Study Note: Introducing GO (O'Reilly)

Duy

Ch1: Getting Started

GO Introduction

- Strong typed (static) language
- GO= programming language + toolset
- 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

- goodc: 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"

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)
    }

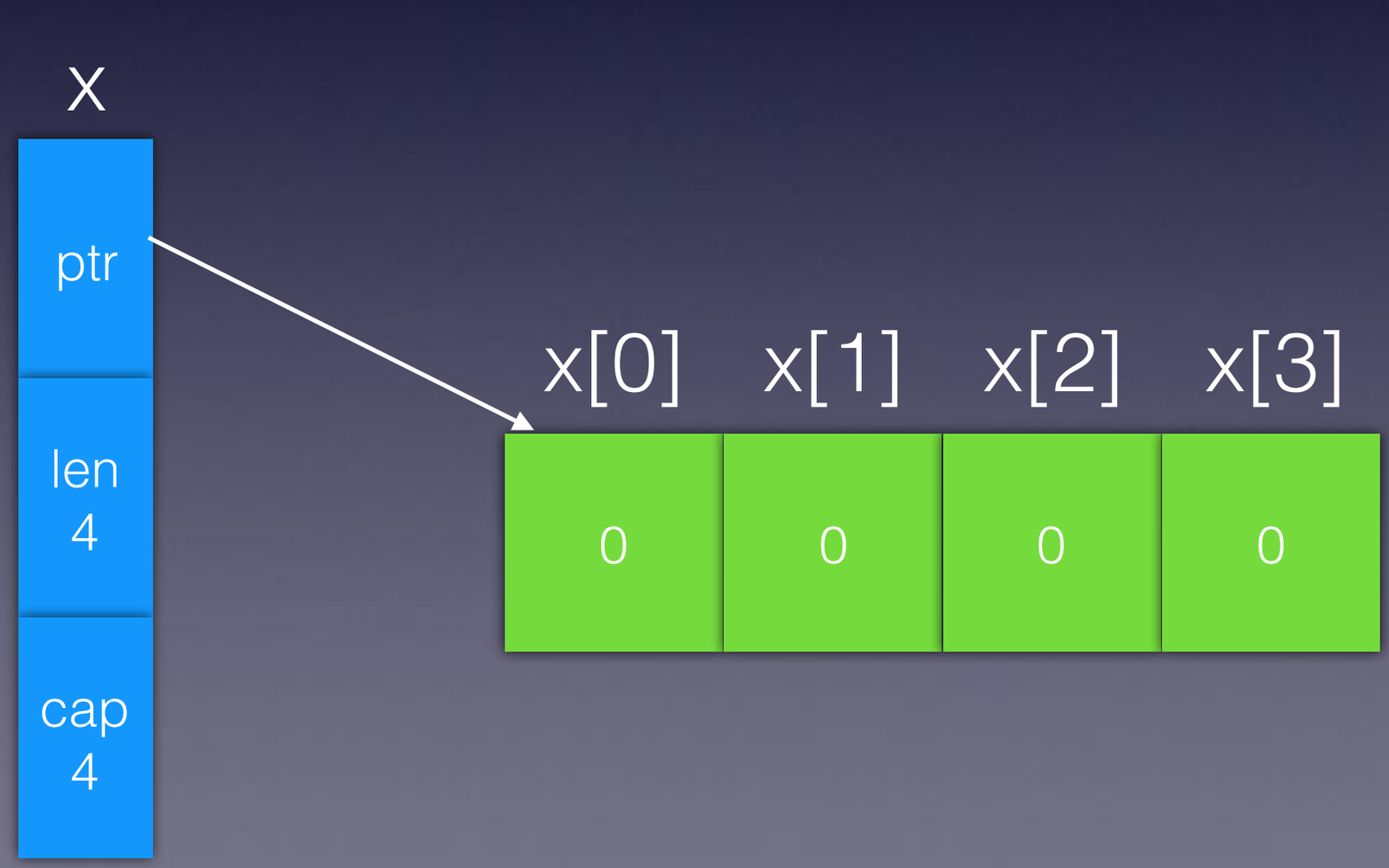
  - for i:=0; i < len(b); i++) {
    fmt.Println(b[i])
