

Study Note: Introducing GO (O'Reilly)

Duy

Ch1: Getting Started

GO Introduction

- Strong typed (static) language
- GO= programming language + toolset (go build, go fmt, go help ...)
- every go program starts with package declaration
 - package main
- Types of GO programs: executable and libraries
- import statement
 - import "fmt"
 - import ("fmt ; "reflect")

GO Environment

- ***go env***: list go environment setting
- GOPATH: environment var to find your source code and 3rd-party packages
 - directory structures: \$GOPATH/src , /\$GOPATH/pkg
 - setup GOPATH as home dir (linux / MAC OSX)
 - `echo 'export GOPATH=$HOME\n' >> ~/.bashrc`
 - will overwrite go env variables
- Other env: <http://wiki.jikexueyuan.com/project/go-command-tutorial/0.14.html>

GO DOC

- godoc: built-in documentation tool
 - `godoc fmt.Println`

Ch2: Types

Primitive Types

- Integer types:
 - *uint8, uint16, uint32, uint64, int8, int16, int32, int64*
 - alias: *uint8=byte, rune=int32*
- Machine dependent types: *uint, int, uintptr*
 - always use *int* when working with integers
- Floating points: *float32, float64*
- Boolean: *bool (true / false)*
- String type: *string*

Print Data Types

- `var a = 1`
`var b = 0.5`
`var c = [4]int { 1, 2, 3 , 4}`
`var d = "hello"`
- `fmt.Println("a is", a, " b is ", b, " c is ", c , " d is ", d)`
 - or `fmt.Print`
- `fmt.Printf("a is %d b is %f c is %v d is %s\n", a, b, c, d)`
- `fmt.Printf("a is %v b is %v c is %v d is %v\n", a, b, c, d)`
 - `%v` : general data types

Ch3: Variables

Var Declaration

- `var x T`
 - declare only variable ; cannot be used before memory allocation
- `x := ...` (No `T` ; type inferred by compiler)
 - declare and allocate memory
- `var (`
 `a = 5`
 `b = 10`
 `c = 15)`
- `const data` (replace *var* with *const*)
 - `const x string = "A const string"`

Global Variable

- variable declared outside of function is global
 - syntax: `var gVariable int=0`
 - Cannot use `gVariable:=0` outside function body

Ch4: Control Structures

Loop

- `for i:=1 ; i< 10; i++ {`
 `...`
 `}`
- infinite loop (no while syntax in GO)
 `for {`
 `....`
 `}`

Branching

- ```
if i == 1 { ...
 } else if i == 2 { // else if must be at the same line as }
 ...
 } else { // else must be at the same line as }
 ...
}
```
- ```
switch i {  
  case 0: ...  
  case 1: ...  
  default: ... // no break is needed as C language  
}
```

Ch5 Arrays, Slices and Maps

Arrays

- declaration:
 - `var x [5]int`
 - `var x = [5]int { 1, 2, 3, 4, 5 }`
 - `var x = [...]int { 1, 2, 3, 4, 5 }` // let compiler count for you
- `[5]int` and `[4]int` are two different data types
 - passing them to functions means copy array
 - GO Argument passing: *call by value*
 - pass array's *pointer* to avoid copy

