Chapter-6 Pointers

In C#, data types are categorized into three categories: pointer types, value types, and reference types based on how they store their value in the memory.

❖ Introduction:

A pointer is a variable whose value is the address of another variable, i.e., the direct address of the memory location. The syntax of a pointer is –

```
type *var-name;
```

The following is how you can declare a pointer type –

```
double *z;      /* pointer to a double */
```

C# allows using pointer variables in a function or code block when it is marked by the '**unsafe**' modifier. The unsafe code or the unmanaged code is a code block that uses a pointer variable.

The following is our module showing how to declare and use a pointer variable. We have used the '**unsafe'** modifier here –

```
static unsafe void Main(string[] args) {
   int val = 50;
   int* x = &val;
   Console.WriteLine("Data: {0} ", val);
   Console.WriteLine("Address: {0}", (int)x);
   Console.ReadKey();
}
```

❖ Unsafe Codes:

The C# statements can be executed either as in a safe or in an unsafe context. The statements marked as unsafe by using the keyword unsafe run outside the control of the Garbage Collector. Remember that in C#, any code involving pointers requires an unsafe context.

Features of Pointer:

1. Pointers save memory space.
2. Execution time with pointers is faster because data is manipulated with the address, i.e., direct access to memory location.
3. Memory is accessed efficiently with pointers. The pointer assigns and releases the memory as well. Hence, it can be said that the Memory of pointers is dynamically allocated.
4. Pointers are used with data structures. They are useful for representing two-dimensional and multidimensional arrays.
5. An array, of any type, can be accessed with the help of pointers, without considering its subscript range.
6. Pointers are used for file handling.
7. Pointers are used to allocate memory dynamically.
8. In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Application of Pointer:

1) To pass arguments by reference.
2) For accessing array elements.
3) To return multiple values.
4) Dynamic memory allocation.
5) To implement data structures.
6) To do system-level programming where memory addresses are useful.

// C# program to demonstrate the unsafe code

```
using System;

namespace GFG {

   class Program {

      // Main Method
      static void Main(string[] args) {
         // Declaring a code block as
         // unsafe to make use of pointers

         unsafe {
            int x = 10;
            int* ptr;
            ptr = &x;
```

```
            // displaying value of x using pointer
            Console.WriteLine("Inside the unsafe code block");
            Console.WriteLine("The value of x is " + *ptr);
        } // end unsafe block

        Console.WriteLine("\nOutside the unsafe code block");
    } // end main
  }
}
```

In C#, a string is an object, and it is stored on the heap in memory. A pointer is a memory address that refers to the location of an object in memory. In C#, pointers can be used to refer to the memory address of value types such as int, char, and structs, but not reference types such as strings.

Instead of using pointers to refer to strings, C# provides the string type, which is a reference type. A string variable holds a reference to the memory location where the string object is stored, rather than the actual string data. This means that you can use the string variable to access the string object, but you cannot use a pointer to directly access the memory location of the string object.

However, C# provides the fixed keyword, which can be used to pin a string object in memory and obtain a pointer to the first character of the string. The fixed keyword is typically used when working with unsafe code and is not recommended for general use.

For instance:

```
using System;

class Program
{
  static unsafe void Main()
  {
    string myString = "Hello World";

    // Use fixed to get a pointer to the first character of the string
    fixed (char* ptr = myString)
    {
      // Access each character using the pointer
      for (int i = 0; i < myString.Length; i++)
      {
        Console.Write(*(ptr + i));
          // Dereference the pointer to get the character
```

```
        }

    }
  }
}
```

This code creates a pointer to the first character of the string and pins the string object in memory, allowing you to access the individual characters in the string using the pointer. However, it is important to note that working with pointers in this way can be dangerous and can lead to memory leaks, buffer overflow, and other issues if not handled carefully.

It is generally recommended to use the string type in C# and to avoid using pointers with strings, unless you have a specific need to do so and you understand the risks.

