Reduce memory allocations using new C# features

Article • 04/06/2023

Important

The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on **Diagnostics and instrumentation**. You can practice a profiling session in the tutorial to **Measure memory usage** in the Visual Studio documentation.

Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, or `out` modifier indicates that parameter is *passed by reference*. The *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. The *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

C#

```csharp
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment

Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and ref assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

Ref safe to escape scope

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

```C#
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe to escape scope is
the body of CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe to escape scope* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe to escape scopes* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

Declaration	***ref safe to escape scope***
non-ref local	block where local is declared
non-ref parameter	current method
`ref`, `in` parameter	calling method
`out` parameter	current method
`class` field	calling method
non-ref `struct` field	current method
`ref` field of `ref struct`	calling method

A variable can be `ref` returned if its *ref safe to escape scope* is the calling method. If its *ref safe to escape scope* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe to escape scope* is the calling method. The *ref safe to escape scope* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

```C#
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

> **Note**
>
> When the `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe to escape scope*. You can use `ref` parameters, `ref return` and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

Safe to escape scope and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type may include `ref` fields. That requires the introduction of a *safe to escape scope*. For most types, the *safe to escape scope* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe to escape scope* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe to escape scopes* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe to escape scope* is the method:

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unify memory types

The introduction of System.Span<T> and System.Memory<T> provide a unified model for working with memory. System.ReadOnlySpan<T> and System.ReadOnlyMemory<T> provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and `ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe to escape scope*. The *safe to escape* scope of a `ref struct` is the *ref safe to escape scope* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and that decreases the work the garbage collector does. It may also decrease memory pressure so the garbage collector runs less often.
- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `in` modifier to indicate that the argument can't be modified. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on reducing memory allocations.