How to avoid Go gotchas

by learning internals

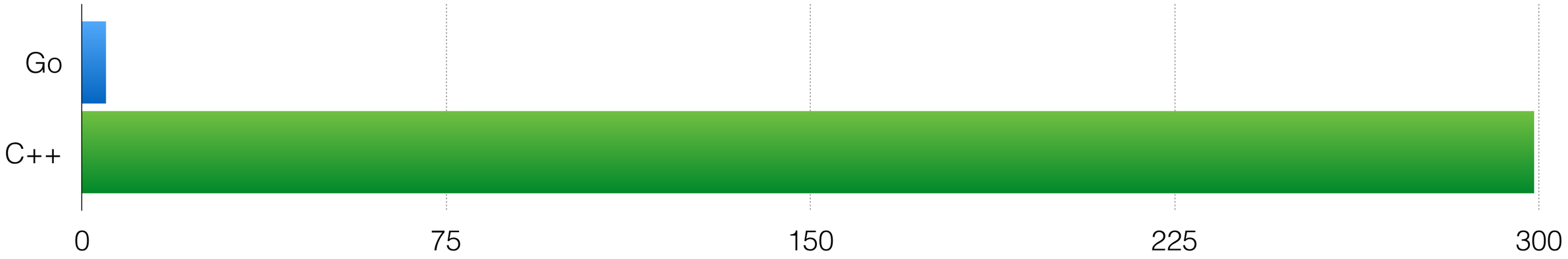
Ivan Danyliuk, Golang BCN meetup
2 Nov 2016

Gotchas

- Go has some gotchas
- Good examples:
  - 50 Shades of Go: Traps, Gotchas, and Common Mistakes for New Golang Devs
  - Go Traps
  - Golang slice append gotcha

Gotchas

- Luckily, Go has very few gotchas
- Especially in comparison with other languages



Gotchas

- So, what is gotcha?
- “a gotcha is a **valid construct** in a system, program or programming language that works as documented but **is counter-intuitive** and almost invites mistakes because it is both easy to invoke and unexpected or unreasonable in its outcome”

Gotchas

- Two solutions:
  - “fix” the language
  - fix the intuition.
- Let’s build some intuition to fight gotchas then.

Gotchas

- Let’s learn some internals and in memory representations
- It worked for me, should work for you as well.

basic types

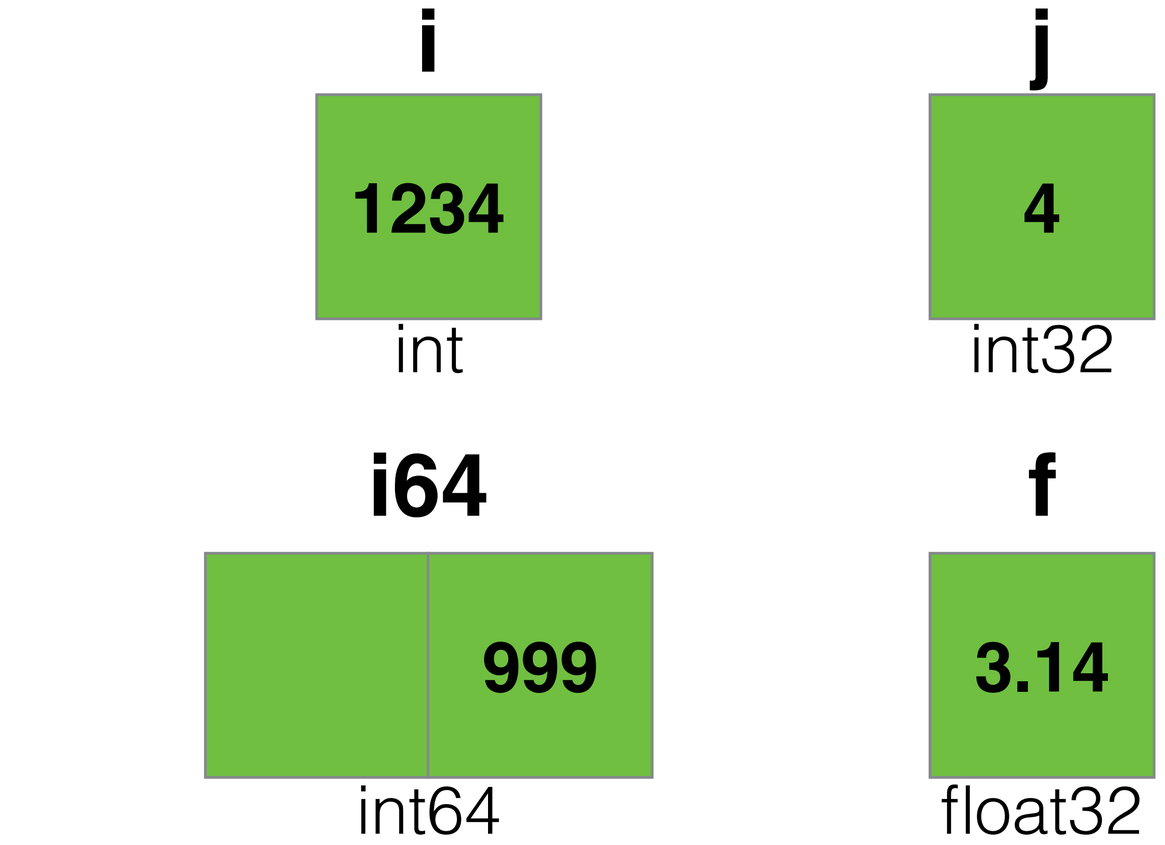
Code:

```
i := 1234
j := int32(4)
i64 := int64(999)
f := float32(3.14)
```

basic types

Code:

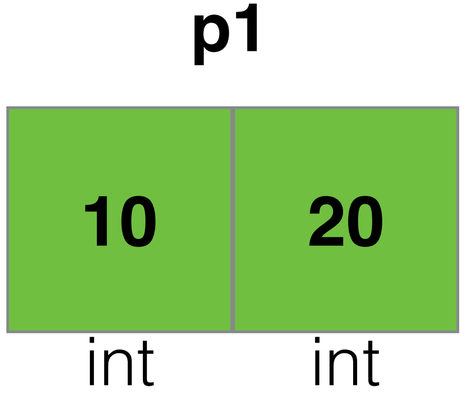
```
i := 1234
j := int32(4)
i64 := int64(999)
f := float32(3.14)
```



structs

Code:

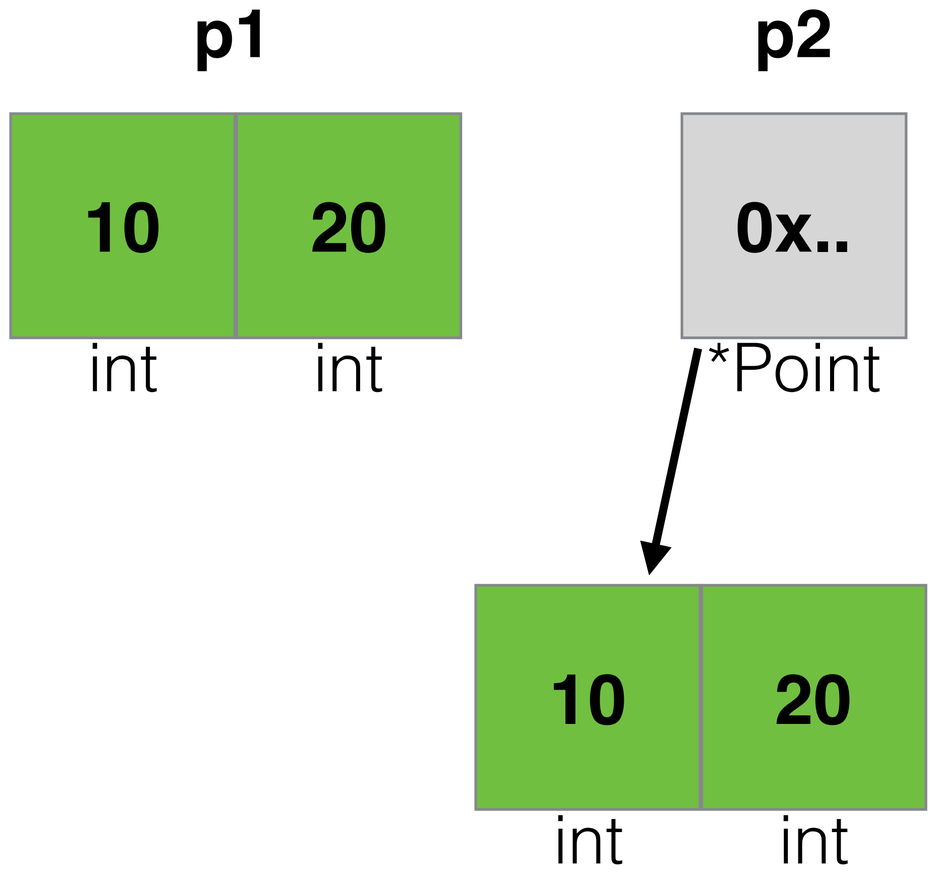
```
type Point struct {
    X, Y int
}

p1 := Point{10, 20}
```



structs

Code:

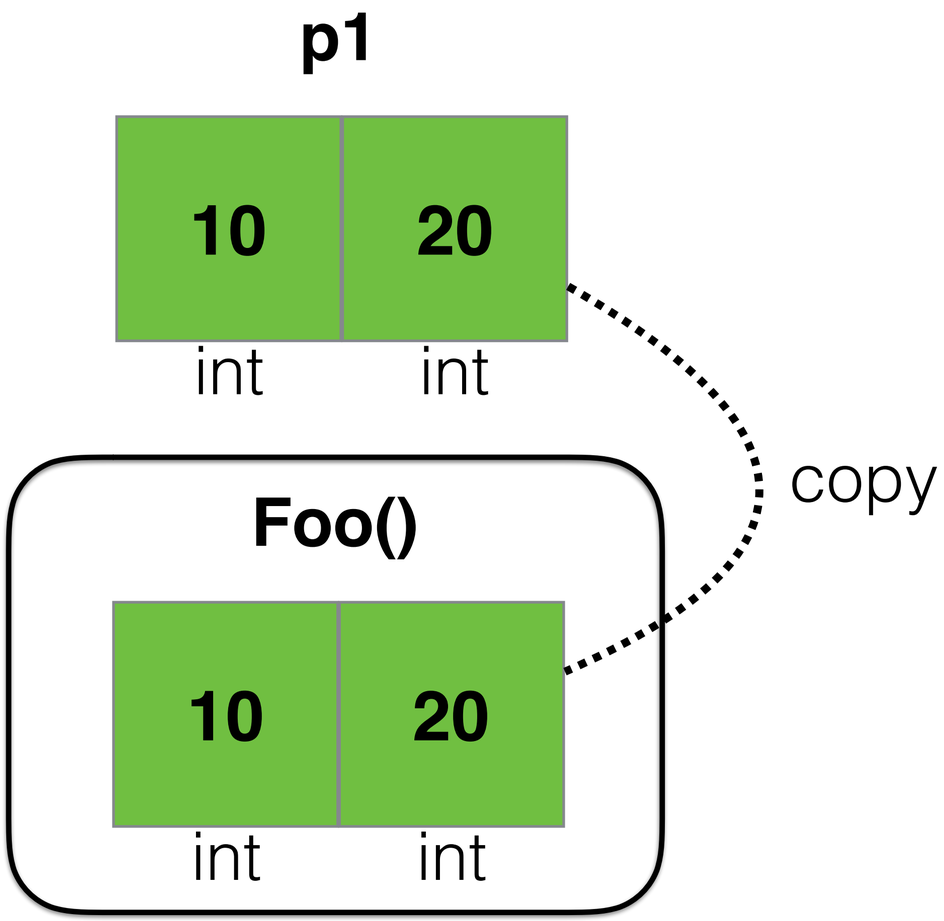
```
type Point struct {
    X, Y int
}

p1 := Point{10, 20}
p2 := &Point{10, 20}
```



structs

Code:

