Write safe and efficient C# code

New features in C# enable you to write verifiable safe code with better performance. If you carefully apply these techniques, fewer scenarios require unsafe code. These features make it easier to use references to value types as method arguments and method returns. When done safely, these techniques minimize copying value types. By using value types, you can minimize the number of allocations and garbage collection passes.

Much of the sample code in this article uses features added in C# 7.2. To use those features, you must configure your project to use C# 7.2 or later. For more information on setting the language version, see configure the language version.

This article focuses on techniques for efficient resource management. One advantage to using value types is that they often avoid heap allocations. The disadvantage is that they're copied by value. This tradeoff makes it harder to optimize algorithms that operate on large amounts of data. New language features in C# 7.2 provide mechanisms that enable safe efficient code using references to value types. Use these features wisely to minimize both allocations and copy operations. This article explores those new features.

This article focuses on the following resource management techniques:

- Declare a `readonly struct` to express that a type is **immutable** and enables the compiler to save copies when using `in` parameters.
- Use a `ref readonly` return when the return value is a `struct` larger than IntPtr.Size and the storage lifetime is greater than the method returning the value.
- When the size of a `readonly struct` is bigger than IntPtr.Size, you should pass it as an `in` parameter for performance reasons.
- Never pass a `struct` as an `in` parameter unless it's declared with the `readonly` modifier because it may negatively affect performance and could lead to an obscure behavior.
- Use a `ref struct`, or a `readonly ref struct` such as Span<T> or ReadOnlySpan<T> to work with memory as a sequence of bytes.

These techniques force you to balance two competing goals with regard to **references** and **values**. Variables that are reference types hold a reference to the location in memory. Variables that are value types directly contain their value. These differences highlight the key differences that are important for managing memory

resources. **Value types** are typically copied when passed to a method or returned from a method. This behavior includes copying the value of `this` when calling members of a value type. The cost of the copy is related to the size of the type. **Reference types** are allocated on the managed heap. Each new object requires a new allocation, and subsequently must be reclaimed. Both these operations take time. The reference is copied when a reference type is passed as an argument to a method or returned from a method.

This article uses the following example concept of the 3D-point structure to explain these recommendations:

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

Different examples use different implementations of this concept.

Declare readonly structs for immutable value types

Declaring a `struct` using 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 `readonly`
- 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` clearly expresses your design intent.

Use `ref readonly return` statements for large structures when possible

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. 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);
```

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 `readonly ref`:

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

    public ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

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 `readonly ref` 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. However, the compiler can't know if any member method modifies the state of the struct. To ensure that the object isn't modified, the compiler creates a copy and calls member references using that copy. Any modifications are to that defensive copy.

Apply the `in` modifier to `readonly struct` parameters larger than `System.IntPtr.Size`

The `in` keyword complements the existing `ref` and `out` keywords to pass arguments by reference. The `in`keyword specifies passing the argument by reference, but the called method doesn't modify the value.

This addition 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 set the value of the argument used as this parameter.
- `in`: This method doesn't modify the value of the argument used as this parameter.

Add the `in` modifier to pass an argument by reference and declare your design intent to pass arguments by reference to avoid unnecessary copying. You don't intend to modify the object used as that argument.

This practice often improves performance for readonly value types that are larger than IntPtr.Size. For simple types (`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` and `bool`, and `enum` types), any potential performance gains are minimial. In fact, 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 adds up when your application calls this method in a tight loop using many different values.

The `in` modifier complements `out` and `ref` in other ways as well. You can't create overloads of a method that differ only in the presence of `in`, `out`, or `ref`. These new rules extend the same behavior that had always been defined for `out` and `ref` parameters. Like the `out` and `ref` modifiers, value types aren't boxed because the `in` modifier is applied.

The `in` modifier may be applied to any member that takes parameters: methods, delegates, lambdas, local functions, indexers, operators.

Another feature of `in` parameters is that you may use literal values or constants for the argument to an `in`parameter. Also, unlike a `ref` or `out` parameter, you don't need to apply the `in` modifier at the call site. The following code shows you 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());
```

