Pointers In C#

Rajesh VS Updated date May 31, 2019 1.1m 10 11

C# supports pointers in a limited extent. A C# pointer is nothing but a variable that holds the memory address of another type. But in C# pointer can only be declared to hold the memory address of value types and arrays. Unlike reference types, pointer types are not tracked by the default garbage collection mechanism. For the same reason pointers are not allowed to point to a reference type or even to a structure type which contains a reference type. We can say that pointers can point to only unmanaged types which includes all basic data types, enum types, other pointer types and structs which contain only unmanaged types.

Declaring a Pointer type

The general form of declaring a pointer type is as shown below,

*type *variable_name;*

Where * is known as the de-reference operator. For example the following statement

*int *x ;*

Declares a pointer variable x, which can hold the address of an int type. The reference operator (&) can be used to get the memory address of a variable.

```
01. int x = 100;
```

The &x gives the memory address of the variable x, which we can assign to a pointer variable

```
int *ptr = & x;.
Console.WriteLine((int)ptr) // Displays the memory address
Console.WriteLine(*ptr) // Displays the value at the memory address.
```

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 runs outside the control of Garbage Collector. Remember that in C# any code involving pointers requires an unsafe context.

We can use the unsafe keyword in two different ways. It can be used as a modifier to a method, property, and constructor etc. For example

```
// Author: rajeshvs@msn.com
using System;
class MyClass
{
    public unsafe void Method()
    {
        int x = 10;
        int y = 20;
        int* ptr1 = &x;
        int* ptr2 = &y;
        Console.WriteLine((int)ptr1);
        Console.WriteLine((int)ptr2);
        Console.WriteLine(*ptr1);
        Console.WriteLine(*ptr2);
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.Method();
    }
}
```

The keyword unsafe can also be used to mark a group of statements as unsafe as shown below.

```
// Author: rajeshvs@msn.com
//unsafe blocks
using System;
class MyClass
{
    public void Method()
    {
        unsafe
        {
            int x = 10;
```

```
            int y = 20;
            int* ptr1 = &x;
            int* ptr2 = &y;
            Console.WriteLine((int)ptr1);
            Console.WriteLine((int)ptr2);
            Console.WriteLine(*ptr1);
            Console.WriteLine(*ptr2);
        }
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.Method();
    }
}
```

Pinning an Object

The C# garbage collector can move the objects in memory as it wishes during the garbage collection process. The C# provides a special keyword fixed to tell Garbage Collector not to move an object. That means this fixes in memory the location of the value types pointed to. This is what is known as pinning in C#.

The fixed statement is typically implemented by generating tables that describe to the Garbage Collector, which objects are to remain fixed in which regions of executable code. Thus as long as a Garbage Collector process doesn't actually occur during execution of fixed statements, there is very little cost associated with this. However when a Garbage Collector process does occur, fixed objects may cause fragmentation of the heap. Hence objects should be fixed only when absolutely necessary and only for the shortest amount of time.

Pointers & Methods

The points can be passed as an argument to a method as showing below. The methods can also return a pointer.

```
// Author: rajeshvs@msn.com
using System;
class MyClass
{
    public unsafe void Method()
    {
        int x = 10;
        int y = 20;
        int* sum = swap(&x, &y);
        Console.WriteLine(*sum);
    }
    public unsafe int* swap(int* x, int* y)
```

```
    {
        int sum;
        sum = *x + *y;
        return ∑
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.Method();
    }
}
```

Pointers & Conversions

In C# pointer types do not inherit from object and no conversion exists between pointer types and objects. That means boxing and un-boxing are not supported by pointers. But C# supports conversions between the different pointer types and pointer types and integral types.

C# supports both implicit and explicit pointer conversions within un-safe context. The implicit conversions are

1. From any type pointer type to void * type.
2. From null type to any pointer type.

The cast operator (()) is necessary for any explicit type conversions. The explicit type conversions are

1. From any pointer type to any other pointer type.
2. From sbyte, byte, short, ushort, int, uint, long, ulong to any pointer type.
3. From any pointer type to sbyte, byte, short, ushort, int, uint, long, ulong types.

For example

```
char c = 'R';
char *pc = &c;
void *pv = pc; // Implicit conversion
int *pi = (int *) pv; // Explicit conversion using casting operator
```

Pointer Arithmetic

In an un-safe context, the ++ and - operators can be applied to pointer variable of all types except void * type. Thus for every pointer type T* the following operators are implicitly overloaded.

```
T* operator ++ (T *x);
T*operator -- (T *x);
```

The ++ operator adds sizeof(T) to the address contained in the variable and - operator subtracts sizeof(--) from the address contained in the variable for a pointer variable of type T*.

In an un-safe context a constant can be added or subtracted from a pointer variable. Similarly a pointer variable can be subtracted from another pointer variable. But it is not possible to add two pointer variables in C#.

In un-safe context = =, ! =, <, >, < =, > = operators can be applied to value types of all pointer types. The multiplication and division of a pointer variable with a constant or with another pointer variable is not possible in C#.

Stack Allocation

In an unsafe context, a local declaration may include a stack allocation initialiser, which allocates memory from the call stack.

The stackalloc T[E] requires T to be an unmanaged type and E to be an expression of type int. The above construct allocates E * sizeof(T) bytes from the call stack and produces a pointer of the type T* to the newly allocated block. The E is negative, a System.OverFlowException is thrown. If there is not enough memory to allocate, a System.StackOverflowException is thrown.

