Safe Systems Programming

in C# and .NET

Joe Duffy

joeduffyblog.com · @xjoeduffyx · joeduffy@acm.org

Introduction

“Systems”?

- Anywhere you’re apt to think about “bits, bytes, instructions, and cycles”
- Demanding performance and reliability requirements
- Classically meant “interacts with hardware,” requiring C and unsafe code
- Abstraction has shifted upwards thanks to concurrency and cloud; security is more important now than ever
- Many scenarios: operating systems, drivers, tools, libraries, cloud infrastructure, web servers, micro services and their frameworks, …

Why C#?

- Productivity and ease of use
- Built-in safety: fewer inherent security and reliability risks
- Powerful async and concurrency models
- Momentum, mature ecosystem of tools and libraries

Why *Not* C#?

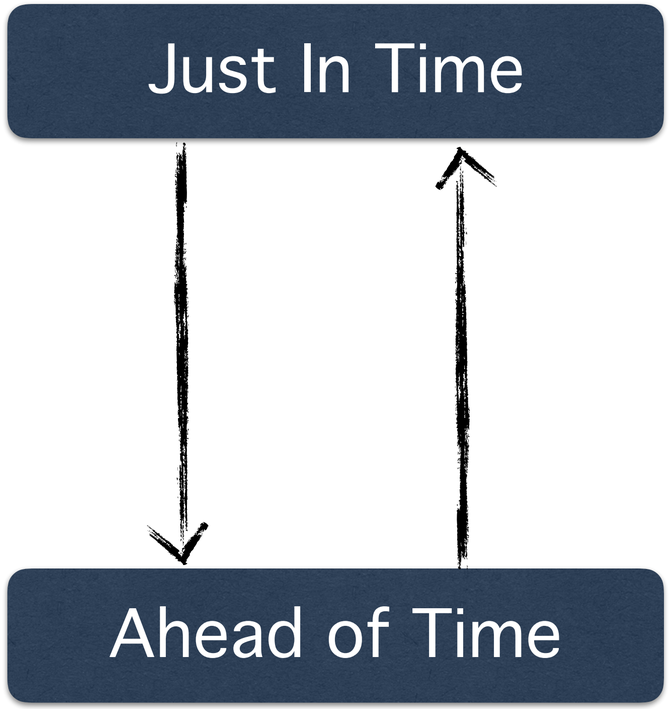
- Garbage collection
- Allocation-rich APIs and patterns
- Error model that makes reliability challenging
- Concurrency is based on unsafe multithreading

This Talk

- We are making progress on all of the above
- New C# and library features
- Patterns that you can apply today, enforce w/ Roslyn Analyzers
- Advances in code generation technologies
- All open source, on GitHub, and developed with the community