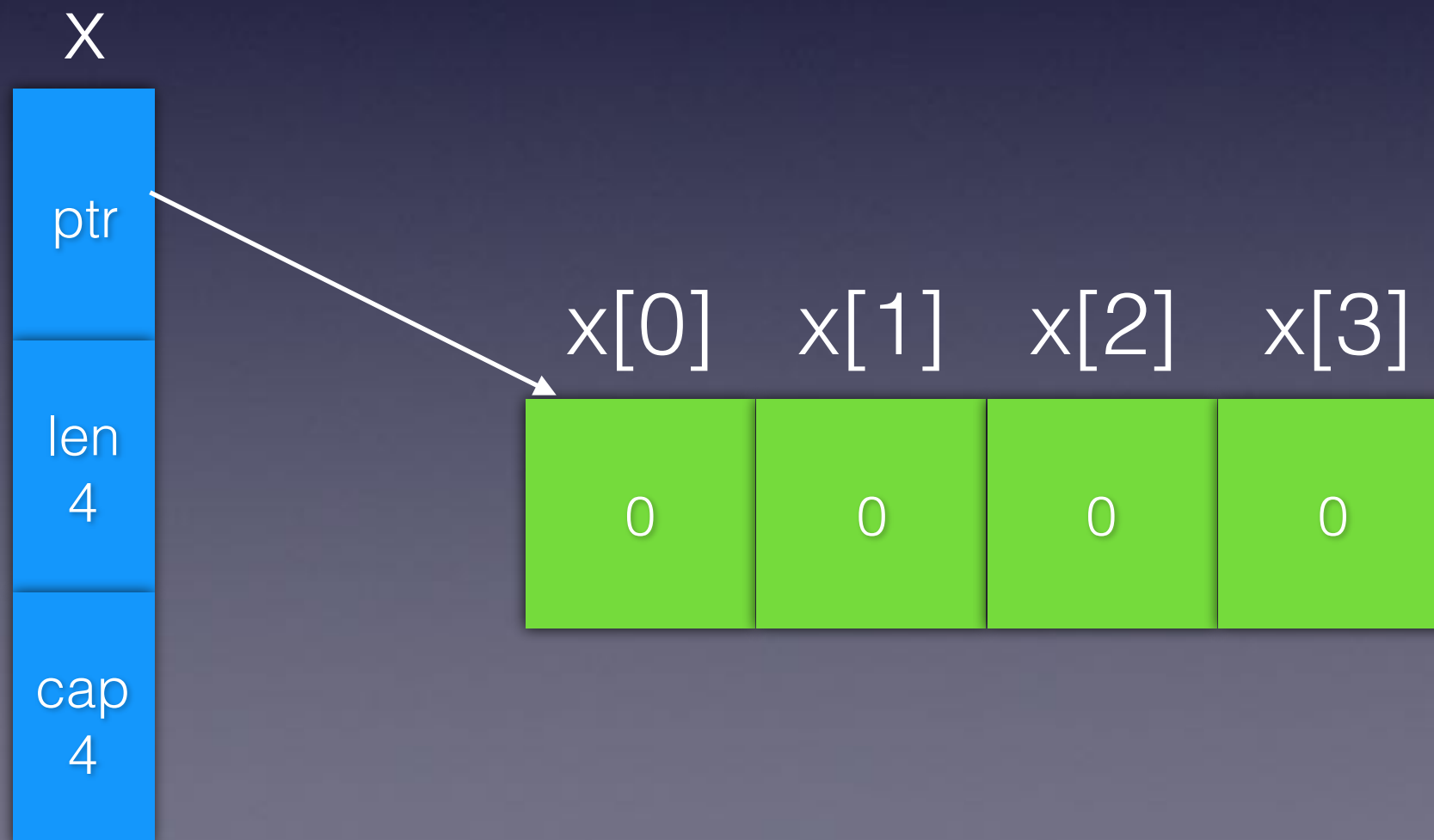
Visiting Arrays

- `var b = [...] { 1, 2, 3, 4, 5 }`
 - `for idx, v := range b {
 fmt.Println(idx, v)
}`
 - `for _, v := range b {
 // if index not used, skip it with _ to avoid compile error
 fmt.Println(v)
}`
 - `for i:=0; i < len(b); i++) {
 fmt.Println(b[i])
}`