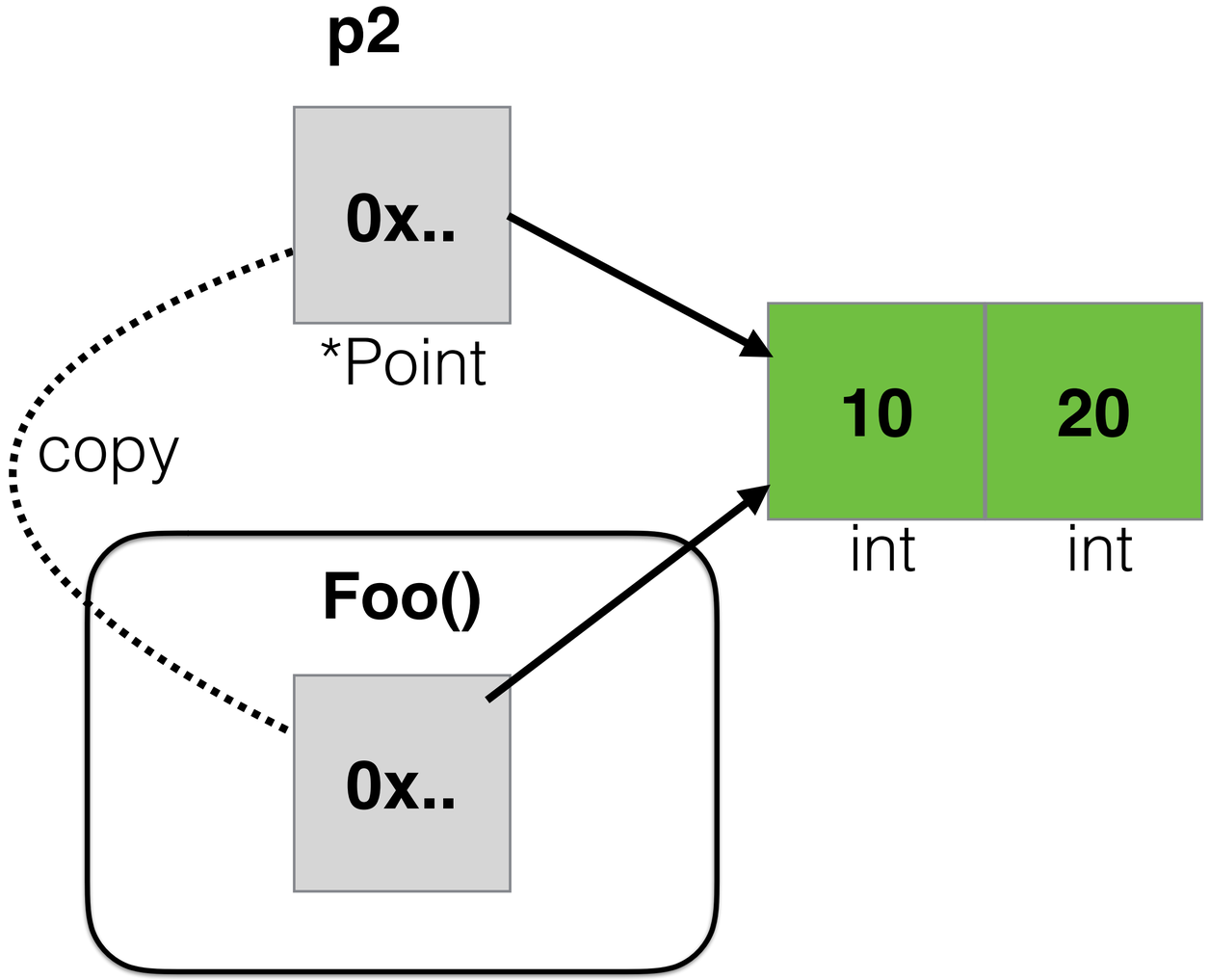
```
func Foo(p Point) {
    // ...
}

p1 := Point{10, 20}
Foo(p1)
```



structs

Code:

```
func Foo(p *Point) {
    // ...
}

p2 := &Point{10, 20}
Foo(p2)
```

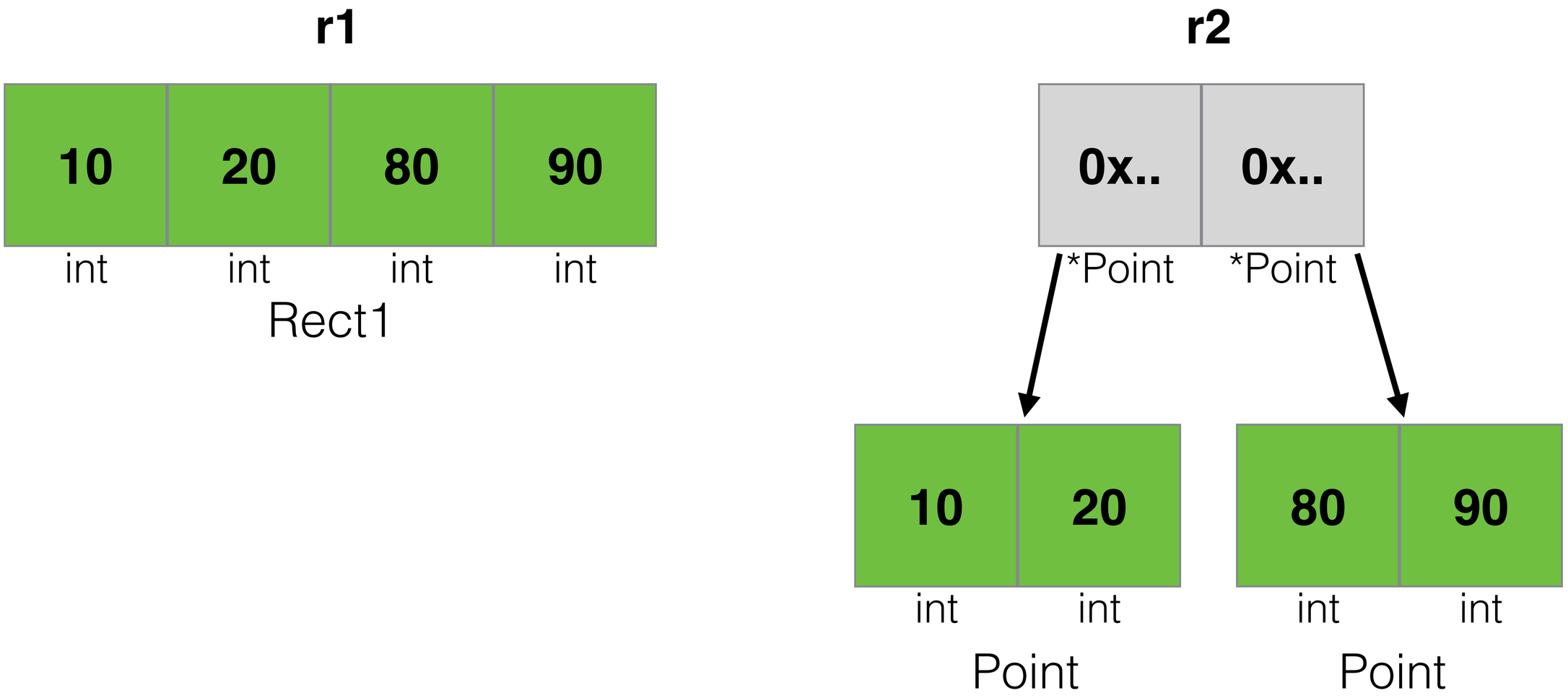


structs

```
type Rect1 struct {
    Min, Max Point
}

r1 := Rect1{
    Min: Point{10, 20},
    Max: Point{80, 90}
}
```

```
type Rect2 struct {
    Min, Max *Point
}

r2 := Rect2{
    Min: &Point{10, 20},
    Max: &Point{80, 90}
}
```

structs


r1
10
20
80
90
int
int
int
int
Rect1
r2
0x..
0x..
*Point
*Point
10
20
80
90
int
int
int
int
Point
Point

array

Code:

```
var arr [5]int
```

array

Code:

```
var arr [5]int
```

Go code: src/runtime/malloc.go

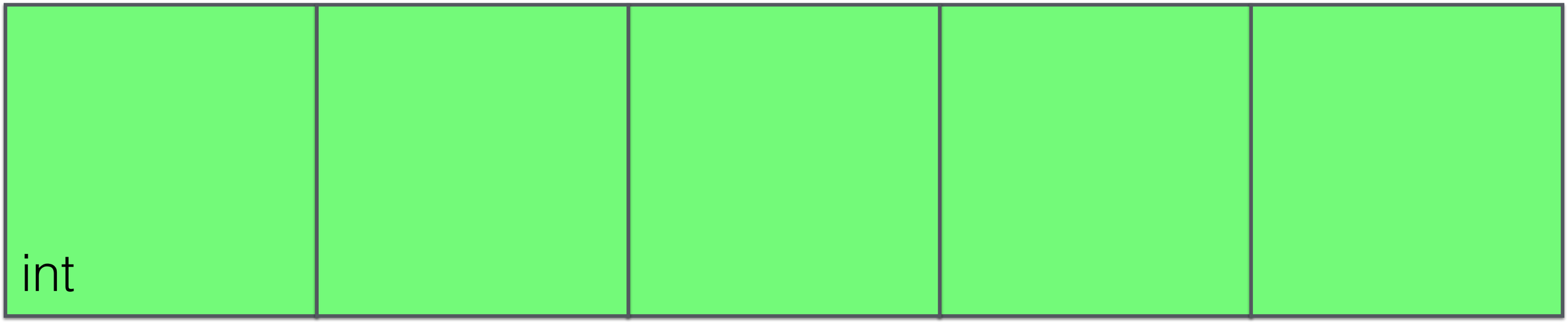
```
// newarray allocates an array of n elements of type typ.
func newarray(typ *_type, n int) unsafe.Pointer {
    if n < 0 || uintptr(n) > maxSliceCap(typ.size) {
        panic(plainError("runtime: allocation size out of range"))
    }
    return mallocgc(typ.size*uintptr(n), typ, true)
}
```

array

Code:

```
var arr [5]int
```

Memory:



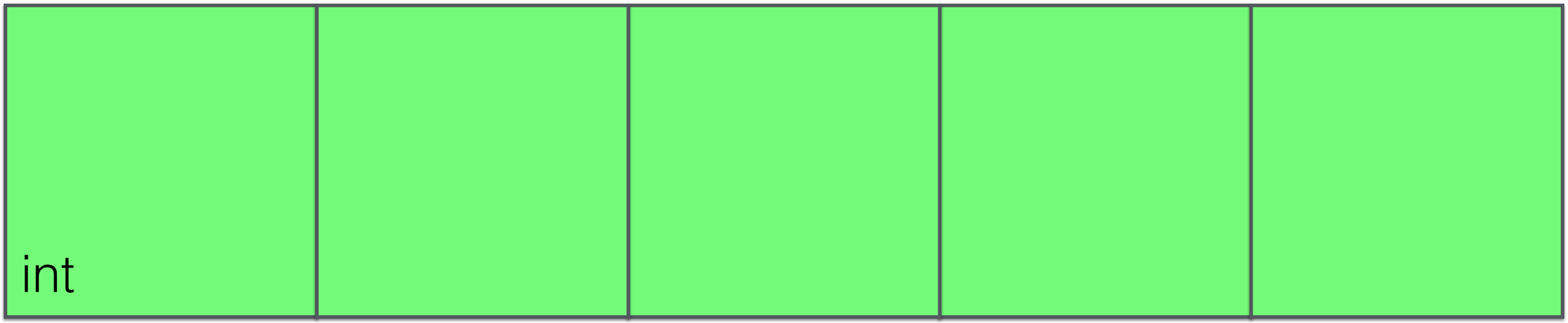
array

Code:

```
var arr [5]int
```

4 or 8 bytes blocks
(32 or 64 bit arch)

Memory:

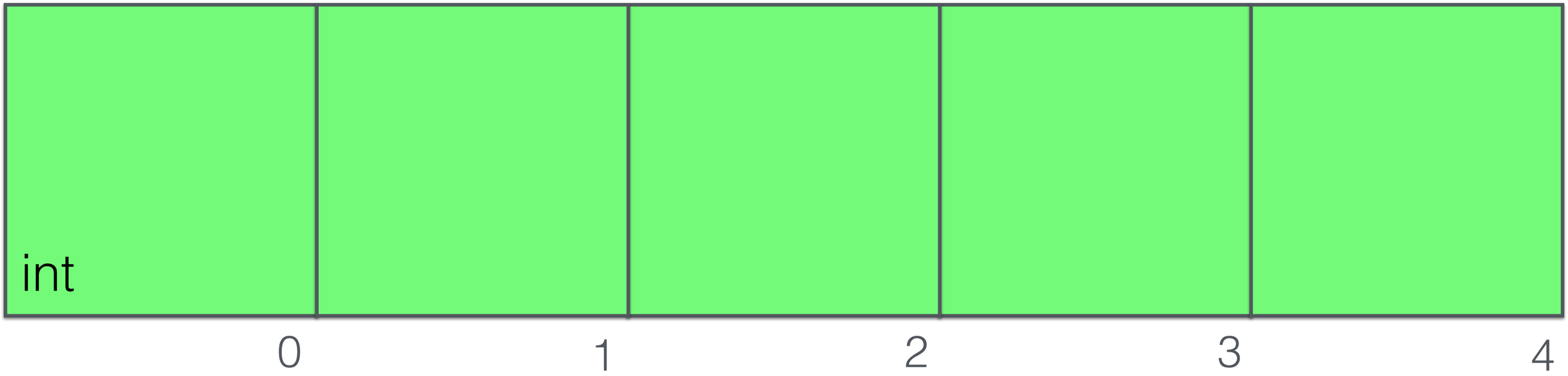


array

Code:

```
var arr [5]int
```

Memory:

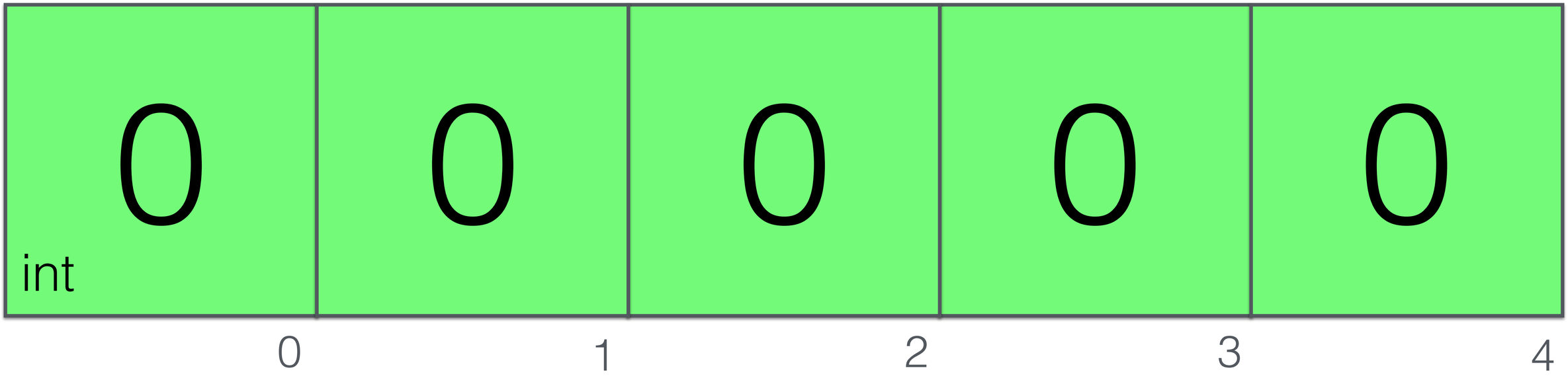


array

Code:

```
var arr [5]int
```

Memory:

