# Escape Analysis & Profiling Of Go Applications

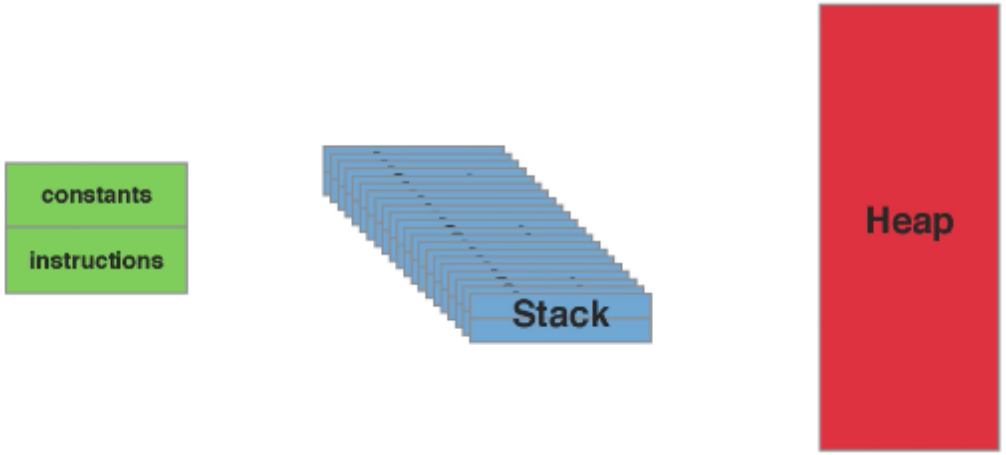
Kanstantsin Shamko
Erlang developer at IDT (https://www.idt.net/)

