

GO
for Tourist



Course Overview

- What is GO (golang)?
- Why learn GO?
- Why this course?
- Course objectives
- Pre-requisites
- Course outline



What is Go?

- Developed by Google in 2007, public 2009
 - **GO** (often referred to as **golang**)
- **Open Source & Cross-platform**
- **Simple, reliable**, and **efficient**
- **Compiled & statically typed**
- **Garbage collection & memory safety features**
- **CSP**-style concurrency

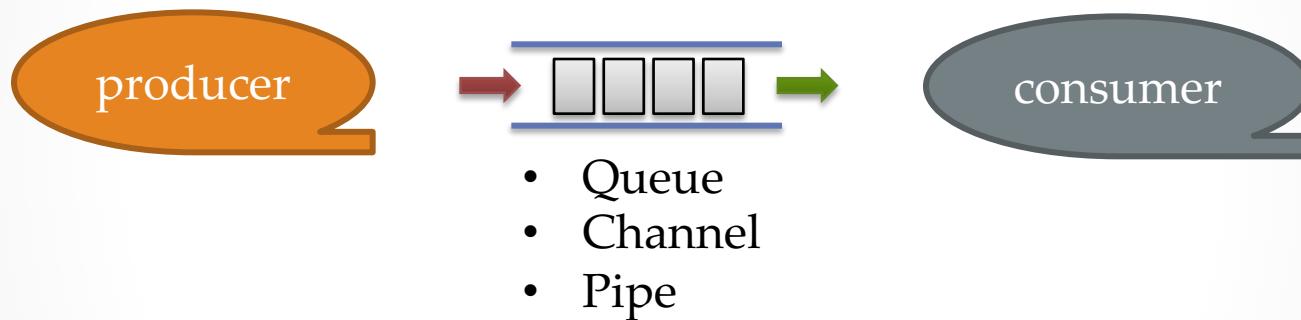


Why learn GO?

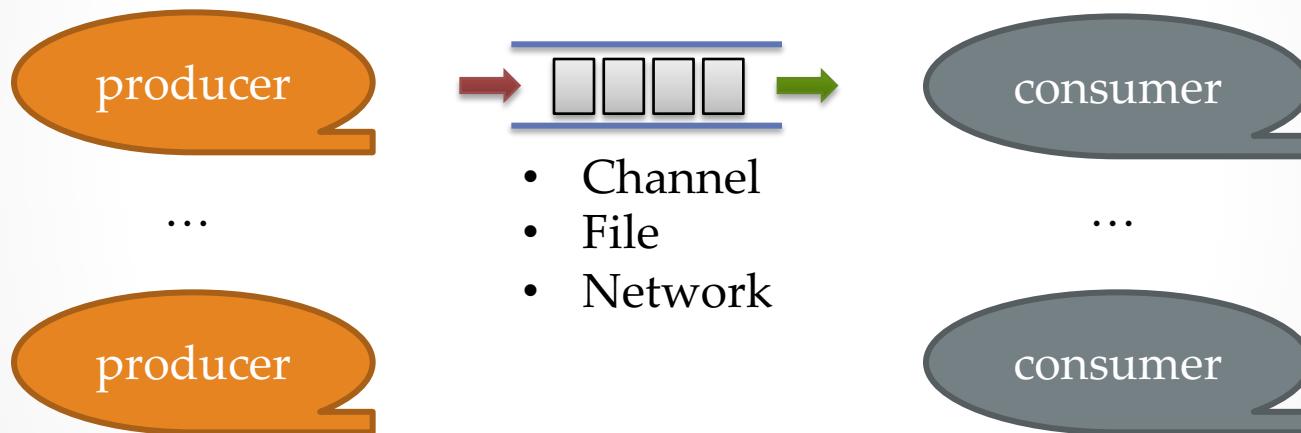
- *Job security*
- *It is fun and fairly easy*
- *Concurrency*
 - easier to implement and explain
- *Build interesting software and tools*



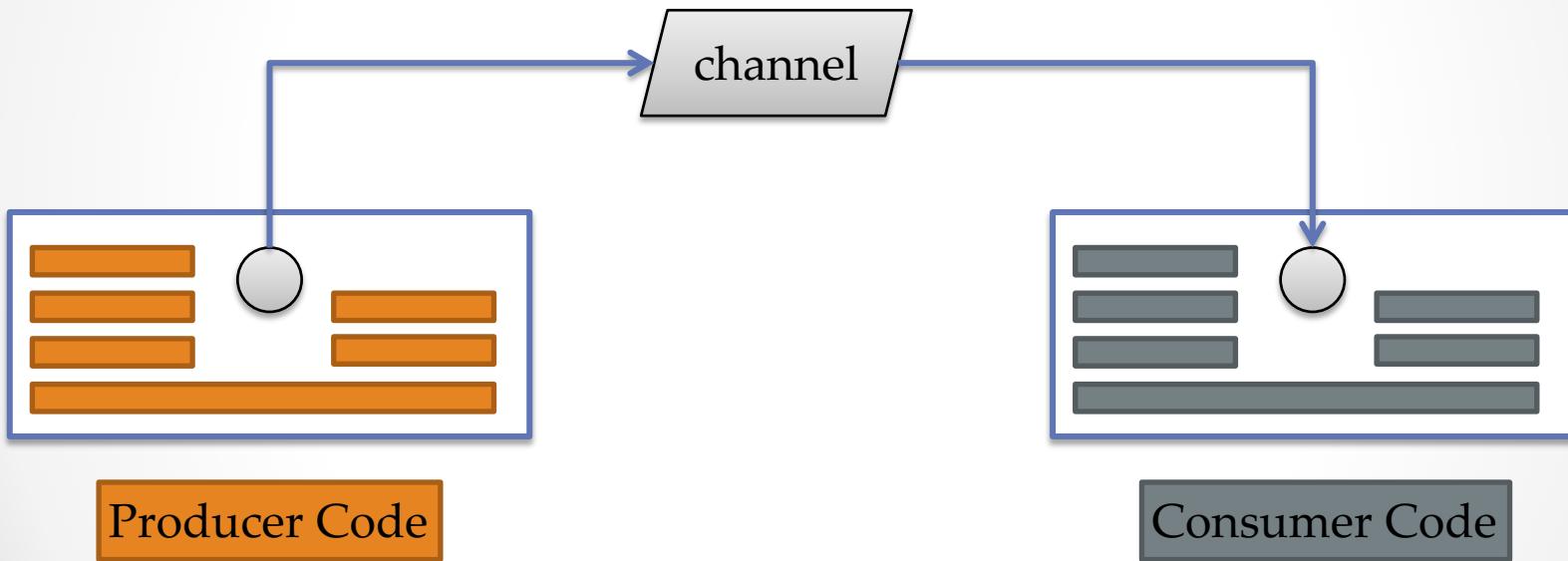
Example



Example



Example



```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func producer(name string, c chan string) {
9     for i := 0; i < 5; i++ {
10         c <- "message from " + name
11     }
12 }
13 func consumer(c chan string) {
14     for v := range c {
15         fmt.Println("Got", v)
16     }
17 }
18 func main() {
19     var c = make(chan string)
20     go producer("producer 1", c)
21     go producer("producer 2", c)
22     go consumer(c)           // launch multiple consumers with 'go consumer(c)'
23     time.Sleep(1 * time.Second) // wait until work is done
24 }
25
```

```
Got message from producer 1
Got message from producer 1
Got message from producer 2
Got message from producer 1
Got message from producer 1
Got message from producer 1
```

```
Got message from producer 1
```



Why this course?

- Scope
 - Introduction to GO Language
 - Covers major language features
 - **NOTE:** Not in too much details



Course objective

- Foundation for everyday GO programming
 - Structure of a GO application
 - Container Types: Arrays, Slices, Maps, Structs
 - GO Concurrency using Channels and GO Routines
 - Data Input/output
 - File Operation
 - Network Programming
- Illustrate through examples



Pre-requisites

- Basic computer knowledge, such as;
 - a. how to navigate around your computer
 - b. how to use a basic text editor
- No programming experience required
 - However, some programming experience will help
 - Non-programmers should be prepared to do all exercises
 - adding reading to fill-in the gaps when necessary



Course Outline

- Section 1: Installation and Setup
- Section 2: Basic Concepts
- Section 3: Arrays & Slices
- Section 4: Maps
- Section 5: Structs
- Section 6: Go-routines
- Section 7: Channels
- Section 8: Pointers
- Section 9: Interfaces
- Section 10: Standard I/O & File I/O
- Section 11: Networking



Section 1: Installation & Setup

- Software Installation
- Configuration/Setup



Software Installation

Section 1 – Lecture 1



Topics

- Installation of:
 - GO Tools
 - Code Editor – Visual Studio Code
 - Optional
 - Version Control - Git



Go Tools Installation

- Available
 - <http://golang.org/dl>
- Follow the installation directions for your OS
 - Best to keep the defaults



Code Editor Installation

- Visual Studio Code Editor
- Simple and fairly easy
- Great support for GO Programming
- Available
 - <https://code.visualstudio.com/download>
 - Or search for “vscode download”



Git Installation

- Source Code Management
- Available
 - <https://git-scm.com>
- Follow the installation directions for your OS
 - Best to keep the defaults
 - **IMPORTANT:**
 - **Windows user, accept ‘Bash’ shell**



Verify GO Tool installation

- Check that the '**go**' command is in your **PATH**:

```
$ go env
```

```
~ > > go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/another/Library/Caches/go-build"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/another/go"
GORACE=""
GOROOT="/usr/local/go"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
```



Resources

- Go Tools
 - <https://golang.org/dl>
- Visual Studio Code
 - <https://code.visualstudio.com/download>
- Git
 - <https://git-scm.com>



