

Writing zero-allocation code with C#

Sergey Nazarov

About me



Sergey Nazarov

I'm software engineer with over more 6 years experience in dotnet development.

Contact me:

- @sanazarov at telegram
- me@nazarovsa.com
- nazarovsa at github.com

1. Main reasons of memory issues in our algorithms

Memory issues in our algorithms

Main reasons why we have issues with memory:

- Incorrect memory allocation. (Create objects in loops; using Linq, when it's not optimal; etc.)
- Weak knowledge about how the .net platform operates. (String is immutable, value types are copied on pass, etc.)
- Weak knowledge about language features.

2. Our instruments for optimization

Structures and their vagaries

As we know, ***struct*** is a value type. One advantage to using value types is that they often avoid heap allocations. The disadvantage is that they're copied by value. This trade-off makes it harder to optimize algorithms that operate on large amounts of data. The language features highlighted in this section provide mechanisms that enable safe efficient code using references to value types. Use these features wisely to minimize both allocations and copy operations.

The section also explains some low-level optimizations that are advisable when you've run a profiler and have identified bottlenecks:

- Use ***ref readonly return*** statements.
- Use the ***in*** parameter modifier.
- Use ***ref struct*** types.

These techniques balance two competing goals:

- **Minimize allocations on the heap:** variables that are reference types hold a reference to a location in memory and are allocated on the managed heap. Only the reference is copied when a reference type is passed as an argument to a method or returned from a method. Each new object requires a new allocation, and later must be reclaimed. Garbage collection takes time.
- **Minimize the copying of values:** variables that are value types directly contain their value, and the value is typically copied when passed to a method or returned from a method. This behavior includes copying the value of this when calling iterators and async instance methods of structs. The copy operation takes time, depending on the size of the type.

Use ref readonly return statements

Use a ***ref readonly return*** when both of the following conditions are true:

- The return value is a ***struct*** larger than ***IntPtr.Size***.
- The storage lifetime is greater than the method returning the value.

You can return values by reference when the value being returned isn't local to the returning method. Returning by reference means that only the reference is copied, not the structure.

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

In the following example, the Origin property can't use a ref return because the value being returned is a local variable:

```
public Point3D Origin => new Point3D(0,0,0);
```

Use ref readonly return statements

However, the following property definition can be returned by reference because the returned value is a static member:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

You don't want callers modifying the origin, so you should return the value by ref readonly:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```


Use ref readonly return statements

Returning ***ref readonly*** enables you to save copying larger structures and preserve the immutability of your internal data members. At the call site, callers make the choice to use the Origin property as a ***ref readonly*** or as a value:

```
var originValue = Point3D.Origin;  
ref readonly var originReference = ref Point3D.Origin;
```

The first assignment in the preceding code makes a copy of the Origin constant and assigns that copy. The second assigns a reference. Notice that the readonly modifier must be part of the declaration of the variable. The reference to which it refers can't be modified. Attempts to do so result in a compile-time error.

The readonly modifier is required on the declaration of ***originReference***.

The compiler enforces that the caller can't modify the reference. Attempts to assign the value directly generate a compile-time error. In other cases, the compiler allocates a defensive copy unless it can safely use the readonly reference. Static analysis rules determine if the struct could be modified. The compiler doesn't create a defensive copy when the struct is a ***readonly struct*** or the member is a ***readonly*** member of the struct. Defensive copies aren't needed to pass the struct as an ***in*** argument.

Declare immutable structs as readonly

Declare a ***readonly struct*** to indicate that a type is immutable. The readonly modifier informs the compiler that your intent is to create an immutable type.

The compiler enforces that design decision with the following rules:

- All field members must be read-only.
- All properties must be read-only, including auto-implemented properties.

These two rules are sufficient to ensure that no member of a ***readonly struct*** modifies the state of that struct. The struct is immutable. The Point3D structure could be defined as an immutable struct as shown in the following example:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Follow this recommendation whenever your design intent is to create an immutable value type. Any performance improvements are an added benefit. The ***readonly struct*** keywords clearly express your design intent.

