**C# Assignment**

1. **Do we define global variable in camel Case or in Pascal Case?**

**Ans:**

While it's technically possible to define global variables using camelCase (the first letter in lowercase with subsequent words capitalized), it is not a common or recommended practice in C#. Using camelCase for global variables could lead to confusion and make it harder to distinguish them from local variables or fields.

In summary, it's best practice to define global variables in Pascal Case to maintain code clarity and consistency with C# naming conventions.

In practice, most C# developers tend to use Pascal Case for global variables because it aligns with the convention for naming classes and properties, making it easier to distinguish global variables from local variables and fields.

**for constant variables**

In C#, constant variables are typically defined using Pascal Case, not camelCase. The convention for naming constant variables follows the same Pascal Case convention as other class-level members like fields, properties, and methods. Using Pascal Case helps distinguish constant variables from other variables and makes your code more readable and consistent with C# naming conventions.

1. **Difference between Variables and Properties?**

**Ans:**

Variables and properties both represent values that you can access. However, there are differences in storage and implementation.

A variable corresponds directly to a memory location. A variable is a name given to a memory location

Properties are the special type of class members that provides a flexible mechanism to read, write, or compute the value of a private field. Properties provide a way to get and set the values of private fields while encapsulating the logic that may be associated with those actions.

Properties are used to provide a controlled interface to the internal state of objects. They enable you to encapsulate data, enforce validation, and add logic when getting or setting values, ensuring that the object's state is consistent and follows certain rules. Properties are used to improve code maintainability and security.

|  |  |  |
| --- | --- | --- |
| **Point of difference** | **Variable** | **Property** |
| Declaration | Single declaration statement | Series of statements in a code block |
| Implementation | Single storage location | Executable code (property procedures) |
| Storage | Directly associated with variable's value | Typically has internal storage not available outside the property's containing class or module  Property's value might or might not exist as a stored element 1 |
| Executable code | None | Must have at least one procedure |
| Read and write access | Read/write or read-only | Read/write, read-only, or write-only |
| Custom actions (in addition to accepting or returning value) | Not possible | Can be performed as part of setting or retrieving property value |

1. **Difference between Functions and Subroutine?**

**Ans:**

A function allows you to encapsulate a piece of code and call it from other parts of your code. You may very soon run into a situation where you need to repeat a piece of code, from multiple places, and this is where functions come in.

Functions and subroutines are used to group and organize code for reusability and modularity, but they have some key differences in terms of their behavior and how they are used:

* **Returning value**

The biggest difference between subs and functions is that function can return value while a sub cannot. A return value is a variable that the procedure sends back to the user. A function should not change the values of actual arguments whereas a subroutine could change them.

* In C#, subroutines are commonly referred to as "methods" or "void methods." Methods in C# can be defined without a return value (void), and they are used to group a set of statements together to perform a specific action or task. In some programming languages, the syntax for defining and calling functions and subroutines may be slightly different. Functions typically specify a return type, while subroutines may use a `void` keyword to indicate no return value.
* **Uses**

Functions are particularly useful at repetitive tasks or tasks that require a return value. You can use functions to make calculations, determine file sizes or classify selected objects. Subroutines are used when you want to group a set of statements together to perform an action or task without the need for returning a value.

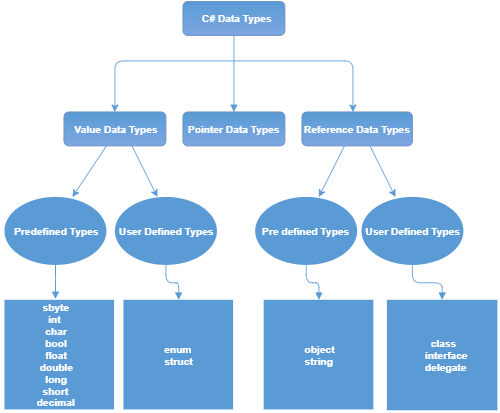
It's important to note that while subroutines (void methods) do not return values, they are essential for encapsulating functionality, improving code organization, and facilitating code reusability. They are commonly used in C# and other object-oriented programming languages to perform actions and modify object states

1. **DATATYPES IN C#**

**Ans:** The data types in C# are divided into three types. These are:

1. **Value Data Types -**These are integer and floating point based. Some examples of value data types are int, char, float etc.
2. **Reference Data Types -**These data types contain a reference to the variables and not the actual data. Some build in reference types are object, dynamic and string.
3. **Pointer Data Types -**This store the memory address of another data type and the data can be accessed using pointers.

