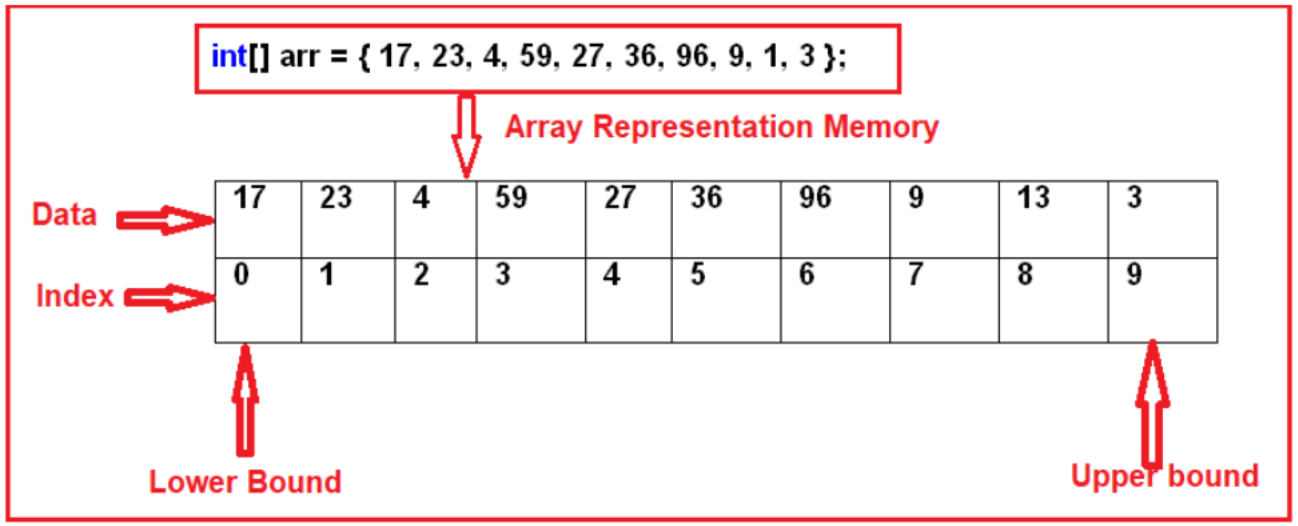
Collections in C#

What is an Array in C#?

In simple words, we can define an array as a collection of similar types of values that are stored in sequential order i.e. they are stored in a contiguous memory location.

Memory Representation of Arrays in C#:



1.Sort(<array>): Sorting the array elements
2.Reverse (<array>): Reversing the array elements
3.Copy (src, dest, n): Copying some of the elements or all elements from the old array to the new array
4.GetLength(int): A 32-bit integer that represents the number of elements in the specified dimension.
5.Length: It Returns the total number of elements in all the dimensions of the Array; zero if there are no elements in the array.

Disadvantages of using Arrays in C#:

1.The array size is fixed. So, we should know in advance how many elements are going to be stored in the array. Once the array is created, then we can never increase the size of an array. If you want then we can do it manually by creating a new array and copying the old array elements into the new array.
2.As the array size is fixed, if we allocate more memory than the requirement then the extra memory will be wasted. On the other hand, if we allocate less memory than the requirement, then it will create the problem.
3.We can never insert an element into the middle of an array. It is also not possible to delete or remove elements from the middle of an array.

What is a Collection in C#?

The **Collections in C#** are a set of predefined classes that are present in the **System.Collections** namespace that provides greater capabilities than the traditional arrays. The collections in C# are reusable, more powerful, more efficient and most importantly they have been designed and tested to ensure quality and performance.

1.Size can be increased dynamically.
2.We can insert an element into the middle of a collection.
3.It also provides the facility to remove or delete elements from the middle of a collection.

Auto-Resizing of collections:

The capacity of a collection increases dynamically i.e. when we keep adding new elements, then the size of the collection keeps increasing automatically. Every collection class has three constructors and the behavior of collections will be as following when created using a different constructor.

1.Default Constructor: It Initializes a new instance of the collection class that is empty and has the default initial capacity as zero which becomes four after adding the first element and whenever needed the current capacity becomes double.

2.Collection (int capacity): This constructor initializes a new instance of the collection class that is empty and has the specified initial capacity, here also when the requirement comes current capacity doubles.

3.Collection (Collection): It Initializes a new instance of the collection class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied, here also when the requirement comes current capacity doubles.

What is ArrayList in C#?

The **ArrayList in C#** is a collection class that works like an array but provides the facilities such as dynamic resizing, adding, and deleting elements from the middle of a collection. It implements the System.Collections.IList interface using an array whose size is dynamically increased as required.

Methods and Properties of ArrayList Collection class in C#:

The following are the methods and properties provided by the ArrayList collection class in C#.

1.Add(object value): This method is used to add an object to the end of the collection.

2.Remove(object obj): This method is used to remove the first occurrence of a specific object from the collection.

3.RemoveAt(int index): This method takes the index position of the elements and removes that element from the collection.

4.Insert(int index, Object value): This method is used to inserts an element into the collection at the specified index.

