Adnans C# windows application learning second document

This will follow the C++ book I got from school we will do all the syntax for it:

We will be starting at chapter 8, arrays and strings:

As a programmer there are a few type of arrays one-dimension and two-dimension is what we will be covering in chapter 8:

A one-dimension array syntax in C++ is (datatype arrayName [integer (length of array)]); OR: (datatype arrayName = {1,2,3,4,5}); this must be hard coded into when declaring the variable.

In C# the syntax is like this:

int[] numbers = {1,2,3,4,5}; OR: int[] numbers = new int[5];

When accessing an array the array starts at number 0 and goes to -1 of the maximum for example: datatype arrayName [10]; will equal 0,1,2,3,4,5,6,7,8,9 NOT 1,2,3,4,5,6,7,8,9,10.

Another example if you want to clear an array in C# use the following line of code (array.Clear(numbers, 0, numbers.Length); the Length has to be an actual hard coded variable before the array.Clear statement needs to be programmed like this.

Int Length = numbers.Length; THEN use the array.Clear statement.

If you want to see the number of dimensions of an array use the following code (int rank = numbers.Rank); this is good practice for the array.Clear statement.

To print each element of an array as a programmer you should use the FOR(int I 0; I < numbners.Length; i++){Console.WriteLine(numbers[i]);} this will print all the elements of an array in order of FIRST to LAST;

To add to an array the syntax is as follows:

Numbers[2] = 10; OR: for (int I = 0; I < array.Length; i++) {numbers[i] = numbers2[j];}

Furthermore another way to print all elements of an array is to use the FOREACH statement for example: foreach (int num in numbers){Console.WriteLine(num);}

Now we are onto 2-dimensional arrays C# syntax:

**Key Points:**

* Two-dimensional arrays in C# are zero-based, meaning the index of the first row and column is 0.
* The **[,]** syntax is used to declare a two-dimensional array in C#.
* The **GetLength** method is used to get the number of rows and columns in the array.
* C# supports multidimensional arrays with more than two dimensions using similar syntax (**[,,]**, **[,,,]**, etc.).

Two-dimensional arrays in C# are commonly used for representing grids, matrices, tables, and other structured data with rows and columns. They provide a convenient way to store and manipulate data in a structured format.

Next we will learn about Strings and C-strings the C# syntax is:

Strings can be in any format listed below:

String Greeting = “$Hello, {name};

String MyString = null

String MyString = “” // empty string

C# C-Strings:

using System;

using System.Text;

class Program {

static void Main() {

// Declaration and initialization of a C string

byte[] cStringBytes = Encoding.ASCII.GetBytes("Hello, World!\0");

// Convert the byte array to a C# string

string cString = Encoding.ASCII.GetString(cStringBytes);

// Output the C string

Console.WriteLine(cString);

// Get the length of the C string

int length = cString.Length;

Console.WriteLine("Length: " + length);

}

}

Chapter 9 RECORDS (structs):

A STRUCT is the use of different variables with in an array.

For example

public struct Point

{

public int X;

public int Y;

}

Point p; // Declares a variable of type Point

p.X = 10; // Accesses members using dot notation

p.Y = 20;

Console.WriteLine($"Point coordinates: ({p.X}, {p.Y})");

**Guidelines:**

* **Keep Them Small:** Avoid defining large structs, as copying large structs can be inefficient.
* **Immutability:** Prefer immutability for struct instances whenever possible to avoid unexpected behavior.
* **Beware of Boxing:** Be cautious when using structs with interfaces or generic collections, as boxing and unboxing can occur.

**Use Cases:**

* **Simple Data Structures:** Structs are commonly used to represent simple data structures that contain a small number of fields.
* **Performance Considerations:** Structs can be useful for performance-critical scenarios where the overhead of heap allocation and garbage collection associated with classes is undesirable.
* **Interoperability:** Structs are often used for interoperability with unmanaged code or when defining binary data structures.
* It seems you might be referring to a typo. If you meant "FileInfo" or "DirectoryInfo" instead of "IOFile," here's a brief explanation of both:
* **FileInfo Class:**
* The **FileInfo** class in C# provides properties and methods for working with files. You can use it to obtain information about a file, such as its size, attributes, creation time, and last access time, and to perform operations like copying, moving, deleting, and opening files for reading or writing.
* Example usage:

using System;

using System.IO;

class Program

{

static void Main(string[] args)