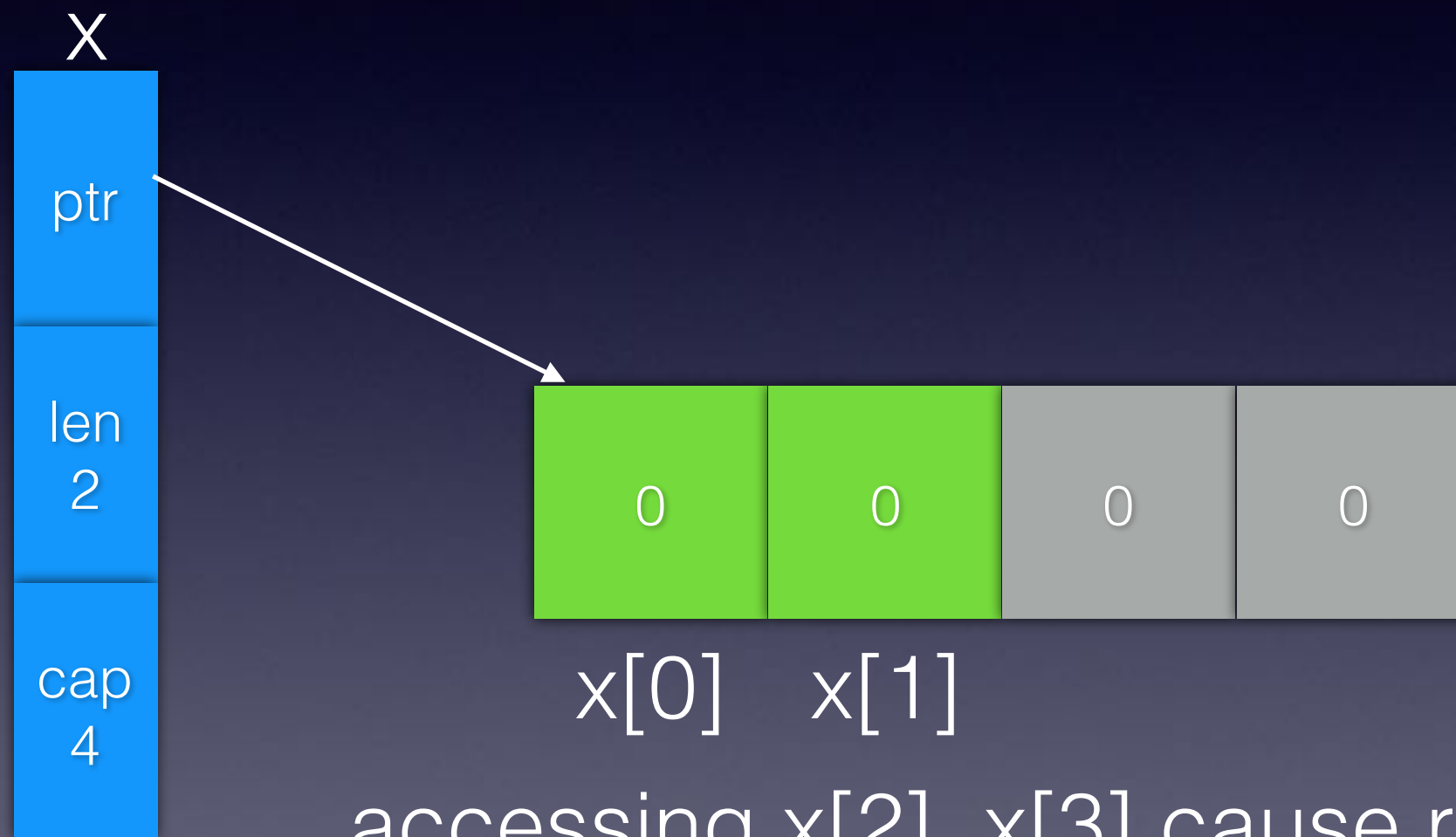
Slices

- A data structure
 - Pointer: a pointer to underlying array
 - Length: length of data currently pointed to (logical boundary)
 - Capacity: number of actual elements in underlying array (beginning at element referred by pointer)
- Functionality:
 - Simulate "dynamic array" (ex. C++ vector)
 - `var x = make([]int, 2, 4) // make a slice with len 2, cap 4`
 - Access sub-part of an array
 - `var y = x[2:] // x[2] to x end`

```
var x := make([]int, 4)  
(Equals: var x:= make([]int, 4, 4))
```

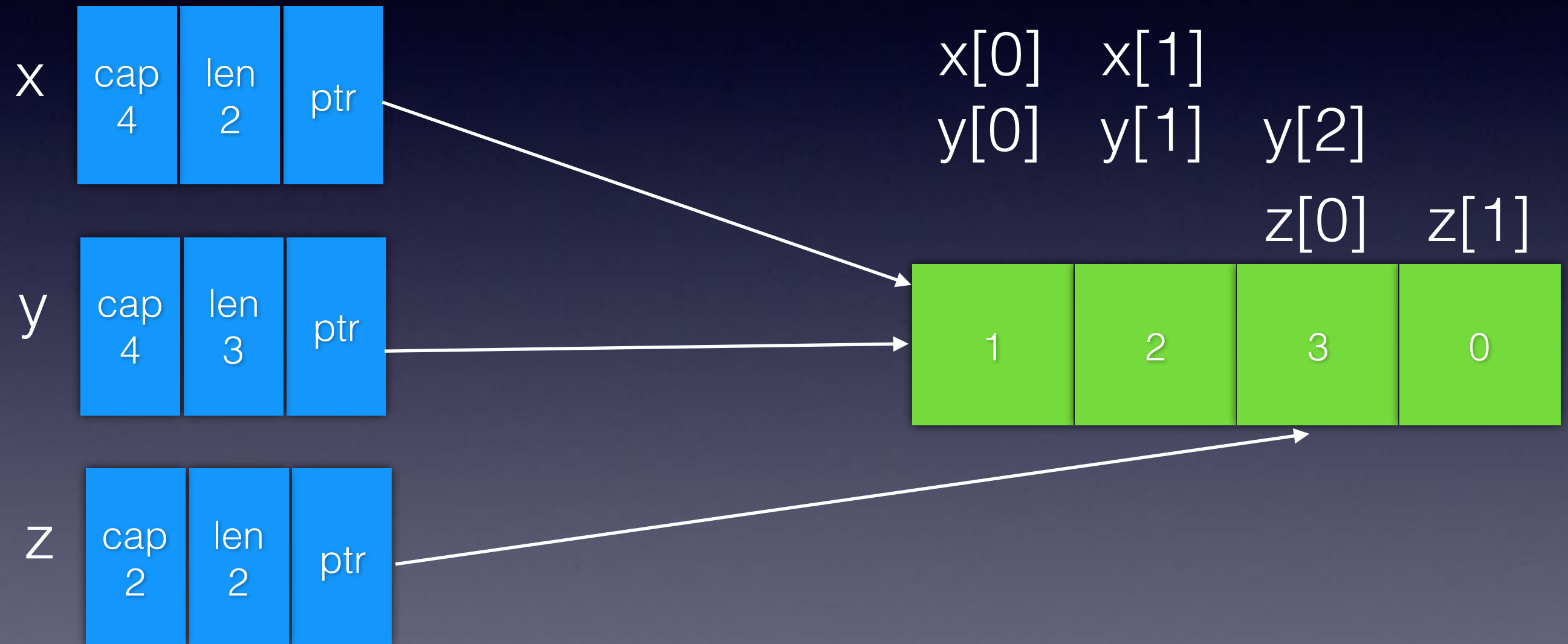


```
var x := make([]int, 2, 4)
```



accessing $x[2]$, $x[3]$ cause runtime error, since
logical length of x is 2
4 is physical length

```
var x := make([]int, 2, 4)
var y:= x[:3]
var z:=x[2:4]
x[0]=1
y[1]=2
z[0]=3
```

