



# Golang Programming

Composite types, functions, error handling

# Where to Find The Code and Materials?

<https://github.com/iproduct/coursego>

# Arrays

- Example:

```
var a [10]int
for i := 0; i < 10; i++ {
    fmt.Printf("Element: a[%d] = %d\n", i, a[i])
}
```

- In Go, arrays are a low-level data structure - blocks of memory.
- In C, array subscripting is just another way of writing pointer arithmetic, but Go does not permit pointer arithmetic - pointers and arrays are distinct types => no arbitrary-sized array.
- The size of a Go array is intrinsic to its type, automatic bounds checking.

# Initializing Arrays

- Creating array of size 50 filled with zeros:

```
var a1 [50]int
```

- Creating 2-D matrix 5x5:

```
var matrix [5][5]float64
```

- Initializing array using array literal:

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

# Length and capacity

```
var a [2]string  
a[0] = "Hello"  
a[1] = "World"  
fmt.Println(a[0], a[1])  
fmt.Println(len(a))  
fmt.Println(cap(a))
```

Results:

Hello World

2

2

# Assigning Array Values

- Assigning array value **copies that value** (potentially very slow):

```
a1 := [...]int{1, 2}
a2 := a1
a2[0] = 3
fmt.Printf("%v, %v, %t\n", a1, a2, &a1 == &a2)
```

Result: [1 2], [3 2], false

- Go makes it **possible** to write **fast code**, but makes it **easy** to write **correct code**. This is the opposite of the C philosophy, which makes it **easy** to write **fast code** and **possible** to write **correct code**. [Chisnall, The Go Programming Language Phrasebook]

# Slices

- Create by slicing existing array:

```
var a3[20]int
firstHalf := a3[:10]
secondHalf := a3[10:]
middle := a3[5:15]
all := a3[:]
fmt.Printf("%v, %v, %v, %v\n", firstHalf, secondHalf, middle, all)
```

- Create using composite literal and by re-slicing:

```
var slice []int = []int{2, 3, 5, 7, 11, 13}
fmt.Println(slice) // [2 3 5 7 11 13]
reslice := slice[2:5]
fmt.Println(reslice) // [5 7 11]
```

# Making Slices, Maps and Channels

Call	Type T	Result
make(T, n)	slice	slice of type T with length n and capacity n
make(T, n, m)	slice	slice of type T with length n and capacity m
make(T)	map	map of type T
make(T, n)	map	map of type T with initial space for approximately n elements
make(T)	channel	unbuffered channel of type T
make(T, n)	channel	buffered channel of type T, buffer size n



# Making Slices and Reslicing

```
func main() {  
    a := make([]int, 5) // Len(a)=5  
    printSlice("a", a) // a Len=5 cap=5 [0 0 0 0 0]  
    b := make([]int, 0, 5) // Len(b)=0, cap(b)=5  
    printSlice("b", b) // b Len=0 cap=5 []  
    b = b[:cap(b)] // Len(b)=5, cap(b)=5  
    printSlice("b", b) // b Len=5 cap=5 [0 0 0 0 0]  
    b = b[1:] // Len(b)=4, cap(b)=4  
    printSlice("b", b) // b Len=4 cap=4 [0 0 0 0]  
}  
  
func printSlice(s string, x []int) {  
    fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x)  
}
```

# Making Slices and Reslicing

```
func main() {  
    a := make([]int, 5, 10)  
    printSlice("a", a)           // a len=5 cap=10 [0 0 0 0 0]  
    b := make([]int, 0, 5)  
    printSlice("b", b)           // b len=0 cap=5 []  
    c := b[:2]  
    printSlice("c", c)           // c len=2 cap=5 [0 0]  
    d := c[2:4:5]  
    printSlice("d", d)           // d len=2 cap=3 [0 0]  
    e := a[2:5:10]  
    printSlice("e", e)           // e len=3 cap=8 [0 0 0]  
}  
  
func printSlice(s string, x []int) {  
    fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x)  
}
```

# Slices Are Like References to Arrays

```
func main() {  
    names := [4]string{"John", "Paul", "George", "Ringo"}  
    fmt.Println(names) // [John Paul George Ringo]  
  
    a := names[1:2]  
    b := a[2:3]  
    fmt.Println(a, b) // [Paul] [Ringo]  
    b[0] = "XXX"  
    fmt.Println(a, b) // [Paul] [XXX]  
    fmt.Println(names) // [John Paul George XXX]  
}
```

# Nil Slices

```
func main() {  
    var s []int  
    fmt.Println(s, len(s), cap(s)) // [] 0 0  
    if s == nil {  
        fmt.Println("nil!") // nil!  
    }  
}
```