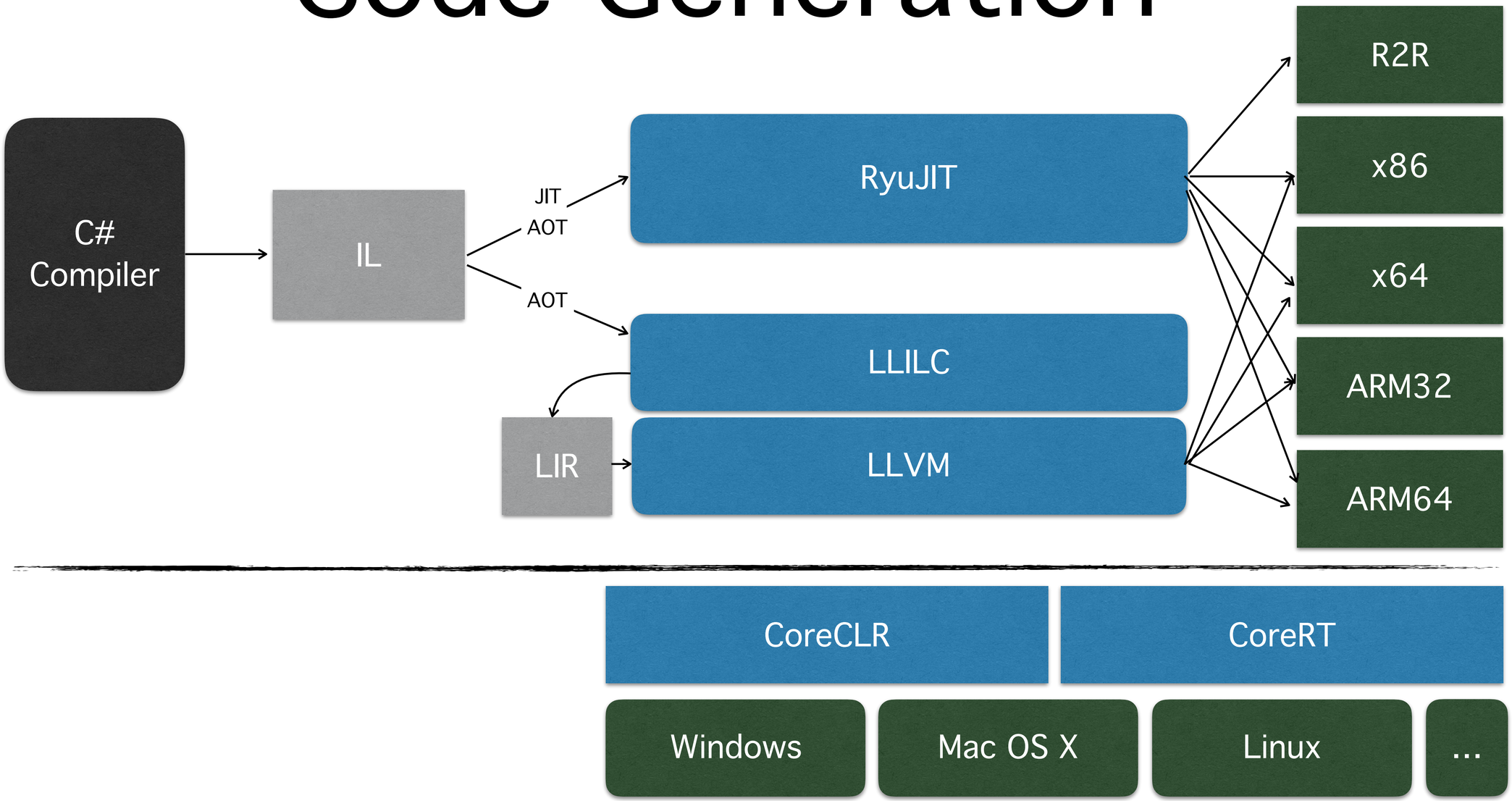
Performance

Code Generation



- A spectrum of code generation
- JIT: fast compile times, simple, decent code quality
- AOT: best code quality, slower compile times, more complex deployment model
- Hybrid: pick and choose, let the system adaptively recompile (future)

Code Generation



Optimizations

- Inlining
- Flowgraph and loop analysis
- Copy/constant propagation
- Range analysis
- Loop invariant code hoisting
- SIMD and vectorization
- SSA and global value numbering
- Common subexpression elimination
- Dead code elimination
- Devirtualization
- Generic sharing
- Stack allocation (work in progress)

Ex. Inlining

```
int a = …, b = …;
Swap<int>(ref a, ref b);

static void Swap<T>(ref T a, ref T b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Calls are not cheap: new stack frame, save return address, save registers that might be overwritten, call, restore
Adds a lot of waste to leaf-level functions (10s of cycles)

Ex. Inlining

```
int a = …, b = …;
int tmp = a;
a = b;
b = tmp;
```

Ex. Range Analysis

```
int[] elems = …;
for (int i = 0; i < elems.Length; i++) {
    elems[i]++;
}
```

Loop “obviously” never goes out of bounds
But a naive compiler will do an induction check (i < elems.Length)
plus a bounds check (elems[i]) on every iteration!

Ex. Range Analysis

```
int[] elems = …;
for (int i = 0; i < elems.Length; i++) {
    elems[i]++;
}
```

