Contents

Unsafe Code and Pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To maintain type safety and security, C# does not support pointer arithmetic, by default. However, by using the unsafe keyword, you can define an unsafe context in which pointers can be used. For more information about pointers, see the topic Pointer types.

> **NOTE**
> In the common language runtime (CLR), unsafe code is referred to as unverifiable code. Unsafe code in C# is not necessarily dangerous; it is just code whose safety cannot be verified by the CLR. The CLR will therefore only execute unsafe code if it is in a fully trusted assembly. If you use unsafe code, it is your responsibility to ensure that your code does not introduce security risks or pointer errors.

Unsafe Code Overview

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- In order for C# to compile unsafe code, the application must be compiled with /unsafe.

Related Sections

For more information, see:

- Pointer types
- Fixed Size Buffers
- How to: Use Pointers to Copy an Array of Bytes
- unsafe

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Fixed Size Buffers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C#, you can use the fixed statement to create a buffer with a fixed size array in a data structure. Fixed size buffers are useful when you write methods that interop with data sources from other languages or platforms. The fixed array can take any attributes or modifiers that are allowed for regular struct members. The only restriction is that the array type must be `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`.

```
private fixed char name[30];
```

Remarks

In safe code, a C# struct that contains an array does not contain the array elements. Instead, the struct contains a reference to the elements. You can embed an array of fixed size in a struct when it is used in an unsafe code block.

The following `struct` is 8 bytes in size. The `pathName` array is a reference:

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

A `struct` can contain an embedded array in unsafe code. In the following example, the `fixedBuffer` array has a fixed size. You use a `fixed` statement to establish a pointer to the first element. You access the elements of the array through this pointer. The `fixed` statement pins the `fixedBuffer` instance field to a specific location in memory.

```
internal unsafe struct MyBuffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class MyClass
{
    public MyBuffer myBuffer = default;
}

private static void AccessEmbeddedArray()
{
    MyClass myC = new MyClass();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = myC.myBuffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = myC.myBuffer.fixedBuffer[0];
        Console.WriteLine(c);
        // modify through the index:
        myC.myBuffer.fixedBuffer[0] = 'B';
        Console.WriteLine(myC.myBuffer.fixedBuffer[0]);
    }

}
```

The size of the 128 element `char` array is 256 bytes. Fixed size char buffers always take two bytes per character, regardless of the encoding. This is true even when char buffers are marshaled to API methods or structs with `CharSet = CharSet.Auto` or `CharSet = CharSet.Ansi`. For more information, see CharSet.

The preceding example demonstrates accessing `fixed` fields without pinning, which is available starting with C# 7.3.

Another common fixed-size array is the bool array. The elements in a `bool` array are always one byte in size. `bool` arrays are not appropriate for creating bit arrays or buffers.

> **NOTE**
>
> Except for memory created by using stackalloc, the C# compiler and the common language runtime (CLR) do not perform any security buffer overrun checks. As with all unsafe code, use caution.

Unsafe buffers differ from regular arrays in the following ways:

- You can only use unsafe buffers in an unsafe context.
- Unsafe buffers are always vectors, or one-dimensional arrays.
- The declaration of the array should include a count, such as `char id[8]`. You cannot use `char id[]`.
- Unsafe buffers can only be instance fields of structs in an unsafe context.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- fixed Statement
- Interoperability

Pointer types (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

In an unsafe context, a type may be a pointer type, a value type, or a reference type. A pointer type declaration takes one of the following forms:

```
type* identifier;
void* identifier; //allowed but not recommended
```

The type specified before the `*` in a pointer type is called the **referent type**. Any of the following types may be a referent type:

- Any integral type: sbyte, byte, short, ushort, int, uint, long, ulong.
- Any floating-point type: float, double.
- char.
- bool.
- decimal.
- Any enum type.
- Any pointer type. This allows expressions such as `void**`.
- Any user-defined struct type that contains fields of unmanaged types only.

Pointer types do not inherit from object and no conversions exist between pointer types and `object`. Also, boxing and unboxing do not support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, the asterisk (*) is written together with the underlying type only; it is not used as a prefix to each pointer name. For example:

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

A pointer cannot point to a reference or to a struct that contains references, because an object reference can be garbage collected even if a pointer is pointing to it. The garbage collector does not keep track of whether an object is being pointed to by any pointer types.

The value of the pointer variable of type `myType*` is the address of a variable of type `myType`. The following are examples of pointer type declarations:

EXAMPLE	DESCRIPTION
`int* p`	`p` is a pointer to an integer.
`int** p`	`p` is a pointer to a pointer to an integer.
`int*[] p`	`p` is a single-dimensional array of pointers to integers.
`char* p`	`p` is a pointer to a char.