# Slices of Slice

*// Create a tic-tac-toe board.*

```
board := [][]string{
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
    []string{"_", "_", "_"},
}
```

*// The players take turns.*

```
board[0][0] = "X"
board[2][2] = "O"
board[1][2] = "X"
board[1][0] = "O"
board[0][2] = "X"
```

```
for i := 0; i < len(board); i++ {
    fmt.Printf("%s\n", strings.Join(board[i], " "))
}
```

# More Examples

```
q := []int{2, 3, 5, 7, 11, 13}
fmt.Println(q) // [2 3 5 7 11 13]
```

```
r := []bool{true, false, true, true, false, true}
fmt.Println(r) // [true false true true false true]
```

```
s := []struct {
    i int
    b bool
}{
    {2, true},
    {3, false},
    {5, true},
    {7, true},
    {11, false},
    {13, true},
}
fmt.Println(s) // [{2 true} {3 false} {5 true} {7 true} {11 false} {13 true}]
```

# Appending to a Slice

```
var s []int
printSlice(s) // len=0 cap=0 []
s2 := append(s, 0) // append works on nil slices.
printSlice(s2) // len=1 cap=1 [0]
s3 := append(s2, 1) // The slice grows as needed.
printSlice(s3) // len=2 cap=2 [0 1]
fmt.Printf("Same array: %t\n", &s3[0] == &s2[0]) // Same array: false

a := [...]int{2,3,5,7,9}
s4 := a[1:3]
printSlice(s4) // len=2 cap=4 [3 5]
s5 := append(s4, 11, 13)
printSlice(s5) // len=4 cap=4 [3 5 11 13]
fmt.Printf("Same array: %t\n", &s5[0] == &s4[0]) // Same array: true
s6 := append(s5, 17)
printSlice(s6) // len=5 cap=8 [3 5 11 13 17]
fmt.Printf("Same array: %t\n", &s6[0] == &s5[0]) // // Same array: false
```

# Slice Range

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

---

```
func main() {
    pow := make([]int, 10)
    for i := range pow {
        pow[i] = 1 << uint(i) // == 2**i
    }
    for _, value := range pow {
        fmt.Printf("%d\n", value)
    }
}
```



# Go Slices: Usage and Internals [Go Blog]

<https://blog.golang.org/go-slices-usage-and-internals>

# Exercise 1: Drawing an Image

- Implement `Pic`. It should return a slice of length `dy`, each element of which is a slice of `dx` 8-bit unsigned integers. When you run the program, it will display your picture, interpreting the integers as grayscale (well, bluescale) values.
- The choice of image is up to you. Interesting functions include  $(x+y)/2$ ,  $x*y$  and  $x^y$ .
- You need to use a loop to allocate each `[]uint8` inside the `[][]uint8`.

```
package main
import ("github.com/iproduct/coursego/simple/mypic"; "log"; "os"; "path")
const baseDir = "d:/CourseGO/workspace/src/github.com/iproduct/coursego/image"
// Pic returns a grayscale pic of size dy * dx
func Pic(dx, dy int) [][]uint8 {
}
func main() {
    file, err := os.Create(path.Join(baseDir, "image.png"))
    defer file.Close()
    if err != nil { log.Fatal(err) }
    mypic.Encode(Pic, file)
}
```

# Maps

```
type Vertex struct {  
    Lat, Long float64  
}  
  
var m map[string]Vertex  
  
func main() {  
    m = make(map[string]Vertex)  
    m["Bell Labs"] = Vertex{  
        40.68433, -74.39967,  
    }  
    fmt.Println(m["Bell Labs"])  
}
```

# Maps Literals

```
type Vertex struct {  
    Lat, Long float64  
}
```

```
var m = map[string]Vertex{  
    "Bell Labs": Vertex{ 40.68433, -74.39967 },  
    "Google"   : Vertex{ 37.42202, -122.08408 },  
}
```

```
func main() {  
    fmt.Println(m)  
}
```

# Maps Literals Shortcut

```
type Vertex struct {  
    Lat, Long float64  
}
```

```
var m = map[string]Vertex{  
    "Bell Labs": {40.68433, -74.39967},  
    "Google":    {37.42202, -122.08408},  
}
```

```
func main() {  
    fmt.Println(m)  
}
```

# Mutating Maps

- `m := make(map[string]int)`

```
m["Answer"] = 42
```

```
fmt.Println("The value:", m["Answer"]) // The value: 42
```

```
m["Answer"] = 48
```

```
fmt.Println("The value:", m["Answer"]) // The value: 48
```

```
delete(m, "Answer")
```

```
fmt.Println("The value:", m["Answer"]) // The value: 0
```

```
v, ok := m["Answer"]
```

```
fmt.Println("The value:", v, "Present?", ok) // 0 Present? false
```