```
; Initialize induction variable to 0:
3D45: 33 C0           xor         eax,eax
; Put bounds into EDX:
3D58: 8B 51 08        mov         edx,dword ptr [rcx+8]
; Check that EAX is still within bounds; jump if not:
3D5B: 3B C2           cmp         eax,edx
3D5D: 73 13           jae         3D72
; Compute the element address and store into it:
3D5F: 48 63 D0        movsxd      rdx,eax
3D62: 89 44 91 10     mov         dword ptr [rcx+rdx*4+10h],eax
; Increment the loop induction variable:
3D66: FF C0           inc         eax
; If still in bounds, then jump back to the loop beginning:
3D68: 83 F8 64        cmp         eax,64h
3D6B: 7C EB           jl          3D58
; ...
; Error routine:
3D72: E8 B9 E2 FF FF  call        2030
```

Ex. Range Analysis

```
int[] elems = …;
for (int i = 0; i < elems.Length; i++) {
    elems[i]++;
}
```

```
; Initialize induction variable to 0:
3D95: 33 C0           xor         eax,eax
; Compute the element address and store into it:
3D97: 48 63 D0        movsxd      rdx,eax
3D9A: 89 04 91        mov         dword ptr [rcx+rdx*4],eax
; Increment the loop induction variable:
3D9D: FF C0           inc         eax
; If still in bounds, then jump back to the loop beginning:
3D9F: 83 F8 64        cmp         eax,64h
3DA2: 7C F3           jl          3D97
```

Ex. Stack Allocation

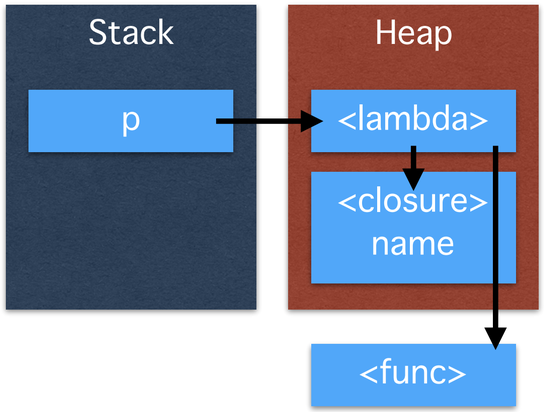
```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = custs.IndexOf(c => c.Name == name);

int IndexOf(Func<T, bool> p) {
    for (int i = 0; i < this.count; i++) {
        if (p(this[i]))
            return i;
    }
    return -1;
}
```

Ex. Stack Allocation

```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = custs.IndexOf(c => c.Name == name);

int IndexOf(Func<T, bool> p) {
    for (int i = 0; i < this.count; i++) {
        if (p(this[i]))
            return i;
    }
    return -1;
}
```

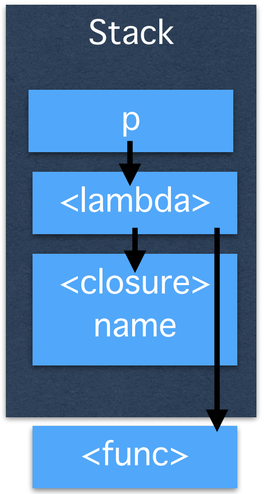


Allocates up to 2 objects (lambda+captured stack frame)

Ex. Stack Allocation

```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = custs.IndexOf(c => c.Name == name);

int IndexOf(Func<T, bool> p) {
    for (int i = 0; i < this.count; i++) {
        if (p(this[i]))
            return i;
    }
    return -1;
}
```

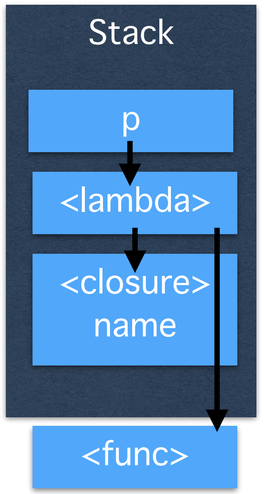


~~Allocates up to 2 objects (lambda+captured stack frame)~~
Automatic escape analysis can determine ‘p’ doesn’t escape IndexOf

Ex. Stack Allocation