Configuration & Setup

Section 1 – Lecture 2



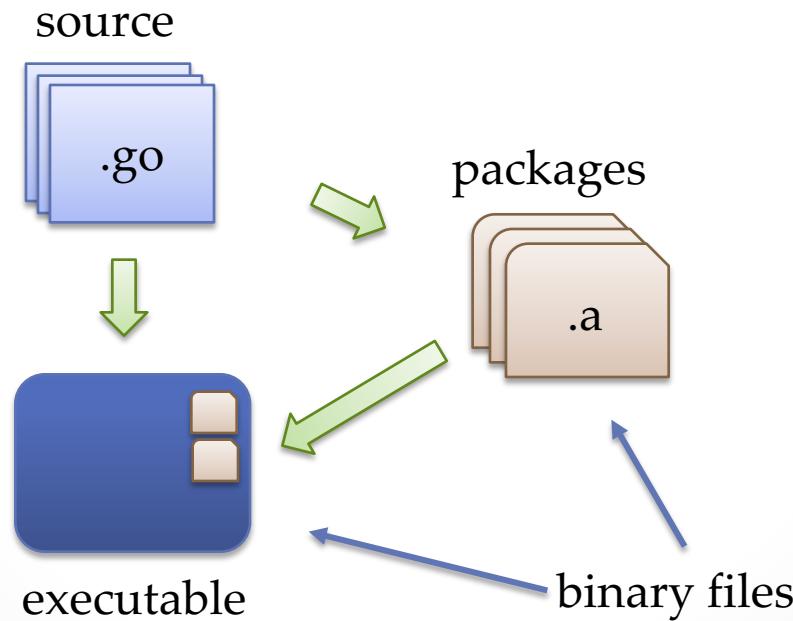
Topics

- GO artifacts
- Code organization
 - Modules
- Set code directory
- Configure VSCode for GO
- Configure Git



GO Artifacts

- GO Source to Binary

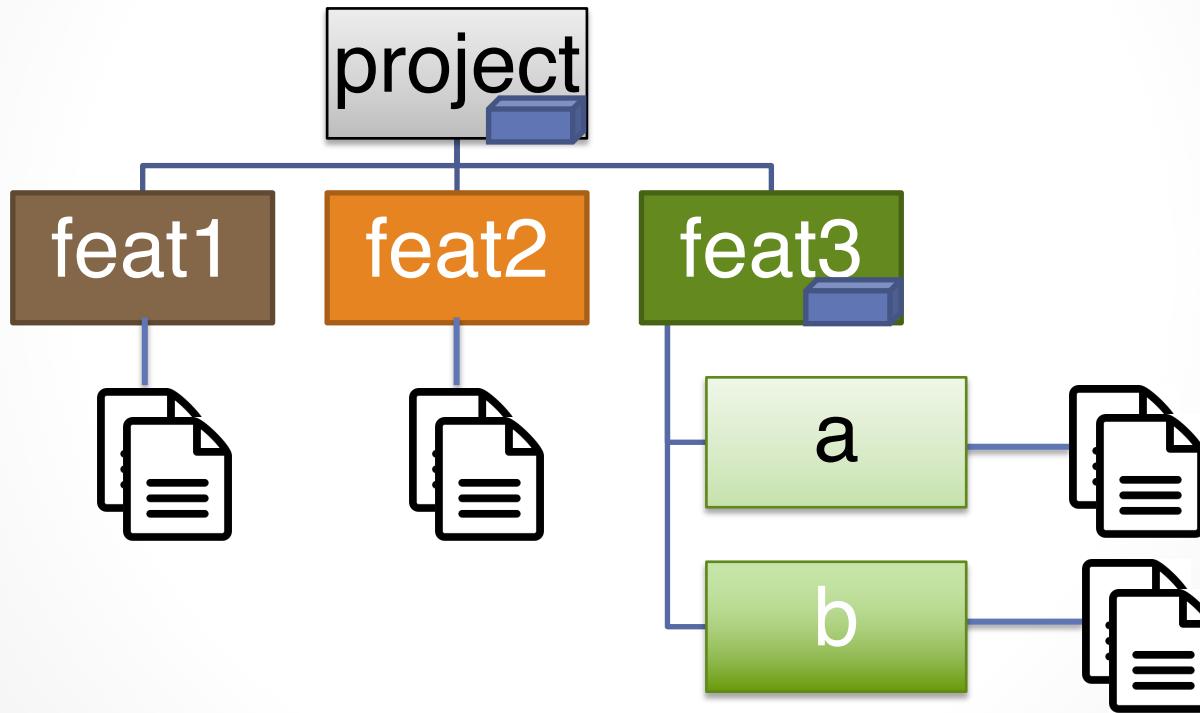


GO Code Organization

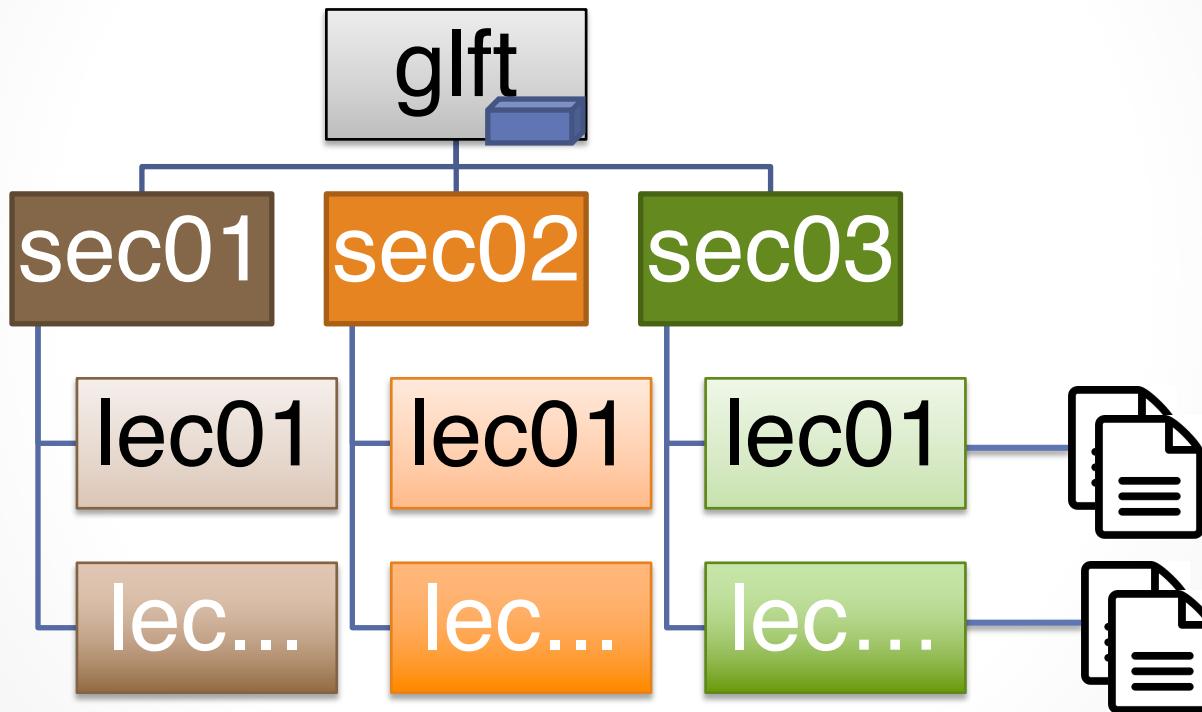
- A **package** contains one or more GO source files
 - NOTE: A GO source file must be in **exactly one package**
 - **Packages** are reusable pieces of code, and works to modularize a large project
- A typically GO project (application or package), is developed in **modules**
- A **module** provides versioning for the **packages** used in your code



Modules



glft Module



Your GO Source Dir

- Is any directory on your filesystem
- Typically, you will one to have a directory into which you create several modules
 - Within which is one directory per module



GOBIN Directory

- GOBIN is the directory GO uses to store binaries
 - These are applications installed using:
 - go get
 - go install



Setting GOBIN & PATH

1. Set GOBIN Variable

- GOBIN=\${HOME}/go/bin
 - Mac/Linux Users
 - Set GOBIN in your shell login file
Eg: ~/.zshrc, ~/.bashrc, ~/.cshrc
 - Windows Users
- ```
$ echo 'GOBIN=${HOME}/go/bin' >> ~/.bashrc
```

## 2. Set PATH Variable

- PATH=\${GOBIN} :\${PATH}  

```
$ echo 'PATH=${GOBIN} :${PATH}' >> ~/.bashrc
```



# Source Dir

- Source directory

- Get the source

```
$ git clone https://github.com/striversity/glft \
 ${HOME}/glft
```

- From scratch

```
$ mkdir -p ${HOME}/glft
```

- Edit/Create go.mod in your source dir

```
$ cat go.mod
module github.com/<your-name>/glft
```

**NOTE:** It doesn't matter if you have a github repo or not.



# VSCode GO Plugin & Tools

- GO plugins for VisualStudio Code (VS Code)
  1. From the commandline  
\$ code
  2. Install VSCode “GO Language’ plubin by  
'lukehoban'
  3. Restart VSCode
  4. Install GO Tools
    - a. Click on ‘View’ -> ‘Command Pallet...’
    - b. Type ‘install’
    - c. Click on Install GO Tools



# Git Setup (optional)

- Git Config File
  1. Type the following commands:  
`$ git config --global user.email "your@email.com"`  
`$ git config --global user.name "your name"`
  2. Confirm the changes by typing:  
`$ cat ~/.gitconfig`



# Section 2: Basic Concepts

- Anatomy of a Go program
- Boolean and numeric Values
- String and rune Values
- Constants & Variables
- The 'if' Statement
- The 'switch' Statement
- The 'for' Statement
- Functions
- Basic Types
- Packages Part