The following are the details about all the data types in C#:



**1. Integral Types:**

* `sbyte`: Signed 8-bit integer.
* `byte`: Unsigned 8-bit integer.
* `short`: Signed 16-bit integer.
* `ushort`: Unsigned 16-bit integer.
* `int`: Signed 32-bit integer.
* `uint`: Unsigned 32-bit integer.
* `long`: Signed 64-bit integer.
* `ulong`: Unsigned 64-bit integer.

| **Data Type** | **Size** | **Range** |
| --- | --- | --- |
| byte | 8 bits | 0 to 255 |
| sbyte | 8 bits | -128 to 127 |
| int | 32 bits | -2,147,483,648 to 2,147,483,647 |
| uint | 32 bits | 0 to 4294967295 |
| short | 16 bits | -32,768 to 32,767 |
| ushort | 16 bits | 0 to 65,535 |
| long | 64 bits | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | 64 bits | 0 to 18,446,744,073,709,551,615 |
| float | 32 bits | -3.402823e38 to 3.402823e38 |
| double | 64 bits | -1.79769313486232e308 to 1.79769313486232e308 |
| bool | 8 bits | True or False |
| decimal | 128 bits | (+ or -)1.0 x 10e-28 to 7.9 x 10e28 |
| DateTime | - | 0:00:00am 1/1/01 to 11:59:59pm 12/31/9999 |

**2. Floating-Point Types** (float): Single-precision floating-point number.

- `double`: Double-precision floating-point number.

**3. Decimal Type** (decimal): A high-precision decimal type for financial and monetary calculations.

**4. Character Types (**char): A single Unicode character.

**5. Boolean Type** (bool): Represents a Boolean value (true or false).

**C# Reference Data Types**

In c#, the **Reference Data Types** will contain a memory address of variable value because the reference types won’t store the variable value directly in memory.

The following table lists the reference data types in c# programming language with memory size and range of values.

**Pointer Types (Unsafe Context Only):** - C# allows the use of pointers in unsafe code blocks to work with memory directly.

1. **STATEMENTS IN C#**

**Ans: -** A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block.

The following table lists the various types of statements in C# and their associated keywords, with links to topics that include more information:

| **Category** | **C# keywords / notes** |
| --- | --- |
| [Declaration statements](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#declaration-statements) | A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required. |
| [Expression statements](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#expression-statements) | Expression statements that calculate a value must store the value in a variable. |
| Selection statements | Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics:   * [if](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements#the-if-statement) * [switch](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements#the-switch-statement) |
| Iteration statements | Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics:   * [do](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-do-statement) * [for](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-for-statement) * [foreach](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement) * [while](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-while-statement) |
| Jump statements | Jump statements transfer control to another section of code. For more information, see the following topics:   * [break](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-break-statement) * [continue](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-continue-statement) * [goto](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-goto-statement) * [return](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements#the-return-statement) * [yield](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield) |
| Exception-handling statements | Exception-handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics:   * [throw](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-throw-statement) * [try-catch](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-catch-statement) * [try-finally](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-finally-statement) * [try-catch-finally](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements#the-try-catch-finally-statement) |
| [checked and unchecked](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked) | The checked and unchecked statements enable you to specify whether integral-type numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. |
| The await statement | If you mark a method with the [async](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/async) modifier, you can use the [await](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await) operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.  For a simple example, see the "Async Methods" section of [Methods](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods). For more information, see [Asynchronous Programming with async and await](https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/). |
| The yield return statement | An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the [yield return](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield) statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.  For more information, see [Iterators](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators). |
| The fixed statement | The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see [fixed](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/fixed). |
| The lock statement | The lock statement enables you to limit access to blocks of code to only one thread at a time. For more information, see [lock](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock). |
| Labeled statements | You can give a statement a label and then use the [goto](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements" \l "the-goto-statement) keyword to jump to the labeled statement. (See the example in the following row.) |
| The [empty statement](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements#the-empty-statement) | The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed. |