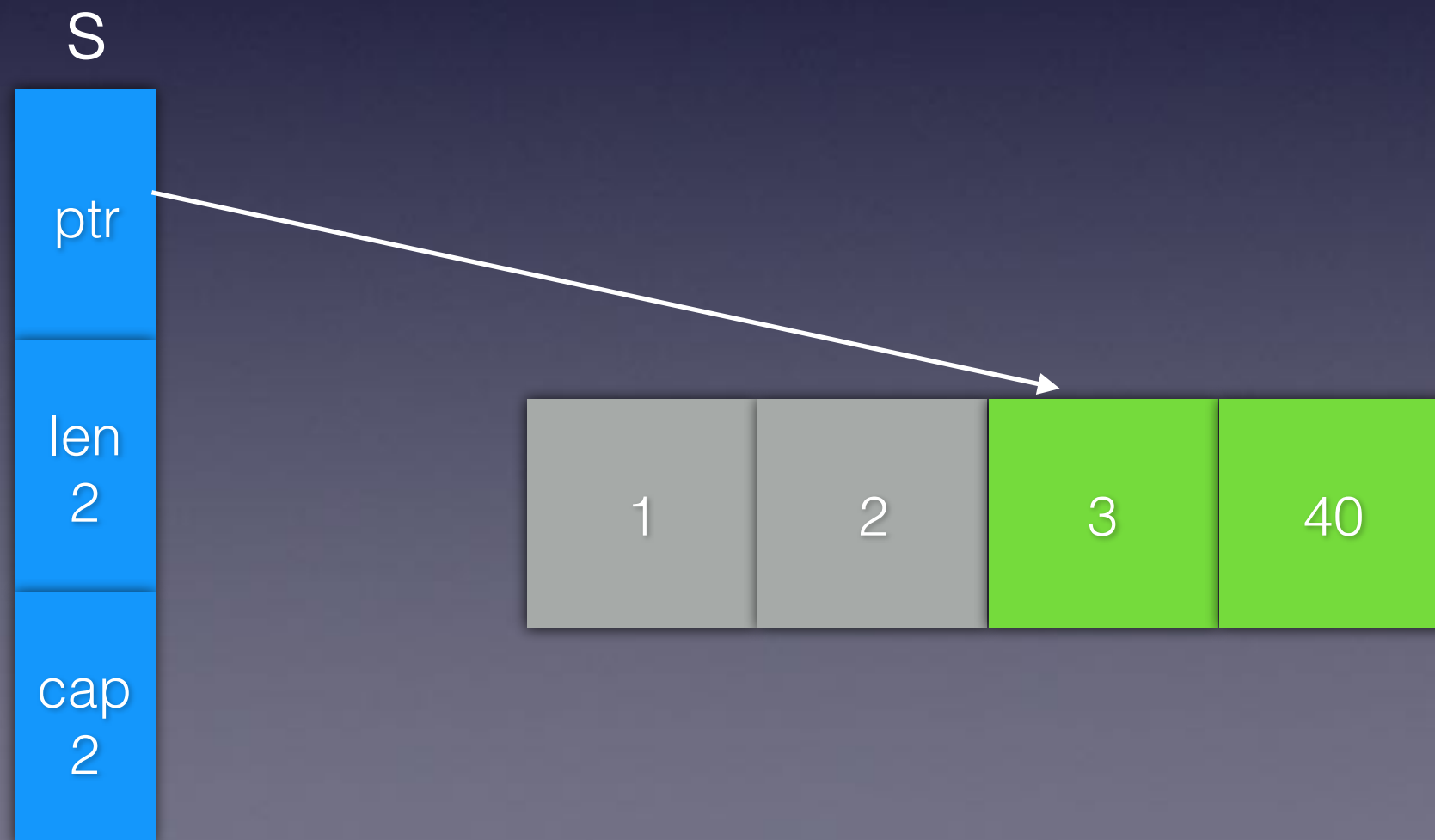


note: cap=number of physical elements beginning from element pointed to
note: `_:= x[2:]` cause panic error since `x[2]` does not exist (while `x[:2]` is ok)