# Anatomy of a Go program

Section 2 – Lecture 1



# Minimal Go Program

1. Start a new terminal
  - Windows user, start 'Bash Shell' installed with Git
2. Change to the 'glft' project directory
  - `$ cd ${GOPATH}/src/github.com/<yourname>/glft`
3. Start code editor in project directory
  - `$ code .`



# Simple Go Program

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Comment

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Package ‘main’

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Special Function ‘main’

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Importing a Package

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Calling a Function

```
1 // Section 02 - Lecture 01 : Anatomy of a Go Program
2 package main
3
4 import "fmt"
5 /* this is a
6 multi-line comment
7 end it with * followed immediately by /
8 */
9 func main() {
10 fmt.Println("Hello, World!")
11 }
```



# Go Program

main

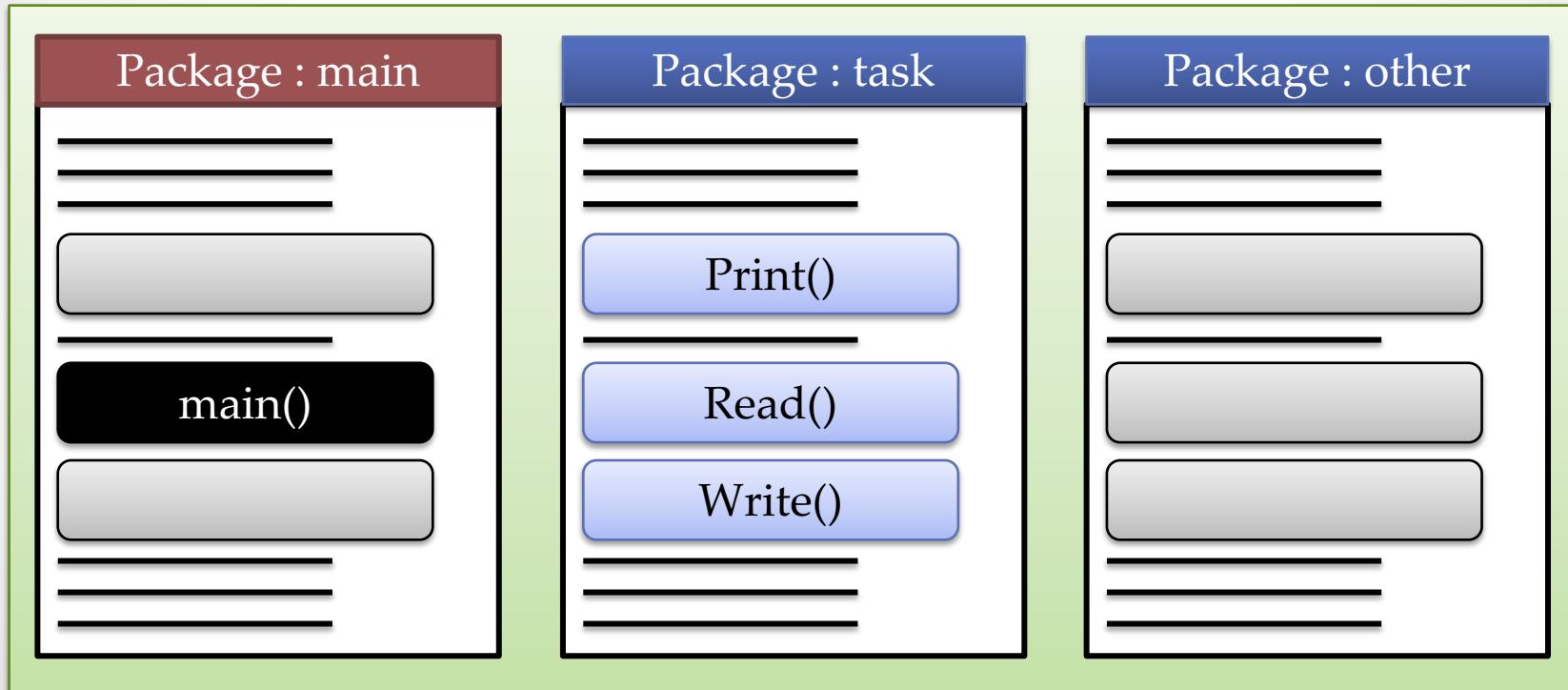
main()

Package A

Package B



# Awesomely Complex Program



# Review

- **Comment** lines begin with ‘//’
- Go source code **must** belong to a **package**
- A Go **program/application** uses one or more **package(s)**
- A **package** provides reusable code as **functions, types, constants, and variables**
- A **package** must be imported before it can be used



# Boolean and Numeric Values

Section 2 – Lecture 2



# Definition: Value

- Verrol's definition:
  - A value is an **abstraction** to **represent** a **computed result**
    - **NOTE:** Not all values are **computed**, some values just **represents themselves**.
    - When a value stands for itself, we call it a **literal**.  
**NOTE**, a **literal** is still a value.



# Examples: Value

- Examples:
  - **6**
    - 6 is a value and also a literal, it stands for itself
  - **3 + 1**
    - will compute the value '4', but we don't see '4' in the expression
    - '4' is the computed resulted or value
  - **3.141592**
  - **true**
    - true is also a literal in Go Language
      - It is the value to represent the boolean 'true' vs 'false'



# Resource

- Logical Operators
  - [https://golang.org/ref/spec#Logical\\_operators](https://golang.org/ref/spec#Logical_operators)
- Integer Literals
  - [https://golang.org/ref/spec#Integer\\_literals](https://golang.org/ref/spec#Integer_literals)
- Floating-points Literals
  - [https://golang.org/ref/spec#Floating-point\\_literals](https://golang.org/ref/spec#Floating-point_literals)
- Imaginary Literals
  - [https://golang.org/ref/spec#Imaginary\\_literals](https://golang.org/ref/spec#Imaginary_literals)
- Arithmetic Operators
  - [https://golang.org/ref/spec#Arithmetic\\_operators](https://golang.org/ref/spec#Arithmetic_operators)
- Comparison Operators
  - [https://golang.org/ref/spec#Comparison\\_operators](https://golang.org/ref/spec#Comparison_operators)



# String and Rune Values

Section 2 – Lecture 3



# String Value

“Hello”

H e l l o



# Runes of a String

“Hello, 世界”

|            |    |    |    |    |    |    |    |      |      |
|------------|----|----|----|----|----|----|----|------|------|
| character  | H  | e  | l  | l  | o  | ,  |    | 世    | 界    |
| rune value | 48 | 65 | 6c | 6c | 6f | 2c | 20 | 4e16 | 754c |



# Strings: A Complicated Story

- A **string** contains 0 or more **characters**
  - Enclosed in “ or ‘
- A **character** is represented by **rune**
  - Enclosed in ‘
- A **rune** is a 32-bit number



# Resource

- ASCII and Unicode Tables:
  - <https://www.asciitable.com/>
- String Literals
  - [https://golang.org/ref/spec#String\\_literals](https://golang.org/ref/spec#String_literals)
- Strings, bytes, runes and characters in Go
  - <https://blog.golang.org/strings>



# Constants and Variables

Section 2 – Lecture 4



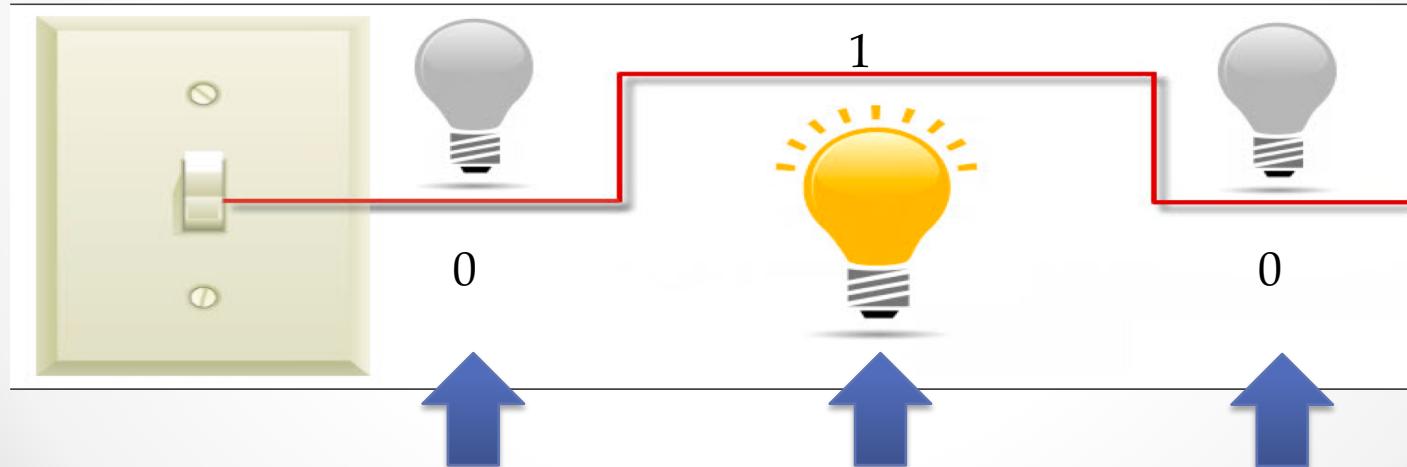
# Everything is a Number

- A **number** is a collection of **bits**
- A **bit** can **store** a value of **0** or **1**



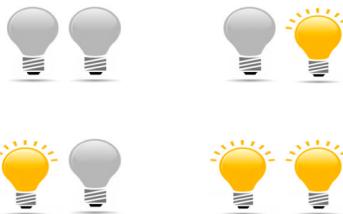
# What is a 'bit'?