1. **Conditional in c#?**

**Ans**: C# has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

**The if statement** is used as a single statement where the codes in the curly {} brackets execute if, and only if, the conditions in the round () brackets are true.

**The if-else statement** allows for two conditions to be checked. If the conditions in the first round () brackets are true, the codes in the first curly {} brackets will execute. Otherwise, the codes in the second curly {} brackets will execute.

**The nested if-else statement.**

This **nested if-else statement** allows for a series of conditions to be checked continuously, from top to bottom, until a condition is met. Then, the code in that block is executed.

**The switch statement**

A **switch statement** in C# checks the value of a variable against multiple cases. If it matches any case, the statements in that case will execute and the break keyword will stop the switch.

Each case in a block of switch cases holds an option called an ‘identifier’. When a value is entered, it is compared with the cases in the switch block until an exact match is found.

If a match is not found, the default statement is executed.

1. **-12 and +12 time zone exist? If yes than it will meet or not?**

**Ans: -** Yes, there are time zones that are 12 hours ahead and 12 hours behind Coordinated Universal Time (UTC). The time zone that is 12 hours ahead of UTC is often referred to as UTC+12, while the time zone that is 12 hours behind UTC is often referred to as UTC-12.

If you were to compare a location in UTC+12 with a location in UTC-12, there would be a 24-hour time difference between them. In other words, when it is noon (12:00 PM) in a UTC+12 time zone, it would be midnight (12:00 AM) in a UTC-12 time zone.

Numerous locations observe UTC+12, including a few Pacific Ocean islands including Fiji, Wallis and Futuna, and Tuvalu.UTC-12 is not a commonly used time zone and is not observed in many places, but it does exist.

1. **Program using bitwise operators, datatypes, conversion, statements, conditional statement?**

**Ans: -**

1. **Loops in c#?**

**Ans:** Looping in a programming language is a way to execute a statement or a set of statements multiple times depending on the result of the condition to be evaluated to execute statements. The result condition should be true to execute statements within loops.

**1 while loop**

It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.

**2 for loop**

It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

**3 do...while loop**

It is similar to a while statement, except that it tests the condition at the end of the loop body

**4 nested loops**

You can use one or more loop inside any another while, for or do..while loop.

1. **Program using bitwise operators, datatypes, conversion, statements, conditional statement?**

using System;

class Program

{

static void Main(string[] args)

{

Console.Write("Enter the first integer: ");

int a = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the second integer: ");

int b = Convert.ToInt32(Console.ReadLine());

int result = 0;

//for a=10 b=3

result = a & b; // for a=10 && b=3 , 1010 & 0011 = 0010 = 3

Console.WriteLine("a & b :" + result);

result = a | b; //for a=10 && b=3 ,1010 | 0011 = 1011 = 11

Console.WriteLine("a | b : "+ result);

result = a ^ b; //for a=10 && b=3 ,1010 ^ 0011 = 1001

Console.WriteLine("a ^ b : "+ result);

result = ~a; // for a=10 , ones compliment of 10

Console.WriteLine("~a : "+ result);

result = a << 2; //1010<<2 = 101000 = 40

Console.WriteLine("a << b : "+result);

result = a >> 2; //1010>>2 = 0010 = 2

Console.WriteLine("a >> b : " + result);

// using Xor Bitwise operator

Console.Write(" swap numbers using xor ");

int num1, num2;

