**What is Data Structure (and why)**

It is a way of organizing all data items and their relationships to each other inside the program in order to deal with them. It affects the design of both structural and functional aspects of the program. It is how you organize, manage and store data for efficiency reasons.

It is not only used for organizing data. It also used for processing, retrieving and storing data.

There are different basic and advanced types of data structures that are sued in almost every program or software system that has been developed. So we must have a good knowledge of data structures.

They are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing processing, accessing and storing data efficiently. But this is not all. Various types of data structures have their own characteristics, features, applications, advantages and disadvantages.

**Calcification of Data Structures**

**Primitive (Basic):**

Is generally a basic structure that is usually built into the language and directly operated upon the machine instructions, such as Integer, Float, Char, Pointer, etc...

**NON-Primitive (Advanced):**

**Linear, Non-Linear:**

* Complex/Sophisticated data structures derived for primitive DS.
* Emphasize on structuring of group of homogenous (same type) or heterogenous (different type) data items.
* The design of an efficient data structure must take operations to be performed on data structure.

**Linear vs Non-Linear Data Structures:**

* **Linear:**
  + Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.
* **Non-Linear:**
  + Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.
  + Example of this data structure are Tree, Graph, etc.

**Static vs Non-Static:**

* **Static:**
  + It has a fixed memory size. It is easier to access the elements in a static data structure.
  + Example of this one is an array.
* **Dynamic:**
  + In this one the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  + Examples are Stack, Queue, etc.

**Operation on Data Structures:**

Create, Update, Search, Select, Sorting, Merging, Destroy or Delete.

**Boxing**

It is the process of converting a value type to a reference type. This involves wrapping a value type like (int, float, char) in an object or any interface type implemented by this value type.

**Example:**

using System;

class Program

{

    static void Main()

    {

        int valType = 10;

        object objType = valType; // Boxing

        Console.WriteLine("Value Type: " + valType);

        Console.WriteLine("Object Type: " + objType);

    }

}

**Expected Output:**

Value Type: 10

Object Type: 10

**Output Explanation:**

The output demonstrates the boxing process where valType, an integer (value type), is boxed into objType (object type). Both display the same value, but objType is a reference type stored in the heap.

**Conclusion:**

Boxing is a fundamental concept in C#, allowing value types to be treated as objects. While it necessary in certain scenarios, developers should be aware of its performance.

**Unboxing**

It is the reverse process of Boxing, where the value type is extracted from the object. It’s crucial to ensure the type being Unboxed matches the type of the object.

**Example:**

using System;

class Program

{

    static void Main()

    {

        int valType = 10;

        object objType = valType; // Boxing

        int unboxedValType = (int)objType; // Unboxing

        Console.WriteLine("Unboxed Value: " + unboxedValType);

    }

}

**Output:**

Unboxed Value: 10

Explanation:

The program demonstrates Unboxing, where the value 10 is retrieved from objType (the boxed object) and stored back in UnboxedValType, a value type.

**Key Point**

Unboxing requires the exact data type match, otherwise, it results in InvalidCastException.

**Conclusion:**

Unboxing is a critical operation in C# that retrieves values from object. Proper type matching is essential for successful unboxing.

**Introduction to Collections**

**What are Collections:**

* Collections are data structures used to store and organize groups of related objects in memory.
* They are sophisticated ways to store and manage data in C#. they offer more flexibility and functionality compared to basic array types.
* They allow for dynamic memory allocation, meaning the size of the collection can grow or shrink as needed.

**Why Use Collections:**

* They are used to store, retrieve, manipulate and communicate aggregate data.
* They provide efficient ways to handle large amount of data with built-int methods for common tasks.
* Collection provide efficient ways to manipulate and manage data, making programming tasks easier and more efficient.

**Common operations on collections:** Adding, removing, modifying, and accessing elements.

**Exploring Types of Collections:**

* The System.Collections and System.Collection.Generic name spaces.
  + These namespaces include various collection types. System.Collections contains non-generic collections.

**Common Collections types in C#:**

1. **List<T>:** A List<T> is a collection of objects that can be accessed by index. It functions like a dynamic array, which can automatically resize as needed. It’s versatile and suitable for sorting and manipulating a list of objects of a specific types.
2. **SortedList<TKey, TValue>:** It is a collection that maintain its elements in sorted order. It’s a combination of an array and hash table, providing fast lookups as well as maintaining a sorted order.
3. **Dictionary<TKey, TValue>:** This collection stores key-value pairs. It enables fast retrieval of values based on keys, making it ideal for situations where you need to access elements quickly and uniquely, like lookup table.
4. **HashSet<T>:** It stores a set of unique elements. It’s useful for operations that requires uniqueness for each element and is efficient in performing set operations like union or intersection.
5. **Stack<T>:** It represents a Last-In-First-Out (LIFO) structure. It’s perfect for scenarios that require reverse order processing, such as undo mechanism in applications.
6. **Queue<T>:** Representing First-In-First-Out (FIFO) structure, it is great for tasks where you need to process items in the order they where added, like task scheduling.
7. **linkedList<T>:** This is a doubly linked list, where each element points to both its previous and next element. It allows for efficient insertions and deletions at any point in the list.
8. **ObservableCollection<T>:** this collection is used primarily in data binding, typically in UI context. It notifies listeners of dynamic changes, like when items get added, removed or the whole list is refreshed.
9. **CurrentDictionary<TKey, TValue>:** A thread safe version of dictionary, this one is designed for concurrent access. It’s useful in multi-threading applications where different threads need to add or remove items simultaneously.
10. **BitArray:** this one manages compact array of bit values, which are represented as Booleans. It’s used in scenarios where you need to store bits but don’t need the overhead of a Boolean array.