- A **bit**, is like a switch, it is either **on** or **off**
- Each **bit** represents a binary value of 0 or 1



# Counting with ‘bits’

- Just as 1 **bit** can designate 2 distinct **states** or **values**, using multiple **bits** can designate even higher numbers of **states/values**.
- A pair of light bulbs, can have 4 distinct states. Hence, 2 **bits** can designate 4 distinct **states/values**.



# Numbering Systems at a Glance

| Binary | Decimal | Hexadecimal |
|--------|---------|-------------|
| 0000   | 0       | 0           |
| 0001   | 1       | 1           |
| 0010   | 2       | 2           |
| 0011   | 3       | 3           |
| 0100   | 4       | 4           |
| 0101   | 5       | 5           |
| 0110   | 6       | 6           |
| 0111   | 7       | 7           |

| Binary | Decimal | Hexadecimal |
|--------|---------|-------------|
| 1000   | 8       | 8           |
| 1001   | 9       | 9           |
| 1010   | 10      | A           |
| 1011   | 11      | B           |
| 1100   | 12      | C           |
| 1101   | 13      | D           |
| 1110   | 14      | E           |
| 1111   | 15      | F           |



# Grouping Bits

- Grouping **bits** to represents **numbers**
  - 4 bits = nibble
  - 8 bits = byte
  - 16 bits
  - 32 bits
  - 64 bits
    - most modern computers use this many bits for **addressing**
  - 128 bits



# Definition: Data & Data Type

- What is 'data'?
  - 'data' is another name for 'value'
- What is 'data type'?
  - Verrol's definition:
    - 'data type' **defines** the:
      - *interpretation* and *representation* of a *value*
      - For example:
        - Integers, floats, boolean
        - **valid operations**



# Signed and Unsigned

| Unsigned |         |             |
|----------|---------|-------------|
| Binary   | Decimal | Hexadecimal |
| 000      | 0       | 0           |
| 001      | 1       | 1           |
| 010      | 2       | 2           |
| 011      | 3       | 3           |
| 100      | 4       | 4           |
| 101      | 5       | 5           |
| 110      | 6       | 6           |
| 111      | 7       | 7           |

| Signed |         |             |
|--------|---------|-------------|
| Binary | Decimal | Hexadecimal |
| 000    | 0       | 0           |
| 001    | 1       | 1           |
| 010    | 2       | 2           |
| 011    | 3       | 3           |
| 111    | -1      | 7           |
| 110    | -2      | 6           |
| 101    | -3      | 5           |
| 100    | -4      | 4           |



# Basic Data Types

- Boolean
  - true/false
- String
- Numeric
  - Integers
    - 8-bits
    - 16-bits
    - 32-bits
    - 64-bits
  - Float
    - 32-bits
    - 64-bits
  - Complex
    - 64-bits
    - 128-bits



# Numeric Types

A *numeric type* represents sets of integer or floating-point values. The predeclared architecture-independent numeric types are:

|                         |                                                                                     |
|-------------------------|-------------------------------------------------------------------------------------|
| <code>uint8</code>      | the set of all unsigned 8-bit integers (0 to 255)                                   |
| <code>uint16</code>     | the set of all unsigned 16-bit integers (0 to 65535)                                |
| <code>uint32</code>     | the set of all unsigned 32-bit integers (0 to 4294967295)                           |
| <code>uint64</code>     | the set of all unsigned 64-bit integers (0 to 18446744073709551615)                 |
| <br>                    |                                                                                     |
| <code>int8</code>       | the set of all signed 8-bit integers (-128 to 127)                                  |
| <code>int16</code>      | the set of all signed 16-bit integers (-32768 to 32767)                             |
| <code>int32</code>      | the set of all signed 32-bit integers (-2147483648 to 2147483647)                   |
| <code>int64</code>      | the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807) |
| <br>                    |                                                                                     |
| <code>float32</code>    | the set of all IEEE-754 32-bit floating-point numbers                               |
| <code>float64</code>    | the set of all IEEE-754 64-bit floating-point numbers                               |
| <br>                    |                                                                                     |
| <code>complex64</code>  | the set of all complex numbers with float32 real and imaginary parts                |
| <code>complex128</code> | the set of all complex numbers with float64 real and imaginary parts                |
| <br>                    |                                                                                     |
| <code>byte</code>       | alias for <code>uint8</code>                                                        |
| <code>rune</code>       | alias for <code>int32</code>                                                        |

The value of an  $n$ -bit integer is  $n$  bits wide and represented using two's complement arithmetic.



# Constants

- What is a 'constant'?
  - A **constant** is an **identifier** used in place of a **value**
- Property:
  - The **value** of a **constant** **can't** be changed after it is assigned
- Declaration:
  1. Un-typed  
*const identifier = value*
  2. Typed  
*const identifier type = value*



# Examples: Constant

- `const pi = 3.1415`
- `const secondsInHour = 60`
- `const hoursInDay = 24`
- `const minAge uint8 = 18`
- `const favoriteLanguage = "Go Language"`



# Variables

- What is a '**variable**'?
  - Associates an identifier with storage in memory
  - Uses the type to enforces a set of values
- Property:
  - **Variables** can have different values over their lives
- Declaration:
  1. Default value  
`var identifier type`
  2. Assigned value  
`var identifier [type] = value`
  3. Short declaration  
`identifier := value`



# Example: Variable

- `var itemCount uint`
- `var price`
- `var myName string`
- `var isLoggingEnabled = true`
- `var _weird = 3 * 25 * (36 – hoursInDay)`



# Resources

- Constants:
  - <https://golang.org/ref/spec#Constants>
- Variables:
  - <https://golang.org/ref/spec#Variables>
- Numeric Types:
  - [https://golang.org/ref/spec#Numeric\\_types](https://golang.org/ref/spec#Numeric_types)
- Two's Complement Calculator
  - <http://www.convertforfree.com/twos-complement-calculator/>



# The 'if' Statement

Section 2 – Lecture 5

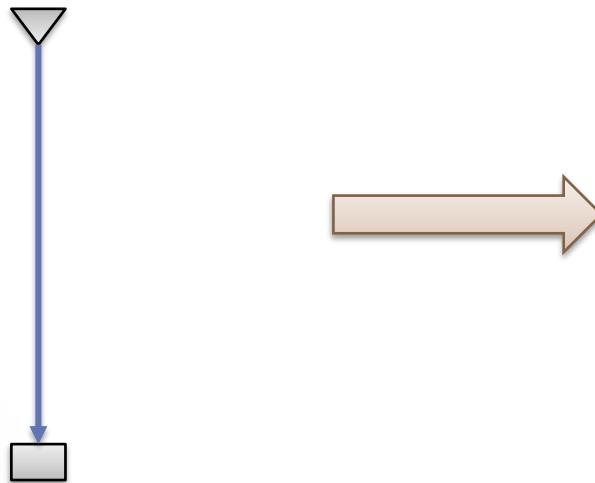


# Our Programs So Far

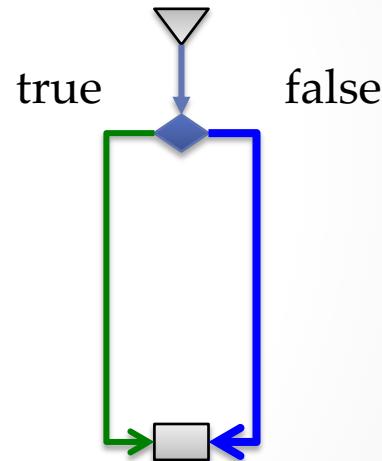


# What “if”?

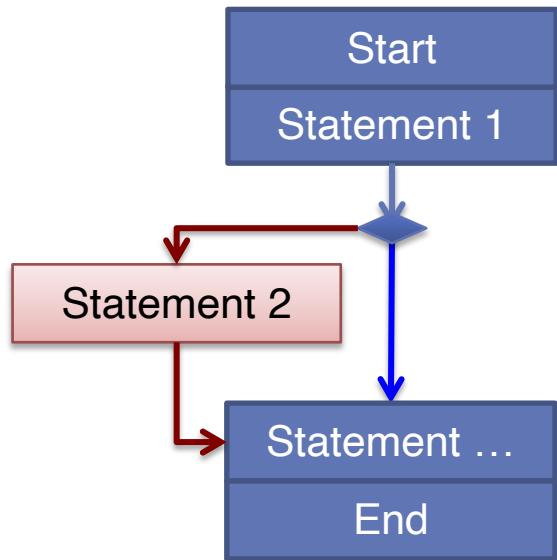
Instead of this



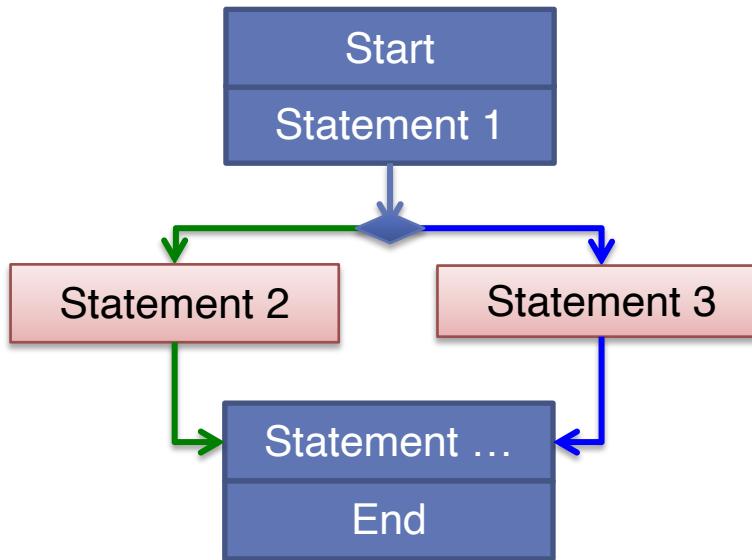
You want this



# Skip



# Choosing One of Two Paths



# Review

1. Control Flow gives the programmer the ability to write more sophisticated and elegant programs
2. The 'if' statement allows for the execution of different branches in the program