# Agenda

- Garbage collector
- Escape analysis
- Profiling tools

2

# Memory Model of Golang

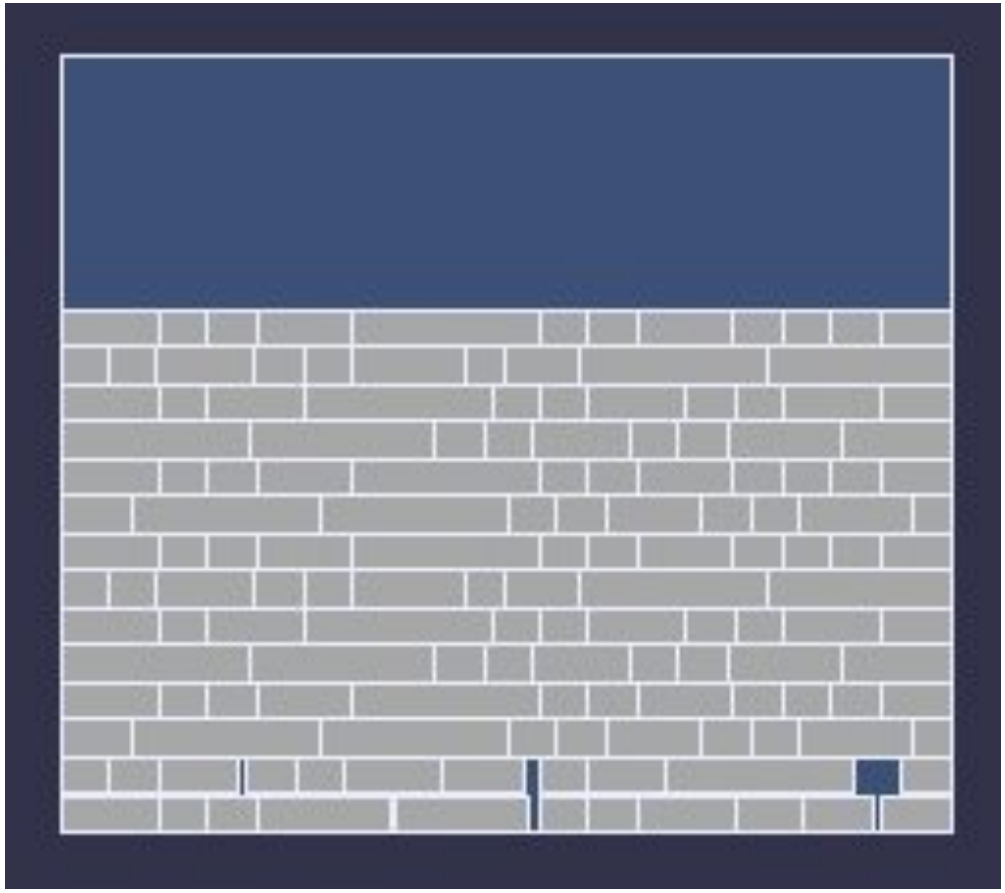



- each goroutine has own **stack**
- **stack** represents a state of execution (holds function calls and their local variables)
- default stack size is 8KB. Could be resided by the Go Runtime
- **heap** holds variables (i.e. pointers, arrays, data structures)
- garbage collector cleans **heap**