5.Capacity: This property gives you the capacity of the collection means how many elements you can insert into the collection.

```
ArrayList al = new ArrayList();
Console.WriteLine("Initial Capacity: " + al.Capacity);
al.Add(10);

// Inserting an element on index
al.Insert(2, false);

// Delete an element on index
al.Remove(true);
```

Generic Collections in C#:

The **Generic Collections in C#** are strongly typed. The strongly typed nature allows these collection classes to store only one type of value into it. This not only eliminates the type mismatch at runtime but also we will get better performance as they don't require boxing and unboxing while they store value type data.

What is Generic List in C#?

The **Generic List in C#** is a collection class that is present in **System.Collections.Generic** namespace. The List Collection class is one of the most widely used generic collection classes in real-time applications. This Generic List collection class represents a strongly typed list of objects which can be accessed by using the index. It also provides methods that can be used for search, sort and manipulate the list items.

We can create a collection of any type by using the generic list class in C#. For example, if we want then we can create a list of strings, a list of integers, and even though it is also possible to create a list of the user-defined complex type such as a list of customers, a list of products, etc. The most important point that we need to keep in mind is the size of the collection grows automatically when we add items into the collection

Methods and Properties of Generic List Collection class in C#:

The following are some of the useful methods and properties of the List collection class in C#.

1.Add(T value): This method is used to add an item to the end of the list collection.

2.Remove(T value): This method is used to remove the first occurrence of a specific item from the collection.

3.RemoveAt(int index): This method takes the index position of the elements and then removes that element from the collection.

4.Insert(int index, T value): This method is used to inserts an element into the collection at a specified index position.

5.Capacity: This property is used to return the capacity of the collection means how many elements you can insert into the collection.

```
List<Customer> listCustomer = new List<Customer> ();
listCustomer.Add(new Customer {id=101,name=xyz});
listCustomer.Remove(listCustomer);
listCustomer. RemoveAt(1);
listCustomer. Insert(1,new customer{id=101,name=abc});
Console.WriteLine("Initial Capacity: " + listCustome.Capacity);
```

The Generic List class provides the following Range methods.

AddRange() Method:

As we already discussed the Add() method of the List class allows us to add only one item at the end of the collection. If you want to add another list of items to the list collection then you need to the AddRange() method.

Syntax: AddRange(IEnumerable<T>)

GetRange() Method:

In our previous article, we discussed that by using the index, we can retrieve only one element from the collection. In many real-time scenarios, we may need to retrieve a list of items from a collection. Then in such scenarios, we need to use the GetRange() method of the List class. The GetRange() method takes 2 parameters. The first parameter is the starting index position and the second parameter is the number of items to return from the list.

Syntax: GetRange(Int32, Int32)

InsertRange() Method:

The Insert() method of the Generic List collection class allows us to insert an element at a specified index position. If you want to insert another list of elements at a specified index, then you need to use the InsertRange() method of the List class. This method takes two parameters. The first parameter is the index position where it will insert the elements and the second parameter is the list of items that you want to insert into the collection.

Syntax: InsertRange(Int32, IEnumerable<T>)

RemoveRange() Method:

This **RemoveAt** takes the index position of the elements and then removes that element from the collection. If you want to remove a range of elements from a specified index position then you need to use the RemoveRange() method. This RemoveRange() method takes 2 parameters. The first parameter is the start index in the list and the second parameter is the number of elements to remove from the list.

Syntax: RemoveRange(Int32, Int32)

Note: The Remove method is used to remove only the first matching element from the list.

How to Sort a List of Simple Types in C#?

In C#, sorting a list of simple types like int, double, char, string, etc. is straightforward. Here, we just need to call the Sort() method which is provided by the Generic List class on the list instance, and then the data will be automatically sorted in ascending order.

```
List<int> numbersList = new List<int>{ 1, 8, 7, 5, 2, 3, 4, 9, 6 };
numbersList.Sort();
numbersList.Reverse();
```

```
List<int> numbersList = new List<int> { 1, 8, 7, 5, 2,
3, 4, 9, 6 };
Console.WriteLine("Numbers before sorting");
foreach (int i in numbersList)
{
Console.WriteLine(i);
}
// The Sort() of List Collection class
// will sort the data in ascending order
numbersList.Sort();
Console.WriteLine("Numbers after sorting");
foreach (int i in numbersList)
{
Console.WriteLine(i);
}
// If you want to to retrieve data in descending order
then use the
//Reverse() method of the List collection class
numbersList.Reverse();
```

Implementing the **IComparable interface in C#**

Let see an example for better understanding. What we want is, we need to sort the employees based on the Salary. To do so, our Employee class should implement the **IComparable** interface and should provide an implementation for the **CompareTo()** method. This method will compare the current object (specified with this) and the object to be compared. The following code exactly does the same.

```
public class Employee : IComparable<Employee>
{
public int ID { get; set; }
public string Name { get; set; }
public string Gender { get; set; }
public int Salary { get; set; }
public int CompareTo(Employee obj)
{
if (this.Salary > obj.Salary)
{
return 1;
}
else if (this.Salary < obj.Salary)
{
return -1;
}
else
{
return 0;
}}}
```