# Resource

- If Statement:
  - [https://golang.org/ref/spec#If\\_statements](https://golang.org/ref/spec#If_statements)

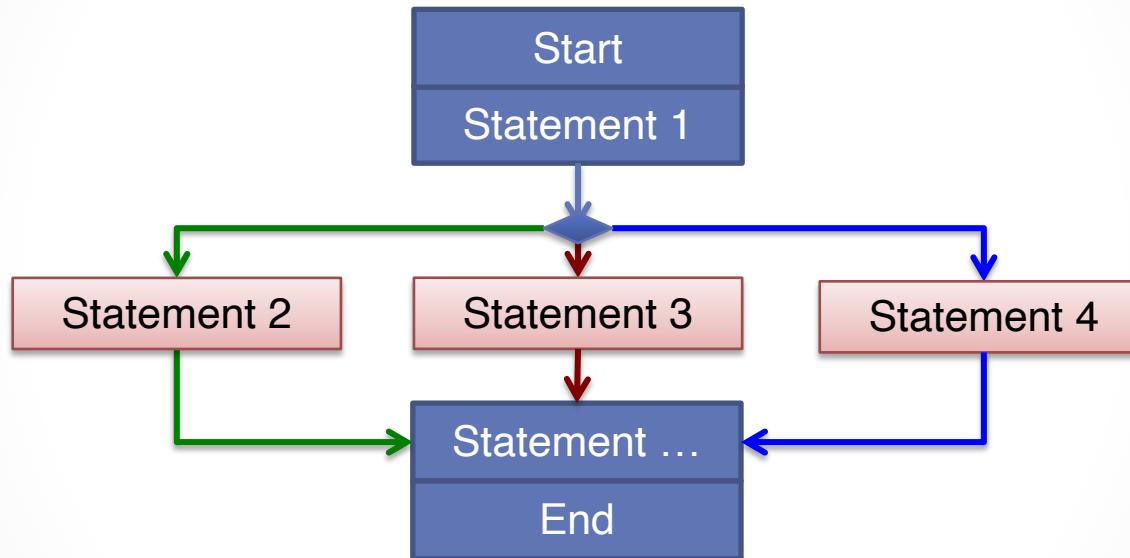


# The ‘switch’ Statement

Section 2 – Lecture 6



# Choosing One Of Many Paths



# Review

1. A 'switch' statement can be used in place of a multi-condition 'if' statement
2. The 'switch' statement provides '***fall through***', but it must be used with care



# Resource

- Switch Statement:
  - [https://golang.org/ref/spec#Switch\\_statements](https://golang.org/ref/spec#Switch_statements)



# The ‘for’ Statement

Section 2 – Lecture 7



# Review

- The ‘for’ Loop Statement
  - Mechanism to repeat statements while a condition is **true**
  - Be careful of the **infinite loop**
  - Skipping iteration(s) within a for loop with ‘**continue**’
  - Ending a for loop early with ‘**break**’



# Resource

- For Statement:
  - [https://golang.org/ref/spec#For\\_statements](https://golang.org/ref/spec#For_statements)
- Modulo Operator (%):
  - <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic>
- 'fmt' Package
  - <https://golang.org/pkg/fmt/>



# Functions - Part 1

Section 2 – Lecture 8



# Definition

- What is a ‘function’?
  - Verrol’s definition:
    - A ‘function’ is a **name** given to a **collection/set** of **statements**
- What are the benefits of functions?
  - **Reuse**
  - **Organization**
  - **Abstraction**



# Review

- Named Function Syntax

```
func name([input]) [(output)]{
 body
}
 o Input
 • Optional comma separated list of variable names
 o Output
 • Optional comma separated list of return variable(s) or type(s)
 o Body
 • Zero or more statements
```



# Resource

- Effective Go:
  - [https://golang.org/doc/effective\\_go.html#functions](https://golang.org/doc/effective_go.html#functions)



# Functions - Part 2

Section 2 – Lecture 9



# Functions with Inputs

- Some examples of function argument list:
  1. No parameter  
`func foo()`
  2. A single parameter  
`func foo(a int)`
  3. One or more parameters of the same type  
`func foo(a, b, c float32)`
  4. One or more different parameters  
`func foo(a int, b string, c float32)`
  5. **Optional variadic** parameter  
`func foo(a int, b string, ...c float32)`



# Functions with Outputs

- Some examples of function return list:
  1. No return  
*func* foo(...)
  2. A single *unnamed* return  
*func* foo(...) int
  3. A single *named* return  
*func* foo(...) (a int)
  4. One or more parameters of the same type  
*func* foo(...) (a, b, c float32)
  5. One or more different parameters  
*func* foo(...) (a int, b string, c float32)



# Example

```
17
18 func foo() {
19 fmt.Println("I am the function 'foo()'!")
20 }
21
22 func myPrintString(value string) {
23 fmt.Printf("-- value: %v ---\n", value)
24 }
25
26 func myAdd(a, b int) int {
27 return (a + b)
28 }
29
30 func swap(a, b int) (int, int) {
31 return b, a
32 }
33
34 func printArgs(args ...string) {
35 fmt.Println("Number of args to function 'printArgs(...)' are:", len(args))
36 }
37
```



# Example

```
17
18 func foo() {
19 fmt.Println("I am the function 'foo()'!")
20 }
21
22 func myPrintString(value string) {
23 fmt.Printf("-- value: %v ---\n", value)
24 }
25
26 func myAdd(a, b int) int {
27 return (a + b)
28 }
29
30 func swap(a, b int) (int, int) {
31 return b, a
32 }
33
34 func printArgs(args ...string) {
35 fmt.Println("Number of args to function 'printArgs(...)' are:", len(args))
36 }
37
```

```
5 func main() {
6 foo()
7 myPrintString("Hello world!")
8 fmt.Println("myAdd of 1, 5 is", myAdd(1, 5))
9 var a, b = swap(5, 1)
10 fmt.Printf("Swap of 5, 1 is %v, %v\n", a, b)
11 printArgs()
12 printArgs("Hello")
13 printArgs("Hello", "World")
14 printArgs("Hello", "World", "!")
15 }
```



# Example

```
17
18 func foo() {
19 fmt.Println("I am the function 'foo()'!")
20 }
21
22 func myPrintString(value string) {
23 fmt.Printf("-- value: %v ---\n", value)
24 }
25
26 func myAdd(a, b int) int {
27 return (a + b)
28 }
29
30 func swap(a, b int) (int, int) {
31 return b, a
32 }
33
34 func printArgs(args ...string) {
35 fmt.Println("Number of args to function 'printArgs(...)' are: ", len(args))
36 }
```

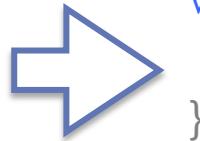
```
5 func main() {
6 foo()
7 myPrintString("Hello world!")
8 fmt.Println("myAdd of 1, 5 is", myAdd(1, 5))
9 var a, b = swap(5, 1)
10 fmt.Printf("Swap of 5, 1 is %v, %v\n", a, b)
11 printArgs()
12 printArgs("Hello")
13 printArgs("Hello", "World")
14 printArgs("Hello", "World", "!")
15 }
```

```
~$ go run main.go
I am the function 'foo()'
-- value: Hello world! ---
myAdd of 1, 5 is 6
Swap of 5, 1 is 1, 5
Number of args to function 'printArgs(...)' are: 0
Number of args to function 'printArgs(...)' are: 1
Number of args to function 'printArgs(...)' are: 2
Number of args to function 'printArgs(...)' are: 3
```



# Anonymous Function Syntax

```
func name([input]) [(output)]{
 body
}
```



```
var name = func([input]) [(output)]{
 body
}
```

# Review

- In Go, a function is introduced with the ‘**func**’ keyword
- Functions can optionally accept **0** or more parameters, inputs
- Functions can optionally return **0** or more values
- Functions create their own scope
  - *variables inside of functions are different from variables outside of the function*



# Review

- Functions are first class citizens in Go
- Support for ‘anonymous functions’
- Functions supports closure
- Functions can return functions



# Resource

- Effective Go:
  - [https://golang.org/doc/effective\\_go.html#functions](https://golang.org/doc/effective_go.html#functions)



# Type Declarations

Section 2 – Lecture 10



# Resource

- Lang Specification
  - <https://golang.org/ref/spec#Types>
  - [https://golang.org/ref/spec#Type\\_declarations](https://golang.org/ref/spec#Type_declarations)



# Packages Part 1

Section 2 – Lecture 11



# Go Program

main

main()