3

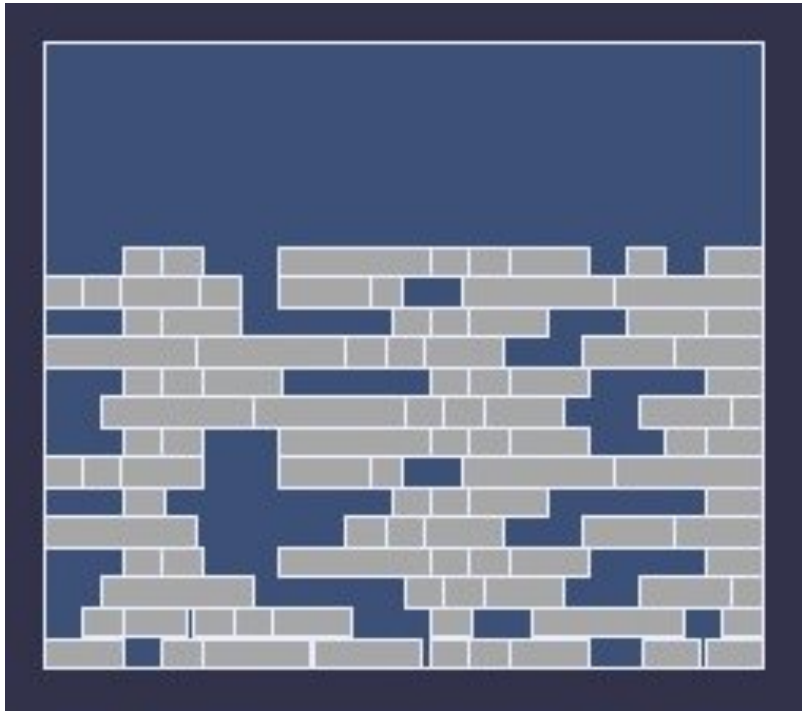
# Garbage Collector

# Heap Before GC



5

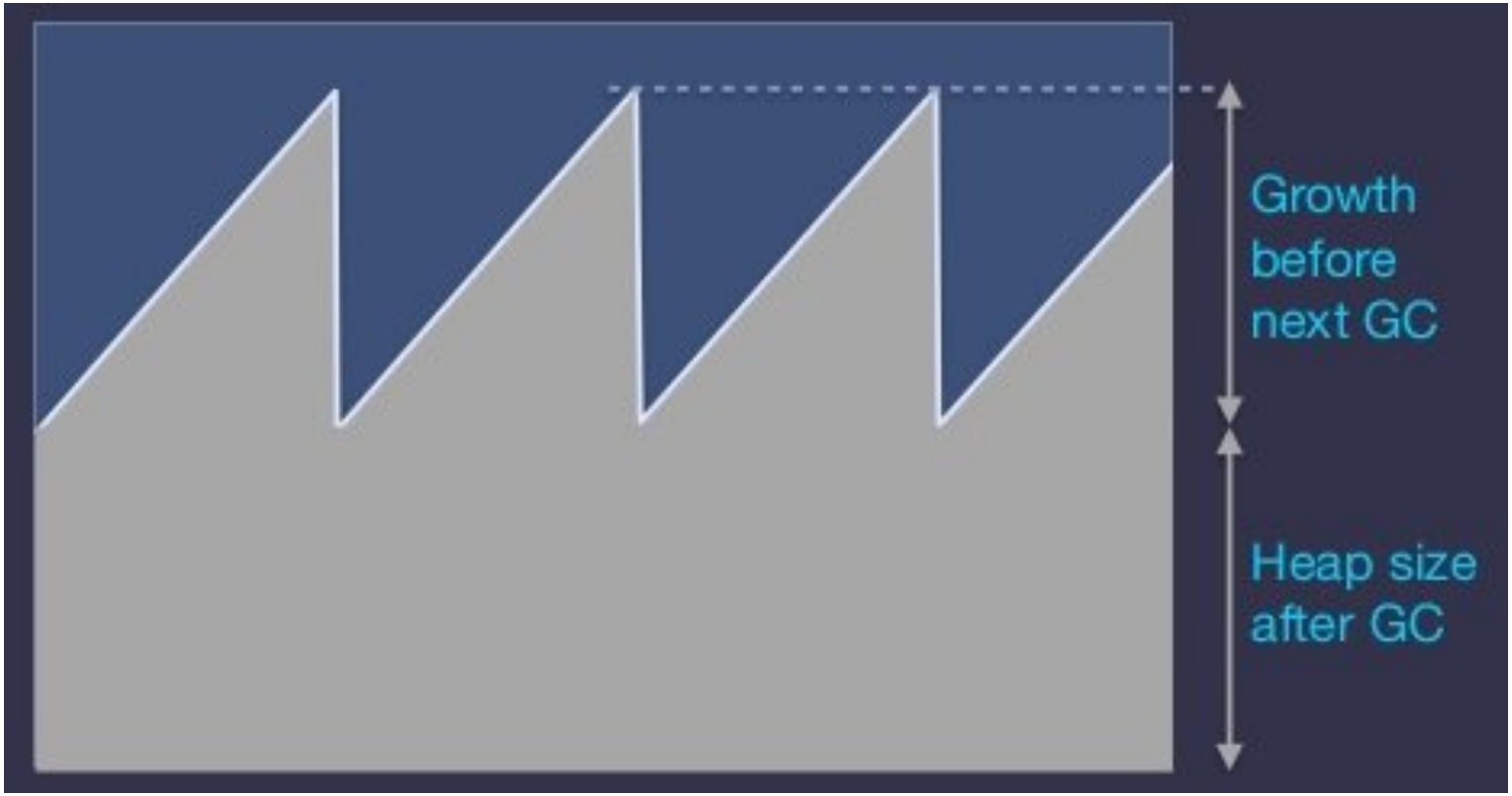
# Heap After GC



- GC locates memory blocks which have no pointers on them
- GC cleans located blocks