Console.Write("Enter the first integer: ");

num1 = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter the second integer: ");

num2 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine($"Before swapping: num1 = {num1}, num2 = {num2}");

num1 = num1 ^ num2;

num2 = num1 ^ num2;

num1 = num1 ^ num2;

Console.WriteLine($"After swapping: num1 = {num1}, num2 = {num2}");

// using conditional statements

int i;

Console.Write("Enter a Number : ");

i = int.Parse(Console.ReadLine());

if (i % 2 == 0)

{

Console.Write("Entered Number is an Even Number");

}

else

{

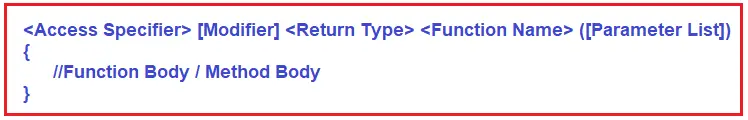
Console.Write("Entered Number is an Odd Number");

}

}

}

1. **FUNCTIONS:** A function is a group of related instructions that performs a specific task.

**** A function allows the programmers to enclose a piece of code and then call that part of the code from another part of the program. It is quite useful when you need to run the same code from different places.

Here,

1. **Function Name:**It is mandatory, and it defines the name of the method or function. The method signature consists of the method name and parameter list. The Methods are identified by their name. The rules for giving function names are the same as the rules for giving variable names. Same rules you should follow for giving function names also.
2. **Parameter List:**It is Optional and defines the list of parameters. A function that can take 0 or more parameters may not take any input.
3. **Return Type:**It is Mandatory and defines the method’s return type value. A function may or may not return a value, but it can return at most one value. It cannot return multiple values but can take multiple values as parameters. If the function is not returning any value, then the return type should be void.
4. **Access Specifier:** It is Optional and defines the method’s scope. That means it defines the accessibility of the method, such as private, protected, public, etc.
5. **Modifier:**It is optional and defines the method’s access type. For example, static, virtual, partial, sealed, etc. If you declare the method with a static modifier, then you can access the method directly without creating an instance. If you declare the method with the sealed modifier, then this method is not going to be overridden under a child class. And if you declare the method with the partial modifier, then you can split the method definition into two parts.
6. **Function Body:**The function’s body defines the code or list of statements you need to execute the function call. It is enclosed within curly braces.
7. **POINTERS:**

Pointers in C# are a powerful feature of the language, but they are not as commonly used or as explicit as in languages like C or C++. In C#, pointers are primarily used within "unsafe" code blocks, which allow you to work with memory addresses directly.

1. **REF AND OUT KEYWORD.**

**Ref**

The ref keyword is used to pass an argument as a reference. This means that when value of that parameter is changed in the method, it gets reflected in the calling method. An argument that is passed using a ref keyword must be initialized in the calling method before it is passed to the called method.

* Allows you to pass a variable by reference, which means that the variable can be modified inside of the method.
* Is a good choice when you need to pass a variable into a method and you want to be able to modify the variable inside of the method.

**Out**

The out keyword is also used to pass an argument like ref keyword, but the argument can be passed without assigning any value to it. An argument that is passed using an out keyword must be initialized in the called method before it returns back to calling method.

* Allows you to pass a variable by reference, but the variable does not have to be initialized before it is passed into the method.
* Can only be used on variables that are declared inside of the method.
* Is a good choice when you need to pass a variable into a method and you want to initialize the variable inside of the method.

1. **HOW MUCH ARGUMENTS WE CAN PASS INTO MAIN METHOD ?**

C# applications have an entry point called Main Method. It is the first method which gets invoked whenever an application started and it is present in every C# executable file. The application may be Console Application or Windows Application. The most common entry point of a C# program is static void Main() or static void Main(String []args).

The args parameter is an array of strings, representing the command-line arguments passed to the program when it is executed.

You can technically pass as many command-line arguments as you want. The args array will contain all the arguments passed to the program, and you can access them using array indexing.