## Exercise 2: Word Counting

- Implement `WordCount`. It should return a map of the counts of each “word” in the string `s`. The `wc.Test` function runs a test suite against the provided function and prints success or failure. (You might find [strings.Fields](#) helpful):

```
package main
import (
    "golang.org/x/tour/wc"
)
func WordCount(s string) map[string]int {
    return map[string]int{"x": 1}
}
func main() {
    wc.Test(WordCount)
}
```

# Map Ranges

```
func countLines(f *os.File, counts map[string]int) {  
    input := bufio.NewScanner(f)  
    for input.Scan() {  
        counts[input.Text()]++  
    }  
  
    if input.Err() != nil {...}  
}
```

```
func main() {  
    files := os.Args[1:]  
    counts := make(map[string]int)  
    countLines(os.Stdin, counts)  
    for key, val := range counts {  
        fmt.Printf("%-20.20s -> %5d\n", key, val)  
    }  
}
```



# Structs

```
type Vertex struct{ X, Y int }
type Line struct{ A, B *Vertex }
var gv Vertex = Vertex{2, 5}
var gv2 Vertex = Vertex{12, 29}
var g1 Line = Line{&gv, &gv2}

func test(l Line) {
    fmt.Printf("%v, same=%v\n", l, l.A == g1.A)
    l.B.X = 42
    fmt.Printf("%v, %v\n", *l.A, *l.B)
}

func main() {
    test(g1)
    fmt.Printf("%v, %v\n", *g1.A, *g1.B)
}
```

# Struct Literals

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

```
var (  
    v1 = Vertex{1, 2} // has type Vertex  
    v2 = Vertex{X: 1} // Y:0 is implicit  
    v3 = Vertex{}     // X:0 and Y:0  
    p  = &Vertex{1, 2} // has type *Vertex  
)
```

```
func main() {  
    fmt.Println(v1, p, v2, v3) // {1 2} &{1 2} {1 0} {0 0}  
    p := &v1  
    p.X = 1e9  
    fmt.Println(v1) // {1000000000 2}  
}
```

# Rules of Struct Literals

- A **key** must be a **field name** declared in the struct type.
- An element list that **does not contain any keys** must list an element for **each struct field** in the order in which the fields are declared.
- If **any element has a key**, every element **must have a key**.
- An element **list that contains keys** does not need to have an element for each struct field. **Omitted fields get the zero value** for that field.
- A literal may **omit the element list**; such a literal evaluates to the **zero value** for its type.
- It is an **error** to specify an element for a **non-exported field** of a struct belonging to a **different package**.

# Recursion

*//This fact function calls itself until it reaches the base case  
//of fact(0).*

```
func fact(n int) int {  
    if n == 0 {  
        return 1  
    }  
    return n * fact(n-1)  
}  
func main() {  
    fmt.Println(fact(7))  
}
```

# Functions – multiple return values

```
func swap(x, y string) (string, string) {  
    return y, x  
}
```

```
func main() {  
    a, b := swap("hello", "world")  
    fmt.Println(a, b)  
}
```

# Functions - named return values

```
func split(sum int) (x, y int) {  
    x = sum * 4 / 9  
    y = sum - x  
    return  
}
```

```
func main() {  
    fmt.Println(split(17))  
}
```

# Value vs. Pointer Parameters

```
• func swapVal(x, y string) (string, string) {  
    return y, x  
}  
func swapRef(x, y *string) {  
    *x, *y = *y, *x  
}  
  
func main() {  
    a, b := swapVal("hello", "world")  
    fmt.Println(a, b)  
    swapRef(&a, &b)  
    fmt.Println(a, b)  
}
```

Output:

```
world hello  
hello world
```

# Variadic Parameters

```
func printf(format string, args ...interface{}) (int, error) {  
    _, err := fmt.Printf(format, args...)   
    return len(args), err  
}
```

```
func main() {  
    argsLen, err := printf("%v, %v\n", "abcd", 15)  
    if err == nil {  
        printf("Number args: %d\n", argsLen)  
    } else {  
        fmt.Printf("Error: %v\n", err)  
    }  
}
```



# Function Values, Anonymous Functions, Closures

```
count := 0
inc := func() int {
    count++
    return count
}

incBy := func(n int) int {
    count += n
    return count
}

printf("%d\n", inc())
printf("%d\n", incBy(10))
```

# Deferred Function Calls

```
func main() {  
    defer fmt.Println("world")  
  
    fmt.Println("hello")  
}
```

Results:

hello

world

# Stacking Deferred Function Calls

```
• func main() {  
    fmt.Println("counting")  
  
    for i := 0; i < 10; i++ {  
        defer fmt.Println(i)  
    }  
  
    fmt.Println("done")  
}
```

Results: ?