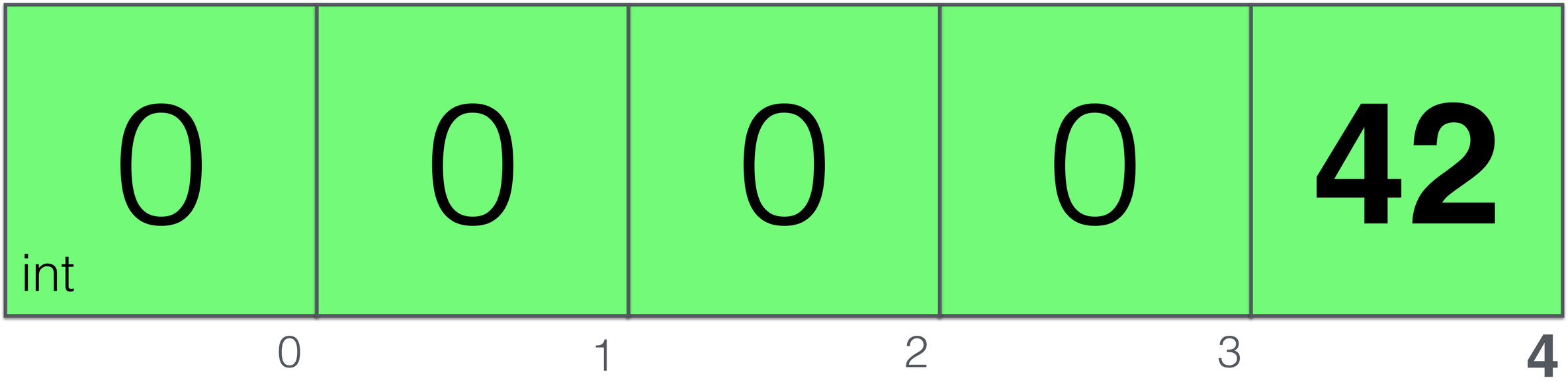


array

Code:

```
var arr [5]int
arr[4] = 42
```

Memory:



slice

Code:

```
var foo []int
```

slice

Code:

```
var foo []int
```

Go code: src/runtime/slice.go

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

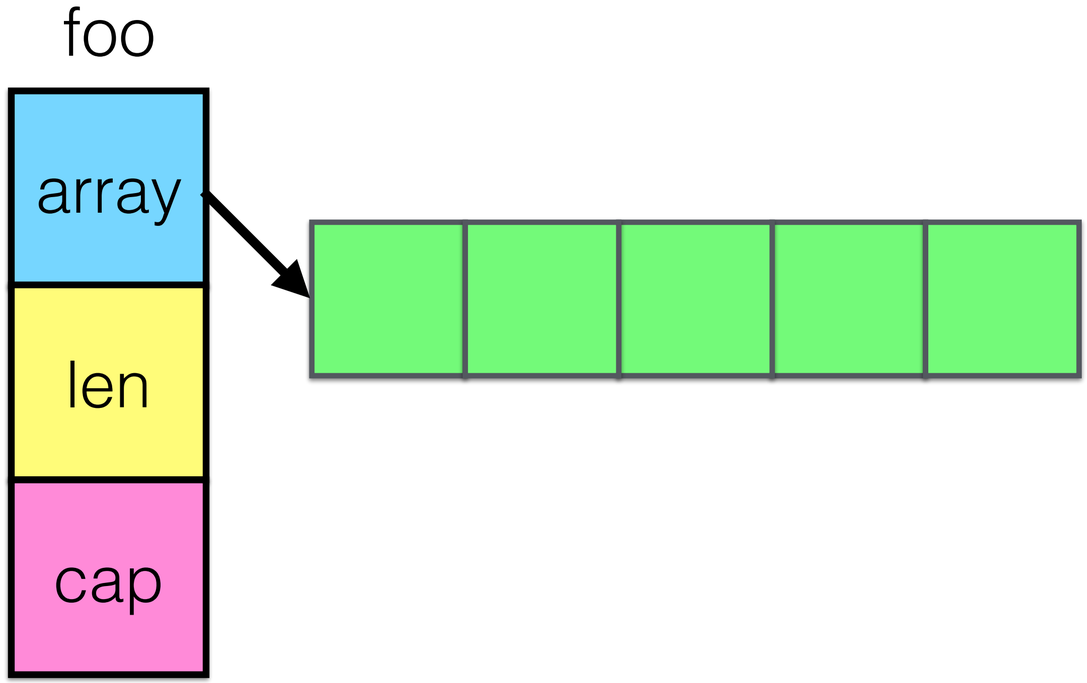
slice

Code:

```
var foo []int
```

Go code: src/runtime/slice.go

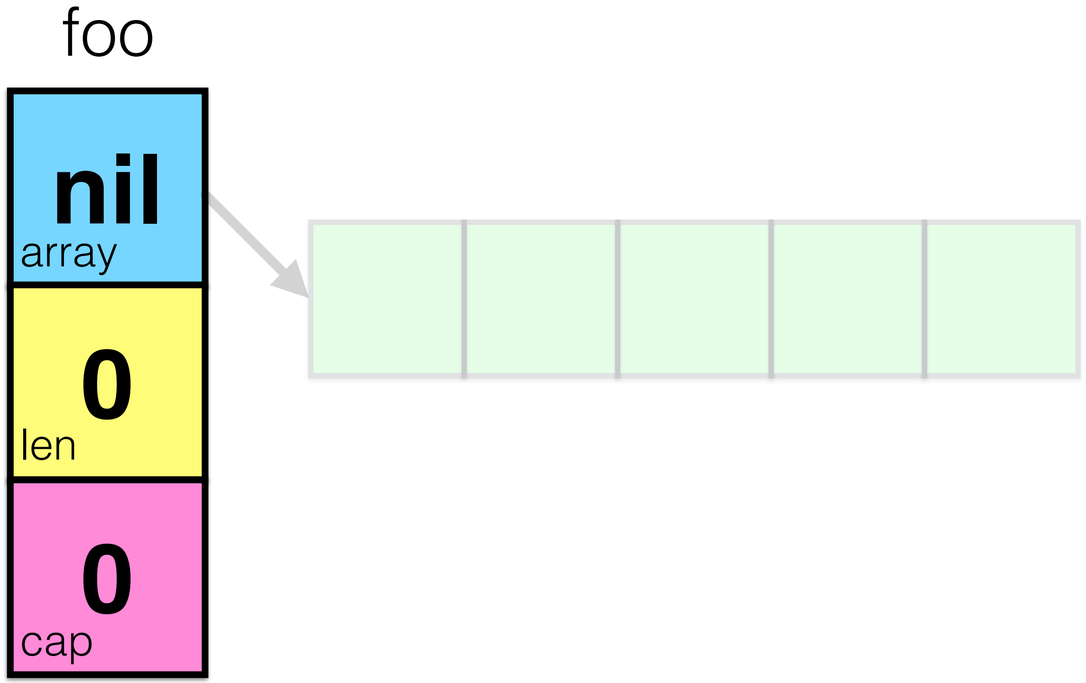
```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```



slice

Code:

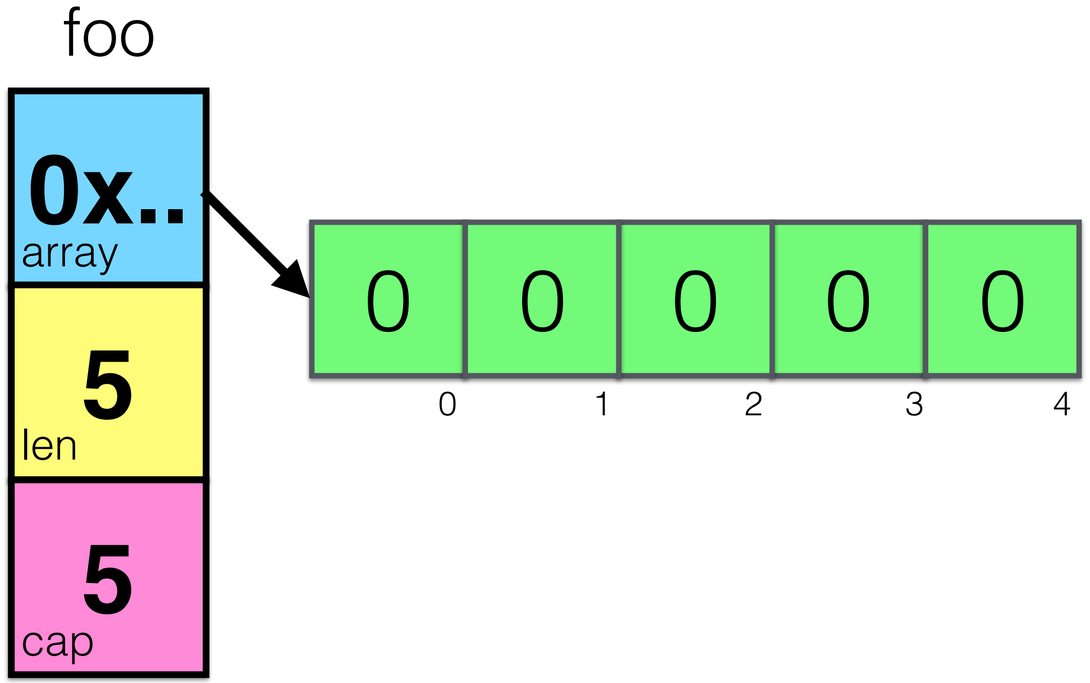
```
var foo []int
```



slice

Code:

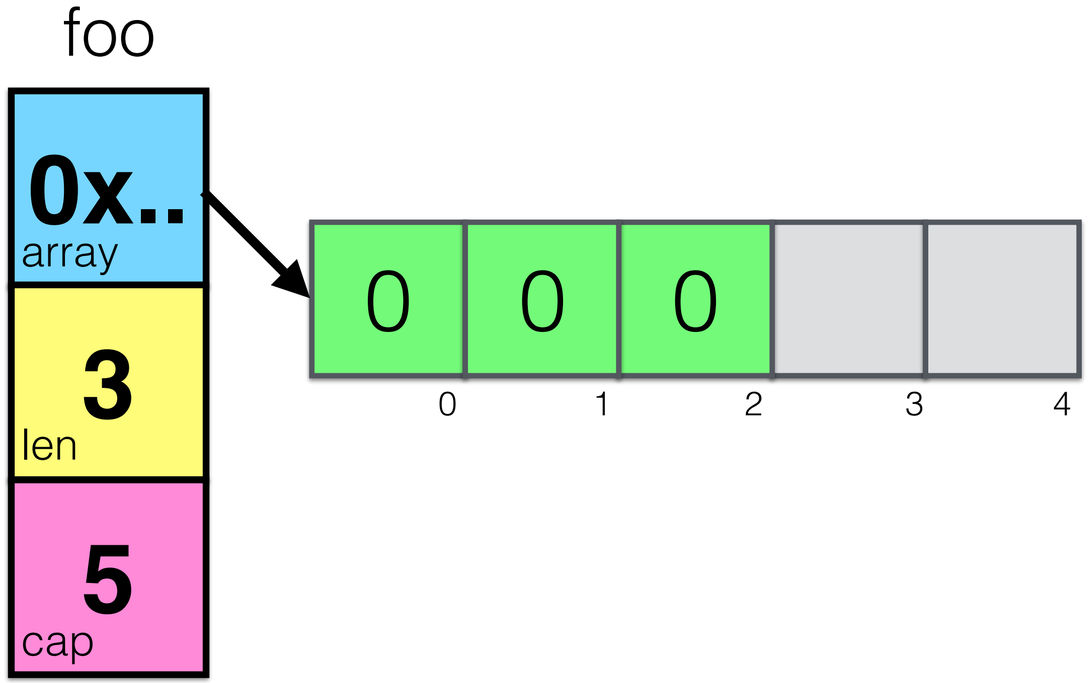
```
var foo []int
foo = make([]int, 5)
```



slice

Code:

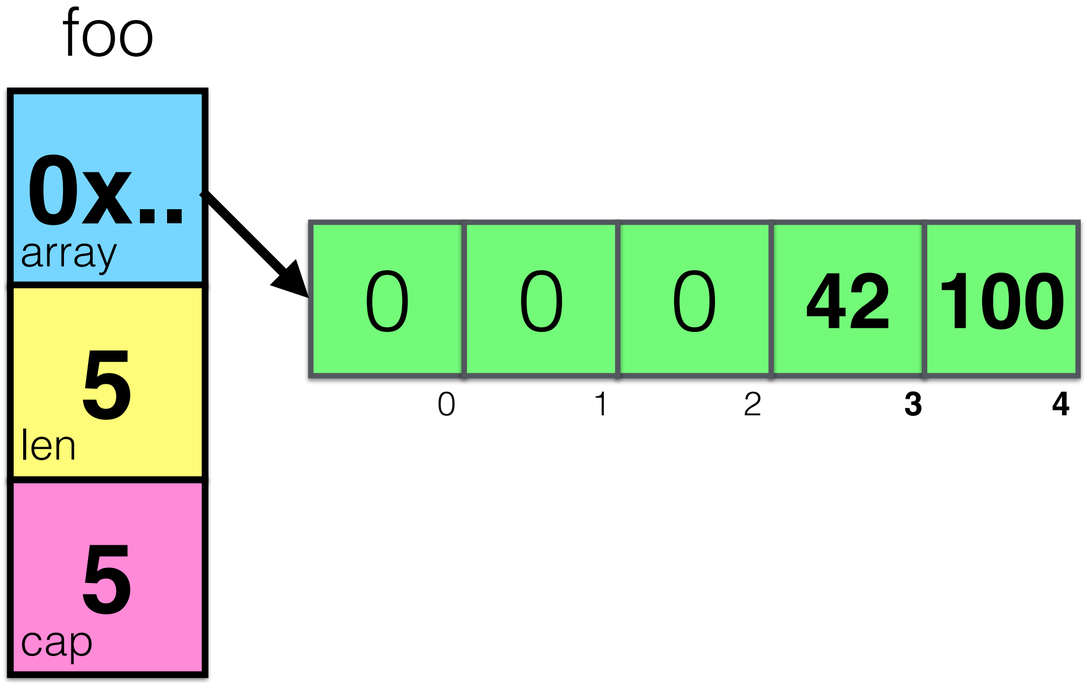
```
var foo []int
foo = make([]int, 3, 5)
```



slice

Code:

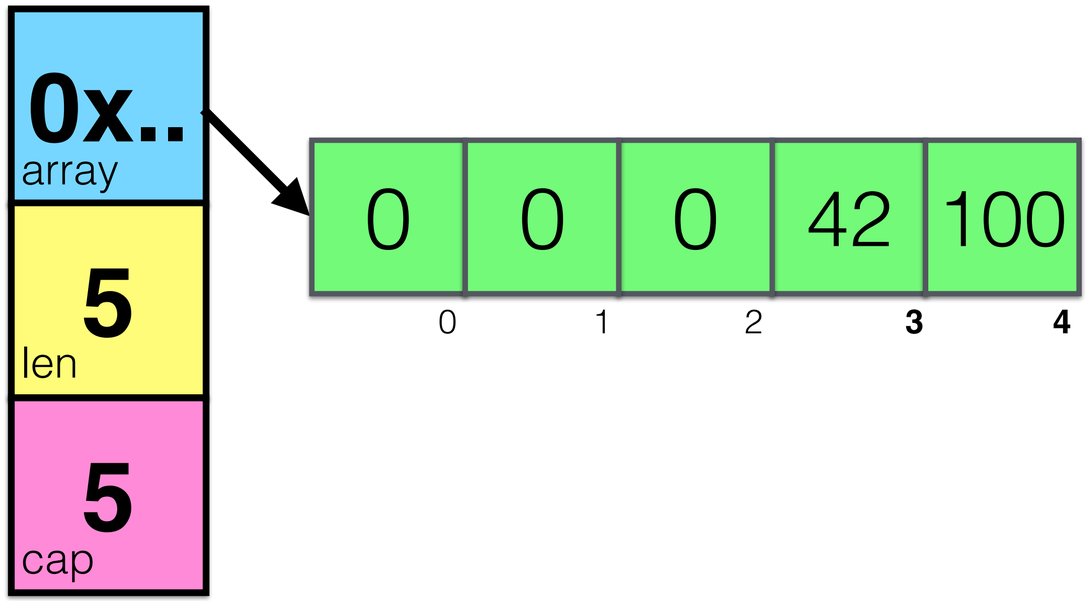
```
var foo []int
foo = make([]int, 5)
foo[3] = 42
foo[4] = 100
```



slice

Code:

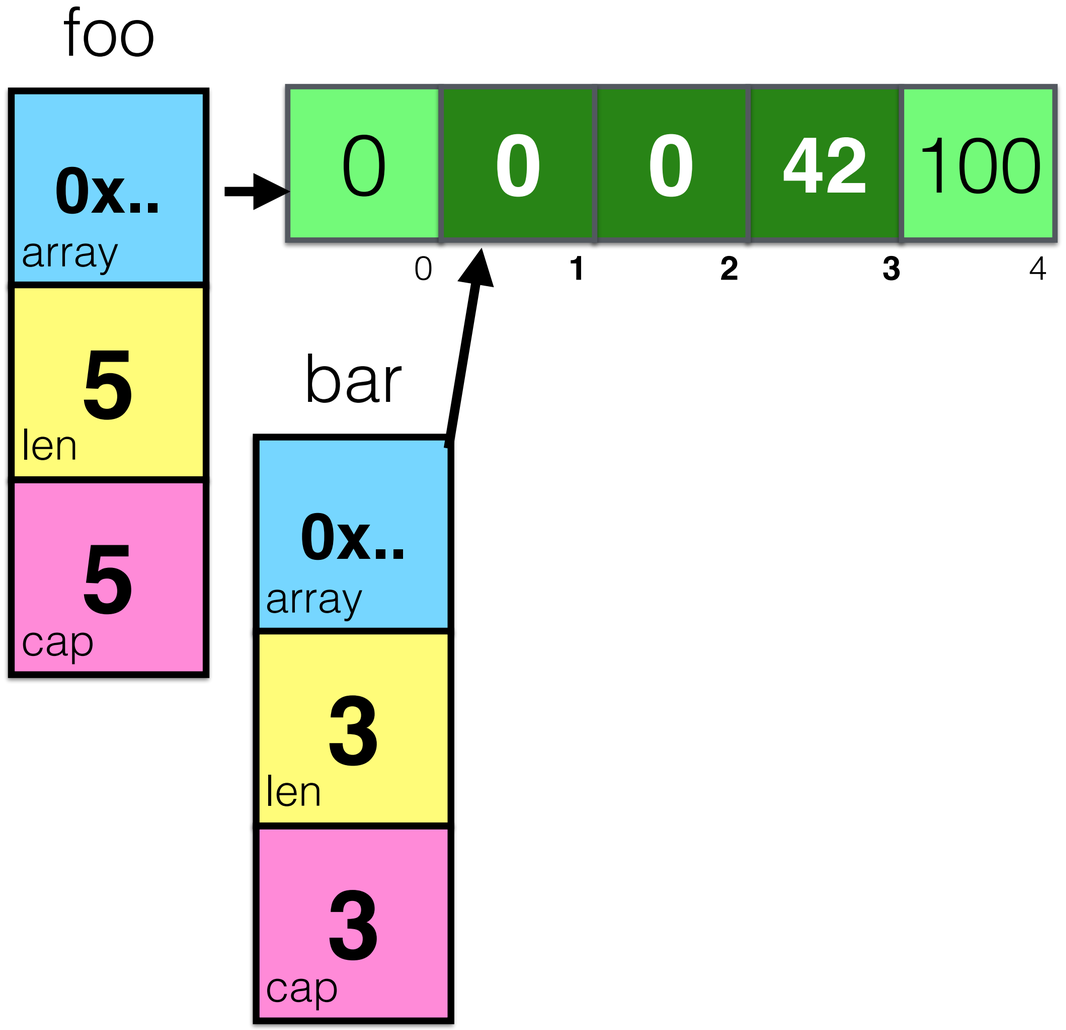
```
var foo []int
foo = make([]int, 5)
foo[3] = 42
foo[4] = 100