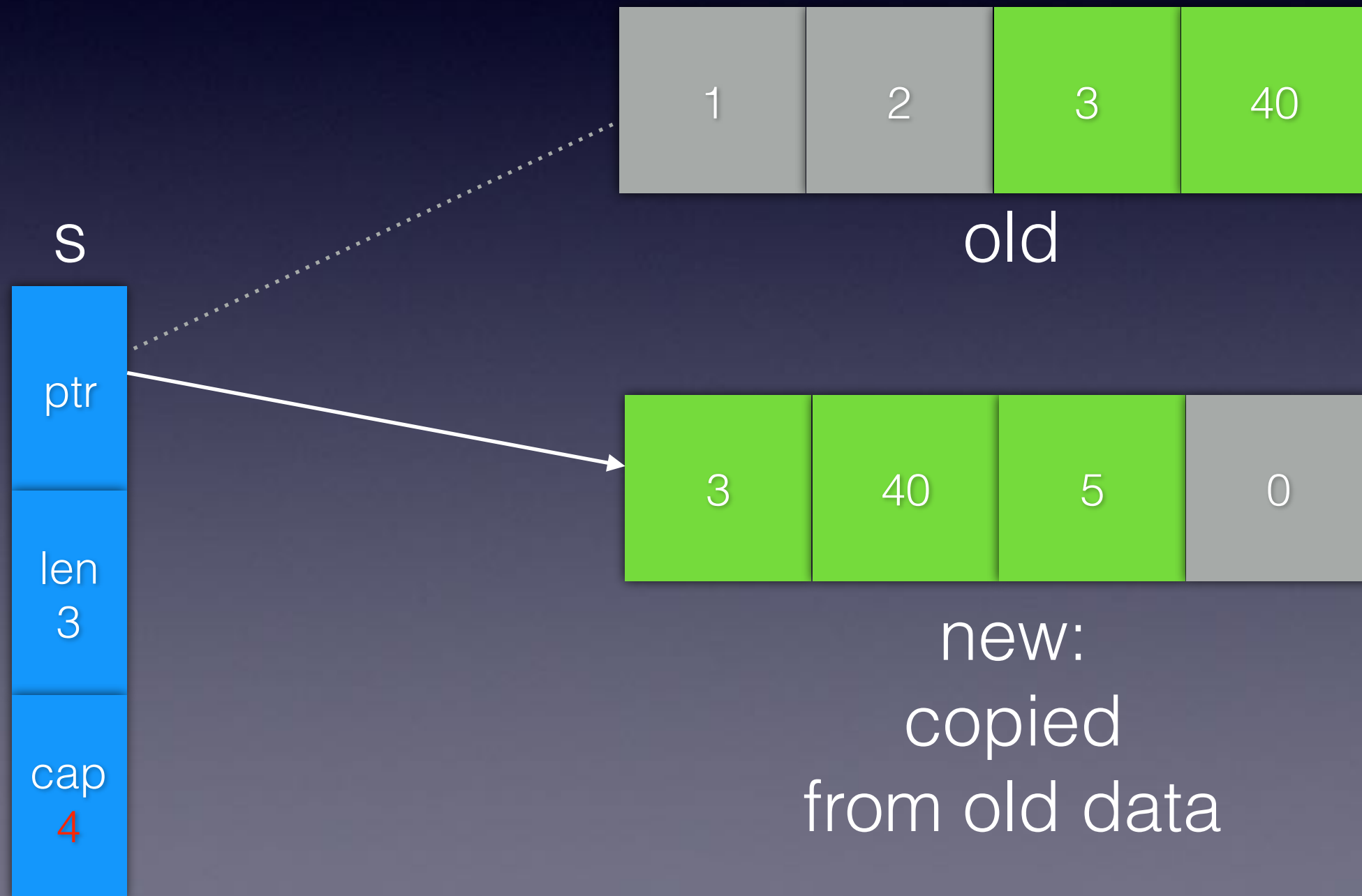
Expanding Slice

- New array will be allocated and old data is copied from old array, and slice is detached from old array to new array

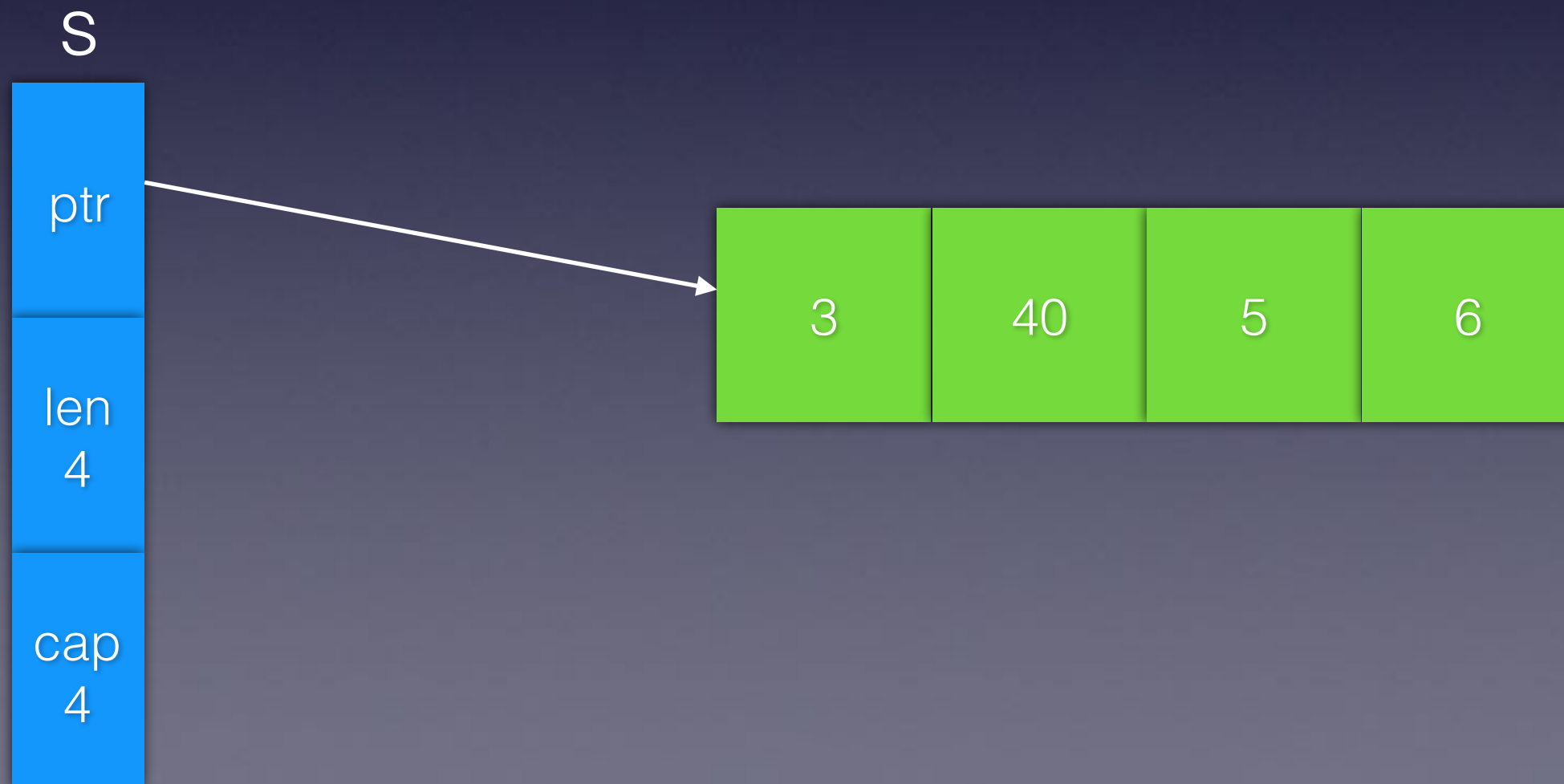
```
var b = [...]int{ 1, 2, 3, 4 }  
    s:= b[2:]  
    s[1]*= 10
```