```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = custs.IndexOf(c => c.Name == name);

int IndexOf(Func<T, bool> p) {
    for (int i = 0; i < this.count; i++) {
        if (p(this[i]))
            return i;
    }
    return -1;
}
```

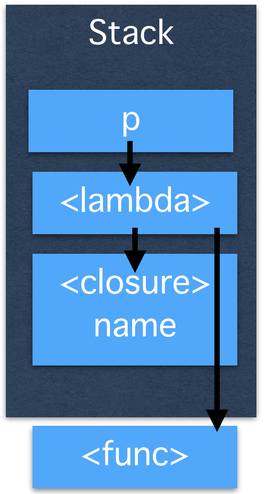


~~Allocates up to 2 objects (lambda+captured stack frame)~~
Automatic escape analysis can determine ‘p’ doesn’t escape IndexOf

Ex. Stack Allocation

```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = custs.IndexOf(c => c.Name == name);

int IndexOf([Scoped] Func<T, bool> p) {
    for (int i = 0; i < this.count; i++) {
        if (p(this[i]))
            return i;
    }
    return -1;
}
```



~~Allocates up to 2 objects (lambda+captured stack frame)~~
Automatic escape analysis can determine ‘p’ doesn’t escape IndexOf

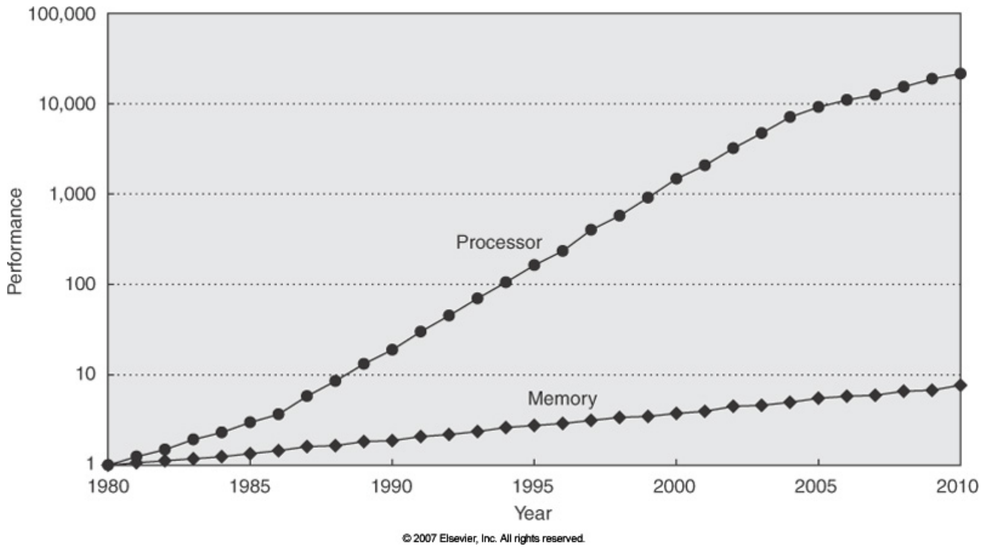
Ex. Stack Allocation