{

// Creating a FileInfo object for a file

FileInfo fileInfo = new FileInfo("example.txt");

// Getting file information

Console.WriteLine($"File Name: {fileInfo.Name}");

Console.WriteLine($"File Size: {fileInfo.Length} bytes");

Console.WriteLine($"Creation Time: {fileInfo.CreationTime}");

Console.WriteLine($"Last Access Time: {fileInfo.LastAccessTime}");

// Deleting the file

fileInfo.Delete();

}

}

**DirectoryInfo Class:**

The **DirectoryInfo** class provides properties and methods for working with directories. You can use it to obtain information about a directory, such as its name, parent directory, creation time, and to perform operations like creating, moving, deleting, and enumerating files and subdirectories within a directory.

Example usage:

using System;

using System.IO;

class Program

{

static void Main(string[] args)

{

// Creating a DirectoryInfo object for a directory

DirectoryInfo directoryInfo = new DirectoryInfo(@"C:\ExampleDirectory");

// Getting directory information

Console.WriteLine($"Directory Name: {directoryInfo.Name}");

Console.WriteLine($"Parent Directory: {directoryInfo.Parent}");

Console.WriteLine($"Creation Time: {directoryInfo.CreationTime}");

// Enumerating files in the directory

FileInfo[] files = directoryInfo.GetFiles();

foreach (FileInfo file in files)

{

Console.WriteLine($"File Name: {file.Name}, File Size: {file.Length} bytes");

}

// Enumerating subdirectories in the directory

DirectoryInfo[] directories = directoryInfo.GetDirectories();

foreach (DirectoryInfo dir in directories)

{

Console.WriteLine($"Subdirectory Name: {dir.Name}");

}

}

}

Lesson 10 classes and data abstraction:

In C#, classes are the fundamental building blocks of object-oriented programming (OOP). They are used to define blueprints for creating objects, which are instances of classes. Here's an overview of how classes work in C#:

**Declaration:**

public class MyClass

{

// Fields (attributes)

public int MyField;

// Constructor

public MyClass()

{

// Constructor logic

}

// Methods (functions)

public void MyMethod()

{

// Method logic

}

}

**Usage:**

// Creating an instance of MyClass

MyClass obj = new MyClass();

// Accessing fields and methods

obj.MyField = 10;

obj.MyMethod();

**Characteristics:**

1. **Fields (Attributes):**
   * Fields are variables declared within a class. They represent the data or attributes associated with objects of the class.
   * Fields can have access modifiers like **public**, **private**, **protected**, etc., controlling their visibility and accessibility.
2. **Constructors:**
   * Constructors are special methods used for initializing objects of a class.
   * They have the same name as the class and do not have a return type.
   * Constructors can be parameterized or parameterless.
3. **Methods (Functions):**
   * Methods are functions defined within a class. They encapsulate behavior associated with objects of the class.
   * Methods can have access modifiers, parameters, and return types.
4. **Properties:**
   * Properties are a type of class member that encapsulates private fields and provides access to them through getter and setter methods.
   * Properties allow controlled access to fields, enabling validation and encapsulation.
5. **Inheritance:**
   * Classes support inheritance, allowing one class to inherit members (fields, properties, methods) from another class.
   * Inheritance promotes code reuse and hierarchical organization of classes.
6. **Encapsulation:**
   * Classes support encapsulation, which means hiding the internal state and functionality of objects and exposing only the necessary features through well-defined interfaces.
7. **Polymorphism:**
   * Classes support polymorphism, allowing methods to behave differently based on the object they are called on.
   * Polymorphism enables dynamic method invocation and method overriding.

**Use Cases:**

* **Modeling Real-World Entities:** Classes are used to model real-world entities, such as people, cars, or bank accounts, by encapsulating their properties and behaviors.
* **Encapsulating Logic:** Classes encapsulate related logic and data into cohesive units, promoting modularity and maintainability.
* **Creating Libraries:** Classes are used to create reusable components and libraries, allowing code to be organized into reusable modules

Functions in C#:

Function are referred to as methods in C# when defined in a class. Methods are block of code that encapsulates a specific behavior or functionality.

