Question Answering Bonus

1. What meant by user defined constructor and its role in initialization

What is a User-Defined Constructor in C#?

A **user-defined constructor** in C# is a special method in a class or struct, explicitly defined by the programmer, that is called automatically when an object is instantiated using the new keyword. It has the same name as the class, no return type (not even void), and is used to initialize an object's fields, properties, or other members to specific values or states. Unlike default constructors which constructors automatically provided by C# if no constructors are defined), user-defined constructors allow customized initialization logic, ensuring objects are created in a valid and meaningful state tailored to the application's needs.

Role in Initialization

The primary role of a user-defined constructor is to **initialize an object's state** by:

1. **Setting Initial Values**: Assigning specific values to fields or properties based on parameters or logic.
2. **Ensuring Valid State**: Enforcing constraints, such as mandatory fields or valid ranges, to prevent objects from being created in an invalid state.
3. **Performing Setup Logic**: Executing additional initialization tasks, like allocating resources, setting defaults, or invoking other methods.
4. **Supporting Encapsulation**: Allowing controlled initialization of private fields, ensuring data integrity.

5. **Enabling Flexibility**: Providing multiple constructors (constructor overloading) to support different ways of creating objects.

Example:

```csharp
using System;

public class Person
{
   // Fields
   private readonly string _name; // Readonly to enforce immutability after
initialization
   private int _age;
   private readonly DateTime _createdAt;

   // User-defined constructor with parameters
   public Person(string name, int age)
   {
       // Validate inputs to ensure valid state
       if (string.IsNullOrWhiteSpace(name))
           throw new ArgumentException("Name cannot be empty or null.");

       if (age < 0)
           throw new ArgumentException("Age cannot be negative.");

       _name = name;
       _age = age;
       _createdAt = DateTime.Now; // Additional setup logic
   }

   // Overloaded constructor for minimal initialization
   public Person(string name) : this(name, 0) // Chaining to the main
constructor
   {
       // Additional logic can be added here if needed
   }

   // Method to display person details
   public void DisplayInfo()
   {
       Console.WriteLine($"Name: {_name}, Age: {_age}, Created:
{_createdAt}");
```

```
    }
}

class Program
{
    static void Main()
    {
        // Create objects using user-defined constructors
        Person person1 = new Person("Alice", 30);
        person1.DisplayInfo(); // Output: Name: Alice, Age: 30, Created: 
[current timestamp]

        Person person2 = new Person("Bob"); // Uses overloaded constructor
        person2.DisplayInfo(); // Output: Name: Bob, Age: 0, Created: 
[current timestamp]

        // Invalid initialization will throw an exception
        try
        {
            Person person3 = new Person("", -5); // Throws ArgumentException
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

2. Compare between Array and Linked List?

Array and **Linked List** are fundamental data structures in C# with different strengths, suited for specific scenarios based on performance, memory, and access patterns.

Array (T[ ]): An array is a fixed-size, contiguous block of elements in memory. It offers fast, O(1) access to elements via indices (e.g., arr[5]), making it ideal for random access. However, its size is fixed, so insertions or deletions are slow (O(n)) due to element shifting, and resizing requires creating a new array. Arrays are memory-efficient for small, static datasets but inflexible for dynamic sizes. Use them in user-facing applications for fixed lists, like menu items or game scores, where quick access by index is needed.

Linked List (LinkedList<t>)</t>: A linked list is a dynamic collection of nodes, each holding data and a reference to the next node. It excels at fast, O(1) insertions and deletions at known positions (e.g., head or tail), but element access is slower (O(n)) due to node traversal. It uses more memory due to node pointers but grows or shrinks easily. Use linked lists for dynamic data with frequent insertions/deletions, like a playlist in a music app or an undo feature in a text editor.

Differences:

- **Access Speed**: Arrays are faster for accessing elements; linked lists are slower due to traversal.
- **Modification Speed**: Linked lists are faster for insertions/deletions; arrays are slower due to shifting.
- **Size Flexibility**: Arrays have fixed sizes; linked lists are dynamic.
- **Memory**: Arrays are more memory-efficient for fixed data; linked lists have higher overhead.
- **Use Cases**: Arrays suit static, index-based access; linked lists suit dynamic, frequent modifications.

Example

```
using System;
using System.Collections.Generic;

class Program
{
   static void Main()
   {
       // Array example
       string[] namesArray = new string[3]; // Fixed size
       namesArray[0] = "Alice";
       namesArray[1] = "Bob";
       namesArray[2] = "Charlie";

       Console.WriteLine("Array Contents:");
       foreach (var name in namesArray)
       {
           Console.WriteLine(name); // Fast access: O(1)
       }

       // Adding a new name requires a new array (not shown, as size is
fixed)

       // LinkedList example
       LinkedList<string> namesList = new LinkedList<string>();
       namesList.AddLast("Alice"); // O(1) insertion
       namesList.AddLast("Bob");
       namesList.AddFirst("Charlie"); // O(1) insertion at head

       Console.WriteLine("\nLinkedList Contents:");
       foreach (var name in namesList)
       {
           Console.WriteLine(name); // Requires traversal: O(n)
       }

       // Remove Bob efficiently
       var bobNode = namesList.Find("Bob");
       if (bobNode != null)
       {
           namesList.Remove(bobNode); // O(1) deletion
       }
```

```
            Console.WriteLine("\nLinkedList After Removing Bob:");
            foreach (var name in namesList)
            {
                Console.WriteLine(name);
            }
    }
}
```