    }

Slices

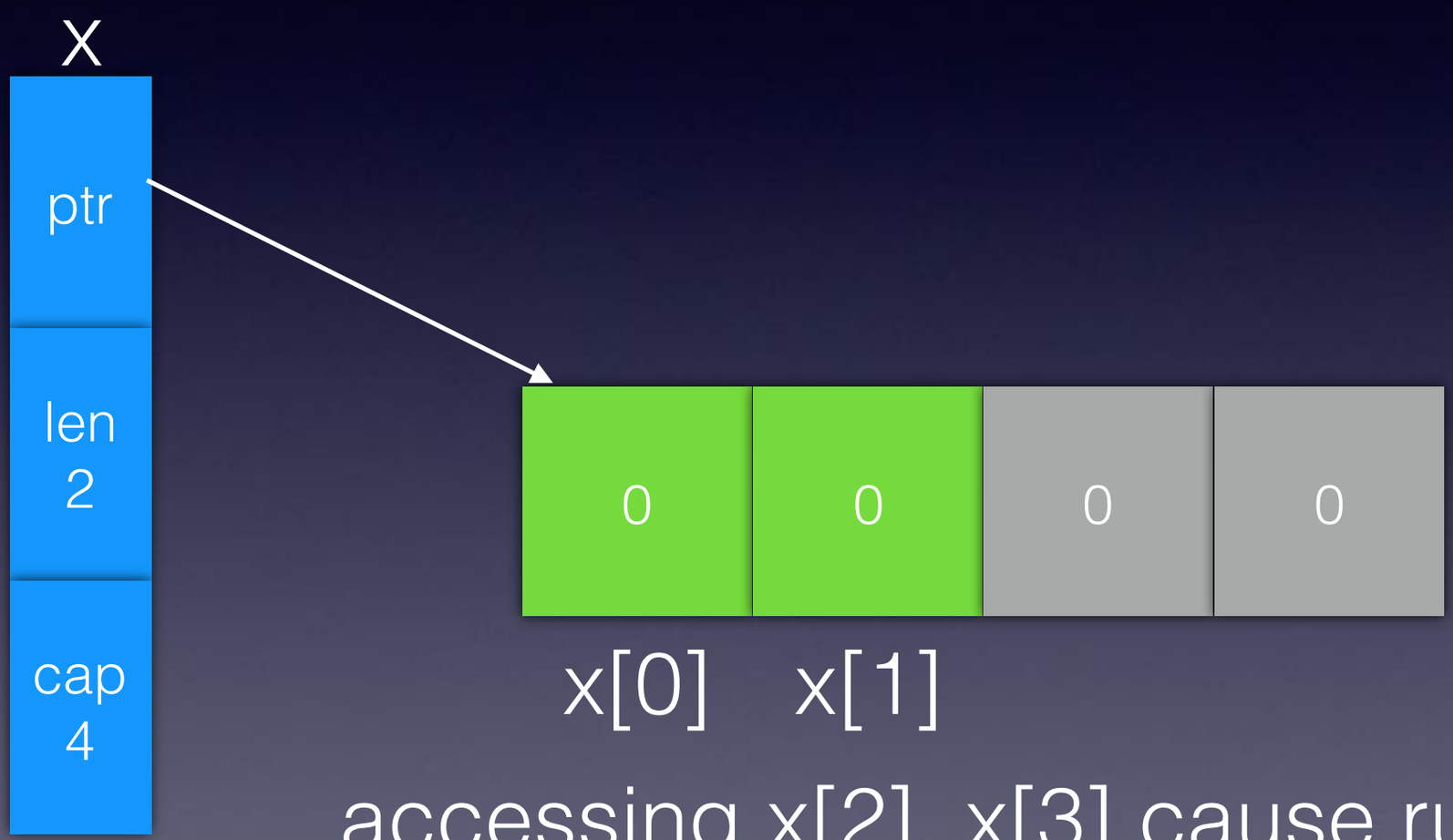
- A data structure
  - Pointer: a pointer to underlying array
  - Length: length of data currently pointed to (logical boundary)
