MATT ELLIS

WRITING ALLOCATION FREE CODE IN C#

DON'T

YOU DON'T NEED TO DO THIS

UNLESS

YOU ACTUALLY NEED TO DO THIS

ALWAYS
BE
MEASURING

WHY?

MANAGED MEMORY IS CHEAP

ALLOCATION IS CHEAP

GARBAGE COLLECTION IS EXPENSIVE

PEAK MEMORY CAN BE LOW, WHILE MEMORY TRAFFIC IS HUGE

THROUGHPUT

THROUGHPUT

- More allocations mean more garbage collections
- Garbage collections introduce pauses
- Pauses disrupt throughput
- Better throughput means smoother, not (necessarily) faster
- Better throughput good for low latency - servers, games, interactive UI, etc.

CONSTRAINED ENVIRONMENTS

- E.g. mobile devices, consoles, etc.
- Using less memory is a Good Thing
- Less powerful device, GC can be slower
- Fewer GCs is better for battery
- Throughput again. Mobile gaming is HUGE…

KNOW YOUR PLATFORM

- Even if you don't use it, you're using it
- Performance has become a key part of the design of the platform. E.g.
    - C# tuples - `System.ValueTuple` not `System.Tuple`
    - `IAsyncEnumerator` (`await foreach`) + `ValueTask`
    - `Span<T>` + slicing

LOW HANGING FRUIT

LOW HANGING FRUIT

- Object reuse. Pass in existing array rather than allocate new one, etc.
- String concatenation. Use (and reuse) `StringBuilder`
- `params` arguments
- Boxing
- Closures
- LINQ
- Iterators
- `async/await`

PARAMS

- Unexpected compiler generated allocations

```
MyParamsMethod("Hello", "world");
MyParamsMethod();

void MyParamsMethod(params string[] args)
{
  // ...
}
```

```
MyParamsMethod(new[] {"Hello", "world"});
MyParamsMethod(new string[0]);

void MyParamsMethod(params string[] args)
{
  // ...
}
```

MEASURE!

- Newer frameworks and compiler use `Array.Empty<T>()`

```
MyParamsMethod("Hello", "world");
MyParamsMethod();

void MyParamsMethod(params string[] args)
{
  // ...
}
```

```
MyParamsMethod(new[] {"Hello", "world"});
MyParamsMethod(Array.Empty<string>());

void MyParamsMethod(params string[] args)
{
  // ...
}
```

BOXING

- Passing value type to method expecting a reference type
- Creates a new object on the heap (box) that contains a *copy* of the value type
- Any changes to the boxed value do not affect the orginal!

```
private static void PrintAnswer(int lifeEtc)
{
    Console.WriteLine("This answer is: {0}", lifeEtc);
}
```

(parameter) int lifeEtc

Boxing allocation: conversion from value type 'int' to reference type 'object'

BACK TO BASICS

REFERENCE TYPES VS VALUE TYPES

REFERENCE TYPES

- `class` keyword
- Allocated on heap
- A variable for a reference type is a reference to the thing on the heap
- Passed around *by reference*
- Assignment is a copy of the reference, not the object

VALUE TYPES

- `struct` keyword
- Allocated on stack
  Or embedded into a reference object
- A variable for a value type is the value itself, e.g. integer, vector, etc.
- Passed around *by value* (i.e. copied)
- Assignment is a copy of the whole value

HEAP VS STACK

- The heap is general purpose memory
  Lasts for the lifetime of the application
- The stack is a block of memory for data required by methods
- Each method pushes space onto the stack for local variables
- Pops the stack on method exit
  Stack allocation is for the lifetime of the method
- Value types are the whole data, so live directly on the stack
- `stackalloc` keyword allows creating blocks of memory on the stack
- Allocation *and* cleanup is cheap, but limited space

CLOSURES

- Compiler rewrites to capture local variables into class
  Lambda rewritten as method on this class
- LINQ - static method allocates Enumerator class
- Heap allocation viewer can show this

```
public static IEnumerable<string> FilterNames(IEnumerable<string> names, string name)
{
    return names.Where(n => n == name);
}
```

Delegate allocation: capture of 'name' parameter

ITERATORS

- Code is rewritten into a state machine
- Allocates state machine

```
foreach (var message in GetMessages())
{

}
```