bar := foo[1:4]
```



slice

Code:

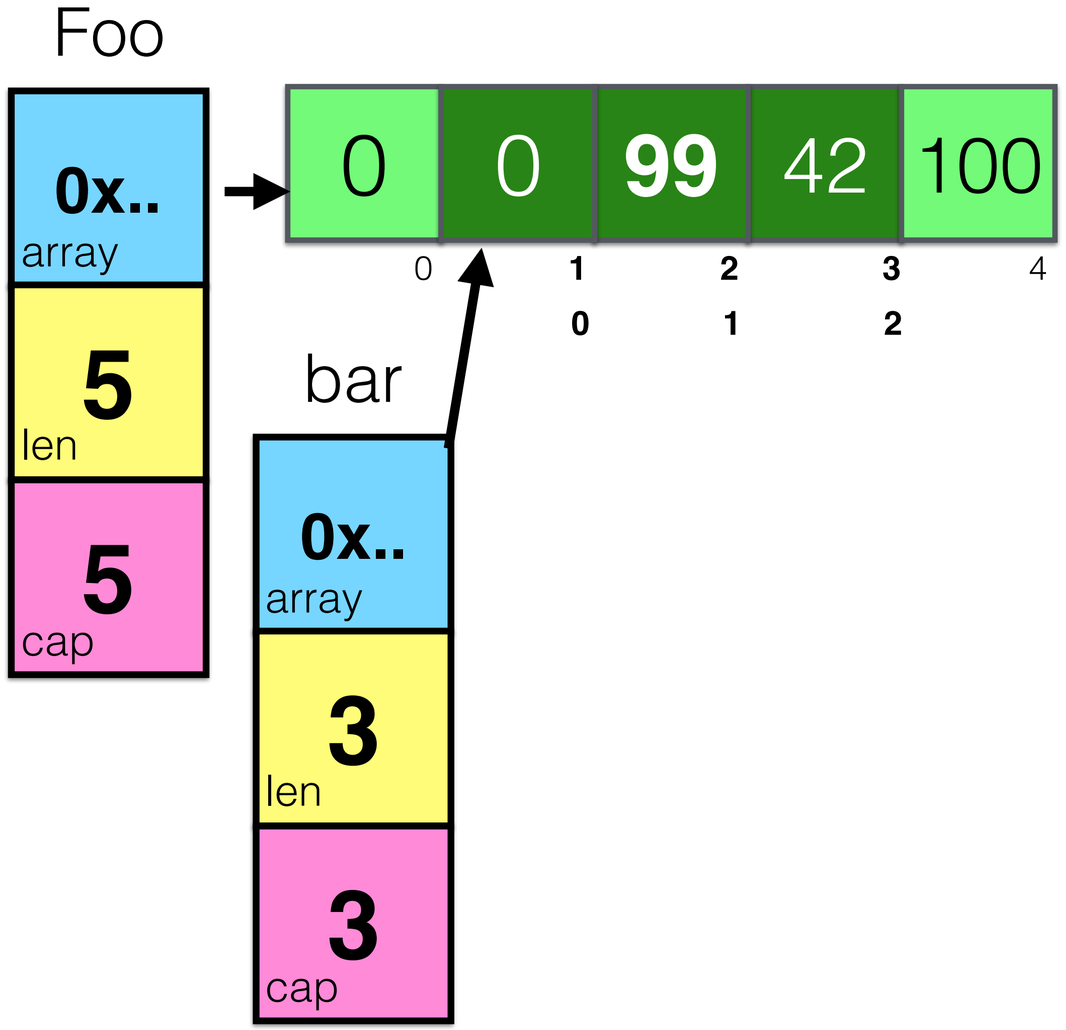
```
var foo []int
foo = make([]int, 5)
foo[3] = 42
foo[4] = 100

bar := foo[1:4]
```



slice

Code:

```
var foo []int
foo = make([]int, 5)
foo[3] = 42
foo[4] = 100

bar := foo[1:4]
bar[1] = 99
```

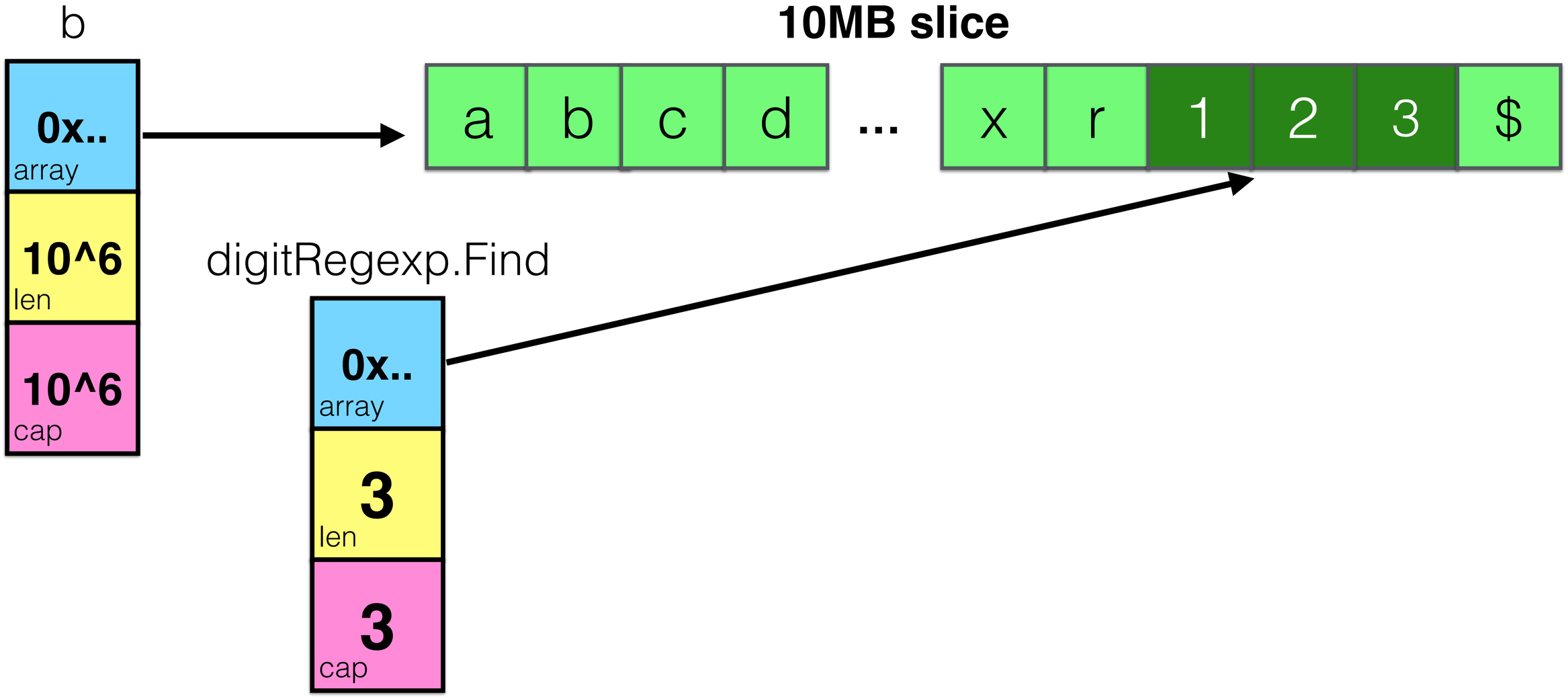


slice

Code:

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

slice


b
0x..
array
10^6
len
10^6
cap
10MB slice
a
b
c
d
...
x
r
1
2
3
$
digitRegexp.Find
0x..
array
3
len
3
cap

append

Code:

```
a := make([]int, 32)
a = append(a, 1)
```

append

Code:

```
a := make([]int, 32)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

What will be the output?

append

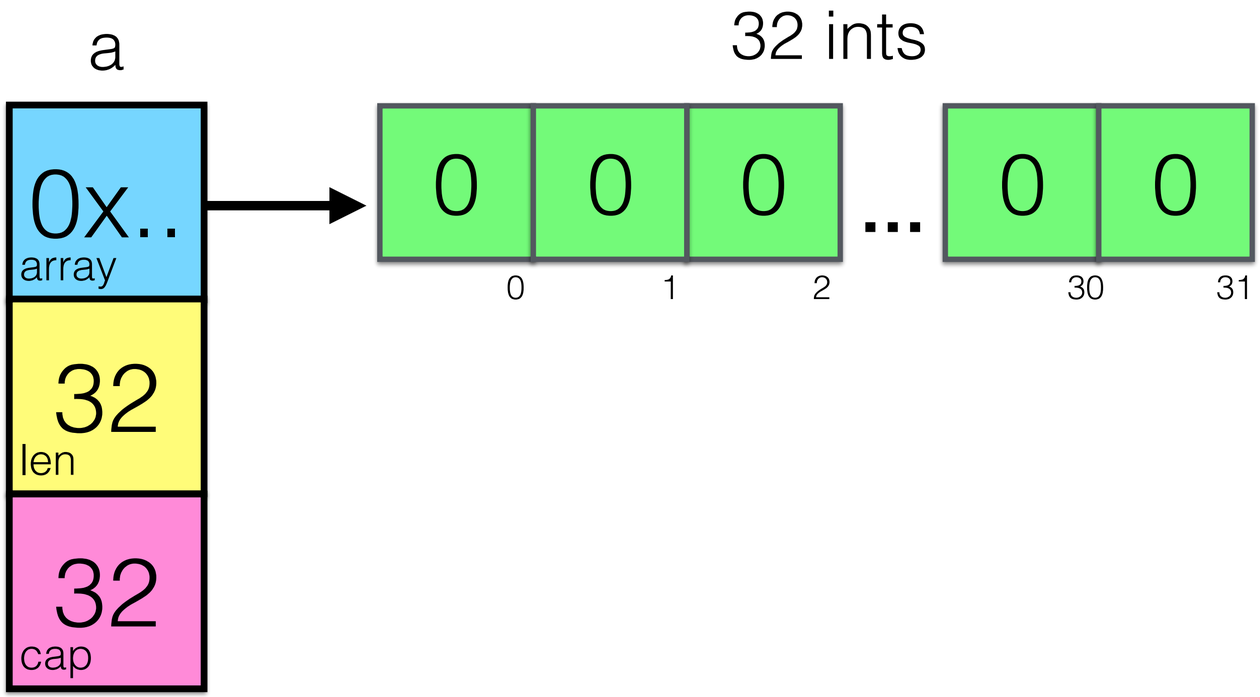
Code:

```
a := make([]int, 32)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

Output:

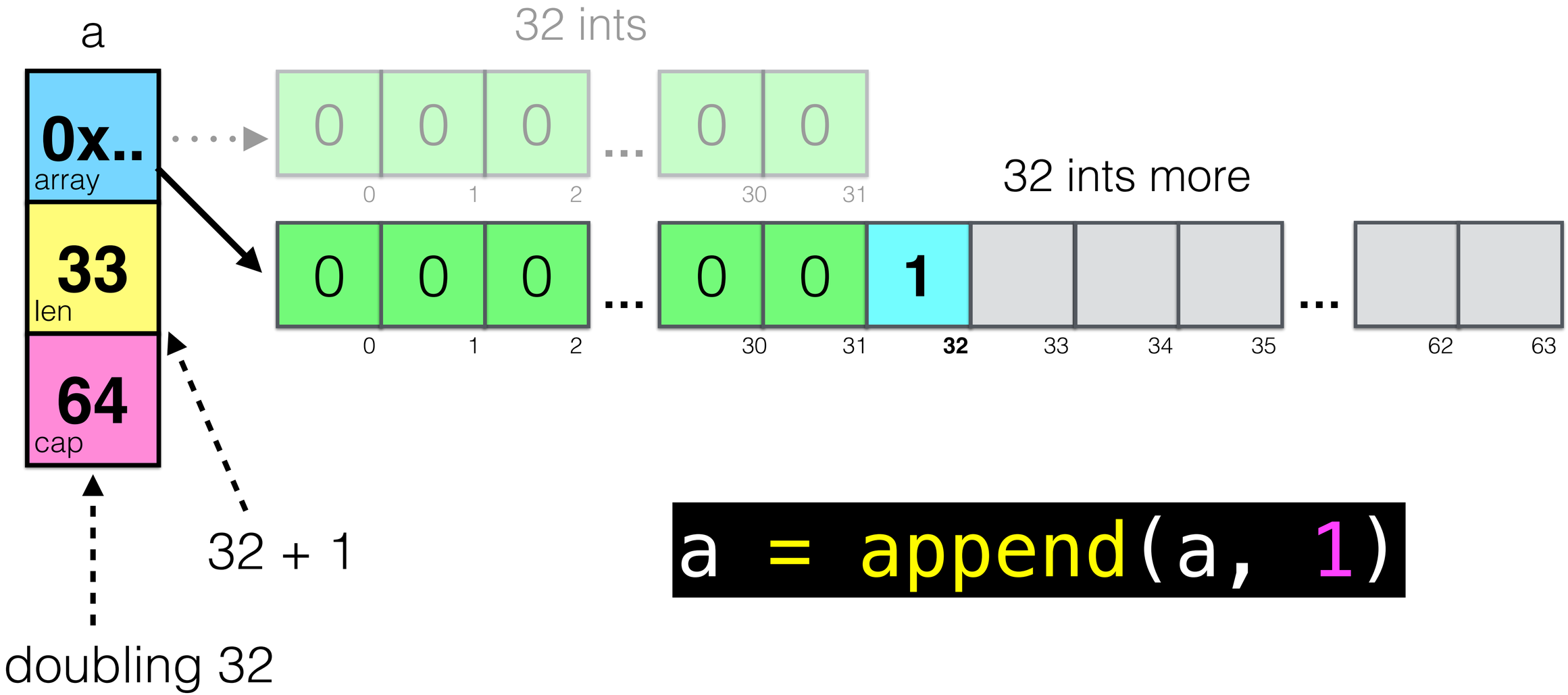
```
len: 33 cap: 64
```

append



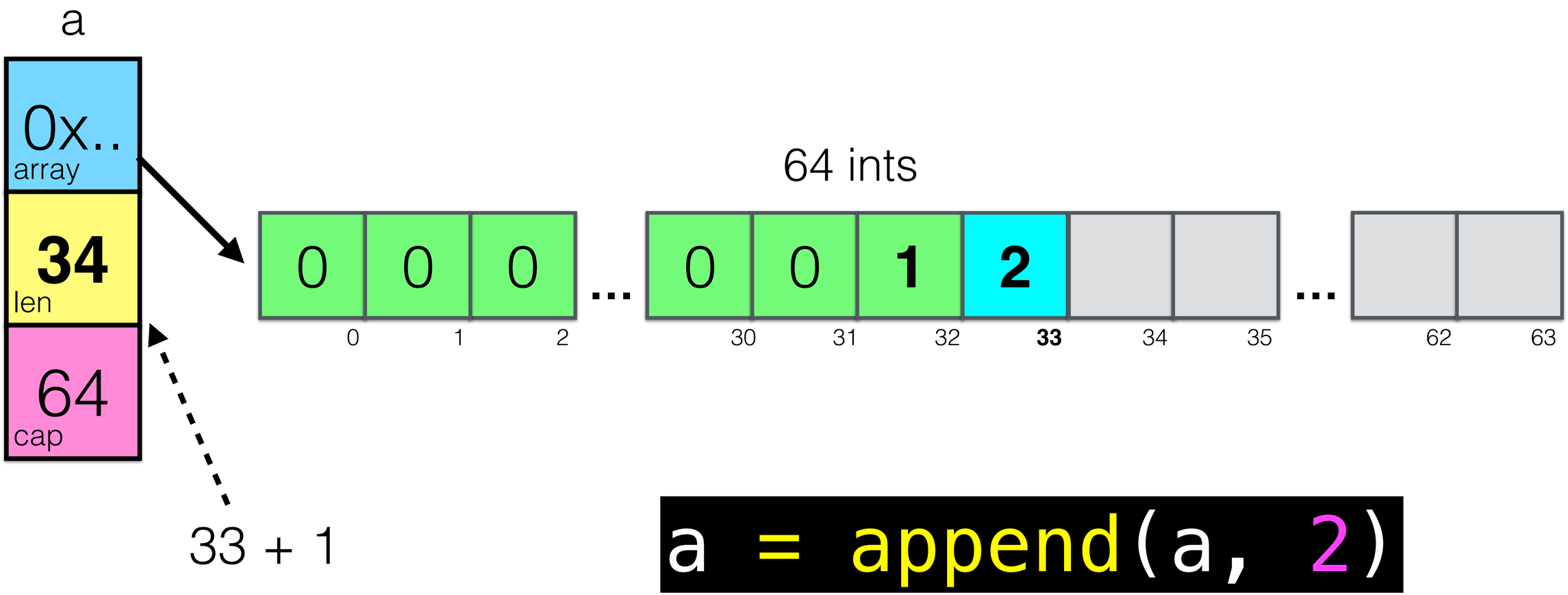

```
a = append(a, 1)
```

append




```
a = append(a, 1)
```

append




```
a = append(a, 2)
```

append

Code:

```
a := make([]int, 1023)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

Output:

What will be the output?

append

Code:

```
a := make([]int, 1023)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

Output:

```
len: 1024 cap: 2048
```

append

Code:

```
a := make([]int, 1024)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

What will be the output?

append

Code:

```
a := make([]int, 1024)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

Output:

```
len: 1025 cap: 1312
```

append

Code:

```
a := make([]int, 1024)
a = append(a, 1)
fmt.Println("len:", len(b), "cap:", cap(b))
```

Output:

