Pointer related operators (C# reference)

Article • 01/25/2022 • 7 minutes to read

You can use the following operators to work with pointers:

- Unary & (address-of) operator: to get the address of a variable
- Unary * (pointer indirection) operator: to obtain the variable pointed by a pointer
- The -> (member access) and [] (element access) operators
- Arithmetic operators +, -, ++, and --
- Comparison operators ==, !=, <, >, <=, and >=

For information about pointer types, see Pointer types.

> **Note**
>
> Any operation with pointers requires an **unsafe** context. The code that contains unsafe blocks must be compiled with the **AllowUnsafeBlocks** compiler option.

Address-of operator &

The unary `&` operator returns the address of its operand:

C#

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

The operand of the `&` operator must be a fixed variable. *Fixed* variables are variables that reside in storage locations that are unaffected by operation of the garbage collector. In the preceding example, the local variable `number` is a fixed variable, because it resides on the stack. Variables that reside in storage locations that can be affected by the garbage

collector (for example, relocated) are called *movable* variables. Object fields and array elements are examples of movable variables. You can get the address of a movable variable if you "fix", or "pin", it with a fixed statement. The obtained address is valid only inside the block of a `fixed` statement. The following example shows how to use a `fixed` statement and the `&` operator:

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

You can't get the address of a constant or a value.

For more information about fixed and movable variables, see the Fixed and moveable variables section of the C# language specification.

The binary `&` operator computes the logical AND of its Boolean operands or the bitwise logical AND of its integral operands.

Pointer indirection operator *

The unary pointer indirection operator `*` obtains the variable to which its operand points. It's also known as the dereference operator. The operand of the `*` operator must be of a pointer type.

C#

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable:
{(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
```

```
    Console.WriteLine($"Value of the `letter` variable after update:
{letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

You cannot apply the `*` operator to an expression of type `void*`.

The binary `*` operator computes the product of its numeric operands.

Pointer member access operator ->

The `->` operator combines pointer indirection and member access. That is, if `x` is a pointer of type `T*` and `y` is an accessible member of type `T`, an expression of the form

C#

```
x->y
```

is equivalent to

C#

```
(*x).y
```

The following example demonstrates the usage of the `->` operator:

C#

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
```

```
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

You cannot apply the `->` operator to an expression of type `void*`.

Pointer element access operator []

For an expression `p` of a pointer type, a pointer element access of the form `p[n]` is evaluated as `*(p + n)`, where `n` must be of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`. For information about the behavior of the `+` operator with pointers, see the Addition or subtraction of an integral value to or from a pointer section.

The following example demonstrates how to access array elements with a pointer and the `[]` operator:

C#

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In the preceding example, a stackalloc expression allocates a block of memory on the stack.

> **Note**
>
> The pointer element access operator doesn't check for out-of-bounds errors.

You cannot use `[]` for pointer element access with an expression of type `void*`.

You can also use the `[]` operator for array element or indexer access.

Pointer arithmetic operators

You can perform the following arithmetic operations with pointers:

- Add or subtract an integral value to or from a pointer
- Subtract two pointers
- Increment or decrement a pointer

You cannot perform those operations with pointers of type `void*`.

For information about supported arithmetic operations with numeric types, see Arithmetic operators.

Addition or subtraction of an integral value to or from a pointer

For a pointer `p` of type `T*` and an expression `n` of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`, addition and subtraction are defined as follows:

- Both `p + n` and `n + p` expressions produce a pointer of type `T*` that results from adding `n * sizeof(T)` to the address given by `p`.
- The `p - n` expression produces a pointer of type `T*` that results from subtracting `n * sizeof(T)` from the address given by `p`.

The sizeof operator obtains the size of a type in bytes.

The following example demonstrates the usage of the `+` operator with a pointer:

C#

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };

    fixed (int* pointerToFirst = &numbers[0])
```

```
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address
{(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address
{(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Pointer subtraction

For two pointers `p1` and `p2` of type `T*`, the expression `p1 - p2` produces the difference between the addresses given by `p1` and `p2` divided by `sizeof(T)`. The type of the result is `long`. That is, `p1 - p2` is computed as `((long)(p1) - (long)(p2)) / sizeof(T)`.

The following example demonstrates the pointer subtraction:

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Pointer increment and decrement

The `++` increment operator adds 1 to its pointer operand. The `--` decrement operator subtracts 1 from its pointer operand.

Both operators are supported in two forms: postfix (`p++` and `p--`) and prefix (`++p` and `--p`). The result of `p++` and `p--` is the value of `p` *before* the operation. The result of `++p` and `--p` is the value of `p` *after* the operation.

The following example demonstrates the behavior of both postfix and prefix increment operators:

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Pointer comparison operators

You can use the `==`, `!=`, `<`, `>`, `<=`, and `>=` operators to compare operands of any pointer type, including `void*`. Those operators compare the addresses given by the two operands as if they were unsigned integers.

For information about the behavior of those operators for operands of other types, see the Equality operators and Comparison operators articles.

Operator precedence

The following list orders pointer related operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators and the `->` and `[]` operators
- Prefix increment `++x` and decrement `--x` operators and the `&` and `*` operators
- Additive `+` and `-` operators
- Comparison `<`, `>`, `<=`, and `>=` operators
- Equality `==` and `!=` operators

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence.

For the complete list of C# operators ordered by precedence level, see the Operator

precedence section of the C# operators article.

Operator overloadability

A user-defined type cannot overload the pointer related operators `&`, `*`, `->`, and `[]`.

C# language specification

For more information, see the following sections of the C# language specification:

- Fixed and moveable variables
- The address-of operator
- Pointer indirection
- Pointer member access
- Pointer element access
- Pointer arithmetic
- Pointer increment and decrement
- Pointer comparison

See also

- C# reference
- C# operators and expressions
- Pointer types
- unsafe keyword
- fixed keyword
- stackalloc
- sizeof operator