There are several ways in which the compiler enforces the read-only nature of an `in` argument. First of all, the called method can't directly assign to an `in` parameter. It can't directly assign to any field of an `in` parameter when that value is a `struct` type. In addition, you can't pass an `in` parameter to any method using the `ref`or `out` modifier. These rules apply to any field of an `in` parameter, provided the field is a `struct` type and the parameter is also a `struct` type. In fact, these rules apply for multiple layers of member access provided the types at all levels of member access are `structs`. The compiler enforces that `struct` types passed as `in`arguments and their `struct` members are read-only variables when used as arguments to other methods.

The use of `in` parameters can avoid the potential performance costs of making copies. It doesn't change the semantics of any method call. Therefore, you don't need to specify the `in` modifier at the call site. Omitting the `in` modifier at the call site informs the compiler that it's allowed to make a copy of the argument for 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.

These rules are useful as you update existing code to use read-only reference arguments. Inside the called method, you can call any instance method that uses by value parameters. In those instances, a copy of the `in`parameter is created. Because the compiler may create a temporary variable for any `in` parameter, you can also specify default values for any `in` parameter. The following code specifies the origin (point 0,0) as the default value for the second point:

```
private static double CalculateDistance2(in Point3D point1, in Point3D 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);
}
```

To force the compiler to pass read-only arguments by reference, specify the `in` modifier on the arguments at the call site, as shown in the following code:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

This behavior makes it easier to adopt `in` parameters over time in large codebases where performance gains are possible. You add the `in` modifier to method signatures first. Then, you can add the `in` modifier at call sites and create `readonly struct` types to enable the compiler to avoid creating defensive copies of `in`parameters in more locations.

The `in` parameter designation can also be used with reference types or numeric values. However, the benefits in both cases are minimal, if any.

Never use mutable structs as in `in` argument

The techniques described above explain how to avoid copies by returning references and passing values by reference. These techniques work best when the argument types are declared as `readonly struct` types. Otherwise, the compiler must create **defensive copies** in many situations to enforce the readonly-ness of any arguments. 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 readonly struct. There are six different property access calls in the body of this method. On first examination, you may have thought these accesses were 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. 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-readonly-reference when the argument type isn't a `readonly struct`.

Instead, if the distance calculation uses the immutable struct, `ReadonlyPoint3D`, temporary objects are not 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.

You can see an example program that demonstrates the performance differences using Benchmark.net in our samples repository on GitHub. It compares passing a mutable struct by value and by reference with passing an immutable struct by value and by reference. The use of the immutable struct and pass by reference is fastest.

Use `ref struct` types to work with blocks or memory on a single stack frame

A related language feature is the ability to declare a value type that must be 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.

You may have similar requirements working with memory created using `stackalloc` or when using memory from interop APIs. You can define your own `ref struct` types for those needs.

`readonly ref struct` type

Declaring a struct as `readonly ref` combines the benefits and restrictions of `ref struct` and `readonly struct`declarations. The memory used by the readonly span is

restricted to a single stack frame, and the memory used by the readonly span can't be modified.

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 are 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 value type return values is heap allocated. The reference to that storage is stored on the stack.

Minimizing allocations comes with tradeoffs. You copy more memory when the size of the `struct` is larger than the size of a reference. A reference is typically 64 bits or 32 bits, and depends on the target machine CPU.

These tradeoffs generally have minimal performance impact. However, for large structs or larger collections, the performance impact increases. The impact can be large in tight loops and hot paths for programs.

These enhancements to the C# language are designed for performance critical algorithms where minimizing memory allocations is a major factor in achieving the necessary performance. You may find that you don't often use these features in the code you write. However, these enhancements have been adopted throughout .NET. As more and more APIs make use of these features, you'll see the performance of your applications improve.