```
string name = "Alexander Hamilton";
List<Customer> custs = …;
int index = -1;
for (int i = 0; i < custs.count; i++) {
    if (custs[i].Name == name) {
        index = i;
        break;
    }
}
```

~~Allocates up to 2 objects (lambda+captured stack frame)~~
~~Automatic escape analysis can determine ‘p’ doesn’t escape IndexOf~~
Best case, IndexOf is inlined, zero allocations, no virtual call!

Memory

CPU, Cache, Memory



L1i/d$: 32KB, L2: 256KB, L3: 8MB*

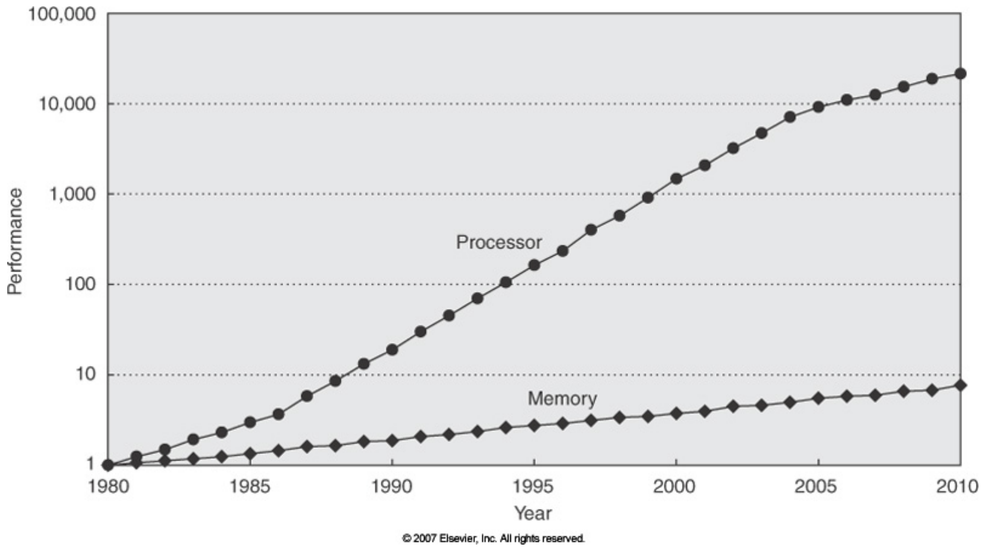
Latency numbers every programmer should know

```
L1 cache reference ..................... 0.5 ns
Branch mispredict ...................... 5 ns
L2 cache reference ..................... 7 ns
Mutex lock/unlock ...................... 25 ns
Main memory reference ..................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

Summary: Instructions matter; memory matters more
(and I/O dwarfs them all...)

* Intel i7 “Haswell” processor cache sizes

CPU, Cache, Memory



L1i/d$: 32KB, L2: 256KB, L3: 8MB*

Latency numbers every programmer should know

```
L1 cache reference ..................... 0.5 ns
Branch mispredict ...................... 5 ns
L2 cache reference ..................... 7 ns
Mutex lock/unlock ...................... 25 ns
Main memory reference ..................... 100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

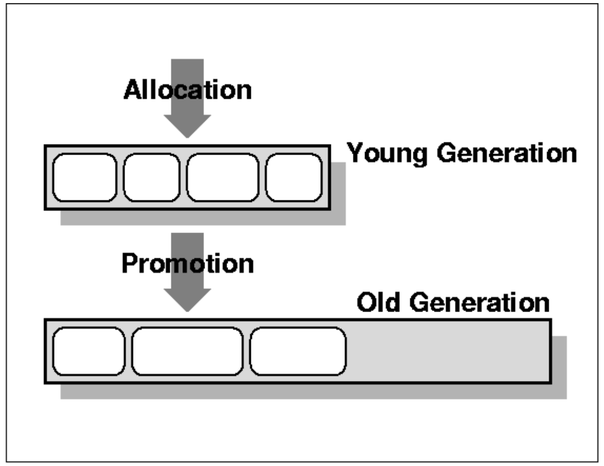
} GC Pause

Summary: Instructions matter; memory matters more
(and I/O dwarfs them all…; and so does GC)

* Intel i7 “Haswell” processor cache sizes

Garbage Collection

- Generational, compacting garbage collector
- Concurrent background scanning for automatically reduced pause times
- Manual “low pause” regions (LowLatency)
- Parallel collector for server workloads; in .NET 4.5, concurrent+parallel in harmony



* https://blogs.msdn.microsoft.com/dotnet/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps/

Pitfalls

- Premature graduation: objects meant to die in Generation 0 live longer
- Mid-life crisis: objects meant to die in Generation 1 live to Generation 2
- Pre-mortem finalization and resurrection: objects get promoted unexpectedly
- LOH “leaks”: very large buffers aren’t a good fit for GC
- In each case, symptom is high GC times; hint: >=10% is bad, <=1% is possible
- Multi-threading complicates things — may perform well in isolation

Values

- C# has two major type kinds: structs and classes