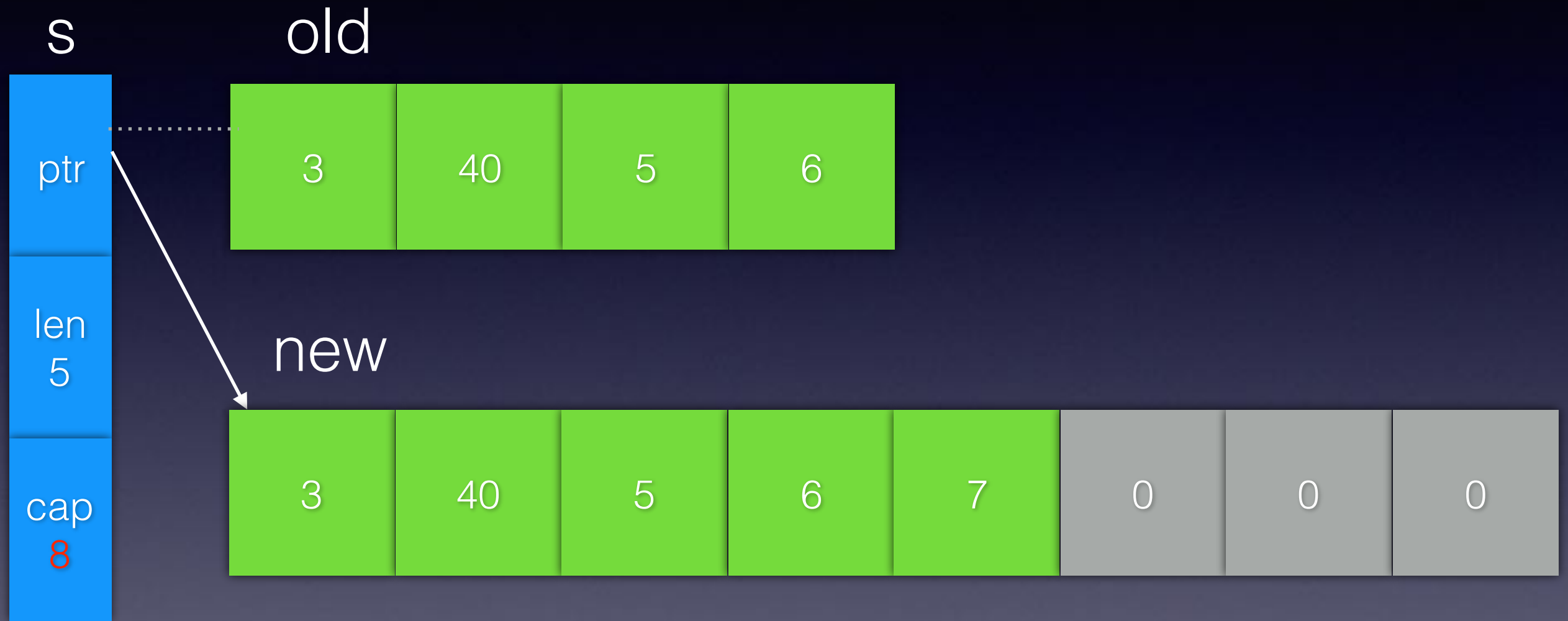
s=append(s, 5)



`s=append(s, 6)`



s=append(s, 7)