6

# Memory Usage Plot




7

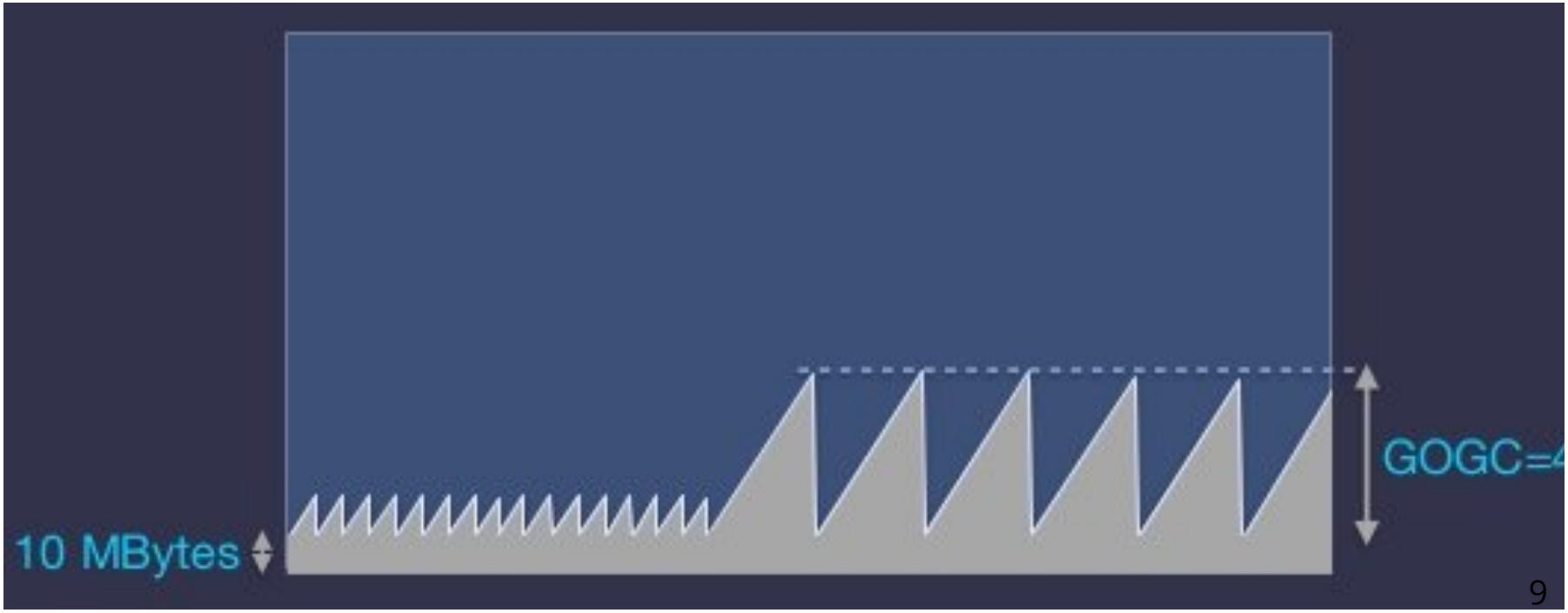
## Garbage Collector Settings

# GOGC

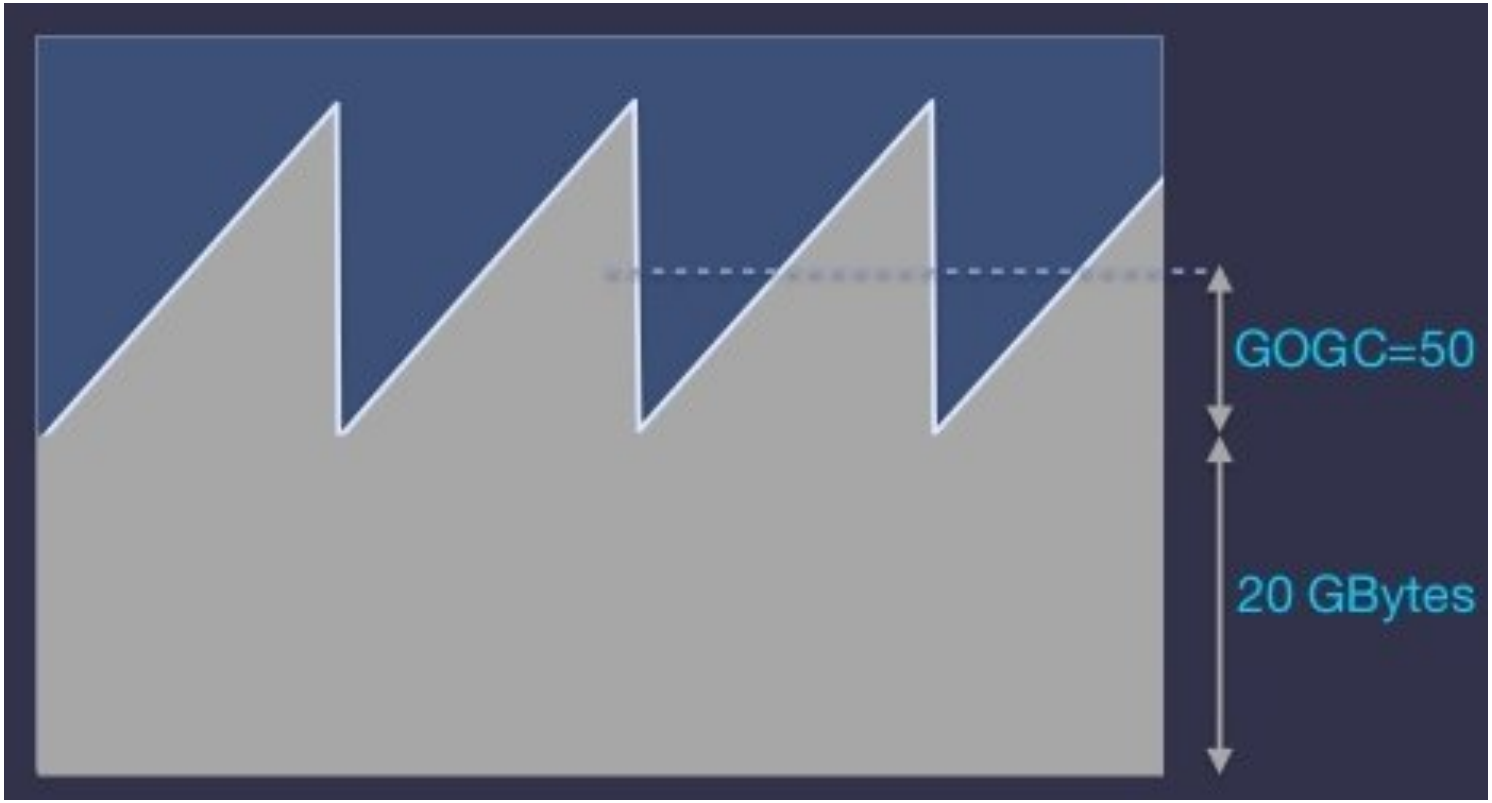
- env variable
- controls the aggressiveness of GC
- GOGC=200 => GC cycle will start when heap grows to 200% of the prev size
- GOGC=off