(method) IEnumerable<string> RefSemantics.Span.GetMessages()

Object allocation: iterator method call

```
private static IEnumerable<string> GetMessages()
{
    yield return "Hello";
    yield return "World";
}
```

ASYNC / AWAIT

- Code is rewritten into a state machine
- Allocates state machine + “task method builder”
- More allocations for `Task` and `Task<T>`

  Expensive for common use cases - synchronous return, no reuse, etc.
- Can use `ValueTask` for common use cases

  https://blogs.msdn.microsoft.com/dotnet/2018/11/07/understanding-the-whys-whats-and-whens-of-valuetask/

LOW HANGING FRUIT

- Object reuse. Pass in existing array rather than allocate new one, etc.
- String concatenation. Use (and reuse) `StringBuilder`
- `params` arguments - *Introduce overloads with common number of arguments*
- Boxing - *Introduce generic overloads*
- Closures - *Avoid in critical paths. Pass state as argument to lambda. Investigate local functions*
- LINQ - *Avoid in critical paths. Use good old `foreach` and `if`*
- Iterators - *Return a collection? Be aware of the cost*
- `async/await` - *Investigate `ValueTask`*

REFERENCE SEMANTICS WITH VALUE TYPES

SAY WHAT?

C# 7.2: REFERENCE SEMANTICS WITH VALUE TYPES

- `in` parameters

  Pass value type by reference. Called method cannot modify it

- `ref` locals and `ref` returns (C# 7.0)

- `ref readonly` returns

  Return a read only value type by reference

- `readonly struct`

  Immutable value types

- `ref struct`

  Stack only value types

WHAT DOES "REFERENCE SEMANTICS WITH VALUE TYPES" EVEN MEAN?

Allocating a reference type has a ***cost***, but passing it around is ***cheap***

Allocating a value type is ***cheap***, but passing it around has a ***cost***

Why can't it be cheap to allocate **AND** cheap to pass around?

REFERENCE SEMANTICS WITH VALUE TYPES

- Allows value types to be used like reference types
  Pass by reference everywhere
- Use value types to reduce allocations, reduce memory traffic, etc.
  Throughput!
- Pass by reference to avoid copies, enable modifying, etc.
- Very low level micro-optimisations…
  But they'll be used in the platform…
  (And games, and parsing, and serialisation, and…)

PASS BY REFERENCE / PASS BY VALUE

- A variable for a reference type is a reference to the actual object on the heap
- Passing a reference type to a method is just passing this reference
  The caller and the called method see the same object on the heap

- A variable for a value type is the value itself
- Passing a value type to a method *copies* the value
- Assigning a value type to a new variable also *copies* the value
- Original value is unmodified
- (Copies aren't actually that expensive)

C# 7.0: REF RETURN

- Return a reference to value type, not a copy of the value
  Return type of method becomes e.g. integer reference - `int&` in IL
- Lifetime of returned value must exceed the lifetime of the called method
  E.g. a reference to a field or method argument. NOT a variable in the called method.
- Modifying this reference is the same as modifying the original value
  E.g. return reference to array element, and update it
- Add `ref` modifier to method declaration return type, and to `return` statement
- Not allowed on async methods

C# 7.0: REF LOCAL

- Assigning a ref return to a new variable will create a copy
  The variable is a value type, not a reference! (Cannot assign `int&` to `int`)
- A ref local is a variable that is a reference to a value type
  Accessing the variable accesses the original value
- Use a ref local to store the ref return result
- Type inference with `var` will get the value type, not the `ref` modifier
  Requires `ref var` to work as expected

REF RETURN

DEMO

C# 7.2: REF READONLY RETURN

- Extends ref returns and ref locals
- Return a value type by reference, but caller is not allowed to modify
- Assign to `ref readonly var`
- Compiler enforces readonly with errors and ***defensive copies***

```
private static Point3D origin = new Point3D(0, 0, 0);
public static ref readonly Point3D Origin => ref origin;

var originValue = Point3D.Origin;   // Copy
ref readonly var originRef = ref Point3D.Origin;    // Reference, but defensive copies for invocation
```