  - Capacity: number of actual elements in underlying array (beginning at element referred by pointer)
- Functionality:
  - Simulate "dynamic array" (ex. C++ vector)
    - 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)

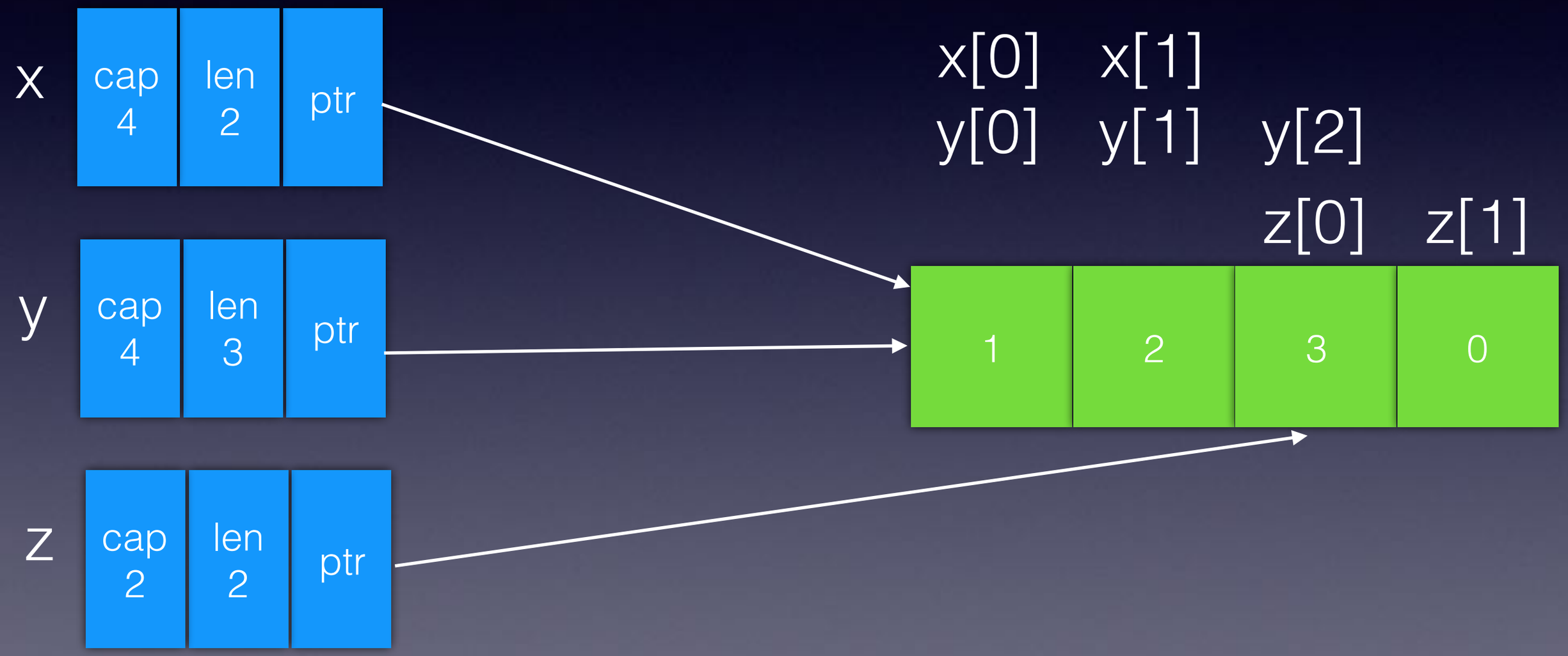


var x := make([]int, 2, 4)



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
```

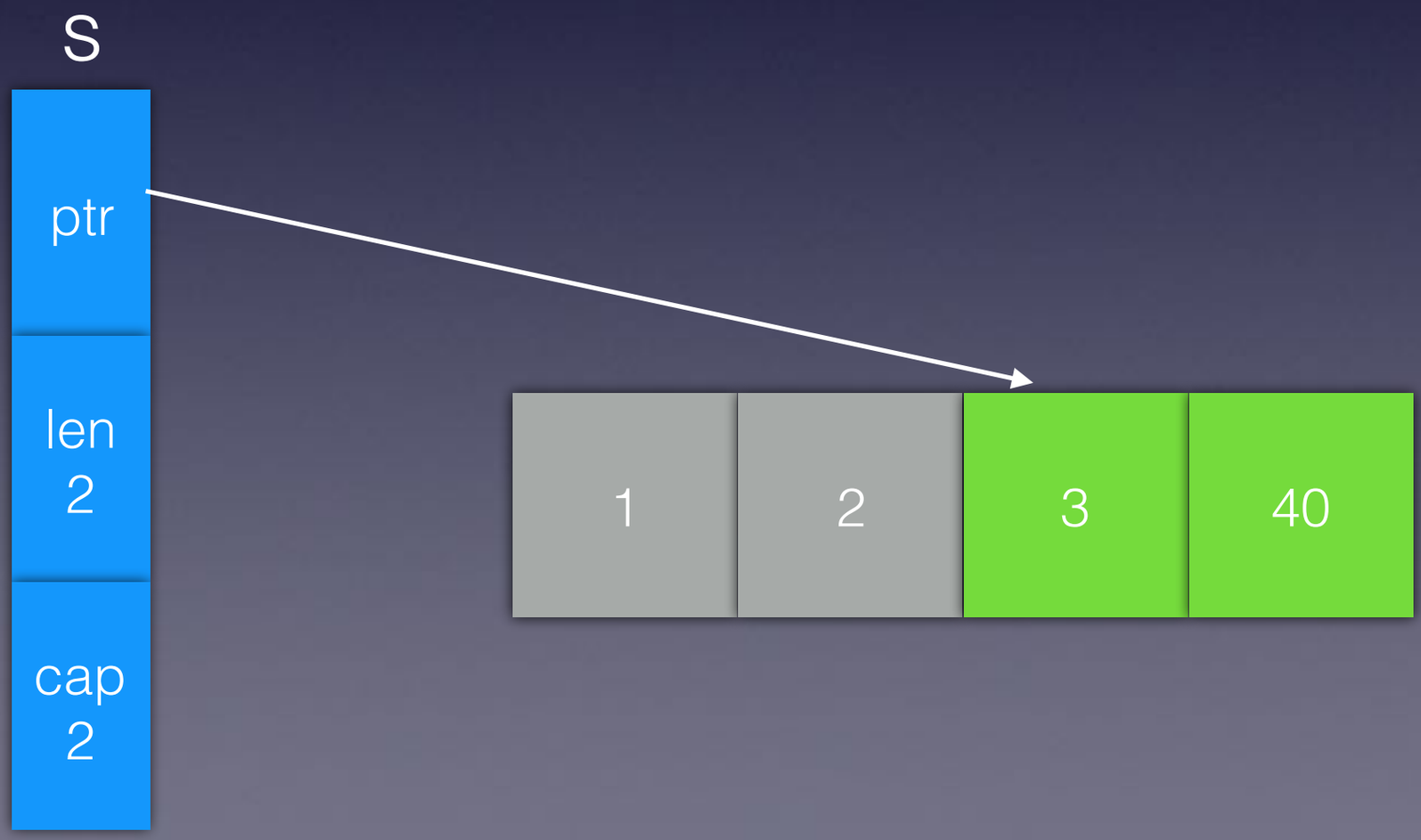


note: cap=number of physical elements beginning from element pointed to