```
len: 1025 cap: 1312
```

WTF?

append

Go code: src/runtime/slice.go

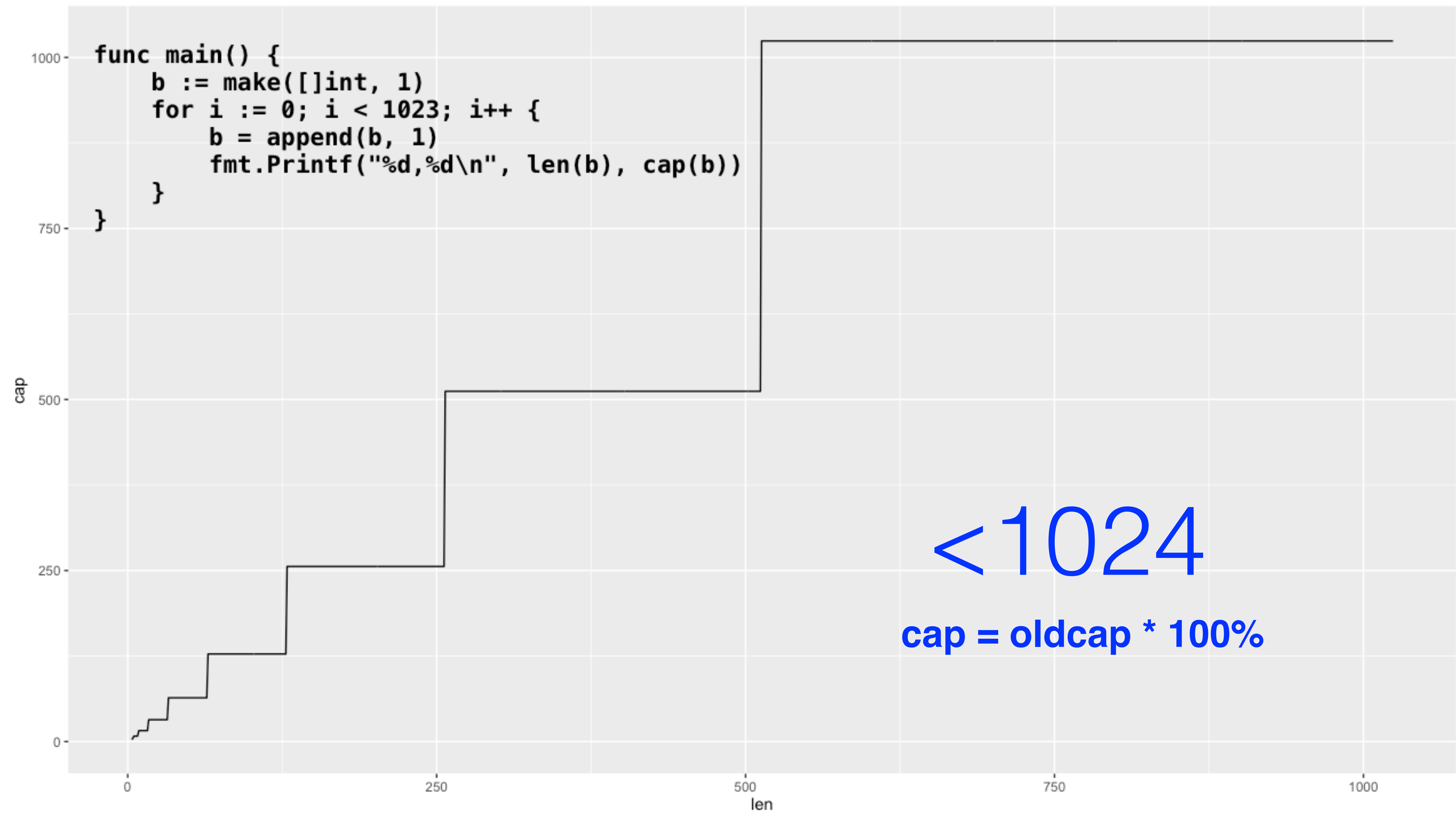
```
func growslice(et *_type, old slice, cap int) slice {
    ...
    if old.len < 1024 {
        newcap = doublecap
    } else {
        for newcap < cap {
            newcap += newcap / 4
        }
    }
    ...
    capmem = roundupsize(uintptr(newcap))
    newcap = int(capmem)
}
```

append

Go code: src/runtime/msize.go

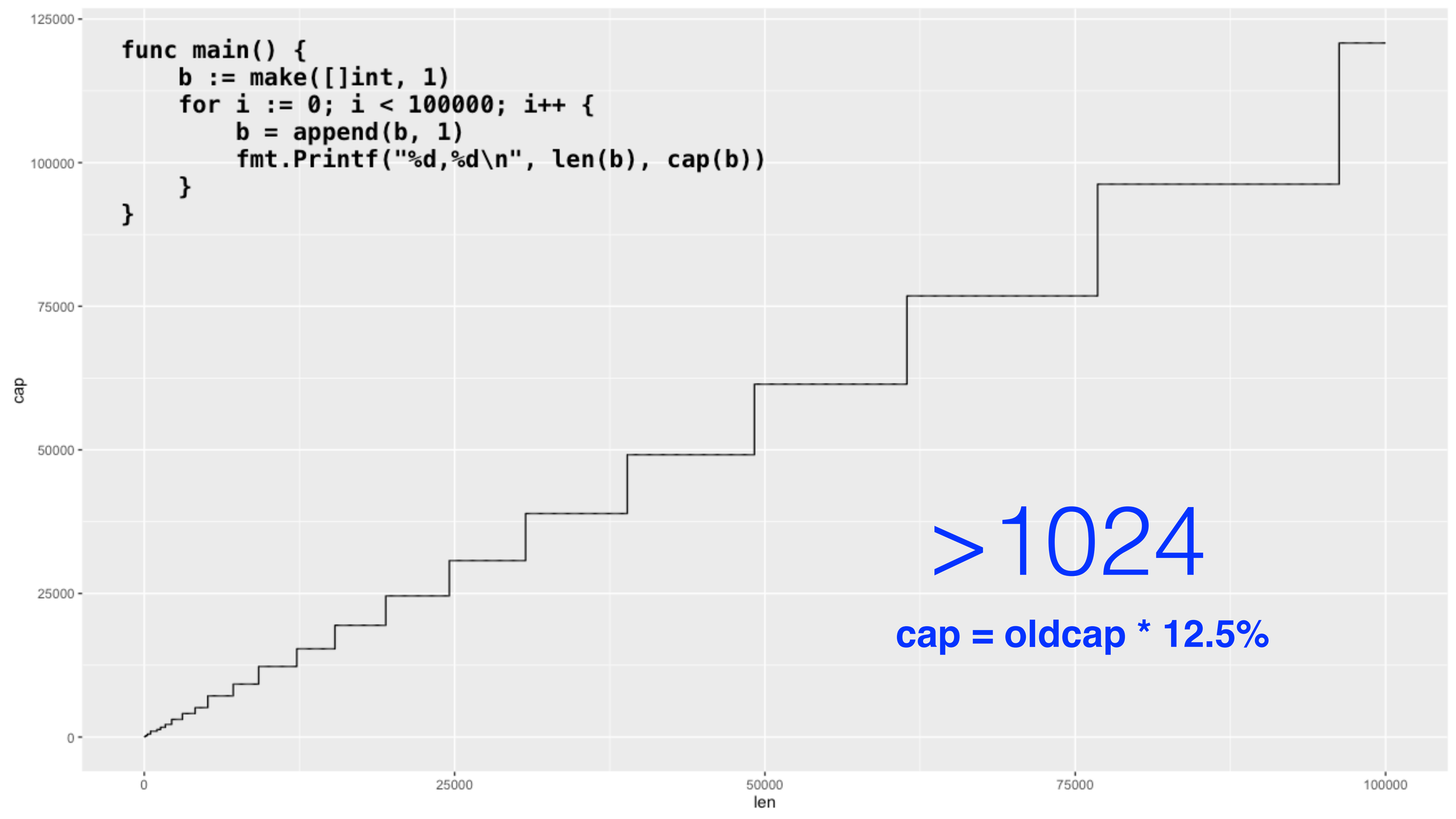
```
NumSizeClasses=67
0 8 16 32 48 64 80 96 112 128 144 160 176 192
208 224 240 256 288 320 352 384 416 448 480
512 576 640 704 768 896 1024 1152 1280 1408
1536 1664 2048 2304 2560 2816 3072 3328 4096
4608 5376 6144 6400 6656 6912 8192 8448 8704
9472  10496  12288  13568  14080  16384  16640
17664 20480 21248 24576 24832 28416 32768
```

```
func main() {
    b := make([]int, 1)
    for i := 0; i < 1023; i++ {
        b = append(b, 1)
        fmt.Printf("%d,%d\n", len(b), cap(b))
    }
}
```



<1024

cap = oldcap * 100%


func main() {
b := make([]int, 1)
for i := 0; i < 100000; i++ {
b = append(b, 1)
fmt.Printf("%d,%d\n", len(b), cap(b))
}
}
125000
100000
75000
50000
25000
0
cap
0
25000
50000
75000
100000
len
>1024
cap = oldcap * 12.5%

append

Code:

```
a := make([]int, 1)
b := append(a, 1)
b[0] = 100
c := append(a, 2)
c[0] = 200
d := append(a, 3, 4)
d[0] = 300
fmt.Println(a, b, c, d)
```

append

Code:

```
a := make([]int, 1)
b := append(a, 1)
b[0] = 100
c := append(a, 2)
c[0] = 200
d := append(a, 3, 4)
d[0] = 300
fmt.Println(a, b, c, d)
```

Output:

```
[0]
[100 1]
[200 2]
[300 3 4]
```

interfaces

Code:

```
type error interface {
    Error() string
}
```

interfaces

Code:

```
type error interface {
    Error() string
}
```

Go code: src/runtime/runtime2.go

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

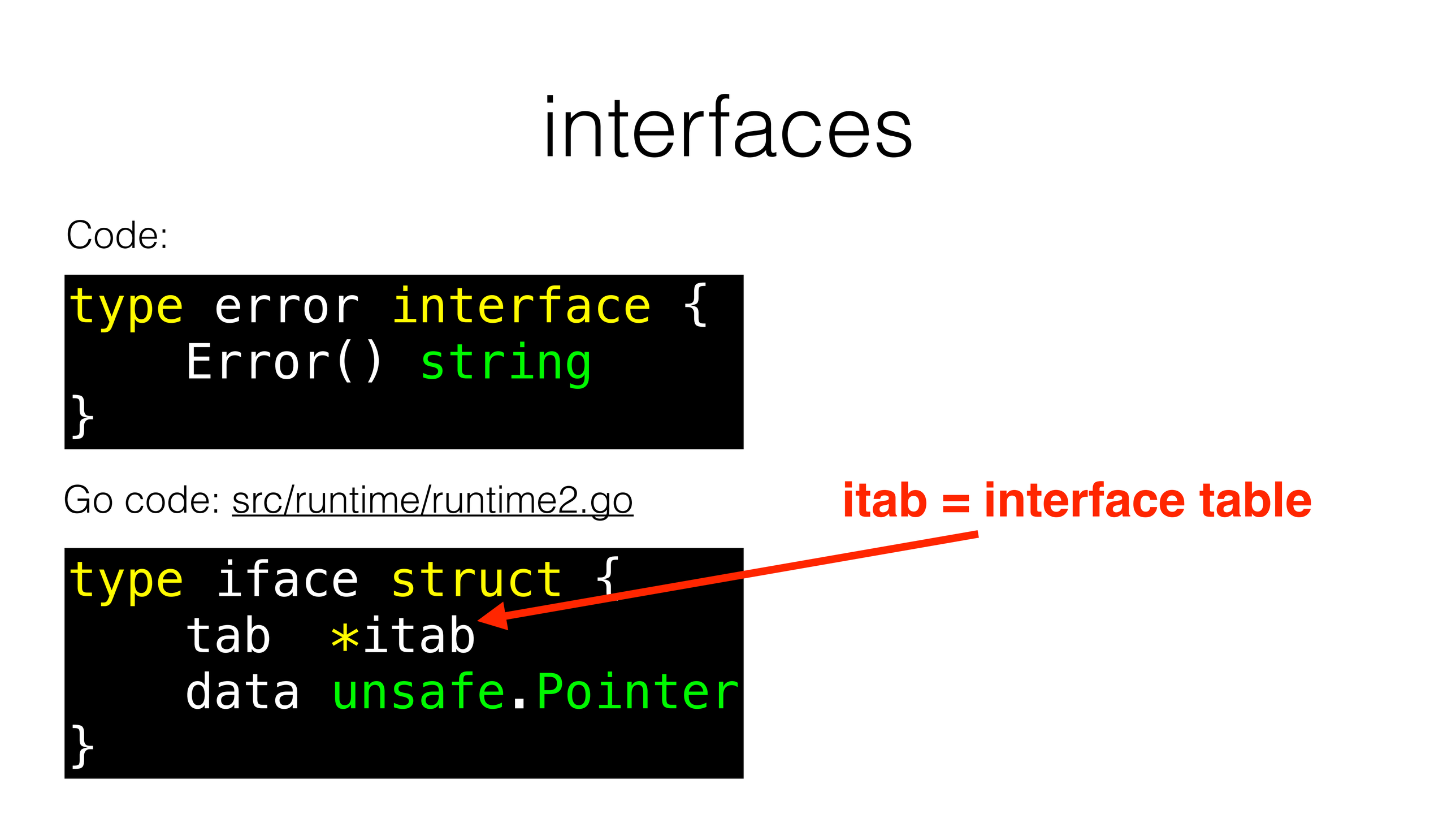
interfaces

Code:

```
type error interface {
    Error() string
}
```

Go code: src/runtime/runtime2.go

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```



interfaces

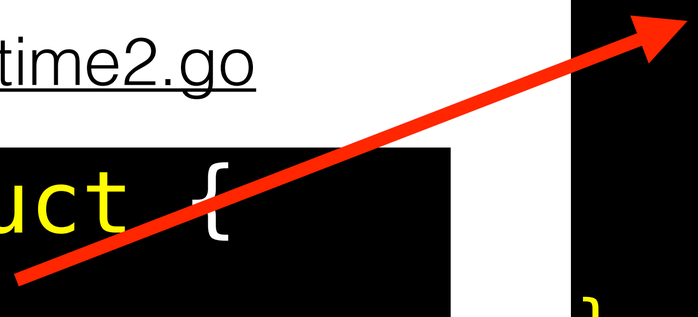
Code:

```
type error interface {
    Error() string
}
```

Go code: src/runtime/runtime2.go