8

# Relatively Low Memory Usage



# Large and Stable Memory Usage




10

# App lives for a short time

```
GOGC=off go build ...
```

11

# Memory Allocation

## Types of Allocation

- stack allocation (cheap)
- heap allocation (expensive)

Go Runtime manages allocations automatically. No way to say where to allocate

## Stack Allocation

- in a local stack of each goroutine
- cheap because requires 2 CPU instructions: push to the stack, release from the stack
- requires that the lifetime and memory footprint of a var can be determined at *compile time*

14

## Heap Allocation

- in a global heap for dynamic allocations
- occurs at *run time*
- expensive because:

1. it is required to search for a chunk of free memory large enough for a var
2. garbage collection

15

# Escape Analysis

## What Is It?

- compiler's technique to choose between 2 types of allocations
- not an optimization but can be used for that
- rules of escape analysis not a part of Go specification
- rules could be changed. make an experimentation

17

# Escape Example

```
package main

import "fmt"

func main() {
    x := 3
    fmt.Println(x)
}
```

```
$ go build -gcflags '-m' ./escape_example.go
```

```
./escape_example.go:9:13: x escapes to heap
./escape_example.go:9:13: main ... argument does not escape
```

```
$ go build -gcflags '-m -m' ./escape_example.go
```

```
escape_example.go:9:13: x escapes to heap
escape_example.go:9:13:         from ... argument (arg to ...) at example1/escape_example.go:9:13
escape_example.go:9:13:         from *(... argument) (indirection) at example1/escape_example.go:9:13
escape_example.go:9:13:         from ... argument (passed to call[argument content escapes]) at examp
escape_example.go:9:13: main ... argument does not escape
```

18

# Main Causes of Heap Allocation

- indirect assignment
- indirect call
- slice and map assignment
- interfaces
- ...

19

# Indirect Assignment 1

```
type x struct {
    data *string
}
```

```
func indirect() {
    str := "xxx"
    o := new(x)
    o.data = &str //BAD
}
```

```
$ go build -gcflags '-m -m' ./assignment.go
```

```
./assignment.go:11:11: &str escapes to heap
./assignment.go:11:11:  from o.data (star-dot-equals) at ./assignment.go:11:9
./assignment.go:9:2: moved to heap: str
./assignment.go:10:10: indirect new(x) does not escape
```

20

# Indirect Assignment 2

```
func direct() {
    str := "xxx"
    o := &x{
        data: &str, //OK
    }
    _ = o
}
```

```
$ go build -gcflags '-m -m' ./assignment.go
```

```
./assignment.go:18:9: direct &str does not escape
./assignment.go:18:3: direct &x literal does not escape
```

21

# Indirect Assignment 3

```
func BenchmarkAssignmentIndirect(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var i int
        x := &X{}
        x.p = &i // BAD: Cause of i escape
    }
}
```

```
func BenchmarkAssignmentDirect(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var i int
        x := &X{
            p: &i, // GOOD: i does not escape
        }
        _ = x
    }
}
```

```
$ go test -benchmem -run=none -bench Assignment -memprofile mem.out
```

```
BenchmarkAssignmentIndirect-8          100000000               14.0 ns/op             8 B/op
BenchmarkAssignmentDirect-8            2000000000               0.27 ns/op            0 B/op
```