note: _:= x[2:] cause panic error since x[2] does not exist (while x[:2] is ok)

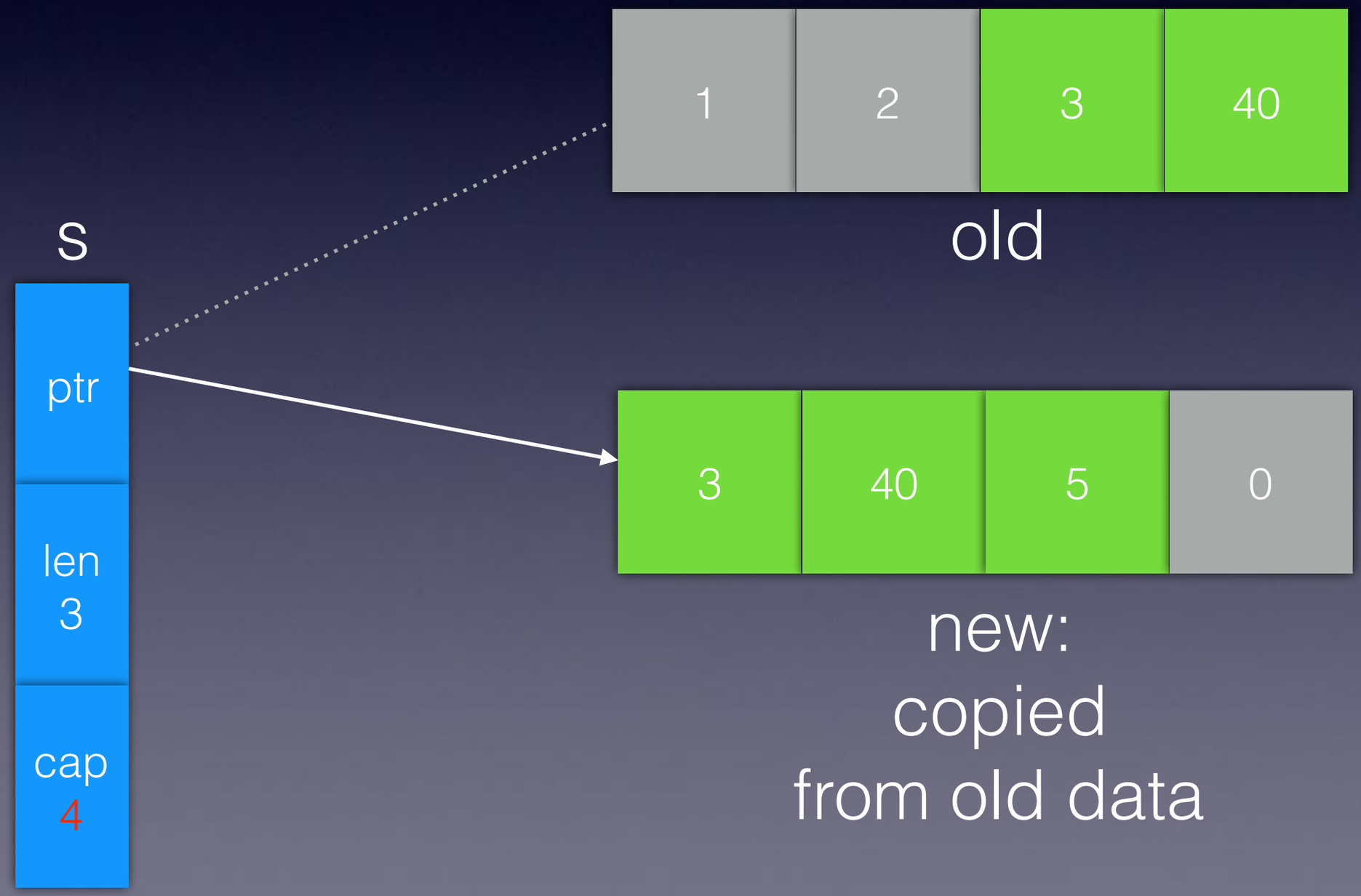
Expanding Slice

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