Package A

Package B

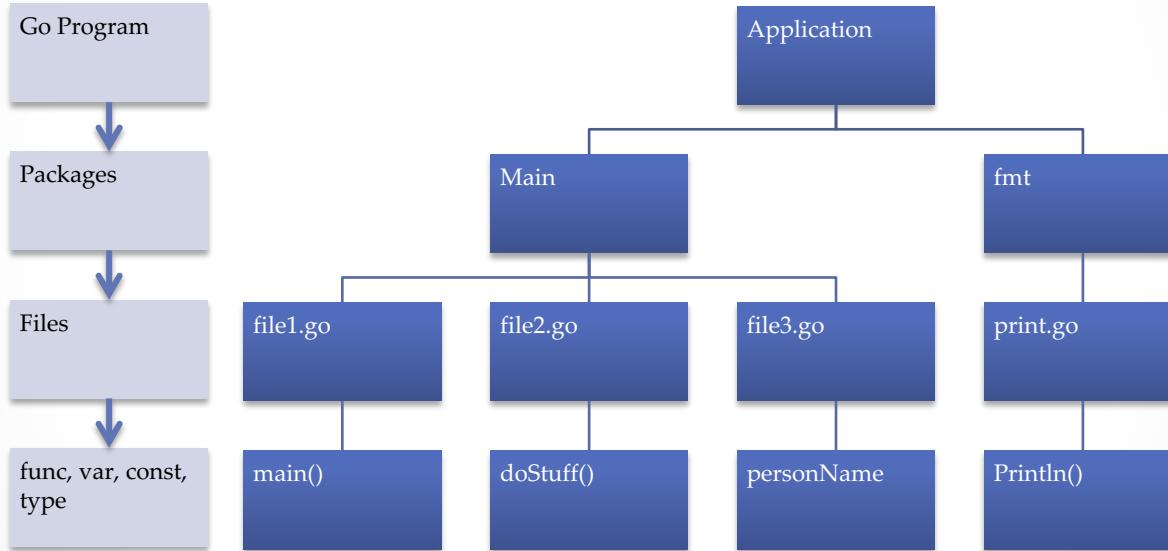


# Definition: Package

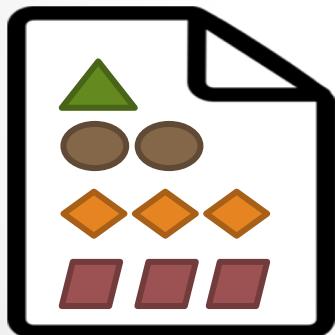
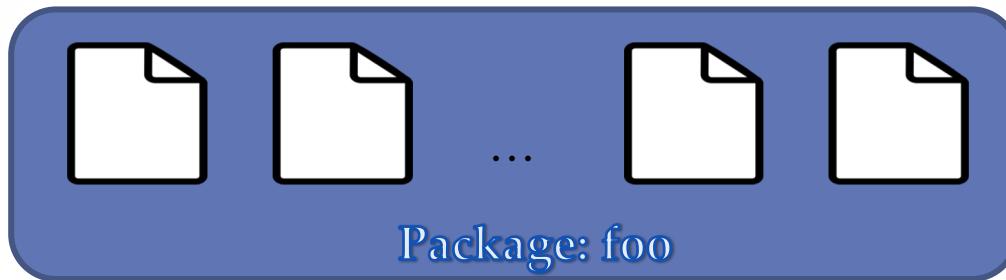
- What is a ‘package’?
  - Go programs are constructed by linking together **packages**.
  - A package in turn is constructed from one or more **source files**.
  - A **source file** may contribute elements to the package such as:
    - **constants**
    - **variables**
    - **functions**
    - **types**
  - Elements in a package are either:
    - **private**
    - **public** (exported)



# Illustration Only



# Package Visualization



- C = Constant
- V = Variable
- F = Function
- T = Type

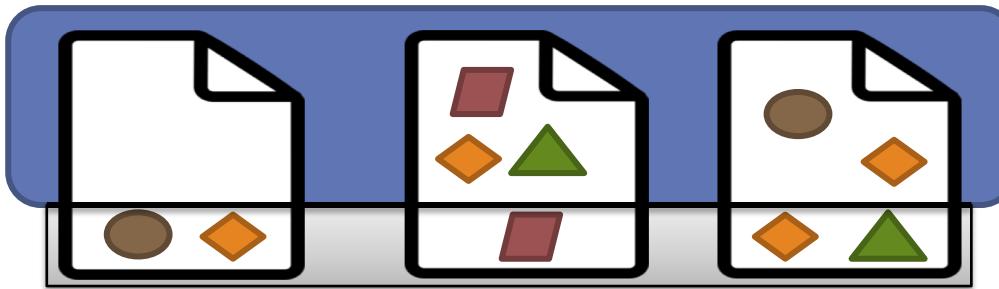


# Package Visibility

Package: foo

private

Public



- C = Constant
- V = Variable
- F = Function
- T = Type



# Resource

- Lang Specification
  - <https://golang.org/ref/spec#Blocks>
  - <https://golang.org/ref/spec#Packages>



# Packages Part 2

Section 2 – Lecture 12



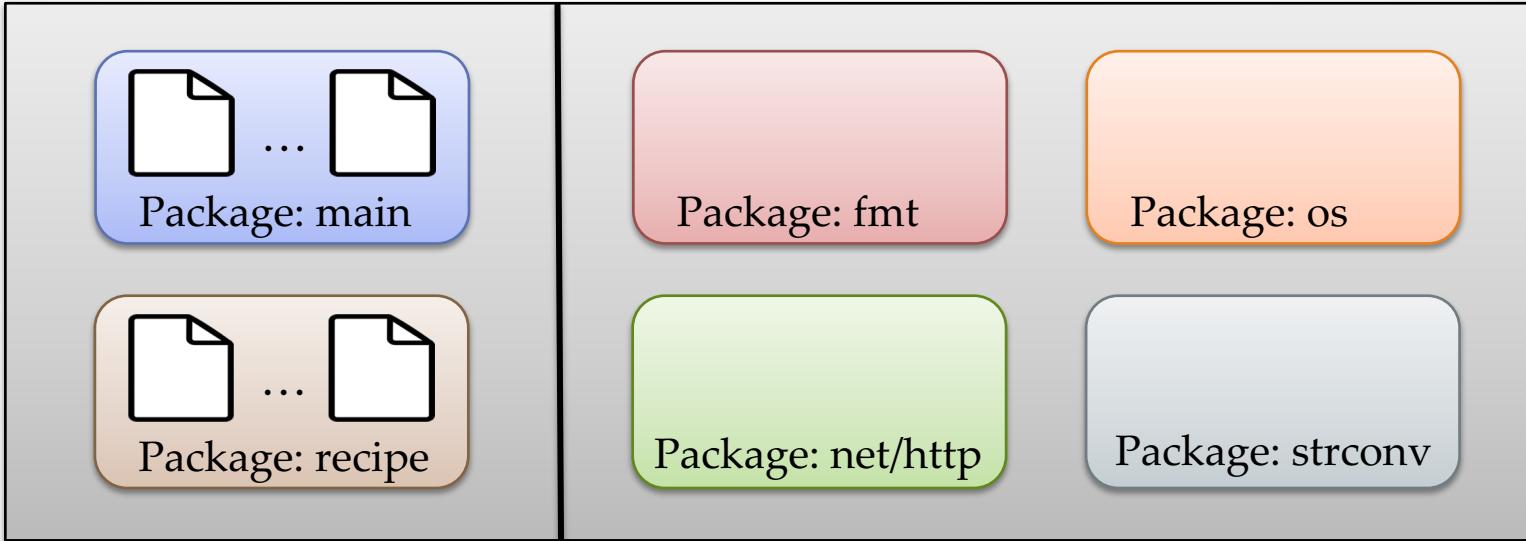
# Importing Packages

- Visibility
  - **Private** members are **package scope** regardless of file
  - **Public/Exported** members are only accessible in the file which imports the package
  - Types of imports:

| Import Method        | Import Declaration  | Local Name of Sin |
|----------------------|---------------------|-------------------|
| Default package name | import "lib/math"   | math.Sin          |
| Assigned name        | import m "lib/math" | m.Sin             |
| Full package         | import . "lib/math" | Sin               |
| Side effect          | import _ "lib/math" | Not Applicable    |



# Executable Visualized



Executable: **recipe-manager**



# What are the benefits of Packages?

- **Abstraction**
- **Reuse**
- **Organization**



# Resource

- Lang Specification
  - <https://golang.org/ref/spec#Blocks>
  - <https://golang.org/ref/spec#Packages>
  - [https://golang.org/ref/spec#Program\\_initialization\\_and\\_execution](https://golang.org/ref/spec#Program_initialization_and_execution)



# Section 2

# Wrap Up

Section 2 – Lecture 13



# Miscellaneous

1. Constants
  - o **iota** identifier
2. Variables
  - o Blank variable identifier
  - o Block scope, '{' and '}'
3. Blank identifier in **if** and **for** statements
4. Functions
  - o 'error' type and 'errors' package
  - o **defer()**
  - o **closure**
5. Packages
  - o Fetching and building packages



# Section 3: Arrays & Slices

- Declaring and Using Arrays
- Arrays and Functions
- Declaring and Using Slices
- Slices and Functions
- Manipulating Slice size at runtime
- Section 3 Wrap Up



