Relational Operators supported by C# are as follows.

Operator	Meaning	Example	Result
==	Checks if two operands are equal	5 == 3	False
!=	Checks if two operands are not equal	5 != 3	True
<	Checks if left operand is less than right operand	5 < 3	False
<=	Checks if left operand is less than or equal to right operand	5 <= 5	True
>	Checks if left operand is greater than right operand	5 > 3	True
>=	Checks if left operand is greater than or equal to right operand	5 >= 5	True

2.2.2:Relational Operators

- When arithmetic expressions are used on either side of a relational operator, the ***arithmetic expressions will be evaluated first and then the results compared.***
- That is, ***arithmetic operators*** have a ***higher priority*** over relational operators.

Eg: (b==a+c)

- Relational expressions are used in ***decision statements*** such **if** and **while** to decide the course of action of running program.
- The following program shows the implementation of relational operators.

```
using System;
```

```
class RelationalOperators
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        bool Result;
```

```
        int Num1 = 5, Num2 = 10;
```

```
        // Equal to Operator
```

```
        Result = (Num1 == Num2);
```

```
        Console.WriteLine("Equal (==) to Operator: " + Result);
```

```
        // Greater than Operator
```

```
        Result = (Num1 > Num2);
```

```
        Console.WriteLine("Greater (>) than Operator: " + Result);
```

```
        // Less than Operator
```

```
        Result = (Num1 < Num2);
```

```
        Console.WriteLine("Less than (<) Operator: " + Result);
```

```
        // Greater than Equal to Operator
```

```
        Result = (Num1 >= Num2);
```

```
        Console.WriteLine("Greater than or Equal to (>=) Operator: " + Result);
```

```
        // Less than Equal to Operator
```

```
        Result = (Num1 <= Num2);
```

```
        Console.WriteLine("Lesser than or Equal to (<=) Operator: " + Result);
```

```
        // Not Equal To Operator
```

```
        Result = (Num1 != Num2);
```

```
        Console.WriteLine("Not Equal to (!=) Operator: " + Result);
```

```
    }
```

```
}
```

2.2.3:Logical Operators

- The operators which are used to ***combines two or more relational expressions*** in order to form a **logical expression** is known as Logical Operators.
- The Logical Operators are **mainly used** in conditional statements and loops for evaluating a condition.
- Like the simple relational expressions, a logical expression also yields (allow) a value of ***true or false***.
- The different types of Logical Operators supported in C# are as follows:

Operator	Meaning	Example	Result / Explanation
	Returns true either if one conditions is true	true false	true – one condition is true.
&&	Returns true only if both conditions are true	true && false	false — both conditions not true
!	Returns true if the condition is false (reverses result)	!true	false — negates the value

Example : Logical Operators

```
//Example to Understand Logical Operators in C#:  
using System;  
  
class LogicalOperators  
{  
    static void Main(string[] args)  
    {  
        bool x = true, y = false, z;  
        //Logical AND operator  
        z = x && y;  
        Console.WriteLine("Logical AND Operator (&&) : " + z);  
        //Logical OR operator  
        z = x || y;  
        Console.WriteLine("Logical OR Operator (||) : " + z);  
        //Logical NOT operator  
        z = !x;  
        Console.WriteLine("Logical NOT Operator (!) : " + z);  
    }  
}
```

2.2.4: Assignment Operators

- The Assignment Operators in C# **are used to assign a value to a variable.**
- The **left-hand side operand** of the assignment operator is a variable and the **right-hand side operand** of the assignment operator **can be a value or an expression** that must return some value and that value is going to assign to the left-hand side variable.
- The most important point that we need to keep in mind is that the **value on the right-hand side must be of the same data type** as the variable on the left-hand side else we will get a compile-time error.
- **Consider an example:**
 - **X += y+1;** This is same as the statement **X = X+(y+1);**
 - Here, += is known as *shorthand assignment operator*.
- The **different Types of Assignment Operators** supported in the C# language are as follows:

2.2.4:Assignment Operators

Op	Operator Name	Meaning	Example
=	Simple Assignment	Assigns the value of the right-hand side operand to the left-hand side variable	int a = 10;
+=	Add Assignment	Adds right operand to left operand and assigns the result to the left variable	a += b; → same as a = a + b;
-=	Subtract Assignment	Subtracts right operand from left operand and assigns the result to the left variable	a -= b; → same as a = a - b;
*=	Multiply Assignment	Multiplies left operand with right operand and assigns the result to the left variable	a *= b; → same as a = a * b;
/=	Division Assignment	Divides left operand by right operand and assigns the result to the left variable	a /= b; → same as a = a / b;
%=	Modulus Assignment	Divides left operand by right operand and assigns the remainder to the left variable	a %= b; → same as a = a % b;

2.5. Unary Operators in C#:

- The Unary Operators in C# need only one operand. They are used to increment or decrement a value. There are two types of Unary Operators. They are as follows:
 - Increment operators (++): Example: (++x, x++)
 - Decrement operators (--): Example: (--x, x--)
- While ++m and m++ mean the **same thing** when they form statements **independently**.
- They **behave differently** when they are used in expressions on the **right-hand** side of an assignment statement. Consider the following:

m = 5;

y = ++m;

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

m = 5;

y = m++;

then, the value of y would be 5 and m would be 6.