EXAMPLE	DESCRIPTION
`void* p`	`p` is a pointer to an unknown type.

The pointer indirection operator * can be used to access the contents at the location pointed to by the pointer variable. For example, consider the following declaration:

```
int* myVariable;
```

The expression `*myVariable` denotes the `int` variable found at the address contained in `myVariable`.

There are several examples of pointers in the topics fixed Statement and Pointer Conversions. The following example uses the `unsafe` keyword and the `fixed` statement, and shows how to increment an interior pointer. You can paste this code into the Main function of a console application to run it. These examples must be compiled with the -unsafe compiler option set.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

You cannot apply the indirection operator to a pointer of type `void*`. However, you can use a cast to convert a void pointer to any other pointer type, and vice versa.

A pointer can be `null`. Applying the indirection operator to a null pointer causes an implementation-defined behavior.

Passing pointers between methods can cause undefined behavior. Consider a method that returns a pointer to a local variable through an `in`, `out`, or `ref` parameter or as the function result. If the pointer was set in a fixed block, the variable to which it points may no longer be fixed.

The following table lists the operators and statements that can operate on pointers in an unsafe context:

OPERATOR/STATEMENT	USE
*	Performs pointer indirection.
->	Accesses a member of a struct through a pointer.
[]	Indexes a pointer.
`&`	Obtains the address of a variable.
++ and --	Increments and decrements pointers.
+ and -	Performs pointer arithmetic.
==, !=, <, >, <=, and >=	Compares pointers.
`stackalloc`	Allocates memory on the stack.
`fixed` statement	Temporarily fixes a variable so that its address may be found.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- Pointer Conversions
- Pointer Expressions
- Types
- unsafe
- fixed Statement
- stackalloc
- Boxing and Unboxing

Pointer Conversions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following table shows the predefined implicit pointer conversions. Implicit conversions might occur in many situations, including method invoking and assignment statements.

Implicit pointer conversions

FROM	TO
Any pointer type	void*
null	Any pointer type

Explicit pointer conversion is used to perform conversions, for which there is no implicit conversion, by using a cast expression. The following table shows these conversions.

Explicit pointer conversions

FROM	TO
Any pointer type	Any other pointer type
sbyte, byte, short, ushort, int, uint, long, or ulong	Any pointer type
Any pointer type	sbyte, byte, short, ushort, int, uint, long, or ulong

Example

In the following example, a pointer to `int` is converted to a pointer to `byte`. Notice that the pointer points to the lowest addressed byte of the variable. When you successively increment the result, up to the size of `int` (4 bytes), you can display the remaining bytes of the variable.

```
// compile with: -unsafe
```

```
class ClassConvert
{
    static void Main()
    {
        int number = 1024;

        unsafe
        {
            // Convert to byte:
            byte* p = (byte*)&number;

            System.Console.Write("The 4 bytes of the integer:");

            // Display the 4 bytes of the int variable:
            for (int i = 0 ; i < sizeof(int) ; ++i)
            {
                System.Console.Write(" {0:X2}", *p);
                // Increment the pointer:
                p++;
            }
            System.Console.WriteLine();
            System.Console.WriteLine("The value of the integer: {0}", number);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Pointer Expressions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In this section, the following pointer expressions are discussed:

Obtaining the Value of a Variable

Obtaining the Address of a Variable

How to: Access a Member with a Pointer

How to: Access an Array Element with a Pointer

Manipulating Pointers

See also

- C# Programming Guide
- Pointer Conversions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: Obtain the Value of a Pointer Variable (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Use the pointer indirection operator to obtain the variable at the location pointed to by a pointer. The expression takes the following form, where `p` is a pointer type:

```
*p;
```

You cannot use the unary indirection operator on an expression of any type other than the pointer type. Also, you cannot apply it to a void pointer.

When you apply the indirection operator to a null pointer, the result depends on the implementation.

Example

In the following example, a variable of the type `char` is accessed by using pointers of different types. Note that the address of `theChar` will vary from run to run, because the physical address allocated to a variable can change.