❖ Advantages of using pointers in C#:

1. It provide direct access to the memory.
2. It provide an alternate way to access array elements.
3. It provide a way to returns more than one value to the function
4. Pointers reduces the storage space and complexity of program.
5. Pointer reduces the execution of the program.
6. Use for memory management

❖ Disadvantages of using pointers in C#:

1. Pointer are slower than normal variables.
2. Uninitialized pointer might cause segmentation fault.
3. Dynamically allocated block needs to be freed explicitly. Otherwise it would lead to memory leaks.
4. Unsafe, If pointer are updated with incorrect values; it might lead to memory corruption.
5. Pointer bugs are difficult to handle.
6. Limited uses
7. Complex to understand and handle

❖ Access Value from Pointer:

A pointer is a variable whose value is the address of another variable. Retrieve the data stored at the located referenced by the pointer variable, using the ToString() method.

Example:

```
using System;

namespace UnsafeCodeApplication {
   class Program {
      public static void Main() {
         unsafe {
            int var = 100;
            int* p = &var;
            Console.WriteLine("Data is: {0} ", var);
            Console.WriteLine("Data is: {0} ", p->ToString());
            Console.WriteLine("Address is: {0} ", (int)p);
         }
         Console.ReadKey();
      }
   }
}
```

Above will require you to set unsafe command line option. After setting it, the following output would be visible.

Output:

```
Data is: 100
Data is: 100
Address is: 77678547
```

- Here is an example of how to access a data value using a pointer in C#:

```
using System;
class Program {
  public  static void Main(string[] args) {
      unsafe{
                    int number = 10;
                    int* p = &number;
                   Console.WriteLine("The value of number is: " + number);
                   Console.WriteLine("The value of the pointer is: " + *p);
                 }
```

```
    }
}
```

In this example, the value of number is set to 10. The pointer 'p' is declared as an int pointer and is initialized with the address of number. The value of number is then printed to the console, followed by the value stored in the memory location pointed to by 'p', which is 10.

Note that this example uses unsafe code, which requires the unsafe keyword and the /unsafe compiler option to be set.

❖ Passing pointers as parameters to methods in C#

In C#, you can pass a pointer to a method as a parameter using the "unsafe" keyword, which allows you to work with pointers within the method. However, the use of unsafe code is generally discouraged and should only be used in cases where it's necessary for specific performance reasons or for interoperability with unmanaged code.

Here's an example of passing a pointer to a method:

```
using System;

namespace UnsafeCodeApplication {
   class TestPointer {

      unsafe void PrintInt(int* p) {
         Console.WriteLine(*p);
      }

      unsafe static void Main(string[] args) {
         TestPointer t1 = new TestPointer();
         int i = 42;
         int* p = &i;
         t1.PrintInt(p);
      }
   }
}
```

❖ Accessing Array Elements Using a Pointer

In C#, pointers can be employed to efficiently access and manipulate array elements directly through their memory addresses. This capability provides advantages in terms of performance and flexibility.

Example:

Consider the following example demonstrating how to use a pointer to access array elements:

```
using System;

namespace ArrayAccessWithPointer {
   class Program {
      static unsafe void Main(string[] args) {
         int[] numbers = { 10, 20, 30, 40, 50 };

         fixed (int* p = numbers) {
            // Accessing array elements using a pointer
            for (int i = 0; i < numbers.Length; i++) {
               Console.WriteLine($"Element {i + 1}: {*(p + i)}");
            }
         }

         Console.ReadKey();
      }
   }
}
```

In this example, the fixed keyword is used to pin the array in memory, and a pointer (p) is then used to access each element of the array directly. The loop iterates through the array, and the value at each memory location is retrieved using the pointer.

Output:

```
Element 1: 10
Element 2: 20
Element 3: 30
Element 4: 40
Element 5: 50
```

This approach provides a more direct and potentially faster way to iterate through array elements, especially when dealing with large datasets.Using pointers for array access should be done with caution, considering the safety

implications associated with unsafe code. It is essential to ensure proper bounds checking to prevent memory-related issues.

//swap value using pointer

```
using System;
namespace Program{
   class TestPointer{
      public unsafe void swap(int *p, int *q){
         int temp = *p;
         *p=*q;
         *q=temp;
      }

      public unsafe static void Main(){
         TestPointer p= new TestPointer();
         int var1 = 20;
         int var2 = 10;

         int *x = &var1;
         int *y = &var2;

         System.Console.WriteLine($"Before swap: var1: {var1} var2: {var2}");

         p.swap(x,y);
         System.Console.WriteLine($"After swap: var1: {var1} var2: {var2}");
      }
   }
}
```