C# syntax for methods:

public class MyClass

{

// Method declaration

public void MyMethod()

{

// Method logic

}

// Method with parameters

public int Add(int a, int b)

{

return a + b;

}

// Static method

public static void StaticMethod()

{

// Static method logic

}

}

Usage:

// Creating an instance of MyClass

MyClass obj = new MyClass();

// Calling methods

obj.MyMethod(); // Calling instance method

int sum = obj.Add(5, 3); // Calling method with parameters

// Calling static method

MyClass.StaticMethod();

**Characteristics:**

1. **Access Modifiers:**
   * Methods can have access modifiers (**public**, **private**, **protected**, **internal**, etc.) controlling their visibility and accessibility.
2. **Return Type:**
   * Methods can have a return type indicating the type of value returned by the method. If a method does not return a value, its return type is **void**.
3. **Parameters:**
   * Methods can accept parameters, allowing data to be passed into the method for processing.
   * Parameters can have different data types and can be optional, using default parameter values.
4. **Static Methods:**
   * Static methods belong to the class rather than to specific instances of the class.
   * They can be called directly on the class without creating an instance of the class.
5. **Overloading:**
   * Methods can be overloaded, meaning you can define multiple methods with the same name but with different parameter lists.
6. **Extension Methods:**
   * Extension methods allow you to add new methods to existing types without modifying their source code. They are defined in static classes.

**Use Cases:**

* **Encapsulating Logic:** Methods encapsulate specific functionality, promoting code organization and modularity.
* **Code Reusability:** Methods allow you to define code once and reuse it multiple times throughout your application.
* **Abstraction:** Methods provide a level of abstraction, allowing you to hide the implementation details and expose a clean interface to users.

A constructor:

In C#, a constructor is a special method that gets called when an instance of a class is created. It is used to initialize the object's state. Here's an overview of constructors in C#:

**Declaration:**

public class MyClass

{

// Default constructor (parameterless)

public MyClass()

{

// Constructor logic

}

// Constructor with parameters

public MyClass(int value)

{

// Constructor logic with parameter

}

}

**Usage:**

// Creating instances of MyClass

MyClass obj1 = new MyClass(); // Calls the default constructor

MyClass obj2 = new MyClass(10); // Calls the constructor with parameter

**Characteristics:**

1. **Name:**
   * Constructors have the same name as the class they belong to.
2. **Return Type:**
   * Constructors don't have a return type, not even **void**.
3. **Initialization:**
   * Constructors are used to initialize the object's state, such as initializing fields, properties, or performing any setup logic required by the class.
4. **Default Constructor:**
   * If a class doesn't define any constructors, a default constructor (parameterless) is provided implicitly by the compiler. It initializes fields to their default values.
5. **Parameterized Constructors:**
   * Constructors can have parameters, allowing you to pass values at the time of object creation.
6. **Constructor Overloading:**
   * Like methods, constructors can be overloaded, meaning you can define multiple constructors with different parameter lists.

**Use Cases:**

* **Initialization:** Constructors are used to initialize the state of objects when they are created.
* **Dependency Injection:** Constructors can be used for dependency injection, where dependencies are passed into a class through constructor parameters.
* **Object Initialization:** Constructors can perform any necessary setup or initialization required by the class before it can be used.

Chapter 11 inheritance and composition:

Inheritance and composition are two fundamental concepts in object-oriented programming, including C#. They both facilitate code reuse and promote modularity, but they approach it in different ways.

### Inheritance:

Inheritance is an "is-a" relationship, where one class (subclass or derived class) inherits characteristics (fields, properties, methods) from another class (base class or superclass). In C#, you can achieve inheritance using the `: baseClassName` syntax.

```csharp

// Base class

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating...");

}

}

// Derived class inheriting from Animal

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking...");

}

}

```

#### Characteristics of Inheritance:

- \*\*Code Reusability:\*\* Inheritance allows you to reuse code from the base class in the derived class.

- \*\*Polymorphism:\*\* Subclasses can override methods of the base class to provide specialized behavior (method overriding).

- \*\*Single Inheritance:\*\* C# supports single inheritance, meaning a class can inherit from only one base class. However, it supports multiple levels of inheritance (multi-level inheritance).

### Composition:

Composition is a "has-a" relationship, where a class contains instances of other classes as members. It allows you to create more complex objects by combining simpler ones. In C#, you can achieve composition by instantiating objects of other classes within your class.

```csharp

// Class representing a Car

public class Car

{

private Engine engine; // Composition: Car "has-a" Engine

public Car()

{

engine = new Engine();

}

public void Drive()

{

engine.Start();

Console.WriteLine("Driving...");

}

}

// Class representing an Engine

public class Engine

{

public void Start()

{

Console.WriteLine("Engine started.");

}

}

```

#### Characteristics of Composition:

- \*\*Flexibility:\*\* Composition allows for greater flexibility in defining relationships between objects compared to inheritance.