```
struct Point3D {…}   class Point3D {…}
```

	Instance	Where	Overhead	"C Type"
Struct	Value	"Inline" (embedded in other object/value, or on stack)	None	T ~= T ref T ~= T&
Class	Object	GC Heap	8 bytes (32-bit) 16 bytes (64-bit)	T ~= T*

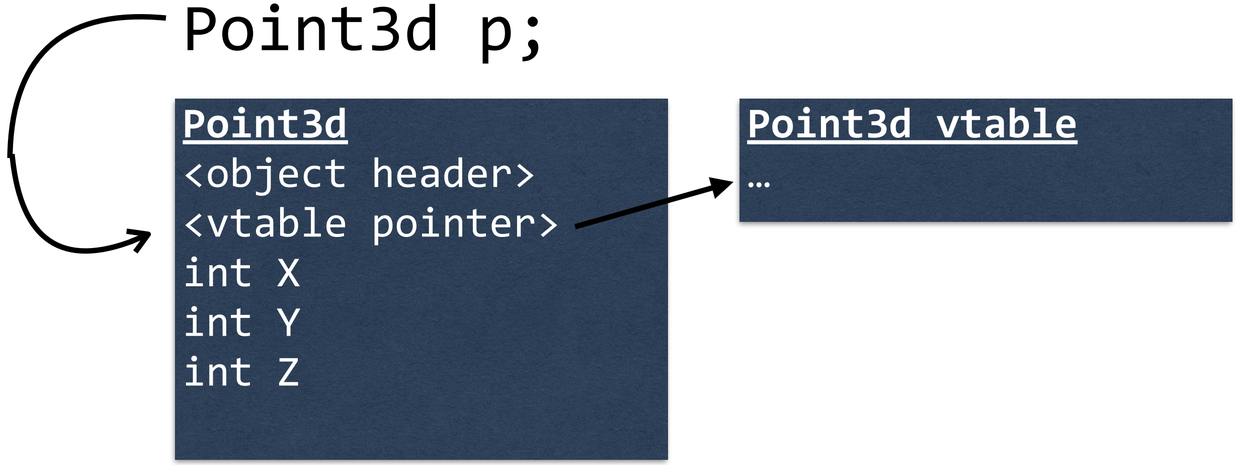
Values

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

```
class Point3D {
    public int X;
    public int Y;
    public int Z;
}
```

```
Point3d p;
```


```
Point3d p;
```



Values and Memory

- Structs can improve memory performance
  - Less GC pressure
  - Better memory locality
  - Less overall space usage
- Beware of copying large structs, however: memcpy
- Byrefs can be used to minimize copying; “ref returns” is a new feature in C# 7
- New features in next rev of C# and .NET: ValueTask, ValueTuple, others

Strings and Arrays

- Strings and arrays often subject to premature graduation, especially Big Data scenarios
- Ex. parse integers from a comma-delimited string; beware code like this!

```
string numbers = "0,1,42,99,128";
string[] pieces = numbers.Split(',');
int[] parsed = (from piece in pieces
                select int.Parse(piece)).ToArray();
long sum = 0;
for (int i = 0; i < pieces.Length; i++) {
    sum += parsed[i];
}
```

Possibly copied UTF-8 to UTF-16
Split allocates 1 array + O(N) strings, copying data
LINQ query allocates O(2Q)+ enumer* objects
ToArray allocates at least 1 array (dynamically grows)

Span

```
// Create over a managed array:
Span<int> ints = new[] { 0, …, 9 };

// Or a string:
SpanView<char> chars = "Hello, Span!";

// Or a native buffer:
byte* bb = …;
Span<byte> bytes = new Span<byte>(bb, 512);

// Or a sub-slice out from an existing slice:
var name = "George Washington";
int space = name.IndexOf(' ');
var firstName = name.Slice(0, space);
var lastName = name.Slice(space + 1);