Use the **IN** parameter modifier

The ***in*** keyword complements the ***ref*** and ***out*** keywords to pass arguments by reference. The ***in*** keyword specifies that the argument is passed by reference, but the called method doesn't modify the value. The ***in*** modifier can be applied to any member that takes parameters, such as methods, delegates, lambdas, local functions, indexers, and operators.

With the addition of the ***in*** keyword, C# provides a full vocabulary to express your design intent. Value types are copied when passed to a called method when you don't specify any of the following modifiers in the method signature. Each of these modifiers specifies that a variable is passed by reference, avoiding the copy.

Each modifier expresses a different intent:

- **out**: This method sets the value of the argument used as this parameter.
- **ref**: This method may modify the value of the argument used as this parameter.
- **in**: This method doesn't modify the value of the argument used as this parameter.

Use the IN parameter modifier: use in parameters for large structs

You can apply the ***in*** modifier to any ***readonly struct*** parameter, but this practice is likely to improve performance only for value types that are substantially larger than ***IntPtr.Size***. For simple types (such as ***sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal*** and ***bool***, and ***enum*** types), any potential performance gains are minimal. Some simple types, such as ***decimal*** at 16 bytes in size, are larger than either 4-byte or 8-byte references but not by enough to make a measurable difference in performance in most scenarios. And performance may degrade by using pass-by-reference for types smaller than ***IntPtr.Size***.

The following code shows an example of a method that calculates the distance between two points in 3D space:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(
        xDifference * xDifference +
        yDifference * yDifference +
        zDifference * zDifference);
}
```

The arguments are two structures that each contain three doubles. A double is 8 bytes, so each argument is 24 bytes. By specifying the ***in*** modifier, you pass a 4-byte or 8-byte reference to those arguments, depending on the architecture of the machine. The difference in size is small, but it can add up when your application calls this method in a tight loop using many different values.

Use the IN parameter modifier: optional use of in at call site

Unlike a ***ref*** or ***out*** parameter, you don't need to apply the ***in*** modifier at the call site. The following code shows two examples of calling the ***CalculateDistance*** method. The first uses two local variables passed by reference. The second includes a temporary variable created as part of the method call

```
var distance = CalculateDistance(pt1, pt2);  
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

Omitting the ***in*** modifier at the call site informs the compiler that it's allowed to make a copy of the argument for any of the following reasons:

- There exists an implicit conversion but not an identity conversion from the argument type to the parameter type.
- The argument is an expression but doesn't have a known storage variable.
- An overload exists that differs by the presence or absence of ***in***. In that case, the by value overload is a better match.

Use the IN parameter modifier: avoid defensive copies

Pass a **struct** as the argument for an **in** parameter only if it's declared with the **readonly** modifier or the method accesses only **readonly** members of the struct. Otherwise, the compiler must create *defensive copies* in many situations to ensure that arguments are not mutated. Consider the following example that calculates the distance of a 3D point from the origin:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference +
        yDifference * yDifference +
        zDifference * zDifference);
}
```

The **Point3D** structure is not a read-only struct. There are six different property access calls in the body of this method. On first examination, you may think these accesses are safe. After all, a **get** accessor shouldn't modify the state of the object. But there's no language rule that enforces that. It's only a common convention. Any type could implement a **get** accessor that modified the internal state.

Without some language guarantee, the compiler must create a temporary copy of the argument before calling any member not marked with the **readonly** modifier. The temporary storage is created on the stack, the values of the argument are copied to the temporary storage, and the value is copied to the stack for each member access as the **this** argument. In many situations, these copies harm performance enough that pass-by-value is faster than pass-by-read-only-reference when the argument type isn't a **readonly struct** and the method calls members that aren't marked **readonly**. If you mark all methods that don't modify the struct state as **readonly**, the compiler can safely determine that the struct state isn't modified, and a defensive copy is not needed.

Use the IN parameter modifier: avoid defensive copies

If the distance calculation uses the immutable struct, ***ReadOnlyPoint3D***, temporary objects aren't needed:

```
private static double CalculateDistance3(  
    in ReadOnlyPoint3D point1,  
    in ReadOnlyPoint3D point2 = default)  
{  
    double xDifference = point1.X - point2.X;  
    double yDifference = point1.Y - point2.Y;  
    double zDifference = point1.Z - point2.Z;  
  
    return Math.Sqrt(xDifference * xDifference +  
        yDifference * yDifference +  
        zDifference * zDifference);  
}
```

The compiler generates more efficient code when you call members of a ***readonly struct***. The ***this*** reference, instead of a copy of the receiver, is always an ***in*** parameter passed by reference to the member method. This optimization saves copying when you use a ***readonly struct*** as an ***in*** argument.

Don't pass a nullable value type as an ***in*** argument. The ***Nullable<T>*** type isn't declared as a read-only struct. That means the compiler must generate defensive copies for any nullable value type argument passed to a method using the ***in*** modifier on the parameter declaration.



Usage of ref struct types

Use a ***ref struct*** or a ***readonly ref struct***, such as ***Span<T>*** or ***ReadOnlySpan<T>***, to work with blocks of memory as a sequence of bytes. The memory used by the span is constrained to a single stack frame. This restriction enables the compiler to make several optimizations. The primary motivation for this feature was ***Span<T>*** and related structures. You'll achieve performance improvements from these enhancements by using new and updated .NET APIs that make use of the ***Span<T>*** type.

ref struct types have a number of restrictions to ensure that they cannot be promoted to the managed heap:

- They can't be boxed.
- They can't be assigned to variables of type ***Object***, ***dynamic*** or to any ***interface*** type.
- They can't be fields in a reference type.
- They can't be used across ***await*** and ***yield*** boundaries.

In addition, calls to two methods, ***Equals(Object)*** and ***GetHashCode***, throw a ***NotSupportedException***.

Struct section conclusions

Using value types minimizes the number of allocation operations:

- Storage for value types is stack-allocated for local variables and method arguments.
- Storage for value types that are members of other objects is allocated as part of that object, not as a separate allocation.
- Storage for value type return values is stack allocated.

Contrast that with reference types in those same situations:

- Storage for reference types is heap allocated for local variables and method arguments. The reference is stored on the stack.
- Storage for reference types that are members of other objects are separately allocated on the heap. The containing object stores the reference.
- Storage for reference type return values is heap allocated. The reference to that storage is stored on the stack.

Span<T> и ReadOnlySpan<T>

Span<T>

Provides a type- and memory-safe representation of a contiguous region of arbitrary memory.

Span<T> is a ***ref struct*** that is allocated on the stack rather than on the managed heap.

A ***Span<T>*** instance is often used to hold the elements of an array or a portion of an array.

Unlike an array, however, a ***Span<T>*** instance can point to managed memory, native memory, or memory managed on the stack.

ReadOnlySpan<T>

Provides a type-safe and memory-safe read-only representation of a contiguous region of arbitrary memory.

Span<T> and arrays

When it wraps an array, ***Span<T>*** can wrap an entire array or any contiguous range within the array. The following example creates a slice of the middle five elements of a 10-element integer array. Note that the code doubles the values of each integer in the slice. As the output shows, the changes made by the span are reflected in the values of the array.

```
using System;

var array = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
var slice = new Span<int>(array, 2, 5);
for (int ctr = 0; ctr < slice.Length; ctr++)
    slice[ctr] *= 2;

// Examine the original array values.
foreach (var value in array)
    Console.Write($"{value} ");
Console.WriteLine();

// The example displays the following output:
//      2  4 12 16 20 24 28 16 18 20
```


Span<T> and slices

Span<T> includes two overloads of the **Slice** method that form a slice out of the current span that starts at a specified index. This makes it possible to treat the data in a **Span<T>** as a set of logical chunks that can be processed as needed by portions of a data processing pipeline with minimal performance impact.

For example, since modern server protocols are often text-based, manipulation of strings and substrings is particularly important. In the **String** class, the major method for extracting substrings is **Substring**. For data pipelines that rely on extensive string manipulation, its use offers some performance penalties, since it:

- Creates a new string to hold the substring.
- Copies a subset of the characters from the original string to the new string. This allocation and copy operation can be eliminated by using either **Span<T>** or **ReadOnlySpan<T>**, as the following example shows:

```
using System;

class Program
{
    static void Main()
    {
        string contentLength = "Content-Length: 132";
        var length = GetContentLength(contentLength.ToCharArray());
        Console.WriteLine($"Content length: <length>");
    }

    private static int GetContentLength(ReadOnlySpan<char> span)
    {
        var slice = span.Slice(16);
        return int.Parse(slice);
    }
}

// Output:
//      Content length: 132
```


stackalloc

A ***stackalloc*** expression allocates a block of memory on the stack. A stack allocated memory block created during the method execution is automatically discarded when that method returns. You cannot explicitly free the memory allocated with ***stackalloc***. A stack allocated memory block is not subject to garbage collection and doesn't have to be pinned with a fixed statement. You can assign the result of a ***stackalloc*** expression to a variable of one of the following types:

- Beginning with C# 7.2, ***System.Span<T>*** or ***System.ReadOnlySpan<T>***
- A pointer type

The amount of memory available on the stack is limited. If you allocate too much memory on the stack, a ***StackOverflowException*** is thrown. To avoid that, follow the rules below:

- Limit the amount of memory you allocate with ***stackalloc***. For example, if the intended buffer size is below a certain limit, you allocate the memory on the stack; otherwise, use an array of the required length, as the following code shows:

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit
    ? stackalloc byte[MaxStackLimit]
    : new byte[inputLength];
```

- Avoid using ***stackalloc*** inside loops. Allocate the memory block outside a loop and reuse it inside the loop.
- The content of the newly allocated memory is undefined.** You should initialize it before the use. For example, you can use the ***Span<T>.Clear*** method that sets all the items to the default value of type T.

Memory<T> и ReadOnlyMemory<T>

Memory<T>

Represents a contiguous region of memory.

Like **Span<T>**, **Memory<T>** represents a contiguous region of memory. Unlike **Span<T>**, however, **Memory<T>** is not a ref struct. This means that **Memory<T>** can be placed on the managed heap, whereas **Span<T>** cannot. As a result, the **Memory<T>** structure does not have the same restrictions as a **Span<T>** instance. In particular:

- It can be used as a field in a class.
- It can be used across **await** and **yield** boundaries.

In addition to **Memory<T>**, you can use **System.ReadOnlyMemory<T>** to represent immutable or read-only memory.

ReadOnlyMemory<T>

Represents a contiguous region of memory, similar to **ReadOnlySpan<T>**. Unlike **ReadOnlySpan<T>**, it is not a byref-like type.

You can rent memory via **MemoryPool** class using **MemoryPool<T>.Shared.Rent(length)**.

IMemoryOwner<T>

Identifies the owner of a block of memory who is responsible for disposing of the underlying memory appropriately.

IMemoryOwner<T> can be used to manage rented memory in a right manner. These cases are occurring when you need to pass a memory block through your calls.

The **IMemoryOwner<T>** interface is used to define the owner responsible for the lifetime management of a **Memory<T>** buffer. An instance of the **IMemoryOwner<T>** interface is returned by the **MemoryPool<T>.Rent** method. While a buffer can have multiple consumers, it can only have a single owner at any given time. The owner can: Create the buffer either directly or by calling a factory method. Transfer ownership to another consumer. In this case, the previous owner should no longer use the buffer. Destroy the buffer when it is no longer in use. Because the **IMemoryOwner<T>** object implements the **IDisposable** interface, you should call its **Dispose** method only after the memory buffer is no longer needed and you have destroyed it. You should not dispose of the **IMemoryOwner<T>** object while a reference to its memory is available. This means that the type in which **IMemoryOwner<T>** is declared should not have a **Finalize** method.

Object pooling

The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object to the pool rather than destroying it; this can be done manually or automatically.

Object pools are primarily used for performance: in some circumstances, object pools significantly improve performance. Object pools complicate object lifetime, as objects obtained from and returned to a pool are not actually created or destroyed at this time, and thus require care in implementation.

Obviously, if we are reusing objects, it allow us to save memory and prevent GC collections. We don't need to create and destroy extra objects anymore, because we taking them from the pool.

Array pooling

Array pooling is similar to object pooling, but you use the pool of arrays instead. C# allows us to use that mechanism via ***ArrayPool*** class.