- A **prefix operator** first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a **postfix operator** first assigns the value to the variable on left and then increments the operand. Following program illustrates this.

2.5. Unary Operators in C#:

Operator Type	Operator	Meaning / Function	Example
Post-Increment	a++	Increases the value of variable after its current value is used	<pre>int x = 5; int y = x++; print(x); // x=6 print(y); // y=5</pre>
Pre-Increment	++a	Increases the value of variable before its current value is used	<pre>int x = 5; int y = ++x; print(x); // x=6 print(y); // y=6</pre>
Post-Decrement	a--	Decreases the value of variable after its current value is used	<pre>int x = 5; int y = x--; print(x); // x=4 print(y); // y=5</pre>
Pre-Decrement	--a	Decreases the value of variable before its current value is used	<pre>int x = 5; int y = --x; print(x); // x=4 print(y); // y=4</pre>

2.5. INCREMENT OPERATOR ILLUSTRATED

```
using System;
namespace OperatorsDemo
{
    class UnaryOperators
    {
        static void Main(string[] args)
        {
            // Post-Increment
            int x = 10;
            // Result1 is assigned 10 only,
            // x is not updated yet
            int Result1 = x++;
            //x becomes 11 now
            Console.WriteLine("x is {0} and Result1 is {1}", x, Result1);
            // Pre-Increment
            int y = 10;
            int Result2 = ++y;
            //y and Result2 have same values = 11
            Console.WriteLine("y is {0} and Result2 is {1}", y, Result2);
        }
    }
}
```

2.6. Ternary Operator in C#:

- The Ternary Operator in C# is also known as the **Conditional Operator** (?:). It is actually the **shorthand of the if-else statement**.
- It is called **ternary because** it has **three operands** or arguments.
- The **first** argument is a **comparison** argument, the **second** is the **result of a true** comparison, and the **third is the result of a false** comparison.

- **Syntax:**

`Condition? first_expression : second_expression;`

- The above statement means that:
 - **first**, we need to **evaluate the condition**. |
 - If **the condition is true** the **first_expression** is executed and becomes the result and
 - if the **condition is false**, the **second_expression** is executed and becomes the result.
- For example, consider the following statements:

```
//ternary statement
```

```
a= 10; b = 15;
```

```
x = (a>b)? a : b;
```

```
//Equivalent if....else statement
```

```
if (a > b)
```

```
    x = a;
```

```
else
```

```
    x=b;
```

Example: Ternary Operator

```
using System;
namespace OperatorsDemo
{
    class TernaryOperator
    {
        static void Main(string[] args)
        {
            int a = 20, b = 10, res;
            res = ((a > b) ? a : b);
            Console.WriteLine("Result = " + res);
        }
    }
}
```

Type Casting in C#

- In simple words, we can say that Type Casting or Type Conversion is the **process to change one data type value into another data type**.
- The Type Conversion is **only possible** if both the data types are **compatible** with each other else we will get compile time error saying **cannot implicitly convert one type to another type**.
- **Example:**
if we declare the variable “a” as int, we cannot assign the string value Hello to it.

```
int a;  
a = "Hello"; // error CS0029: cannot implicitly convert type  
string to in
```
- So, the process of converting the value of one data type (**int, float, double, etc.**) to another data type (**(int, float, double, etc.)**) is known as type conversion or typecasting.
- There are two types of type casting :
 - **Implicit Type Casting**
 - **Explicit Type Casting**

Types of Type Casting

1. Implicit Conversion or Implicit Type Casting / Automatic Type Conversion in C#

- The Implicit Conversion or Implicit Type Casting in C# is **automatically done by the compiler** and in this case, there will be **no data loss**.
- Here, the typecasting or type conversion is done **from a smaller data type to a larger data type**. This type of type conversion is safe.
- **Implicit Type Casting happens when:**
 - The two data types are **compatible**.
 - When we assign a **value of a smaller** data type **to a bigger** data type.