```
// compile with: -unsafe
```

```
unsafe class TestClass
{
    static void Main()
    {
        char theChar = 'Z';
        char* pChar = &theChar;
        void* pVoid = pChar;
        int* pInt = (int*)pVoid;

        System.Console.WriteLine("Value of theChar = {0}", theChar);
        System.Console.WriteLine("Address of theChar = {0:X2}",(int)pChar);
        System.Console.WriteLine("Value of pChar = {0}", *pChar);
        System.Console.WriteLine("Value of pInt = {0}", *pInt);
    }
}
```

Value of theChar = Z Address of theChar = 12F718 Value of pChar = Z Value of pInt = 90

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: obtain the address of a variable (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To obtain the address of a unary expression, which evaluates to a fixed variable, use the address-of operator `&`:

```
int number;
int* p = &number; //address-of operator &
```

The address-of operator can only be applied to a variable. If the variable is a moveable variable, you can use the fixed statement to temporarily fix the variable before obtaining its address.

It's your responsibility to ensure that the variable is initialized. The compiler doesn't issue an error message if the variable is not initialized.

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

Example

In this example, a pointer to `int`, `p`, is declared and assigned the address of an integer variable, `number`. The variable `number` is initialized as a result of the assignment to `*p`. If you comment out this assignment statement, the initialization of the variable `number` is removed, but no compile-time error is issued.

> **NOTE**
> Compile this example with the `-unsafe` compiler option.

```
class AddressOfOperator
{
    static void Main()
    {
        int number;

        unsafe
        {
            // Assign the address of number to a pointer:
            int* p = &number;

            // Commenting the following statement will remove the
            // initialization of number.
            *p = 0xffff;

            // Print the value of *p:
            System.Console.WriteLine("Value at the location pointed to by p: {0:X}", *p);

            // Print the address stored in p:
            System.Console.WriteLine("The address stored in p: {0}", (int)p);
        }

        // Print the value of the variable number:
        System.Console.WriteLine("Value of the variable number: {0:X}", number);

        System.Console.ReadKey();
    }
}
/* Output:
        Value at the location pointed to by p: FFFF
        The address stored in p: 2420904
        Value of the variable number: FFFF
*/
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: access a member with a pointer (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To access a member of a struct that is declared in an unsafe context, you can use the member access operator as shown in the following example in which `p` is a pointer to a struct that contains a member `x`.

```
CoOrds* p = &home;
p -> x = 25; //member access operator ->
```

Example

In this example, a struct, `CoOrds`, that contains the two coordinates `x` and `y` is declared and instantiated. By using the member access operator `->` and a pointer to the instance `home`, `x` and `y` are assigned values.

> **NOTE**
> Notice that the expression `p->x` is equivalent to the expression `(*p).x`, and you can obtain the same result by using either of the two expressions.

```
// compile with: -unsafe
```

```
struct CoOrds
{
    public int x;
    public int y;
}

class AccessMembers
{
    static void Main()
    {
        CoOrds home;

        unsafe
        {
            CoOrds* p = &home;
            p->x = 25;
            p->y = 12;

            System.Console.WriteLine("The coordinates are: x={0}, y={1}", p->x, p->y );
        }
    }
}
```

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types

- Types
- unsafe
- fixed Statement
- stackalloc

How to: access an array element with a pointer (C# Programming Guide)

1/15/2019 • 2 minutes to read • Edit Online

In an unsafe context, you can access an element in memory by using pointer element access, as shown in the following example:

```
char* charPointer = stackalloc char[123];
for (int i = 65; i < 123; i++)
{
    charPointer[i] = (char)i; //access array elements
}
```

The expression in square brackets must be implicitly convertible to `int`, `uint`, `long`, or `ulong`. The operation `p[e]` is equivalent to `*(p+e)`. Like C and C++, the pointer element access does not check for out-of-bounds errors.

Example

In this example, 123 memory locations are allocated to a character array, `charPointer`. The array is used to display the lowercase letters and the uppercase letters in two for loops.

Notice that the expression `charPointer[i]` is equivalent to the expression `*(charPointer + i)`, and you can obtain the same result by using either of the two expressions.

```
// compile with: -unsafe
```

```
class Pointers
{
    unsafe static void Main()
    {
        char* charPointer = stackalloc char[123];

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

        // Print uppercase letters:
        System.Console.WriteLine("Uppercase letters:");
        for (int i = 65; i < 91; i++)
        {
            System.Console.Write(charPointer[i]);
        }
        System.Console.WriteLine();

        // Print lowercase letters:
        System.Console.WriteLine("Lowercase letters:");
        for (int i = 97; i < 123; i++)
        {
            System.Console.Write(charPointer[i]);
        }
    }
}
```

Uppercase letters:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Lowercase letters:
abcdefghijklmnopqrstuvwxyz

See also

- C# Programming Guide
- Pointer Expressions
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Manipulating Pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This section includes the following pointer operations:

Increment and Decrement

Arithmetic Operations

Comparison

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Pointer types
- /unsafe (C# Compiler Options)

How to: increment and decrement pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Use the increment and the decrement operators, `++` and `--`, to change the pointer location by `sizeof(pointer-type)` for a pointer of the type `pointer-type*`. The increment and decrement expressions take the following form:

```
++p;
p++;
--p;
p--;
```

The increment and decrement operators can be applied to pointers of any type except the type `void*`.