The content of newly allocated memory is undefined. There is no way to implicitly free memory allocated using stackalloc. Instead all stack allocated memory block are automatically discarded once the function returns.

```
01. class Test
02. {
03.    char *buffer =
04. }
```

Pointers & Arrays

In C# array elements can be accessed by using pointer notations.

```
01. // Author: rajeshvs@msn.com
02. using System;
03. class MyClass
04. {
05.     public unsafe void Method()
06.     {
07.         int[] iArray = new int[10];
08.         for (int count = 0; count < 10; count++)
09.         {
10.             iArray[count] = count * count;
11.         }
12.         fixed (int* ptr = iArray)
13.             Display(ptr);
14.         //Console.WriteLine(*(ptr+2));
15.         //Console.WriteLine((int)ptr);
16.     }
17.     public unsafe void Display(int* pt)
```

```
    {
        for (int i = 0; i < 14; i++)
        {
            Console.WriteLine(*(pt + i));
        }
    }
}
class MyClient
{
    public static void Main()
    {
        MyClass mc = new MyClass();
        mc.Method();
    }
}
```

Pointers & Structures

The structures in C# are value types. The pointers can be used with structures if it contains only value types as its members. For example

```
// Author: rajeshvs@msn.com
using System;
struct MyStruct
{
    public int x;
    public int y;
    public void SetXY(int i, int j)
    {
        x = i;
        y = j;
    }
    public void ShowXY()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
class MyClient
{
    public unsafe static void Main()
    {
        MyStruct ms = new MyStruct();
        MyStruct* ms1 = &ms;
        ms1->SetXY(10, 20);
        ms1->ShowXY();
    }
}
```

Pointer | Pointer type | Pointers Conversions | Pointers Methods | Pointers Structures | Pointers in C# | Stack Allocation | Unsafe Codes

Next Recommended Reading

Use Of Pointers In C#

OUR BOOKS

Rajesh VS TOP 1000

Rajesh V.S is a software engineer in the area of C/C++/JAVA for the last 5 years. Currently he is interested in core C# language. He is Sun Certified Java Programmer. His other area of interest includes Design Patterns a... Read more

https://www.c-sharpcorner.com/members/rajesh-vs

519 **8.1m**

11 **10**

Type your comment here and press Enter Key (Minimum 10 characters)

Very good organization of content . and good explanation as well.

Ammar Shaukat

549 **4k** **921.3k**

Oct 24, 2018

1 0 Reply

Can we use unsafe keyword in class defination?

Shubham Jain

993 **1.5k** **832.3k**

Nov 03, 2017

2 0 Reply

Nice

Ramesh Palaniappan

219 **11k** **2.2m**

Aug 29, 2016

3 0 Reply

Thank you for sharing.

Joe Wilson

301 **7.9k** **380.8k**

Dec 30, 2015

3 0 Reply

Noman, You can always add an "Article Extension" to add your version to an article. Cheers!

Mahesh Chand

1 **343.2k** **216.8m**

Dec 06, 2012

3 0 Reply

you have mention at beginning of your article that " A pointer is nothing but a variable that holds the memory address of another type" is wrong because pointer is a memory address not a variable.Pointer Variable is a variable that holds the address of variable. Please Correct it. Thanks. From : Noman Ameer Khan

Noman Ameer

2125 **2** **0**

Nov 19, 2011

3 0 Reply

I have one question: why do you declare a pointer like this: int *ptr? one thing is int and another int*! it`s confusing when reading because *ptr refers the value of the variable to which the pointer ptr points to. it is not the same thing to write: int *ptr = &x; and *ptr = &x; someone who will learn from your article will get confused.

eonsimi

2126 **1** **0**

Apr 13, 2011

3 1 Reply

The reason we put int *p is simple. It comes about because we can declare multiple variables on the same line, such as int a, b, c; We can also do int a=1, b=2, c; To do pointers, we write int *p, *q, *r; If we were to write int *p, q, r; then only p would be a pointer. q and r would be simply integers. We can also do int *p=&x, *q=p, *r; Here, p points to the variable x (which you should assume is valid and came earlier in a code chunk), q also points to x, and r isn't initialized. So, the asterisk (*) is put right before the pointer name to keep straight exactly which of the variables being declared are pointers.

Gail Steitz

2126 **1** **0**

Jun 05, 2013

3

i got this error Unsafe code may only appear if compiling with /unsafe

Rahat

2125 **2** **0**

Mar 14, 2011

3 0 Reply

kalyan kamesh

2120 **7** **0**

May 29, 2009

3 0 Reply

FEATURED ARTICLES

Create An Android App With .NET MAUI And Visual Studio 2022

Sealed Classes - A Java 17 New Feature

CRUD Operation And Microservice Communication Using gRPC In .NET Core 6 Web API

How To Post Data To The Controller Using AJAX With Validations In ASP.NET Core

JWT Token Creation, Authentication And Authorization In ASP.NET Core 6.0 With Postman

TRENDING UP

01 Build A Blazor Hybrid App with .NET MAUI for Cross-Platform Application

02 Python Constants

03 Getting Started With .NET MAUI For Cross Platform Application Development

04 Asynchronous Communication Between Microservices Using .NET Core API, RabbitMQ, and Docker

05 What Is DMARC ? | Why DMARC Is Important ? | Understanding DMARC Records

06 SQL - Clone Tables😉

07 Create QR Code Using Google Charts API In VB.Net

08 gRPC Introduction And Implementation Using .NET Core 6

09 Implementation And Containerization Of Microservices Using .NET Core 6 And Docker

10 Index() Method In Python

Learn C# 8.0

CHALLENGE YOURSELF

C# Skill

GET CERTIFIED

Python Developer