Convert from Data Type

byte
short
int
long
float

Convert to Data Type

short, int, long, float, double
int, long, float, double
long, float, double
float, double
double

Types of Type Casting

2. Explicit Conversion or Explicit Type Casting in

- In **C#**, **explicit conversion (type casting)** means **manually converting one data type into another** using a **cast operator ()** or **conversion methods**.
- If we want to convert the **large data type** **to** a **small data type** in C#, then we need to do the same explicitly using the **cast operator**.
- In the case of Explicit Conversion or Explicit Type Casting. there is a **chance of data loss** (e.g., converting double → int) or the conversion might not be succeeded for some reason. This is **an unsafe type of conversion**.
- In C#, Explicit Type Casting can be achieved in following ways:
 1. Using Cast Operator (type)
 2. Using Convert Class Methods
 3. Using Parse() Method
 4. Using TryParse() Method
 5. Using ToString() Method

Using Cast Operator (type)

- It is simplest and most common form of explicit conversion.
- We can explicitly convert a value by placing the **target data type in parentheses before the value**.
- It performs a **low-level conversion** without checking for overflow.
- It is useful when we are sure the value can fit into the target type.

Example	Description
<code>int x = (int)3.75;</code>	Converts double → int (The fractional part .75 is truncated/ losted (not rounded))
<code>float f = (float)10.25;</code>	Converts double → float
<code>short s = (short)num;</code>	Converts int → short

Example: Explicit Type Conversion

```
using System;

class ExplicitTypeCasting
{
    static void Main(string[] args)
    {
        double numDouble = 1.23;
        // Explicit Type Casting
        int numInt = (int)numDouble;
        // Value Before Conversion
        Console.WriteLine("Original double Value: " + numDouble);
        // Value After Conversion
        Console.WriteLine("Converted int Value: " + numInt);
    }
}
```

```
Original double Value: 1.23
Converted int Value: 1
```

Using Cast Operator (type)

- Is it always we lose data when we convert a larger type to a smaller type in C#?
- NO. It basically depends on the value that we are converting and the size of the data type which is going to store the converted value. For a better understanding, please have a look at the below code.

```
int IntNum1 = 100;  
byte ByteNum1 = (byte)IntNum1; // Explicit Type Casting
```

- In the above case, **we will not lose any data**. This is because the integer variable holds the value 100 and in byte data type, we can store the values from 0 to 255, and 100 comes under this range and hence no data loss. Now, observe the following code

```
int IntNum2 = 500;  
byte ByteNum2 = (byte)IntNum2; // Explicit Type Casting
```

- In the above case, **we will lose the data**. This is because the integer variable holds the value 500 and in byte data type, we can store the values from 0 to 255, and 500 does not come under this range, and hence there is data loss.

Conversion with Helper Methods in C#:

- Observe the following example. Here, we have a **string variable** that holds the value **100** and we **try to convert** that value to an **integer type**.
- But this is **not possible with the cast operator**. Because cast operator will **first check the type compatibility** and it found **that string and int are not compatible** with each other because the string is used to store textual data which contains both alphanumeric and **int contains only numeric data**

```
using System;

class ConversionWithCastOperator
{
    static void Main(string[] args)
    {
        string str= "100";
        int i = (int)str;
        Console.WriteLine(" i = "+i);
    }
}
```

error CS0030: Cannot convert type 'string' to 'int'

- So, for the conversion between non-compatible types like **integer** and **string**, the .NET Framework provided us with the **Convert** class, **Parse** method, and **TryParse** method.

Using Convert Class Methods

- The Convert class (from the System namespace) provides **safe and flexible methods** to convert between most built-in data types.
- It handles **nulls** and **exceptions (type checking)** better than direct casting

Example	Description
<pre>double d = 15.67; int i = Convert.ToInt32(d); Console.WriteLine(i); // Output: 16 (rounds the value)</pre>	• Converts to integer • Unlike casting, Convert.ToInt32() rounds the decimal part instead of truncating.)
<pre>double d = Convert.ToDouble("12.34");</pre>	Converts string → double
<pre>string s = Convert.ToString(123);</pre>	Converts number → string
<pre>bool b = Convert.ToBoolean(1);</pre>	Converts integer → bool

Convert Class Helper Methods in C#:

- The Convert class provides the following methods to convert a value to a particular type.