- \*\*Code Reusability:\*\* Composition promotes code reuse by allowing you to reuse existing classes as components within other classes.

- \*\*Encapsulation:\*\* Composition helps encapsulate the functionality of individual components, making the code easier to manage and maintain.

### When to Use Each:

- \*\*Inheritance:\*\* Use inheritance when there is a clear "is-a" relationship between classes and when you want to promote code reuse and polymorphism.

- \*\*Composition:\*\* Use composition when there is a "has-a" relationship between classes and when you want to create complex objects by assembling simpler components.

In summary, both inheritance and composition are essential tools in object-oriented programming, each with its own strengths and use cases. Understanding when to use each approach is crucial for designing robust and maintainable object-oriented systems.

Chapter 12 pointers, virtual functions and abstract functions:

Pointers in C# provide a way to directly manipulate memory addresses, similar to pointers in languages like C and C++. However, in C#, pointer operations are only allowed in an unsafe context. Here's an overview of how pointers work in C#:

### Declaration:

To declare a pointer variable, you use the `unsafe` keyword followed by the pointer type (`\*`) and the variable name:

```csharp

unsafe

{

int\* ptr;

}

```

### Initialization:

You can initialize a pointer variable by assigning it the address of another variable using the address-of operator (`&`):

```csharp

unsafe

{

int x = 10;

int\* ptr = &x;

}

```

### Dereferencing:

To access the value stored at the memory address pointed to by a pointer, you use the dereference operator (`\*`):

```csharp

unsafe

{

int x = 10;

int\* ptr = &x;

int value = \*ptr; // value now holds the value stored at the memory address pointed to by ptr

}

```

### Pointer Arithmetic:

You can perform pointer arithmetic to navigate through memory addresses. This is useful for working with arrays and manipulating memory directly:

```csharp

unsafe

{

int[] arr = { 1, 2, 3, 4, 5 };

fixed (int\* ptr = arr)

{

int\* p = ptr;

for (int i = 0; i < 5; i++)

{

Console.WriteLine(\*p);

p++;

}

}

}

```

### Unsafe Context:

All operations involving pointers must be done within an `unsafe` context. You can define an unsafe block using the `unsafe` keyword:

```csharp

unsafe

{

// Pointer operations go here

}

```

### Fixed Statement:

When working with pointers to managed objects (e.g., arrays), you need to use the `fixed` statement to pin the object in memory to prevent it from being moved by the garbage collector:

```csharp

unsafe

{

int[] arr = { 1, 2, 3, 4, 5 };

fixed (int\* ptr = arr)

{

// Pointer operations go here

}

}

```

### Restrictions:

- Pointer operations are inherently unsafe and can lead to memory corruption and security vulnerabilities if not used carefully.

- Pointer arithmetic can only be performed within an `unsafe` context.

- Code containing unsafe operations must be compiled with the `/unsafe` compiler option.

### Use Cases:

- Interoperability: Pointers are often used in interop scenarios when calling unmanaged code or interacting with hardware.

- Performance Optimization: Pointers can be used to optimize performance-critical code by directly manipulating memory.

### Safety Considerations:

- Avoid using pointers unless absolutely necessary, as they bypass many of the safety features provided by managed code.

- Ensure proper bounds checking and validation to prevent buffer overflows and other memory-related issues.

- Use tools like code analysis and code review to catch potential issues when working with pointers.

In summary, pointers in C# provide a way to work with memory addresses directly, but they come with significant risks and should be used judiciously and only when absolutely necessary. Most C# development can be done without using pointers, leveraging the safety and convenience of managed code.

In C#, both `virtual` and `abstract` members are used in the context of inheritance and polymorphism to provide flexibility in class design. Here's an overview of each:

### Virtual Members:

A `virtual` member in a base class allows derived classes to override its behavior by providing their own implementation. This enables polymorphic behavior, where the method called depends on the runtime type of the object.

#### Syntax:

```csharp

public class BaseClass

{

public virtual void Method()

{

Console.WriteLine("Base class method");

}

}

public class DerivedClass : BaseClass

{

public override void Method()

{

Console.WriteLine("Derived class method");

}

}

```

#### Characteristics:

- Derived classes can choose whether or not to override a virtual member.