# Declaring and Using Arrays

Section 3 – Lecture 1

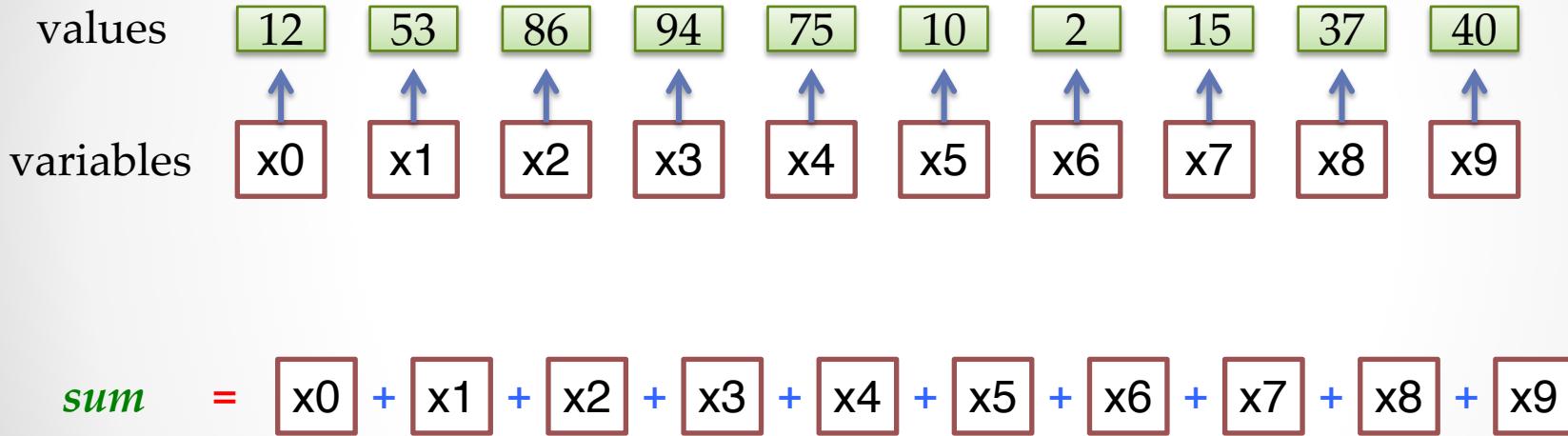


# Topics

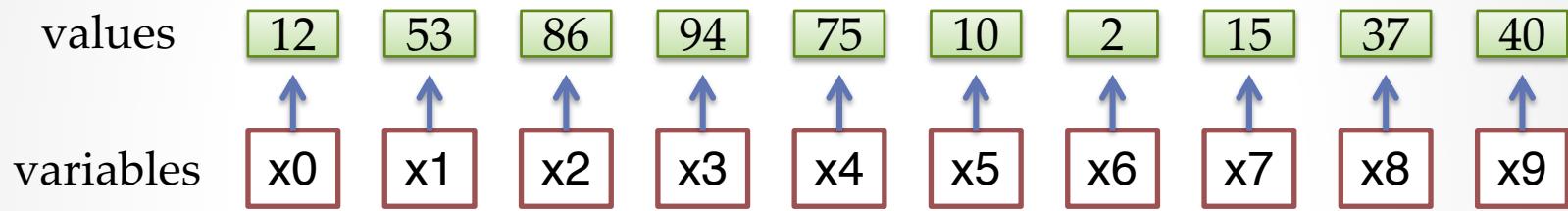
- What is an Array?
- How to declare an array
- How to store and retrieve values from an array
- How calculate the length of an array
- Array iteration



# Life Without Arrays



# Array Illustrated



Array: *nums*

|    |    |    |    |    |    |   |    |    |    |
|----|----|----|----|----|----|---|----|----|----|
| 12 | 53 | 86 | 94 | 75 | 10 | 2 | 15 | 37 | 40 |
| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  |

Array length:  $\text{len}(\text{nums}) = 10$



# Array Analogy



# Review : What is an Array?

- Attributes of an Array
  - An **array** in Go lang is a built in datatype with the following features:
    - Stores **several values** of the **same type**
    - Provides **compile-time bounds checking** on integer literal index
    - Provides **runtime-time bounds checking** on integer variable index
    - Provides **non-negative integer index**
    - **Uses 0-base indexing**



# Review : Declaration & Usage

- Declaring an array if very simple:
  - `var name [N]type`
  - `var name = [N]type{value, value1, value2, ...}`
    - Eg:
      - `var testResults [10]int`
      - `var testResults = [10]int{90, 75, 83, 69, 45, 99, 55, 28, 71, 93}`
- Assigning values to an array's element
  - `arrayName[index] = value`
    - Eg:
      - `testResults[0] = 12`
      - `testResults[9] = 103`
- Bound checking prevents illegal or dangerous access:
  - `arrayName[invalid-index] = value`
    - Eg:
      - `testResults[-1] = 12`
        - Negative index
      - `testResults[10] = 103`
        - Index out of bounds, greater than the number of defined elements



# Resource

- Array Types - Lang Specification
  - [https://golang.org/ref/spec#Array\\_types](https://golang.org/ref/spec#Array_types)
- For Statements with ‘Range clause’
  - [https://golang.org/ref/spec#For\\_statements](https://golang.org/ref/spec#For_statements)



# Arrays and Functions

Section 3 – Lecture 2



# Topics

- Passing Arrays as Function Parameter
- Arrays as Function Return Types
- Understanding '*pass-by-value*'



# Sorting Illustrated

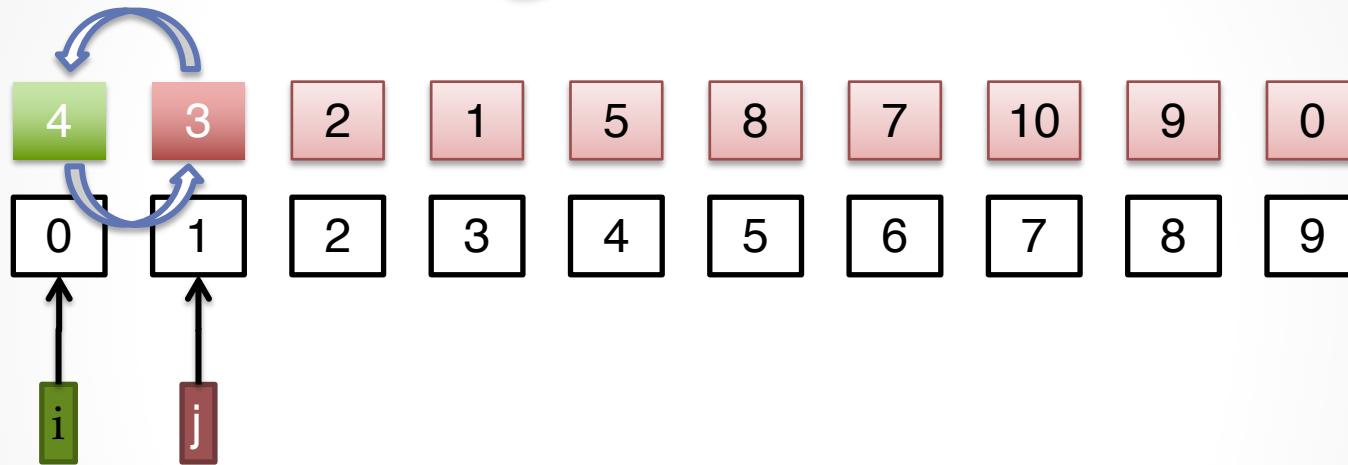


# Sorting Illustrated

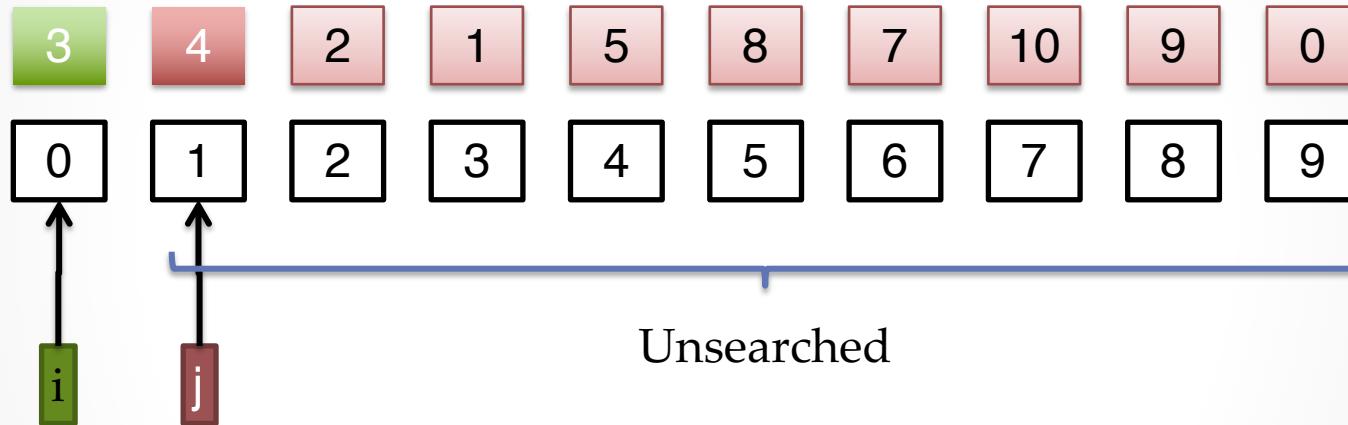
|   |   |   |   |   |   |   |    |   |   |
|---|---|---|---|---|---|---|----|---|---|
| 4 | 3 | 2 | 1 | 5 | 8 | 7 | 10 | 9 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8 | 9 |