Array Copy

- syntax: `copy(destination, source)`
 - if length of destination < source, source data will be truncated
- `slice1:=[]int{1, 2, 3}`
`slice2:=make([]int, 2)`
`copy(slice2, slice1) // slice2 = [1,2].`

Map

- Declaration
 - `var m1 = make(map[string]int)`
`m1["A"]=1`
 - `var m2 = map[string]string {`
 `"007": "James Bond",`
 `"MissionImpossible": "Ethan",`
 `"SpiderMan": "Peter Park" }`

Nested Map

- ```
var m3 = map[string]map[string]int {
 "John":map[string]int {
 "Math":80,
 "Physics":60 },
 "Mary":map[string]int {
 "Math":90, }
}
```

# Accessing Map

- Traverse
  - for k,v := range m2 {  
    fmt.Printf("key=%v value=%v\n", k, v) }
- Check key and do
  - if val, ok := m2["SpiderMan"]; ok {  
    fmt.Printf("SpiderMan also exists: actor=%v\n", val) }

# Ch6 Function

# Syntax

- `func Foo(param1 int, param2 string) int {  
 return 1  
}`
  - begins with capital case: public (**exported**)
- `func foo(param1 int, param2 string) (int, int) {  
 return 1, 2  
}`
  - begins with lower case: protected (**not exported**)
    - accessible within package



# Named Return Data

- `func Foo(param1 int, param2 string) (ret int) {  
 ret = 1  
 return // must return, otherwise get compiler error  
}`

# Variadic Functions

- ```
fun add(args ...int) int {  
    total:=0  
    for k, v := range args {  
        total+=v  
    }  
    return v  
}
```

- args is called a variadic parameter

- variadic parameters must be the last parameter

- ```
fun add(int, string, args...int) {

}
add(1, "hello")
add(1, "hello", 2, 3)
```

# Closure

- Declare function within a function
- Closure can keep its internal state (state cleared once closure not referenced)
- Closures are passed to other functions by reference
  - passed closure is the original one

```
func main() {
 g1 := MakeNumberGeneratorClosure()
 g2 := MakeNumberGeneratorClosure()
 fmt.Println(g1(),g1(),g2(),g2()) // 1, 2, 1, 2
 foo(g1)
 fmt.Println(g1()) // 4
}
```

```
func MakeNumberGeneratorClosure() func() int {
 i:=0
 return func() int {
 i+=1
 return i
 }
}
```

```
func foo(funObj func() int) { // closure is not copied.
 funObj()
}
```

# defer

- deferred function is called after original function completes
- ```
fun main() {  
    f, _ := os.Open(filename)  
    defer f.Close() // called after leaving main  
}
```
- write Open and Close together : easy to understand
- deferred function are run **even if a runtime panic occurs**

Pointers

```
func main() {  
    x := 0  
    foo(&x)  
    fmt.Println(x) // 1  
    bar(x)  
    fmt.Println(x) // 1  
}
```

```
func foo(x *int) { // pointer variable is copied  
    *x+=1  
}
```

```
func bar(x int) { // value is copied  
    x+=1  
}
```

new

- get a pointer to a newly allocated memory
- ```
y:=new(int)
fmt.Println(*y) // 0
```

# Ch7 Struct and Interfaces



# Struct

- type Circle struct {  
    x float64  
    y float64  
    r float64  
}
- type Circle struct {  
    x, y, r float64  
}
- **type** keyword creates a new data type
- Initialization
  - c:=Circle{x:0, y:0, r:0 } or c:=Circle {0,0,5}

# Struct Method

- `func(c* Circle) area() float64 {  
    return math.pi * c.r*c.r  
}`
- `c* Circle`: Receiver
  - So we can use `c.area()` syntax

# Embedded Type(Anonymous Field)

- type Person struct { Name string }  
func (p \*Person) Talk() {  
    fmt.Println("I am Person, Name is", p.Name) }

- Has a relationship:  
type Android struct {  
    Person Person;  
    Model string }

- Is-a relationship:  
type Android struct {  
    **Person**  
    Model string }

```
a:=new(Android)
```

```
a.Person.Talk()
```

```
a.Talk() // Android is Person, so Android can Talk (and Talk can be overridden)
```

# interface

- An interface defines a method set
- An interface is like an pointer to an object
  - However interface cannot access anything of the object except the methods it knows/defines
- An interface is like a "Protocol" or "Contract" when a function wish to receive infinite unknown types but can still work, by calling some common "methods" the unknown types should have (i.e., implement the interface)

# Examples

- type Shape interface {  
    area() float64  
}
- Implement interface: just have a method of same signature  
type Circle struct {...}  
func(c\* Circle) area() float64 {  
    ....  
}

# Ch8 Packages

# Creating Packages

- `import golang-book/chapter8/math`
  - `math` = real package name
    - package path is: `$GOPATH/src/golang-book/chapter8/math`
  - all source files put right in `math` folder will be scanned for searching "package `math`"
    - subfolder will not be searched recursively
    - source file name does not matter
- Best practice: let package name equals the folder name they fall in
- Data in package source file with capital letter is exposed ; others are only accessible within same package