- Virtual members have a default implementation in the base class.

- The `virtual` keyword allows methods to be overridden by derived classes using the `override` keyword.

- Virtual members can have a body in the base class, providing a default behavior.

### Abstract Members:

An `abstract` member in a base class defines a method signature without providing any implementation. It mandates that derived classes must provide an implementation for the abstract member.

#### Syntax:

```csharp

public abstract class BaseClass

{

public abstract void Method(); // Abstract method declaration

}

public class DerivedClass : BaseClass

{

public override void Method()

{

Console.WriteLine("Derived class method");

}

}

```

#### Characteristics:

- Abstract members do not have a default implementation and must be overridden by derived classes.

- Abstract classes cannot be instantiated directly; they are meant to be subclassed.

- Abstract methods are declared using the `abstract` keyword and do not have a body.

- Any class that inherits from an abstract class must provide concrete implementations for all abstract members.

### Use Cases:

- \*\*Virtual Members:\*\*

- Useful when you want to provide a default implementation in a base class but allow derived classes to override it if needed.

- Provides flexibility in class design, allowing customization of behavior in derived classes.

- \*\*Abstract Members:\*\*

- Useful when you want to define a contract that derived classes must adhere to.

- Ensures that specific methods must be implemented in derived classes, enforcing consistency in behavior across subclasses.

### When to Use Each:

- Use `virtual` members when you want to provide a default implementation but allow customization by derived classes.

- Use `abstract` members when you want to define a contract that derived classes must fulfill, ensuring consistent behavior across subclasses.

In summary, both `virtual` and `abstract` members play important roles in class design in C#. They facilitate polymorphism and provide mechanisms for defining flexible and extensible class hierarchies. The choice between them depends on the specific requirements of your design.

Chapter 13 overloading and templates:

In C#, overloading and generics (templates) are two powerful features that enable you to create more flexible and reusable code.

### Method Overloading:

Method overloading allows you to define multiple methods with the same name but with different parameter lists within the same class. The compiler determines which overload to call based on the number and types of arguments provided.

#### Example:

```csharp

public class Calculator

{

// Method overload with two int parameters

public int Add(int a, int b)

{

return a + b;

}

// Method overload with three int parameters

public int Add(int a, int b, int c)

{

return a + b + c;

}

}

```

#### Usage:

```csharp

Calculator calculator = new Calculator();

int sum1 = calculator.Add(1, 2); // Calls the first overload

int sum2 = calculator.Add(1, 2, 3); // Calls the second overload

```

### Generics (Templates):

Generics, also known as templates in other languages, allow you to create classes, structures, methods, and interfaces that operate with specified types. This enables you to write code that can work with any data type.

#### Example (Generic Method):

```csharp

public class GenericList<T>

{

private T[] array = new T[10];

private int currentIndex = 0;

// Generic method to add an item to the list

public void Add(T item)

{

if (currentIndex < array.Length)

{

array[currentIndex] = item;

currentIndex++;

}

else

{

Console.WriteLine("List is full.");

}

}

}

```

#### Usage:

```csharp

GenericList<int> intList = new GenericList<int>();

intList.Add(10);

GenericList<string> stringList = new GenericList<string>();

stringList.Add("Hello");

```

### Key Differences:

- \*\*Overloading:\*\* Provides multiple methods with the same name but different parameter lists.

- \*\*Generics:\*\* Provides a way to define classes, methods, or structures that can work with any data type.

### Use Cases:

- \*\*Method Overloading:\*\*

- Use when you want to provide multiple ways to call a method with different parameters.

- Useful for creating methods that perform similar operations but on different types or numbers of parameters.

- \*\*Generics:\*\*

- Use when you want to create reusable components that can work with any data type.

- Allows you to write code that is more flexible, efficient, and type-safe.

In summary, method overloading and generics are both important features in C# that allow you to write more flexible and reusable code. Method overloading provides flexibility in method signatures, while generics enable you to create reusable components that can work with any data type. Choosing between them depends on the specific requirements of your code.

Chapter 14 exception handling:

Exception handling in C# allows you to gracefully handle runtime errors and unexpected situations that may occur during program execution. It helps prevent your application from crashing and allows you to handle errors in a controlled manner. Here's an overview of how exception handling works in C#:

### Try-Catch Block:

The primary mechanism for handling exceptions in C# is the `try-catch` block. Code that might throw an exception is enclosed within a `try` block, and any exceptions that occur are caught and handled in the corresponding `catch` block.