```
type iface struct {
    tab  *itab
    data unsafe.Pointer
}
```

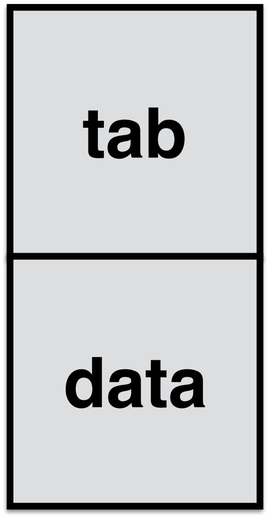
```
type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    bad    int32
    unused int32
    fun    [1]uintptr
}
```



interfaces

Code:

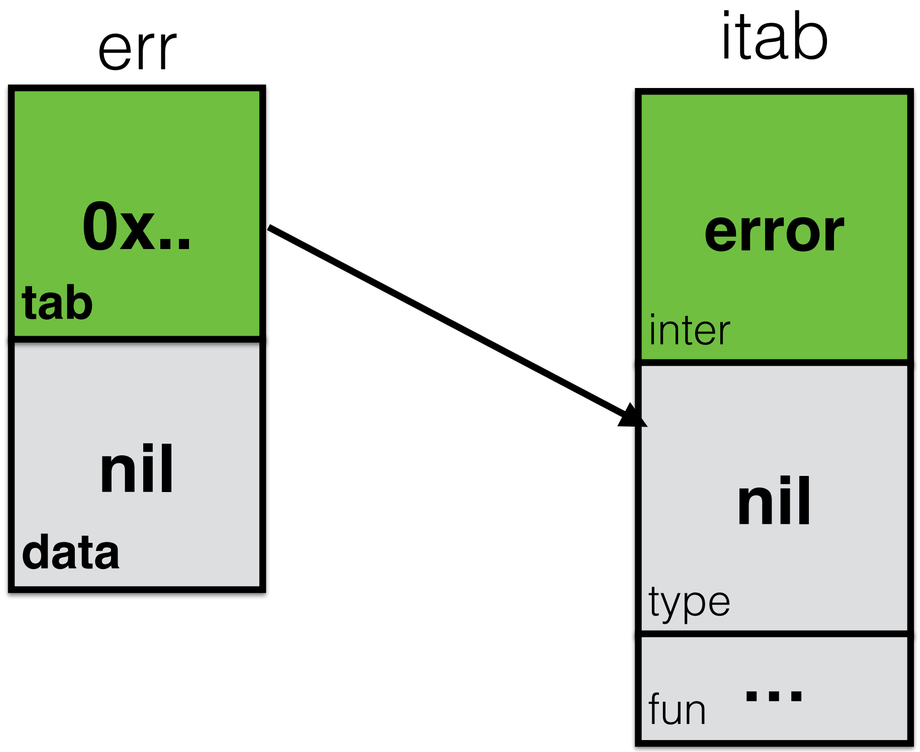
```
type error interface {
    Error() string
}
```



interfaces

Code:

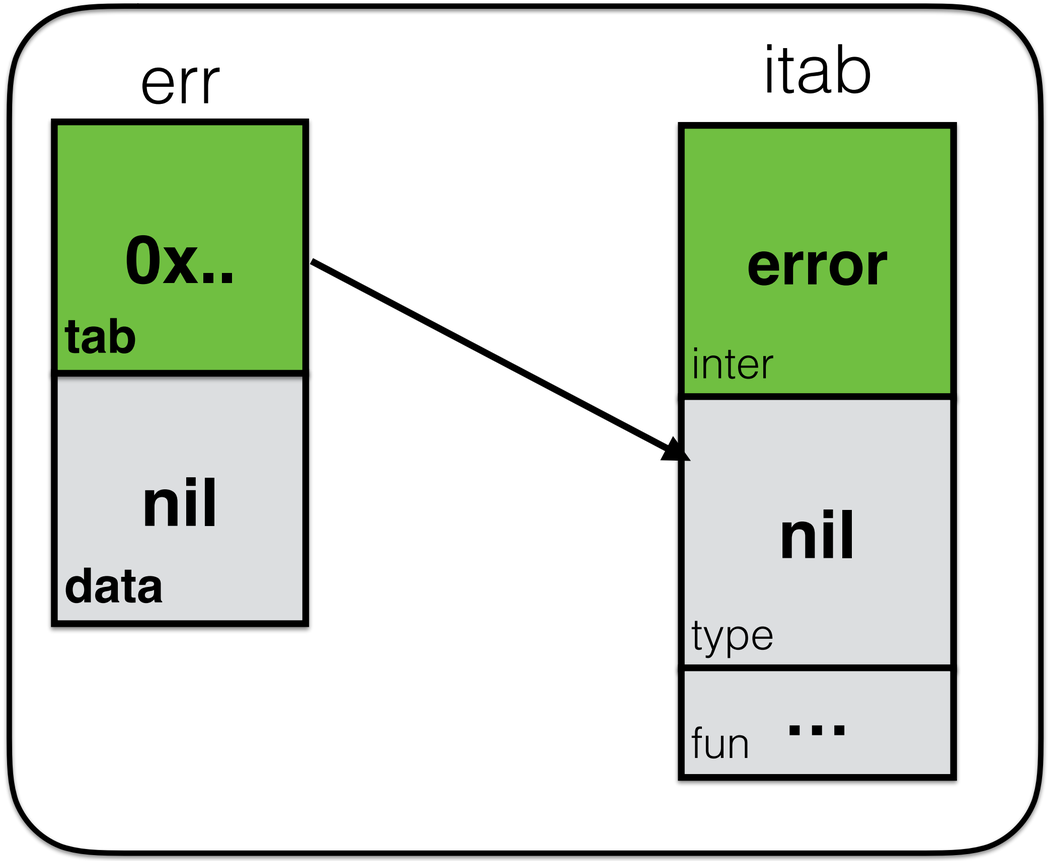
```
type error interface {
    Error() string
}

var err error
```



interfaces

Code:

```
type error interface {
    Error() string
}

var err error
```

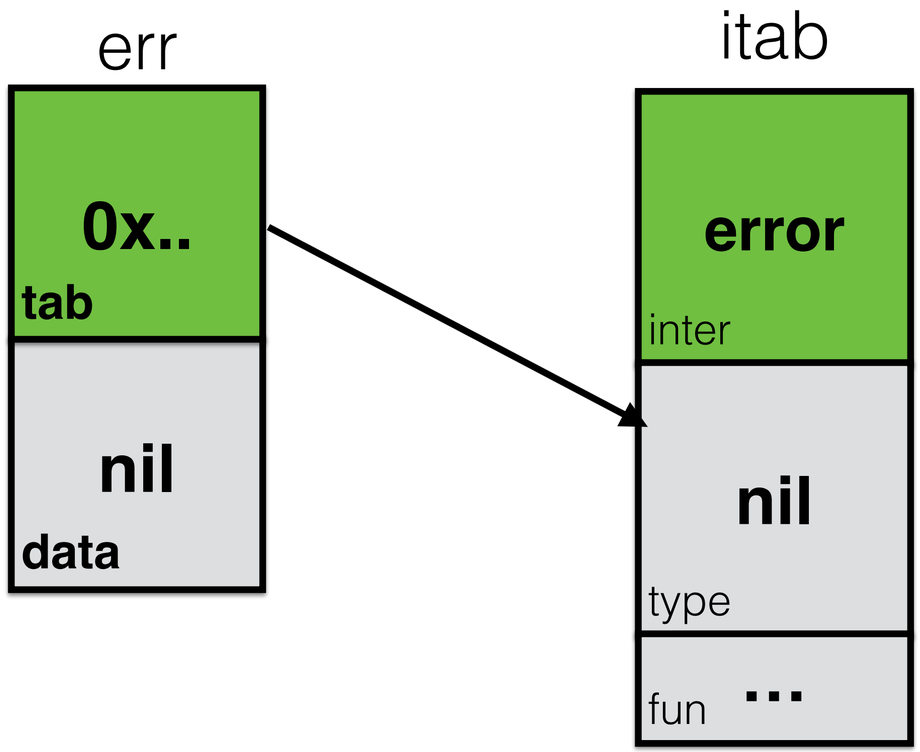


nil interface

interfaces

Code:

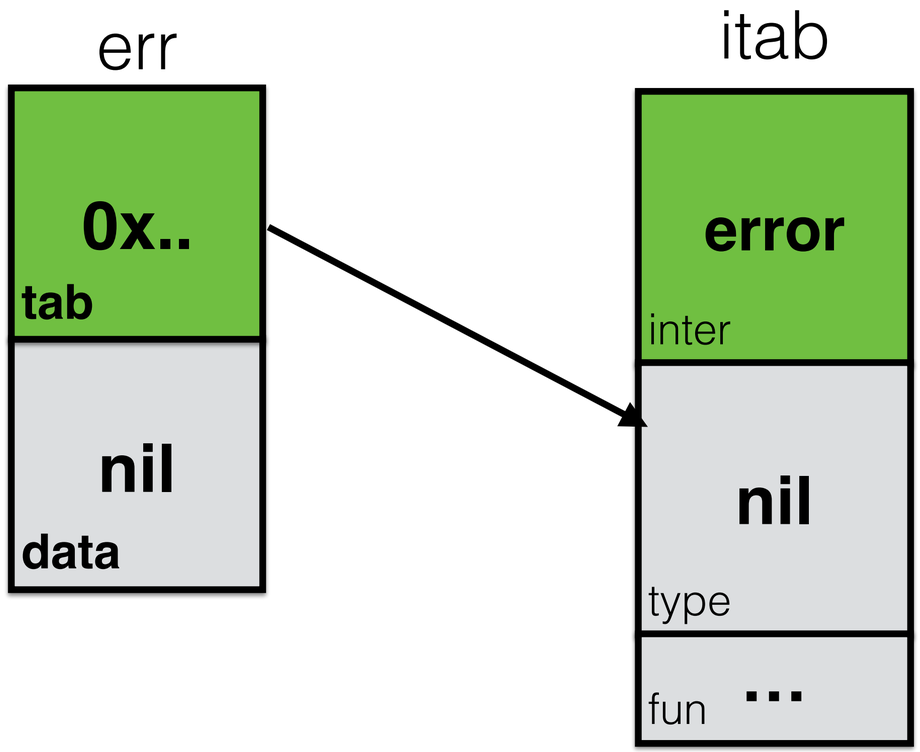
```
type error interface {
    Error() string
}

func foo() error {
    return nil
}
```



interfaces

Code:

```
func foo() error {
    var err error
    // err == nil
    return err
}

err := foo()
if err != nil { // false
}
```



interfaces

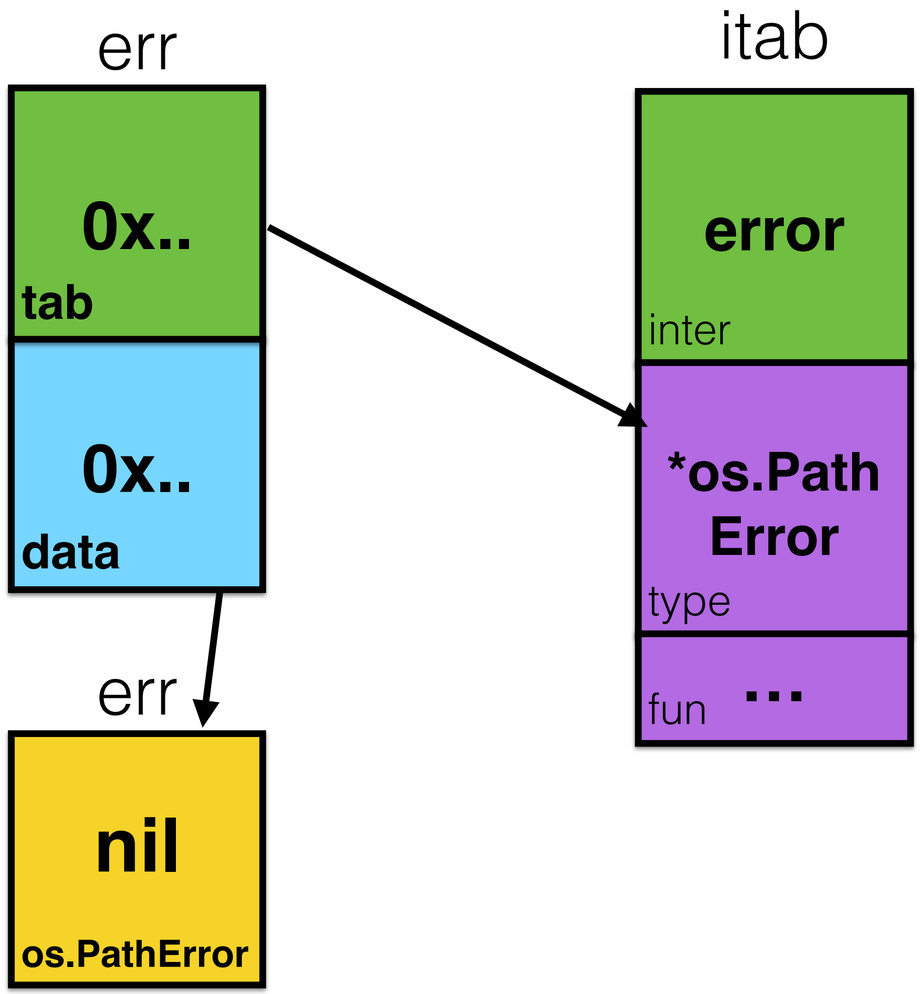
Code:

```
func foo() error {
    var err *os.PathError
    // err == nil
    return err
}

err := foo()
if err != nil { // ???
}
```

interfaces

Code:

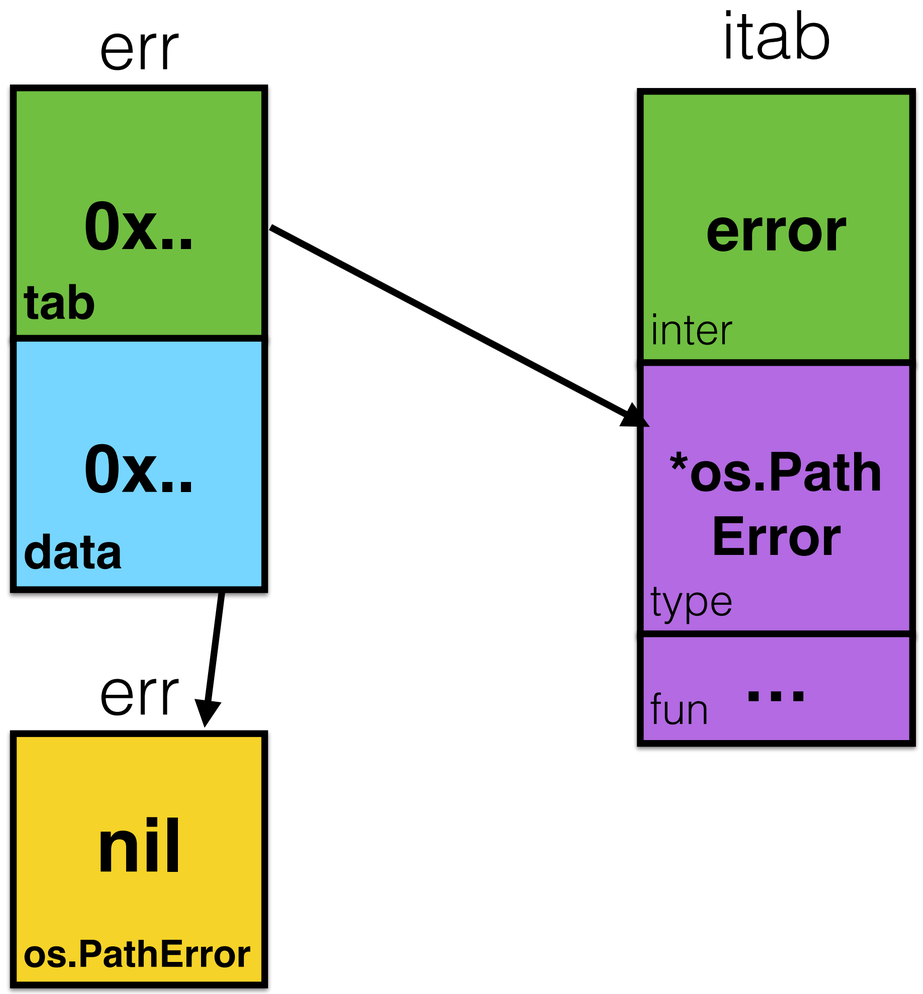
```
func foo() error {
    var err *os.PathError
    // err == nil
    return err
}

err := foo()
if err != nil { // ???
}
```



interfaces

Code:

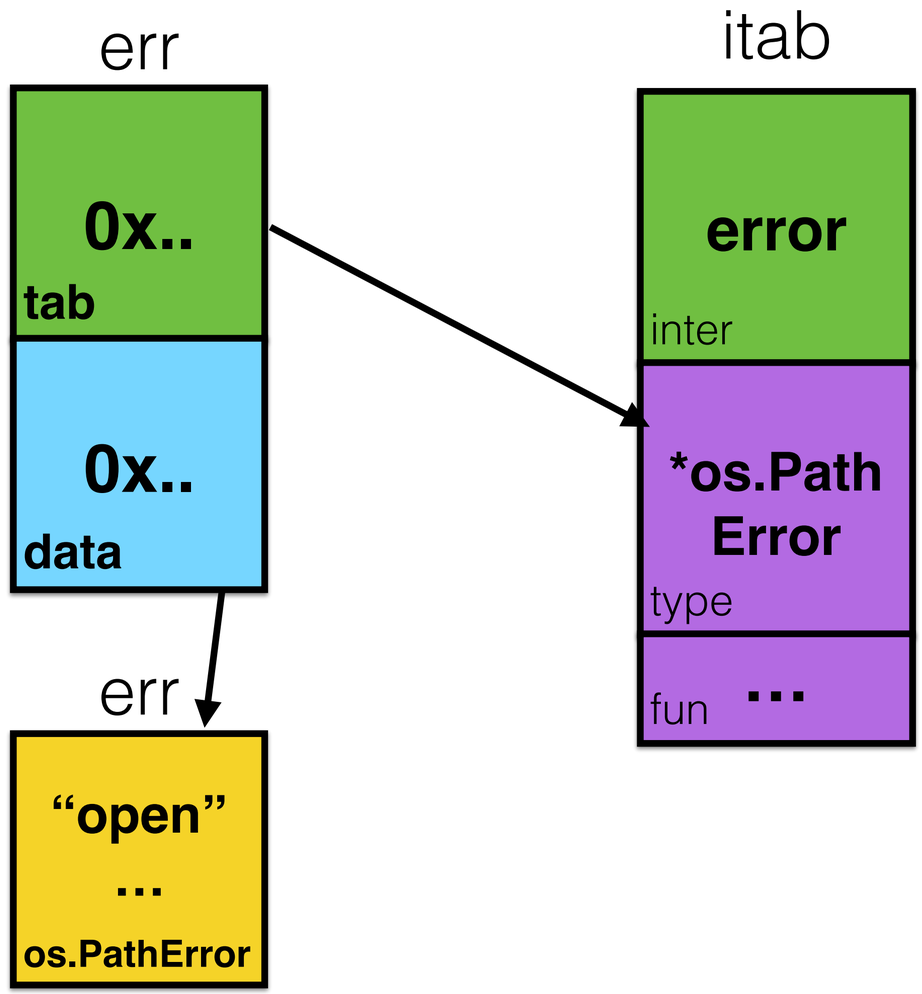
```
func foo() error {
    var err *os.PathError
    // err == nil
    return err
}

err := foo()
if err != nil { // true
}
```



interfaces

Code:

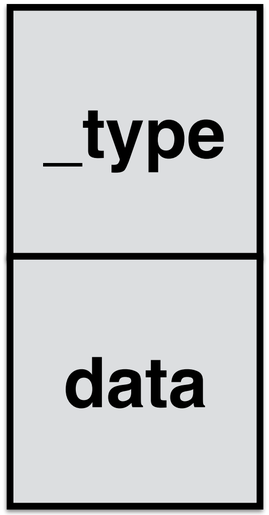
```
func foo() error {
    err := &os.PathError{
        "open", name, e
    }
    return err
}

err := foo()
if err != nil { // true
}
```



interfaces

Code:

```
type empty interface{}
```



Go code: src/runtime/runtime2.go

```
type eface struct {
    _type  *_type
    data unsafe.Pointer
}
```

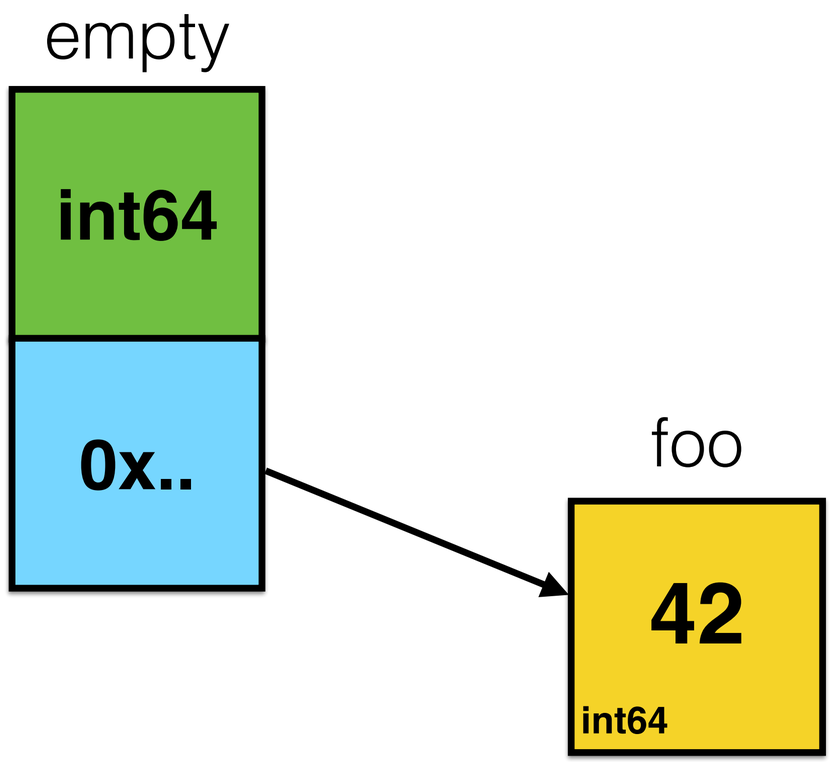
interfaces

Code:

```
type empty interface{}
foo := int64(42)
empty = foo
```

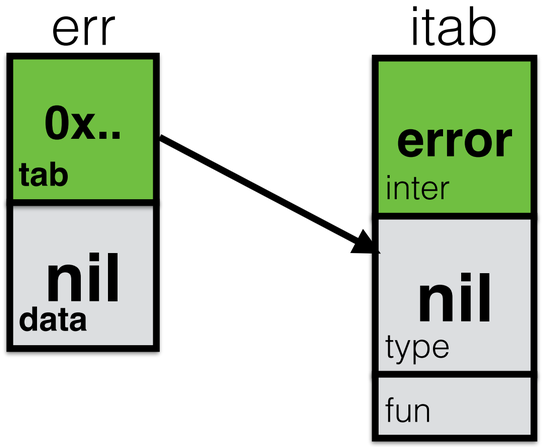
Go code: src/runtime/runtime2.go

```
type eface struct {
    _type  *_type
    data unsafe.Pointer
}
```

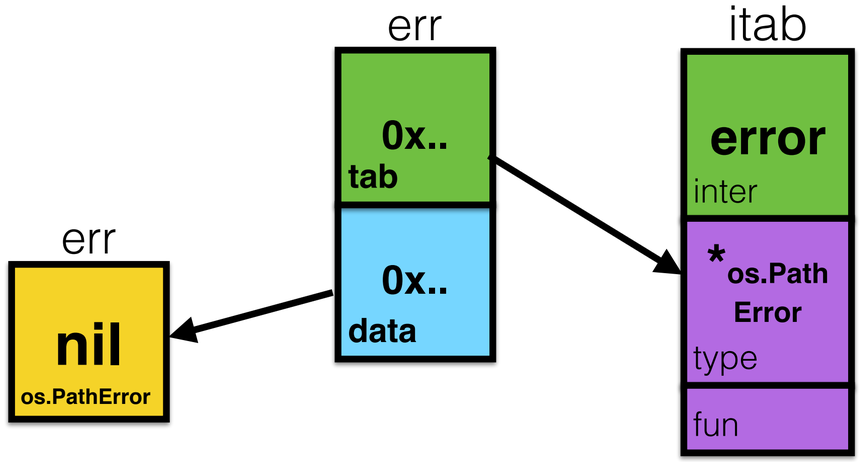


interfaces

```
func foo() error {
    var err error
    // err == nil
    return err
}
```




```
func foo() error {
    var err *os.PathError
    // err == nil
    return err
}
```



init() func

Code:

```
var foo = Foo()

func init() {
    fmt.Println("1")
}

func main() {
    fmt.Println("2")
}
```

init() func

Code:

```
var foo = Foo()

func init() {
    fmt.Println("1")
}

func main() {
    fmt.Println("2")
}
```

Initialization order:

- variables
- init() func
- main() func

init() func

Code:

```
var foo = Foo()

func init() {
    fmt.Println("1")
}

func main() {
    fmt.Println("2")
}
```

Guaranteed by spec:

A package with no imports is initialized by assigning initial values to all its package-level variables followed by calling all init functions in the order they appear in the source, possibly in multiple files, as presented to the compiler.

init() func

```
func init() {
    fmt.Println("1")
}

func init() {
    fmt.Println("3")
}

func main() {
    fmt.Println("2")
}
```

init() func

```
func init() {
    fmt.Println("1")
}

func init() {
    fmt.Println("3")
}

func main() {
    fmt.Println("2")
}
```

Output:

```
1
3
2
```

A package with no imports is initialized by assigning initial values to all its package-level variables followed by calling all init functions **in the order they appear in the source**, possibly in multiple files, as presented to the compiler.

init() func

foo.go:

```
package main

func init() {
    fmt.Println("1")
}

func init() {
    fmt.Println("2")
}
```

bar.go:

```
package main

func init() {
    fmt.Println("3")
}

func init() {
    fmt.Println("4")
}
```

init() func

foo.go:

```go
package main

func init() {
    fmt.Println("1")
}

func init() {
    fmt.Println("2")
}
```

bar.go:

```go
package main

func init() {
    fmt.Println("3")
}

func init() {
    fmt.Println("4")
}
```

Output:

```
3
4
1
2
```

To ensure reproducible initialization behavior, build systems are encouraged to present multiple files belonging to the same package **in lexical file name order** to a compiler.

In our case: bar.go, foo.go

init() func

Code:

```
func init() {
    fmt.Println("1")
}
```

Go code: src/cmd/compile/internal/gc/init.go

```
// a function named init is a special case.
// it is called by the initialization before
// main is run. to make it unique within a
// package and also uncallable, the name,
// normally "pkg.init", is altered to "pkg.init.1".
```

init() func

Code:

```
func init.1() {
    fmt.Println("1")
}
```

Go code: src/cmd/compile/internal/gc/init.go

```
// a function named init is a special case.
// it is called by the initialization before
// main is run. to make it unique within a
// package and also uncallable, the name,
// normally "pkg.init", is altered to "pkg.init.1".
```

init() func

Code:

```
func init.1() {
    fmt.Println("1")
}
```

Shell:

```
$ strings foo
...
main.init.1
main.init.2
...
```

Go code: src/cmd/compile/internal/gc/init.go

```
// a function named init is a special case.
// it is called by the initialization before
// main is run. to make it unique within a
// package and also uncallable, the name,
// normally "pkg.init", is altered to "pkg.init.1".
```

Links

- Must read:
  - Go Data Structures
  - Go Data Structures: Interfaces
  - Go Slices: usage and internals
  - Gopher Puzzlers
- And, of course:
  - Go source code
  - Effective Go
  - Go spec

Thank you