# Package Structure Example & Import

- Assume \$GOPATH = ~
- Folder structure:
  - ~/src/mylib/sum.go
  - ~/src/mylib/mul.go
  - ~/src/mylib/sublib/inc.go
  - package mylib
  - package sublib
- ```
import mylib
import mylib/sublib
x:=mylib.Sum(1,2)
y:=mylib.Multiply(2,3)
z:=sublib.Inc(3)
```


System Packages

- GOROOT determines which version of GO to use
 - Ex. /usr/local/Cellar/go/1.8/libexec
- System packages will be imported and reused (i.e., no need to compile) if they exist
 - Ex. MAC OS example:
 - /usr/local/Cellar/go/1.8/libexec/pkg/darwin_amd64

Import Mechanism Testing

- MacOS system package path:
`/usr/local/Cellar/go/1.8/libexec/pkg/darwin_amd64`
 - By removing system pkg folder, go run still works until it's slow since all libs should be re-compiled every time
 - By removing `/usr/local/Cellar/go/1.8/libexec/src` and system pkg, go run will fail
 - By only removing `/usr/local/Cellar/go/1.8/libexec/src` (system pkg exists), go run still fail
- Conclusion: GO seeks src source first, and then check if system package has corresponding existing library

Ch9 Testing

Run Test

- go test (go test -v: verbose mode): execute all test files under current directory
 - "testing" GO package should be imported
 - test file names should end with `_test.go`
 - ex. mylib_test.go
 - Function signature should be:
 - ```
func TestXXXX(t *testing.T) {

}
```
  - Trigger test error: `t.Error(msg1, msg2, msg3 ...)`

# Ch10 Concurrency

# Goroutines

- coroutine that runs concurrently
- ```
fun main() {  
    go func() {  
        fmt.Println("Hello")  
    }() // will not wait goroutine to complete here.  
    var input string  
    fmt.Scanln(&input) // to hold the main goroutine
```

Some Example

```
func main() {  
    var i = 0  
    fmt.Println("main %v Adr=%v", i, &i)  
    for i=0;i<5;i++ {  
        go func() {  
            fmt.Println("main goroutine %v Adr=%v", i, &i)  
        }()  
    }  
    foo()  
    var input string  
    fmt.Scanln(&input) // to prevent main from leaving  
}
```

```
func foo() {  
    var i = 0  
    fmt.Println("foo %v Adr=%v", i, &i)  
    for i=0;i<5;i++ {  
        go func() {  
            fmt.Println("foo goroutine %v Ard=%v", i, &i) // note: they will not print what you may be thinking  
!            }()  
        }  
    }  
}
```

About Stack/Heap Data Allocation

- The goroutine in previous example can refer to local variable in foo, even after foo exits
- The GO compiler will infer whether to allocate data in stack or heap
 - If compiler cannot determine a data won't be referenced after function leave, it will allocate it on heap. Otherwise, it prefers to allocate on stack for better performance
 - See GO FAQ: https://golang.org/doc/faq#stack_or_heap

Goroutines Race Condition

```
func UnSafe() {  
    var wg sync.WaitGroup  
    var counter int = 0  
    for i:=0; i<1000; i++ {  
        wg.Add(1)  
        go Foo1(&wg, &counter)  
    }  
    wg.Wait()  
    fmt.Println("UnSafe:counter=", counter) // will not equal 10000000  
}  
func Foo1(wg *sync.WaitGroup, counter *int) { // race conditon !  
    defer wg.Done()  
    for i:=0; i<10000; i++ {  
        add(counter)  
    }  
}  
func add(i *int) {  
    *i+=1  
}
```

GO Slogan:

Don't communicate by shared
memory. Share memory by
communicating

However some experiment shows that **channels are less efficient than Mutex**.
When using channels, you are still using a lock wrapped by channel

- So for those frequently accessed shared memory, Mutex is still an option.
For architectural factor that is less frequently accessed, channels may be considered

Channel size is fixed at declaration, while you can have growable built-in data structure by using mutex

Channels

- Channels are reference types
 - Other reference types: slice, map, pointers, function
 - Reference type: only the data structure referring to source memory is copied; referred memory is not copied
 - So you can pass a channel to multi functions without using &
 - ```
var c chan string = make(chan string)
fun1(c)
fun2(c) // fun1 and fun2 receive the same channel instance, not
copied separate instances
```

# Channels

- Channels are synchronized by default
  - send / receive blocks, until both ends are ready
    - The first goroutine to take action is blocked; the second one will free the blocking goroutine on the other side, and the second goroutine is able to keep going without blocking
- Non-blocking channel operation:
  - ```
select {  
    case ch<-1: ... ch1 has data  
    default: ... // ch1 has no data  
}
```

UnBlocked Channel Select

- `select {`
 `case msg1 := <- ch: ...`
 `default: ... // triggers immediately if ch1 has no data`
}

Channel Direction

- restrict a channel to be either sending or reading channels
 - read only: `func pinger(c chan<- string) {...}`
 - send only: `func longer(c <-chan string) {...}`

Buffered Channels

- Asynchronous channel:
 - `c := make(chan int, 100)` // channel with buffer size 100
- Default channel is synchronous (buffer size = 0)
 - `make(chan int) == make(chan int, 0)`