#### Syntax:

```csharp

try

{

// Code that might throw an exception

}

catch (ExceptionType1 ex)

{

// Handle ExceptionType1

}

catch (ExceptionType2 ex)

{

// Handle ExceptionType2

}

finally

{

// Optional finally block executes regardless of whether an exception occurred

}

```

#### Example:

```csharp

try

{

int result = 10 / 0; // This will throw a DivideByZeroException

}

catch (DivideByZeroException ex)

{

Console.WriteLine("Error: " + ex.Message);

}

catch (Exception ex) // Catching a general exception

{

Console.WriteLine("An error occurred: " + ex.Message);

}

finally

{

Console.WriteLine("Cleanup code here."); // Optional: Cleanup code

}

```

### Throw Statement:

You can manually throw exceptions using the `throw` statement. This allows you to create custom exceptions or rethrow exceptions caught in a `catch` block.

#### Syntax:

```csharp

try

{

throw new Exception("Custom error message");

}

catch (Exception ex)

{

Console.WriteLine("An error occurred: " + ex.Message);

}

```

### Finally Block:

The `finally` block, if provided, is executed regardless of whether an exception occurs. It's commonly used for cleanup code that should always run, such as releasing resources.

### Exception Types:

C# provides a wide range of built-in exception types, such as `ArgumentException`, `InvalidOperationException`, `NullReferenceException`, etc. Additionally, you can create custom exception types by deriving from the `Exception` class.

### Best Practices:

- Only catch exceptions that you can handle properly. Avoid catching general `Exception` types unless necessary.

- Always log exceptions or provide appropriate error messages to aid in troubleshooting.

- Clean up resources in the `finally` block to ensure proper resource management.

- Use multiple `catch` blocks to handle different types of exceptions appropriately.

### Exception Filters (C# 6 and later):

Starting from C# 6, you can use exception filters to specify conditions under which a `catch` block should handle an exception. This allows for more precise exception handling.

#### Syntax:

```csharp

try

{

// Code that might throw an exception

}

catch (Exception ex) when (condition)

{

// Handle the exception only if the condition is true

}

```

#### Example:

```csharp

try

{

// Code that might throw an exception

}

catch (DivideByZeroException ex) when (ex.Message.Contains("zero"))

{

Console.WriteLine("Error: Attempt to divide by zero.");

}

```

In summary, exception handling in C# provides a robust mechanism for dealing with errors and unexpected situations in your code. By using `try-catch` blocks, you can gracefully handle exceptions and ensure that your application remains stable and responsive.

Chapter 15 recursion:

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem. In C#, recursion is a powerful tool for solving problems that can be broken down into smaller, similar sub-problems. Here's an overview of how recursion works in C#:

### Basic Example:

Let's consider the classic example of calculating the factorial of a number using recursion.

```csharp

public class RecursionExample

{

public int Factorial(int n)

{

// Base case: factorial of 0 or 1 is 1

if (n == 0 || n == 1)

return 1;

// Recursive case: n! = n \* (n-1)!

return n \* Factorial(n - 1);

}

}

```

### Key Components:

1. \*\*Base Case:\*\*

- Every recursive function must have a base case that specifies when the recursion should stop. Without a base case, the function will keep calling itself indefinitely, leading to a stack overflow error.

2. \*\*Recursive Case:\*\*

- The recursive case defines how the problem is broken down into smaller instances. It typically involves calling the function with a smaller or simpler argument.

### Usage:

```csharp

RecursionExample example = new RecursionExample();

int result = example.Factorial(5); // Calculate factorial of 5

Console.WriteLine("Factorial of 5 is: " + result); // Output: 120

```

### Visualizing Recursion:

Recursion can be challenging to understand because it involves a function calling itself. Visualizing the recursive calls using a call stack can help understand how recursion works.

### Use Cases:

- \*\*Tree and Graph Traversal:\*\* Recursion is commonly used for traversing tree and graph data structures, such as binary trees, where each node has zero or more child nodes.

- \*\*Sorting Algorithms:\*\* Some sorting algorithms, like quicksort and mergesort, use recursion to divide the problem into smaller sub-problems.

- \*\*Dynamic Programming:\*\* Recursion is used in dynamic programming to break down complex problems into simpler sub-problems and cache their solutions for efficiency.