Use arrays from the pool when you need an array of the same length and can't allocate it on the stack. For example, instead of creating a new array at each method call, use the array pool.

Use ***ArrayPool<T>.Shared.Rent(length)*** to rent array from the pool.

Use ***ArrayPool<T>.Shared.Return(array, clearArray)*** to return array into the pool.

```
public string DoSomeWork(some args)
{
    var array = new double[2048];
    // Do some work
}
```

Without pooling



```
public string DoSomeWork(some args)
{
    var array = ArrayPool<double>.Shared.Rent(2048);
    try
    {
        // Do some work
    }
    finally
    {
        ArrayPool<double>.Shared.Return(array);
    }
}
```

With pooling

3. Practice

Guid transformer

Sample of optimizing simple guid to efficient string transformer.

The efficient string is a string that represents a guid in user-friendly format. For example:

1. Convert GUID to Base64.
2. Replace '/' with '-' (To make it works in url string)
3. Replace '+' with '_' (To make it works in url string)
4. Remove the tailing '=' characters (Because the length of our guid is always constant and we can return it while transforming from string to guid)



Source code

Guid transformer benchmarks

Method	Mean	Error	StdDev	Allocated
ToGuidFromString	101.63 ns	0.244 ns	0.190 ns	184 B
ToGuidFromStringEfficient	48.69 ns	1.087 ns	0.908 ns	-
ToStringFromGuid	117.85 ns	1.724 ns	1.528 ns	256 B
ToStringFromGuidEfficient	40.45 ns	0.195 ns	0.152 ns	72 B

Benchmark results

Abstract lottery ticket's combination generator

Consider a lottery where you should guess the order of ten numbers with predefined tickets. We will generate ticket combinations for that lottery.



Source code

Abstract lottery ticket's combination generator benchmarks

Method	Count	Mean	Error	StdDev	Gen 0	Allocated
GenerateCombinations	1	558.5 ns	1.96 ns	1.83 ns	0.2251	472 B
GenerateCombinationsEfficient	1	141.1 ns	0.25 ns	0.24 ns	-	-
GenerateCombinations	10000	5,357,173.9 ns	45,280.68 ns	40,140.15 ns	2250.0000	4,720,009 B
GenerateCombinationsEfficient	10000	1,412,395.9 ns	2,739.52 ns	2,287.63 ns	-	2 B
GenerateCombinations	100000	62,100,165.6 ns	204,302.19 ns	191,104.39 ns	22500.0000	47,201,427 B
GenerateCombinationsEfficient	100000	16,507,169.5 ns	27,985.89 ns	26,178.02 ns	-	36 B

Benchmark results



Source code

Object pools

Imagine a process where we need to do some work in parallel. In this example, we have a large object that we need for calculations. This example is synthetic, but it is enough to show the difference in memory.

```
public class ExampleObject
{
    public int[] Nums { get; set; }

    public ExampleObject()
    {
        Nums = new int[1000000];
        var rand = new Random();
        for (int i = 0; i < Nums.Length; i++)
        {
            Nums[i] = rand.Next();
        }
    }

    public double GetValue(long i) => Math.Sqrt(Nums[i]);
}
```



Source code

Example large object

Method	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
ProcessExample	132.03 ms	2.880 ms	8.446 ms	5500.0000	5250.0000	5250.0000	382 MB
ProcessExampleMicrosoftObjectPool	15.26 ms	0.529 ms	1.561 ms	750.0000	718.7500	718.7500	39 MB

Benchmark results

Conclusions

Conclusions

- A simple solution does not mean an effective solution.
- When designing and writing code, you need to take the time to analyse and find effective solutions, according to the requirements. This is especially important when writing shared libraries.
- It is necessary to identify gaps in the knowledge of developers and eliminate them.
- Use dotMemory for memory traffic analysis. Spend some time to ace that instrument. It will help you a lot.
- Performance issues are often connected with memory allocation. If you eliminate them, resources spent on GC work will be used for code execution. At the same time, it will prevent extra garbage collections. It will boost your code execution speed.



Repo with all materials

Thanks for your attention!

Nazarov Sergey