Method Name	Description / Purpose	Example
ToByte()	Converts a value to a byte (0–255).	<code>Console.WriteLine(Convert.ToByte(123)); // 123</code>
ToChar()	Converts a number or string to a character.	<code>Console.WriteLine(Convert.ToChar(65)); // 'A'</code>
ToDateTime()	Converts a string or number to a DateTime object.	<code>Console.WriteLine(Convert.ToDateTime("2025-10-16"));</code>
ToDecimal()	Converts a value to a high-precision decimal type.	<code>Console.WriteLine(Convert.ToDecimal(12.56));</code>
ToSingle()	Converts a value to a single-precision floating number (float).	<code>Console.WriteLine(Convert.ToSingle("12.3"));</code>
ToString()	Converts any type to its string representation.	<code>int n = 20; Console.WriteLine(Convert.ToString(n)); // "20"</code>
ToUInt16()	Converts a value to an unsigned 16-bit integer.	<code>Console.WriteLine(Convert.ToUInt16(65000));</code>

Convert Class Helper Methods in C#:

ToDouble()	Converts a value to a double (floating-point number).	<code>Console.WriteLine(Convert.ToDouble("123.45"));</code>
ToInt16()	Converts a value to a 16-bit integer.	<code>Console.WriteLine(Convert.ToInt16(200));</code>
ToInt32()	Converts a value to a 32-bit integer (int).	<code>Console.WriteLine(Convert.ToInt32("123")); // 123</code>
ToInt64()	Converts a value to a 64-bit integer (long).	<code>Console.WriteLine(Convert.ToInt64(99999999999));</code>

- For example, if we want to convert a **string to an Int type**, then we need to use either **`Convert.ToInt16`**, or **`Convert.ToInt32`**, or **`Convert.ToInt64`**.
- These helper methods are implemented **as static methods** inside the **`Convert` class** and hence we can access them directly.
- For a better understanding, look at the following example

Example:

```
using System;
namespace TypeCastingDemo
{
    class ConvertClassMethods
    {
        static void Main(string[] args)
        {
            string str = "100";
            int i1 = Convert.ToInt32(str); //Converting string to Integer

            double d = 123.45;
            int i2 = Convert.ToInt32(d); //Converting double to Integer

            float f = 45.678F;
            string str2 = Convert.ToString(f); //Converting float to string

            Console.WriteLine("Original value str: "+str+" and Converted Value i1: "+i1);
            Console.WriteLine("Original value d: "+d+" and Converted Value i2:"+i2);
            Console.WriteLine("Original value f: "+f+" and Converted Value str2: "+str2);
        }
    }
}
```

```
Original value str: 100 and Converted Value i1: 100
Original value d: 123.45 and Converted Value i2:123
Original value f: 45.678 and Converted Value str2: 45.678
```

Example 2 : Calculate Area of Circle

```
using System;

class AreaCalculations1{
    public static void FullCircle(){
        double r;
        double pi=3.14F; //implicit type casting

        Console.WriteLine("Enter the radius of circle :");
        //Return type of ReadLine() method is string
        string strRadius=Console.ReadLine();

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

        double result=pi*r*r;
        Console.WriteLine("Area of circle="+result);
    }
}

class MainClass{
    public static void Main(string[] args){
        AreaCalculations1.FullCircle();
    }
}
```

```
Enter the radius of circle :
10
Area of circle=314.000010490417
```

Assignment 1

- Write a program to compute and display the average of the number 25,75 and 100.
- Write a program to calculate the **area** and **perimeter** of circle.
- Write a program to calculate the **simple interest** of given amount, rate and period by user.

Type Conversion using Parse() Method in C#

- In C#, we can also use the **built-in Parse()** method to perform type conversion. So, while performing type conversion between **non-compatible types** like **int** and **string**, we **can also use Parse() method** like the Convert class helper methods.
- Now, if we go to the definition of built-in value data types such as **int**, **short**, **long**, **bool**, etc., then we **will see that the Parse method** is implemented as a **static method** in those built-in value data types.
- So, using the built-in type, we can call the Parse method
- Used to convert **string** → **numeric** type.

Example	Description
<code>int num = int.Parse("123");</code>	Converts string to int
<code>double d = double.Parse("12.45");</code>	Converts string to double

- **Throws exception** if the string is invalid or null.

Example : Type casting with Parse() method

```
using System;
namespace TypeCastingDemo
{
    class CastingWithParseMethod
    {
        static void Main(string[] args)
        {
            string str1 = "100";
            //Converting string to int type
            int i = int.Parse(str1);
            Console.WriteLine("Original String value: "+str1+" and Converted int value: "+i);

            string str2 = "TRUE";
            //Converting string to boolean type
            bool b= bool.Parse(str2);
            Console.WriteLine("Original String value: "+str2+" and Converted bool value: "+b);
        }
    }
}
```

Original String value: 100 and Converted int value: 100

Original String value: TRUE and Converted bool value: True