### Considerations:

- \*\*Performance:\*\* Recursive solutions may be less efficient than iterative solutions for some problems due to the overhead of function calls and the potential for stack overflow.

- \*\*Stack Overflow:\*\* Recursion can lead to stack overflow errors if not implemented correctly or if the problem size is too large.

### Tail Recursion Optimization (TCO):

C# does not provide native support for tail call optimization (TCO), which eliminates the need for additional stack frames in tail-recursive functions. Therefore, in C#, tail-recursive functions can still lead to stack overflow errors for large input values.

In summary, recursion is a powerful programming technique in C# for solving problems by breaking them down into smaller instances. Understanding how recursion works and using it appropriately can help you write elegant and efficient code. However, it's essential to handle base cases properly and be mindful of potential stack overflow errors for large input sizes.

Chapter 16 searching, sorting, and vector types:

In C#, searching and sorting algorithms are typically implemented using collections provided by the .NET Framework, such as arrays, lists, dictionaries, etc. Additionally, C# provides a `System.Array` class for arrays, LINQ (Language-Integrated Query) for querying collections, and various sorting algorithms in the `System.Linq` namespace. Here's an overview:

### Searching:

#### Linear Search:

```csharp

int[] array = { 10, 20, 30, 40, 50 };

int target = 30;

int index = Array.IndexOf(array, target); // Returns the index of the target element

if (index != -1)

{

Console.WriteLine($"Element {target} found at index {index}");

}

else

{

Console.WriteLine("Element not found");

}

```

#### Binary Search (for sorted arrays):

```csharp

int[] array = { 10, 20, 30, 40, 50 };

int target = 30;

int index = Array.BinarySearch(array, target); // Returns the index of the target element

if (index >= 0)

{

Console.WriteLine($"Element {target} found at index {index}");

}

else

{

Console.WriteLine("Element not found");

}

```

### Sorting:

#### Sorting Arrays:

```csharp

int[] array = { 5, 2, 9, 1, 3 };

Array.Sort(array); // Sorts the array in ascending order

foreach (int num in array)

{

Console.WriteLine(num);

}

```

#### Sorting Lists:

```csharp

List<int> list = new List<int> { 5, 2, 9, 1, 3 };

list.Sort(); // Sorts the list in ascending order

foreach (int num in list)

{

Console.WriteLine(num);

}

```

#### Custom Sorting:

```csharp

List<int> list = new List<int> { 5, 2, 9, 1, 3 };

list.Sort((x, y) => x.CompareTo(y)); // Custom sorting using a lambda expression

foreach (int num in list)

{

Console.WriteLine(num);

}

```

### Vector Types:

C# does not have built-in support for vector types like some other languages (e.g., C++ with `std::vector`). However, you can use arrays or collections to achieve similar functionality.

#### Example:

```csharp

// Using arrays to represent a vector

int[] vector = { 1, 2, 3 };

```

Alternatively, you could use third-party libraries or specialized numeric libraries that provide vector types and operations if needed.

In summary, in C#, you can use built-in methods provided by the .NET Framework for searching and sorting arrays and collections. While there are no built-in vector types in C#, you can use arrays or collections to represent vectors and perform operations accordingly. Additionally, third-party libraries can be used for more specialized numeric operations if required.

Chapter 17 linked lists:

In C#, a linked list is a linear data structure where elements, called nodes, are connected by pointers. Unlike arrays, linked lists do not have a fixed size and allow for efficient insertion and deletion of elements anywhere in the list. The LinkedList<T> class in C# provides an implementation of a doubly linked list. Here's an overview of how to work with linked lists in C#:

### Creating a Linked List:

```csharp

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

// Creating a linked list of integers

LinkedList<int> linkedList = new LinkedList<int>();

// Adding elements to the linked list

linkedList.AddLast(10);

linkedList.AddLast(20);

linkedList.AddLast(30);

}

}

```

### Basic Operations:

#### Adding Elements:

```csharp

// Adding elements to the end of the list

linkedList.AddLast(40);

// Adding elements to the beginning of the list

linkedList.AddFirst(5);

```

#### Removing Elements:

```csharp

// Removing the first element

linkedList.RemoveFirst();

// Removing the last element

linkedList.RemoveLast();

// Removing a specific element

linkedList.Remove(20); // Removes the first occurrence of 20