Each of this collection types in C# serves specific purpose and choosing the right type depends on the requirements of the application or the specific problem you’re solving.

**Conclusion:**

In this lesson, we explored the fundamental of collections in C#, including various collection types and their characteristics. By understanding collection, you will be able to efficiently mange and manipulate data in your applications.

**Generic vs Non-Generic Collections**

1. **Generic Collections:**

**What are Generic Collections?**

* They are part of the System.Collections.Generic namespace.
* Generics allow us to create reusable code that can work with different types.
* Generics introduce the concept of type parameters to collections, making them more flexible and type-safe.
* They allow the collections to store any data type and prevent runtime type errors.
* They allow you to specify the type of objects they store, for example List<int>.
* They offer type safety, better performance, and reduced need for boxing/unboxing.

Advantages of Generic Collections

* Type Safety: They store elements of a specified type, reducing runtime errors.
* Performance: No need for boxing/unboxing of value types, which improves performance.
* Reduce Memory Overhead: They directly store elements without converting them to object type.
* Code Reusability: Avoid code duplication by creating generic algorithms and data structures.

**Key Generic Collections**

1. List<T>: A list of elements that can be accessed by index.
2. Dictionary<TKey, TValue>: A collection of key-value pairs.
3. Queue<T>: A first-in, first-out (FIFO) collection of objects.
4. Stack<T>: A last-in, first-out (LIFO) collection of objects.
5. HashSet<T>: A collection of unique and unordered elements.
6. LinkedList<T>: A double – linked list.
7. SortedSet<T>: A collection of objects that maintains order.
8. SortedDictionary<TKey, TValue>: A dictionary with sorted keys.
9. SortedList<TKey, TValue>: Similar to SortedDictionary but with different performance characteristics.
10. ConcurrentDictionary<TKey, TValue>: A thread-safe dictionary used in concurrent scenarios.
11. BlockingCollection<T>: Provides blocking and bounding capabilities for thread-safe collections.
12. ConcurrentBag<T>: An unordered collection of objects suitable for concurrent scenarios.
13. ConcurrentQueue<T>: A thread-safe FIFO collection.
14. ConcurrentStack<T>: A thread-safe LIFO collection.
15. **Non – Generic Collections**

**What are Non – Generic Collections?**

* Non – Generic collections are part of the System.Collections namespace.
* They store elements as object types, allowing them to hold any data type.
* They require boxing/unboxing for value types.

**Disadvantages of Non – Generic?**

* Type Unsafe: Can store any type of object, leading to runtime errors.
* Performance Overhead: Boxing/Unboxing of value types impacts performance.
* Memory Overhead: Storing value types as object consumes more memory.

**Key Non – Generic Collections:**

1. ArrayList: A dynamically resizable collection.
2. Hashtable: A collection of key-value pairs organized based on the hash code of the key.
3. Queue: A first-in, first-out (FIFO) collection.
4. Stack: A last-in, first-out (LIFO) collection.
5. SortedList: A collection of key-value pairs that are sorted by the keys and are accessible by key and index.
6. BitArray: Manages compact array of bit values, which are represented as Booleans.
7. HybridDictionary: Implements IDictionary using ListDictionary while the collection is small, and then switching to Hashtable as the collection grows.
8. ListDictionary: A simple, small dictionary implemented as a singly linked list.
9. NameValueCollection: Represents a collection of associated string keys and string values that can be accessed either with the key or with the index.
10. OrderedDictionary: A collection of key-value pairs that are accessible by the key or index.
11. StringCollection: A collection of strings.
12. StringDictionary: A collection of associated string keys and string values with a hash table implementation.

**Conclusion:**

Understanding the distinction between generic and non-generic collections is crucial for selecting the right type of collection in C#. while generic collections are preferred for their type safety and performance benefits, non – generic collections can still be useful in scenarios requiring heterogenous data storage.

These collections offer a wide range of functionalities and characteristics, making them suitable for various scenarios in programming. The choice between generic and non-generic collections typically depends on factors like type safety, performance requirement and specific use cases.