// Uniform access regardless of how it was created:
void Print<T>(SpanView<T> span) {
    for (int i = 0; i < span.Length; i++)
        Console.Write("{0} ", span[i]);
   Console.WriteLine();
}
```

- Span is a struct “slice” out of an array, string, native buffer, or another span
- Uniform access regardless of creation
- All accesses are safe and bounds checked

* Currently incubating for future C#/.NET: https://github.com/dotnet/corefxlab/tree/master/src/System.Slices

Strings and Arrays

- Returning to our previous example:

```
string numbers = "0,1,42,99,128";
int sum = 0;
foreach (Span<char> piece in numbers.SplitEnum(',')) {
    sum += int.Parse(piece);
}
```

- Zero-alloc, zero-copy; still possible without Span, just more work

- And, even better, if UTF-8, no need to decode and copy to GC heap

```
SpanView<byte> numbers = "0,1,42,99,128";
// As above, but with Span<byte> instead of Span<char> …
```

Packs

- Small arrays often end up on the GC heap needlessly

```
int[] array = new int[8] { 0, …, 7 };
// Heap allocation!  For short-lived arrays, this is bad!
```

- Pack is a fixed size, struct-based array that interoperates with Span APIs

```
Pack8<int> pack = new Pack8<int>(0, …, 7);
pack[0] = pack[1]; // Normal indexers, just like an array.
Span<int> span = pack; // OK, we can treat a Pack like a Span!
```

- Too bad it's limited to PackN; currently incubating a "safe" `stackalloc` for the future

```
Span<int> span = stackalloc int[8] { 0, …, 7 };
```

Zero Copy

- Do not copy memory needlessly
- `byte[]` is almost always a sign of danger ; if it's in native memory, keep it there (common LOH pitfall)
- Span/Primitive make it convenient to work with `byte*` in a "safe" way

```
[StructLayout(...)]
struct TcpHeader {
    ushort SourcePort;
    ushort DestinationPort;
    ...
    ushort Checksum;
    ushort UrgentPointer;
}
```

```
void HandleRequest(byte* payload, int length) {
    var span = new Span<byte>(payload, length);
    // Parse the header:
    var header = Primitive.Read<TcpHeader>(ref span);
    … header.SourcePort …; // etc.
    // Keep parsing …
    for (…) {
        byte b = Primitive.Read<byte>(ref span);
        …;
    }
}
```

Reliability

Bugs

- Exceptions are meant for recoverable errors; but many errors are not recoverable!
- A bug is an error the programmer didn't expect; a recoverable error is an expected condition, resulting from programmatic data validation

Bugs		Recoverable
Incorrect cast	Out of memory	I/O failure
Dereferencing null	Stack overflow	Parsing error
Array index out-of-bounds	Precondition violation	Data validation error
Divide by zero	Assertion failure	
Arithmetic under/overflow	Explicit abandonment	

- Treating bugs and recoverable errors homogeneously creates reliability problems

Fail-fast

- If you’re going to fail, do it fast
- For places where exceptions delay the inevitable, and invites abuse

```
try {
    BigHunkOfCode();
}
catch (ArgumentNullException) {
    // Ignore, and keep going! 😞
}
```

- Fail-fast ensures bugs are caught promptly before they can do more damage
- In our experience, 1:10 ratio of recoverable errors (exceptions) to bugs (fail-fast)

Contracts and Asserts

- `Contract.Requires`: preconditions that must be met before calling an API
- `Contract.Assert`: conditions that must hold at a specific point in the program
- `Contract.Fail`: initiate an explicit fail-fast (alternatively `Environment.FailFast`)
- Can be debug-only, `Contract.Debug.*` (generally bad idea for preconditions)

```
int Read(char[] buffer, int index, int count) {
    Contract.Requires(buffer != null);
    Contract.Requires(
        Range.IsValid(index, count, buffer.Length));
    // … we know the conditions hold here …
}

// Elsewhere:
char[] buffer = …;
Contract.Debug.Assert(index >= 0 && index < count);
Contract.Debug.Assert(count <= stream.Count);
stream.Read(buffer, index, count);

// Of course, it’s better to do this by-construction:
char[] buffer = …;
SpanView<char> slice = buffer.Slice(index, count);
stream.Read(slice);
```

Immutability

- Immutability can improve concurrency-safety, reliability (no accidental mutation), and performance (enables compiler optimizations)
- In C#, readonly means “memory location cannot be changed”