```

#### Accessing Elements:

```csharp

// Accessing the first and last elements

int firstElement = linkedList.First.Value;

int lastElement = linkedList.Last.Value;

// Iterating over the linked list

foreach (int item in linkedList)

{

Console.WriteLine(item);

}

```

### Advanced Operations:

#### Finding Elements:

```csharp

LinkedListNode<int> node = linkedList.Find(20); // Finds the first occurrence of 20

if (node != null)

{

Console.WriteLine("Found: " + node.Value);

}

```

#### Inserting Elements After/Before a Node:

```csharp

LinkedListNode<int> node = linkedList.Find(10);

if (node != null)

{

linkedList.AddAfter(node, 15); // Inserts 15 after the node

linkedList.AddBefore(node, 5); // Inserts 5 before the node

}

```

#### Clearing the List:

```csharp

linkedList.Clear(); // Removes all elements from the list

```

### Advantages of Linked Lists:

- Dynamic Size: Linked lists can grow or shrink in size dynamically without needing to preallocate memory.

- Efficient Insertion and Deletion: Insertion and deletion operations are typically more efficient than arrays, especially for large lists.

### Disadvantages of Linked Lists:

- Lack of Random Access: Accessing elements by index is less efficient compared to arrays since you have to traverse the list from the beginning.

- Additional Memory Overhead: Linked lists use additional memory to store pointers/references to the next and previous nodes.

In summary, linked lists in C# are a useful data structure for scenarios where dynamic size and efficient insertion/deletion operations are important. The LinkedList<T> class in C# provides a convenient implementation of doubly linked lists.

Chapter 18 stacks and ques:

In C#, stacks and queues are common data structures used for managing collections of items with specific behaviors. Both are provided by the .NET Framework and have their own respective classes: `Stack<T>` for stacks and `Queue<T>` for queues. Here's an overview of how to use stacks and queues in C#:

### Stacks:

A stack is a last-in, first-out (LIFO) data structure, where elements are inserted and removed from the top of the stack. The `Stack<T>` class in C# provides an implementation of a generic stack.

#### Example:

```csharp

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

// Creating a stack of integers

Stack<int> stack = new Stack<int>();

// Pushing elements onto the stack

stack.Push(10);

stack.Push(20);

stack.Push(30);

// Popping elements from the stack

int poppedElement = stack.Pop(); // Removes and returns the top element (30)

Console.WriteLine("Popped Element: " + poppedElement);

// Accessing the top element without removing it

int topElement = stack.Peek(); // Returns the top element without removing it (20)

Console.WriteLine("Top Element: " + topElement);

}

}

```

### Queues:

A queue is a first-in, first-out (FIFO) data structure, where elements are inserted at the rear and removed from the front of the queue. The `Queue<T>` class in C# provides an implementation of a generic queue.

#### Example:

```csharp

using System;

using System.Collections.Generic;

class Program

{

static void Main(string[] args)

{

// Creating a queue of strings

Queue<string> queue = new Queue<string>();

// Enqueuing elements into the queue

queue.Enqueue("John");

queue.Enqueue("Alice");

queue.Enqueue("Bob");

// Dequeuing elements from the queue

string dequeuedElement = queue.Dequeue(); // Removes and returns the front element ("John")

Console.WriteLine("Dequeued Element: " + dequeuedElement);

// Accessing the front element without removing it

string frontElement = queue.Peek(); // Returns the front element without removing it ("Alice")

Console.WriteLine("Front Element: " + frontElement);

}

}

```

### Use Cases:

- \*\*Stacks:\*\*

- Used for implementing function calls in programming languages (call stack).

- Reversing the order of elements.

- Undo operations in text editors.

- \*\*Queues:\*\*

- Process scheduling (task queues).

- Breadth-first search algorithms.

- Handling requests in web servers.

### Advantages:

- Stacks and queues provide efficient insertion and removal operations with constant time complexity.

- They enforce specific orderings (LIFO for stacks, FIFO for queues), which can be useful in various scenarios.

### Disadvantages:

- Stacks and queues do not support random access to elements (e.g., accessing elements by index).

- The `.Peek()` method in both stacks and queues can throw an exception if called on an empty collection.

In summary, stacks and queues are fundamental data structures in computer science and are widely used in various applications. The `Stack<T>` and `Queue<T>` classes in C# provide convenient implementations for working with stacks and queues in .NET applications.

This is the end of the C++ book I got from school.

You should now understand the basic and some advance syntax of C# code.