# Error Handling Strategies

- Propagate the error, so that the failure of the subroutine becomes caller's failure. Using `fmt.Errorf` function formats and returns a new error value possibly extending the error description with more context.
- Retry the failed operation, possibly with (exponential) delay between tries
- Print the error and stop the program gracefully – `log.Fatal()` / `os.Exit(1)`
- Just log the error and then continue, possibly with alternative approach
- Using `panic()` and `recover()`
- More about error handling in Go:

<https://blog.golang.org/error-handling-and-go>

[https://golang.org/doc/effective\\_go.html#errors](https://golang.org/doc/effective_go.html#errors)

# Errors <https://golang.org/pkg/errors/>, <https://blog.golang.org/go1.13-errors>

```
type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s",
        e.When, e.What)
}

func run() error {
    return &MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```

# Errors Summary

- Errors should implement the built-in, universally accessible error interface:

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

- They can have additional fields capturing the complete error context - Ex:

```
type PathError struct {  
    Op string    // "open", "unlink", etc.  
    Path string  // The associated file.  
    Err error      // Returned by the system call.  
}  
  
func (e *PathError) Error() string {  
    return e.Op + " " + e.Path + ": " + e.Err.Error()  
}
```

- Callers that care about the error details can use a type switch or assertion:

```
if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC { ...
```

# Example Handling PathError

```
for try := 0; try < 2; try++ {  
    file, err := os.Create(filename)  
    if err == nil {  
        return  
    }  
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {  
        deleteTempFiles() // Recover some space.  
        continue  
    }  
    return  
}  
  
// Do something useful with the created file ...
```

# Panic [[https://golang.org/doc/effective\\_go.html#panic](https://golang.org/doc/effective_go.html#panic)]

```
func badFunction() {  
    fmt.Printf("Select Panic type (0=no, 1=int, 2= panic)\n")  
    var choice int  
    fmt.Scanf("%d", &choice)  
    switch choice {  
        case 1:  
            panic(0)  
        case 2:  
            var invalid func();  
            invalid()  
    }  
}
```



# Recover [[https://golang.org/doc/effective\\_go.html#recover](https://golang.org/doc/effective_go.html#recover)]

```
func main() {  
    defer func() {  
        if x := recover(); x != nil {  
            switch x.(type) {  
            default:  
                panic(x)  
            case int:  
                fmt.Printf("Function panicked with an error: %d\n", x)  
            }  
        }  
    }()  
    badFunction()  
    fmt.Printf("Program exited normally\n")  
}
```

# Using Panic/Recover to Shut Down Failing Goroutine

```
func server(workChan <-chan *Work) {  
    for work := range workChan {  
        go safelyDo(work)  
    }  
}  
  
func safelyDo(work *Work) {  
    defer func() {  
        if err := recover(); err != nil {  
            log.Println("work failed:", err)  
        }  
    }()  
    do(work)  
}
```

# Converting Panic to Error at API Boundary (regex)

*// Error is the type of a regex parse error; it satisfies the error interface.*

```
type Error string
func (e Error) Error() string {
    return string(e)
}
```

*// error is a method of \*Regexp that reports parsing errors by panicking with an Error.*

```
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}
```

*// Compile returns a parsed representation of the regular expression.*

```
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error.
        }
    }()
    return regexp.doParse(str), nil
}
```

# Homework 1 (algorithmic problem)

Имаме **n** човека наредени в кръг с номера от 1 до **n**, които участват в игра на броење наречена броенка. Играта е със следните правила:

Започваме да броим от човека с номер 1.

Отброяваме **m** човека участващи в кръга. Последният отброен човек (с номер **m**) излиза от кръга.

Повтаряме стъпка 2 (продължавайки да броим от следващия участник), докато в кръга остане само един участник. Нека номерът на участника да бъде **p**.

Създайте функция **findWinner(n, m int) int**, която по подадени като аргументи **n** и **m** връща **p**.

Напишете и **main** функция която да въвежда от клавиатурата **n** и **m** и да отпечата **p** на екрана.

Примерни данни **findWinner( 8, 3 ) --> 7, findWinner( 11, 5 ) --> 8.**

# Recommended Literature

- The Go Documentation - <https://golang.org/doc/>
- The Go Bible: Effective Go - [https://golang.org/doc/effective\\_go.html](https://golang.org/doc/effective_go.html)
- David Chisnall, *The Go Programming Language Phrasebook*, Addison Wesley, 2012
- Alan A. A. Donovan, Brian W. Kernighan, *The Go Programming Language*, Addison Wesley, 2016
- Nathan Youngman, Roger Peppé, *Get Programming with Go*, Manning, 2018
- Naren Yellavula, *Building RESTful Web Services with Go*, Packt, 2017

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>