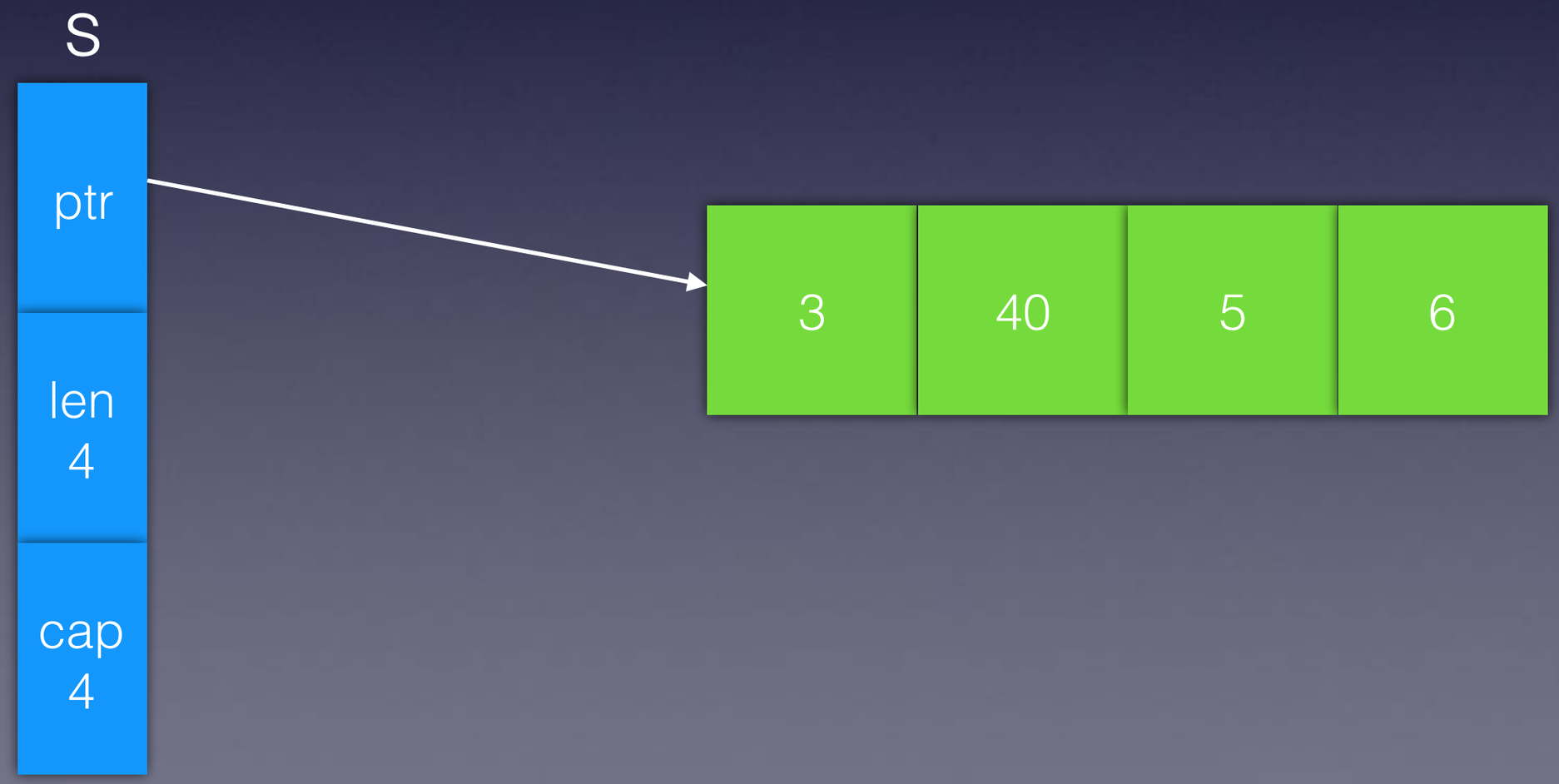
```
var b = […]int{ 1, 2, 3, 4 }
s:= b[2:]
s[1]*= 10
```



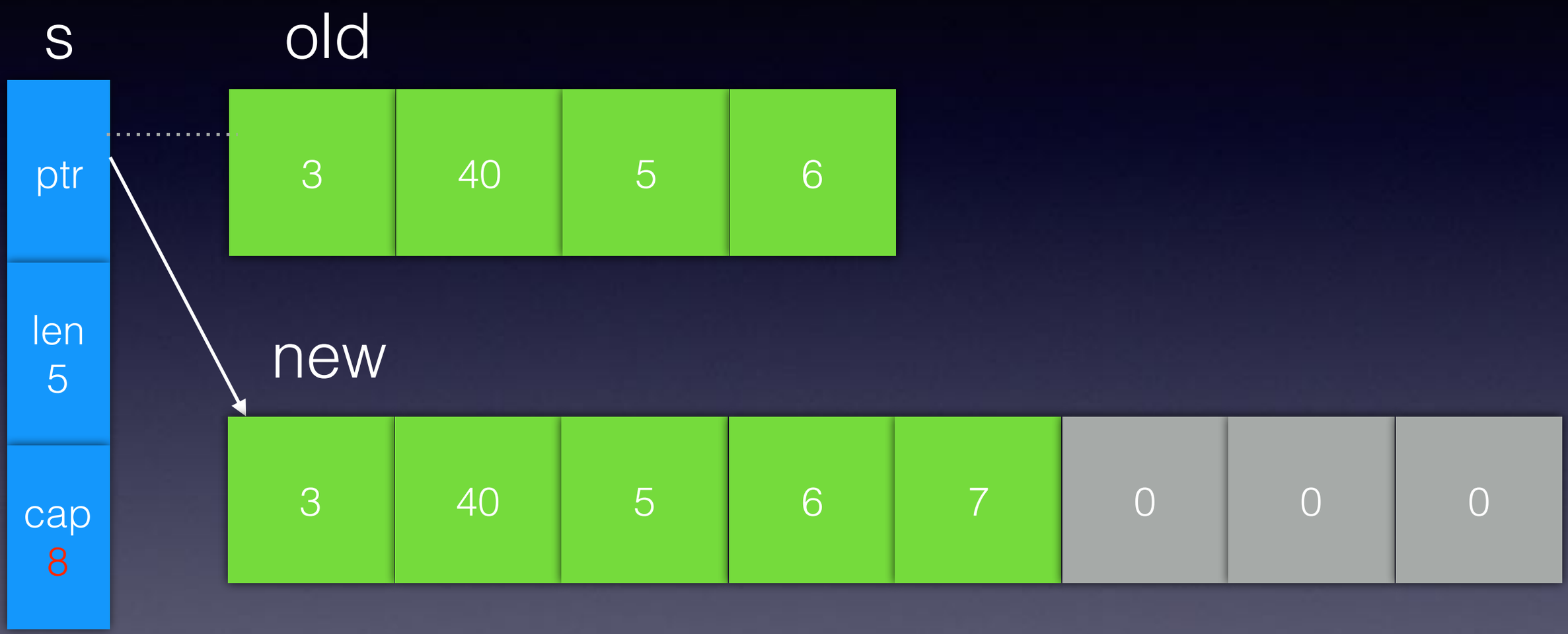
s=append(s, 5)



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
  ```
  ```