22

# Indirect Assignment 4

```
$ go tool pprof -alloc_space mem.out
```

```
(pprof) list example2.BenchmarkAssignmentIndirect
Total: 747.51MB
ROUTINE ======================== _/home/kostik/Code/gosrc/escape/example2.BenchmarkAssignmentIndirec
  747.51MB   747.51MB (flat, cum)   100% of Total
         .          .      6:   p *int
         .          .      7:}
         .          .      8:
         .          .      9:func BenchmarkAssignmentIndirect(b *testing.B) {
         .          .     10:   for i := 0; i < b.N; i++ {
  747.51MB   747.51MB     11:           var i int
         .          .     12:           x := &X{}
         .          .     13:           x.p = &i // BAD: Cause of i escape
         .          .     14:   }
         .          .     15:}
         .          .     16:
```

23

# Indirect Calls

```
package example3

type X struct {
    c int
}

func (x *X) add(i int) {
    x.c += i
}

func indirectCall() {
    x := new(X)
    f := x.add // BAD
    f(5)

    //x.add(5) //GOOD
}
```

```
$ go build -gcflags '-m -m' ./indirect_call.go
```

```
./indirect_call.go:12:10: new(X) escapes to heap
./indirect_call.go:12:10:       from x (assigned) at ./indirect_call.go:12:4
./indirect_call.go:12:10:       from x.add (call part) at ./indirect_call.go:13:8
```

24

# Map & Slice Assignment

- related to indirect assignment case
- size of a slice/map is not fixed

25

# Maps Assignment

```
type x struct {
    data string
}
```

```
func BenchmarkMap(b *testing.B) {
    for i := 0; i < b.N; i++ {
        c := new(x)
        m := make(map[string]*x, 0)
        m["foo"] = c
    }
}
```

```
$ go test -gcflags '-m -m' ./map_and_slice.go
```

```
./map_and_slice.go:13:11: new(x) escapes to heap
./map_and_slice.go:13:11:       from c (assigned) at ./map_and_slice.go:13:5
./map_and_slice.go:13:11:       from m["foo"] (value of map put) at ./map_and_slice.go:15:12
```

26

# Slice Assignment 1

```
func BenchmarkSlice(b *testing.B) {
    for i := 0; i < b.N; i++ {
        c := new(x)
        s := make([]*x, 1)
        s[0] = c
    }
}
```

```
$ go test -gcflags '-m -m' ./map_and_slice.go
```

```
./map_and_slice.go:21:11: new(x) escapes to heap
./map_and_slice.go:21:11:       from c (assigned) at ./map_and_slice.go:21:5
./map_and_slice.go:21:11:       from s[0] (slice-element-equals) at ./map_and_slice.go:23:8
```

27

# Slice Assignment 2

## Not about escape analysis

```
package example4

func sliceAppend(count int) []int {
    out := []int{}
    for j := 0; j < count; j++ {
        out = append(out, j)
    }

    return out
}

func sliceIndex(count int) []int {
    out := make([]int, count)
    for j := 0; j < count; j++ {
        out[j] = j
    }
    return out
}
```

28

```
BenchmarkAppend-8        3000000               567 ns/op            2040 B/op          8 allocs/op
BenchmarkArray-8        10000000               182 ns/op             896 B/op          1 allocs/op
```

# Interfaces

- can't define exact type at compile time
- good idea to use code generation (i.e. easyjson)

29

# Profiling Tools

## Quick List

- go test framework (benchmarks)
- pprof (mem usage, cpu utilisation, goroutine traces, etc.. )
- benchstat

31

# Conclusion

- avoid premature optimization
- don't guess but measure
- more likely, performance problems caused by heap allocations

32

# Links

- https://docs.google.com/document/d/1CxgUBPlx9iJzkz9JWkb6tIpTe5q32QDmz8l0BouG0Cw/edit#heading=h.llaiaboyeyo3
- https://github.com/kshamko/allocations_talk

33

# Thank you

Kanstantsin Shamko
Erlang developer at IDT (https://www.idt.net/)