The effect of applying the increment operator to a pointer of the type `pointer-type*` is to add `sizeof(pointer-type)` to the address that is contained in the pointer variable.

The effect of applying the decrement operator to a pointer of the type `pointer-type*` is to subtract `sizeof(pointer-type)` from the address that is contained in the pointer variable.

No exceptions are generated when the operation overflows the domain of the pointer, and the result depends on the implementation.

Example

In this example, you step through an array by incrementing the pointer by the size of `int`. With each step, you display the address and the content of the array element.

```
// compile with: -unsafe
```

```
class IncrDecr
{
    unsafe static void Main()
    {
        int[] numbers = {0,1,2,3,4};

        // Assign the array address to the pointer:
        fixed (int* p1 = numbers)
        {
            // Step through the array elements:
            for(int* p2=p1; p2<p1+numbers.Length; p2++)
            {
                System.Console.WriteLine("Value:{0} @ Address:{1}", *p2, (long)p2);
            }
        }
    }
}
```

Value:0 @ Address:12860272 Value:1 @ Address:12860276 Value:2 @ Address:12860280 Value:3 @ Address:12860284 Value:4 @ Address:12860288

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc
- sizeof

Arithmetic operations on pointers (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This topic discusses using the arithmetic operators `+` and `-` to manipulate pointers.

> **NOTE**
> You cannot perform any arithmetic operations on void pointers.

Adding and subtracting numeric values to or from pointers

You can add a value `n` of type int, uint, long, or ulong to a pointer. If `p` is a pointer of the type `pointer-type*`, the result `p+n` is the pointer resulting from adding `n * sizeof(pointer-type)` to the address of `p`. Similarly, `p-n` is the pointer resulting from subtracting `n * sizeof(pointer-type)` from the address of `p`.

Subtracting pointers

You can also subtract pointers of the same type. The result is always of the type `long`. For example, if `p1` and `p2` are pointers of the type `pointer-type*`, then the expression `p1-p2` results in:

```
((long)p1 - (long)p2)/sizeof(pointer-type)
```

No exceptions are generated when the arithmetic operation overflows the domain of the pointer, and the result depends on the implementation.

Example

```
// compile with: -unsafe
```

```
class PointerArithmetic
{
    unsafe static void Main()
    {
        int* memory = stackalloc int[30];
        long difference;
        int* p1 = &memory[4];
        int* p2 = &memory[10];

        difference = p2 - p1;

        System.Console.WriteLine("The difference is: {0}", difference);
    }
}
// Output:  The difference is: 6
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Unsafe Code and Pointers
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

Pointer Comparison (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can apply the following operators to compare pointers of any type:

== != < > <= >=

The comparison operators compare the addresses of the two operands as if they are unsigned integers.

Example

```
// compile with: -unsafe
```

```
class CompareOperators
{
    unsafe static void Main()
    {
        int x = 234;
        int y = 236;
        int* p1 = &x;
        int* p2 = &y;

        System.Console.WriteLine(p1 < p2);
        System.Console.WriteLine(p2 < p1);
    }
}
```

Sample Output

```
True
```

```
False
```

See also

- C# Programming Guide
- Pointer Expressions
- C# Operators
- Manipulating Pointers
- Pointer types
- Types
- unsafe
- fixed Statement
- stackalloc

How to: Use Pointers to Copy an Array of Bytes (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following example uses pointers to copy bytes from one array to another.

This example uses the unsafe keyword, which enables you to use pointers in the `Copy` method. The fixed statement is used to declare pointers to the source and destination arrays. The `fixed` statement *pins* the location of the source and destination arrays in memory so that they will not be moved by garbage collection. The memory blocks for the arrays are unpinned when the `fixed` block is completed. Because the `Copy` method in this example uses the `unsafe` keyword, it must be compiled with the -unsafe compiler option.

This example accesses the elements of both arrays using indices rather than a second unmanaged pointer. The declaration of the `pSource` and `pTarget` pointers pins the arrays. This feature is available starting with C# 7.3.

Example

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
```

```
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

See also

- C# Programming Guide
- Unsafe Code and Pointers
- -unsafe (C# Compiler Options)
- Garbage Collection