C# 7.2: IN PARAMETERS

- Method argument modifier
- Complements `out` and `ref`
- Passed by reference
- Method cannot modify original value
- Compiler enforces safety with ***defensive copy*** when calling members

```
private static double CalculateDistance(in Point3D p1, in Point3D p2)
{
  double dx = p1.X - p2.X;
  double dy = p1.Y - p2.Y;
  double dz = p1.Z - p2.Z;

  return Math.Sqrt(dx * dx + dy * dy + dz * dz);
}
```

C# 7.2: READONLY STRUCT

- `in` parameters and `ref readonly` can create defensive copies

  The compiler doesn't know if the struct's methods will modify state
- `readonly struct` - compiler enforces all fields and properties are readonly
- Immutable
- More efficient - no copies made when calling members

  Improves performance (micro-optimisation)

NO CHANGES TO CLR

C# 7.2: REF STRUCT

- Declare a value type that can only be stack allocated

  I.e. can never be part of a reference type

- This constrains lifetime to calling method

  Also, cannot be boxed, cannot use inside a non-ref struct

  Cannot use with async methods or iterators

  Cannot be a generic parameter

- Limited use cases
  - Working with `stackalloc` memory
  - Primarily for `Span<T>`

SPAN<T>

SPAN<T>

- New type to unify working with any kind of contiguous memory

  Arrays, array segments, strings and substrings, native memory, `stackalloc`, etc.

- Provides array-like API - indexer

  `ReadOnlySpan<T>` provides getter indexer only

- Type safe - each element is of type `T`

- Array-like performance

  Not quite, but newer runtimes have special support

- Slicing

  Create a new `Span<T>` with a sub-section of existing `Span` - without allocations!

SPAN<T> IMPLEMENTATION

- Value Type - `struct`
- System.Memory NuGet package
  .NET Standard 1.1 (.NET Framework 4.5)+
- New APIs and overloads in the BCL
  E.g. `String.AsSpan()`, `Stream.ReadAsync()`, `Utf8Parser.TryParse()`
  Significant usage of ref semantics - allocation free!
- `Span<T>`, `ReadOnlySpan<T>`, `Memory<T>`
- Two versions - “portable” and “fast”
  Fast requires runtime support

SPAN<T> PERFORMANCE - PORTABLE IMPLEMENTATION

- Portable works on .NET Standard 1.1 and above
  .NET Framework 4.5+
- Portable is not slow!
  But not as fast as arrays
- Three fields - object reference, internal offset and length
  Slightly larger than fast version, dereferencing is *slightly* more complex operation

SPAN<T> PERFORMANCE - FAST IMPLEMENTATION

- Fast requires runtime support
  .NET Core 2.1
- Only has two fields - “byref” internal pointer and length
  Slightly smaller struct and accessing an element is *slightly* simpler operation
- Specific JIT optimisations
  E.g. eliding bounds check in loop, like arrays
- Very close to array performance

WHAT DOES THIS HAVE TO DO WITH REF STRUCT?

- For thread safety, need to update all fields of `Span<T>` atomically (tearing)

  Whole point is performance - cannot use synchronisation

- Internal pointers require special GC tracking

  Too many in flight at once is expensive

- How could `Span<T>` represent `stackalloc` memory is `Span<T>` was on the heap?

- Solution: `Span<T>` is a `ref struct` - can only be created on the stack

  Constrained lifetime, single thread access

SPAN<T>

DEMO

LINKS

- Reference semantics with value types
  https://docs.microsoft.com/en-us/dotnet/csharp/reference-semantics-with-value-types
- Channel 9 - C# 7.2: Understanding `Span<T>` - Jared Parsons
  https://channel9.msdn.com/Events/Connect/2017/T125
- Adam Sitnik's `Span<T>` post (July 13, 2017)
  http://adamsitnik.com/Span/
- RyuJIT optimisations for `Span<T>`
  https://blogs.msdn.microsoft.com/dotnet/2017/10/16/ryujit-just-in-time-compiler-optimization-enhancements/
- Understanding the whys, whats and whens of `ValueTask`
  https://blogs.msdn.microsoft.com/dotnet/2018/11/07/understanding-the-whys-whats-and-whens-of-valuetask/