```
readonly int x = 42; // 42 forever.
```

- An immutable structure is one with all readonly fields
- A deeply immutable structure is one with all readonly fields, where each field refers to another immutable structure (including primitives)
- New data features in C# make working with immutability easier

```
struct Point3D {
    public readonly int X;
    public readonly int Y;
    public readonly int Z;
}

struct Line {
    public readonly Point3D A;
    public readonly Point3D B;
}
```

Immutability

- Immutability can improve concurrency-safety, reliability (no accidental mutation), and performance (enables compiler optimizations)
- In C#, readonly means “memory location cannot be changed”

```
readonly int x = 42; // 42 forever.
```

- An immutable structure is one with all readonly fields
- A deeply immutable structure is one with all readonly fields, where each field refers to another immutable structure (including primitives)
- New data features in C# make working with immutability easier

```
[Immutable]
struct Point3D {
    public readonly int X;
    public readonly int Y;
    public readonly int Z;
}
[Immutable]
struct Line {
    public readonly Point3D A;
    public readonly Point3D B;
}
```

Wrap Up

Summary

- Systems programming landscape is changing
- Safety is more important than ever, and performance bottlenecks have largely shifted elsewhere
- C# delivers productivity and safety, while still delivering good performance — from JIT to AOT and everything in between
- It will only get better from here
- Vibrant community, please check out GitHub and help us build this stuff!

Q&A

- Thank you!
- C# Systems Programming Pack (types+analyzers):

  https://github.com/joeduffy/csysprog (later today)
- .NET GitHub repositories — where all the action is at!

  https://github.com/dotnet/corefx

  https://github.com/dotnet/coreclr

  https://github.com/dotnet/corefxlab

  https://github.com/dotnet/roslyn

joeduffyblog.com · @xjoeduffyx · joeduffy@acm.org

Backup

Security

- Borderline crisis in our industry
- “There are two kinds of companies in the world: those that know they’ve been hacked, and those that don’t.”
- “Gartner, Inc. forecasts that 6.4 billion connected things will be in use worldwide in 2016, up 30 percent from 2015, and will reach 20.8 billion by 2020. In 2016, 5.5 million new things will get connected every day.”
- Buffer errors >20% of all exploits in 2016, up from 18% in 2015
- It’s time to start building our most mission critical software in safe languages

* https://nvd.nist.gov/visualizations/cwe-over-time

SIMD

Ready-to-Run (R2R)

- New native image file format used by CoreCLR
- Version resilient, so (for instance) moving fields doesn't have cascading recompilation problems
- All standard libraries ship using it

* https://github.com/dotnet/coreclr/blob/master/Documentation/botr/readytorun-overview.md

Error Models

	The Good	The Bad	The Ugly
Error Codes	• All function that can fail are explicit annotated. • All error handling at callsites is explicit.	• You can forget to check them. • Performance of success paths suffers.	• Usability is often subpar.
All Exceptions	• First class language support.	• Performance is typically worse than it could be. • Handling is often done in a non-local manner, where less information about an error is known (goto-like).	
Unchecked Exceptions	• Conducive to rapid development where dealing with errors reliably isn't critical.	• Anything can fail without warning from the language.	• Reliability is as bad as it gets.
Checked Exceptions	• All function that can fail are explicitly annotated.	• Callsites aren't explicit about what can fail and error propagation. • Systems that let some subset of exceptions go unchecked poison the well (not all errors are explicit).	• People hate them (in Java, at least).

Annotations+Analyzers

- Do not use mutable statics
- Don’t implicitly ignore expressions; use `Result.Ignore`
- Don’t implicitly box; use `Result.Box`
- `[ReadOnly]` for shallow immutability
- `[Immutable]` for deep immutability
- `[Throws]` for methods that can throw
- `[NoAlloc]` ensures a method doesn’t allocate
- `[MustNotCopy]` ensures a struct isn’t copied
- `[Scoped]` ensures a value doesn’t escape the callee
- ... and more ...