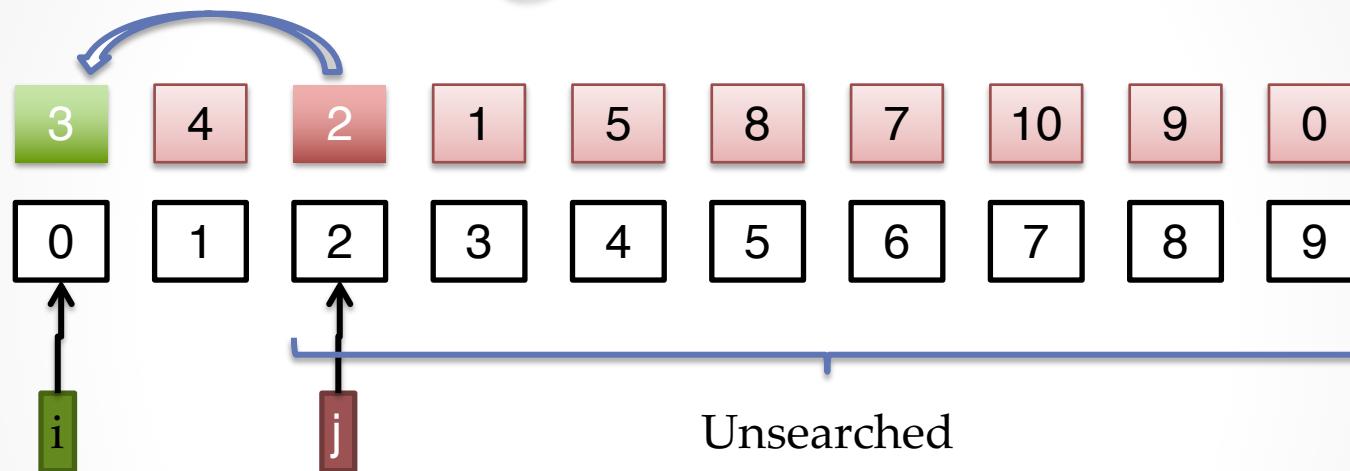
# Sorting Illustrated



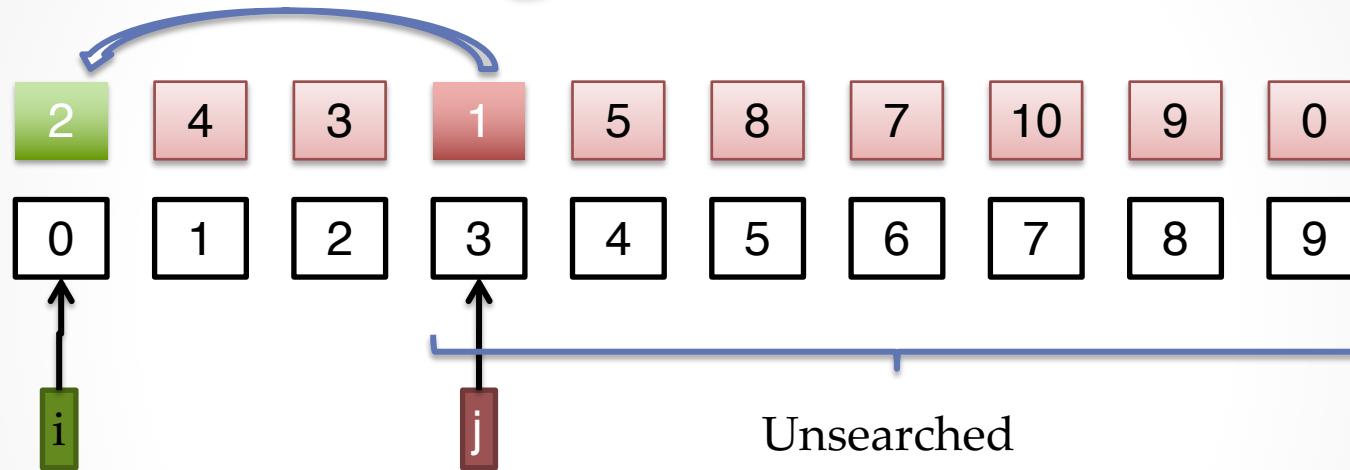
# Sorting Illustrated



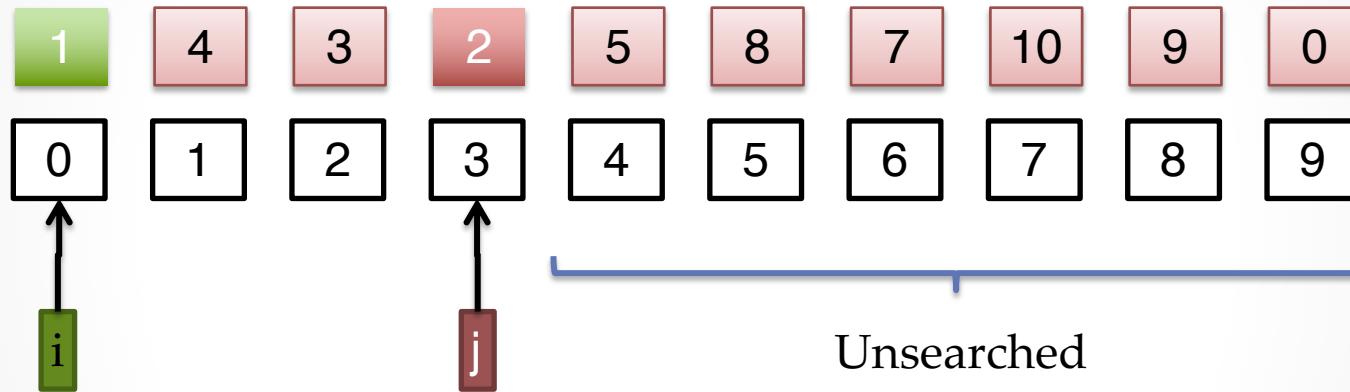
# Sorting Illustrated



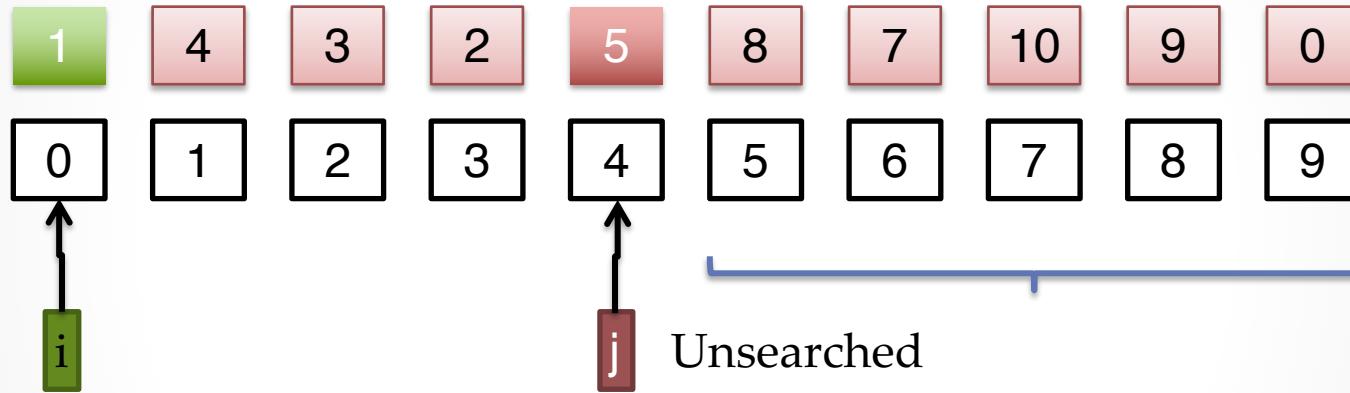
# Sorting Illustrated



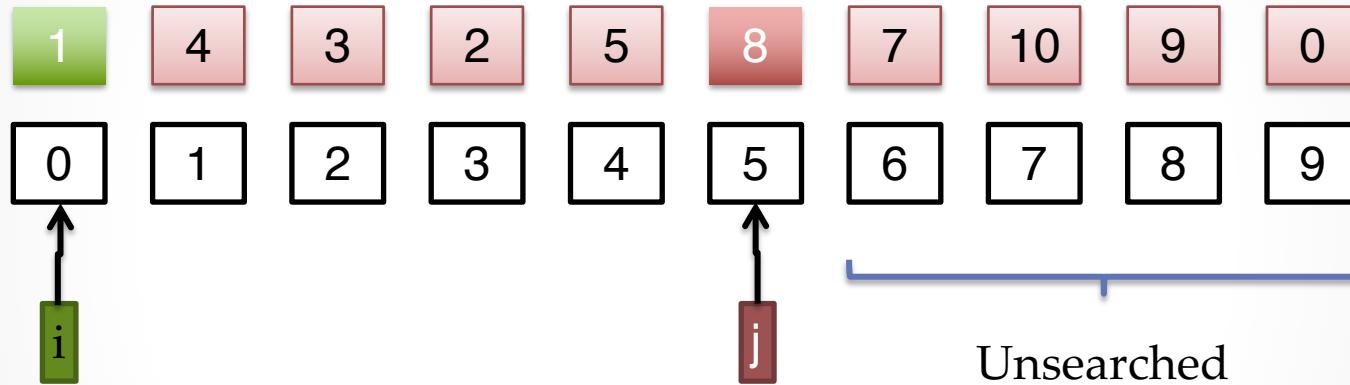
# Sorting Illustrated



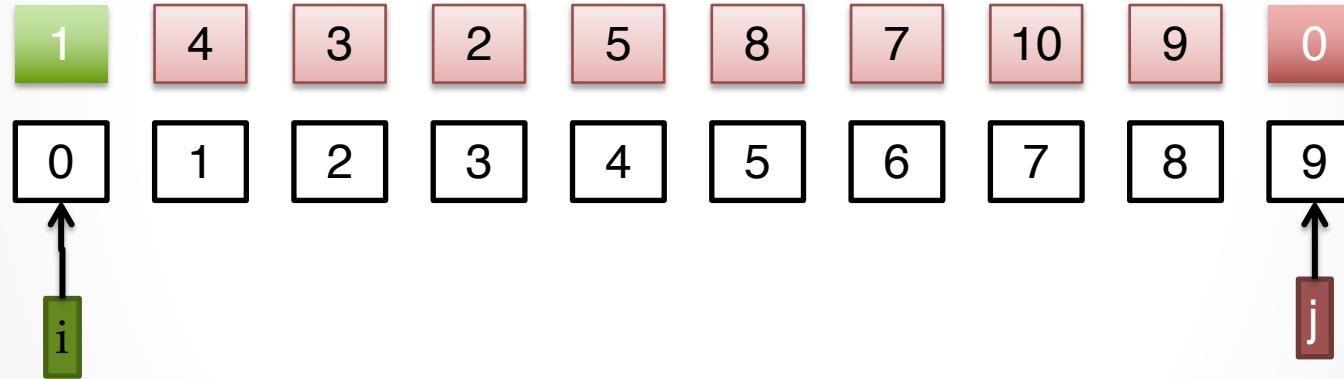
# Sorting Illustrated



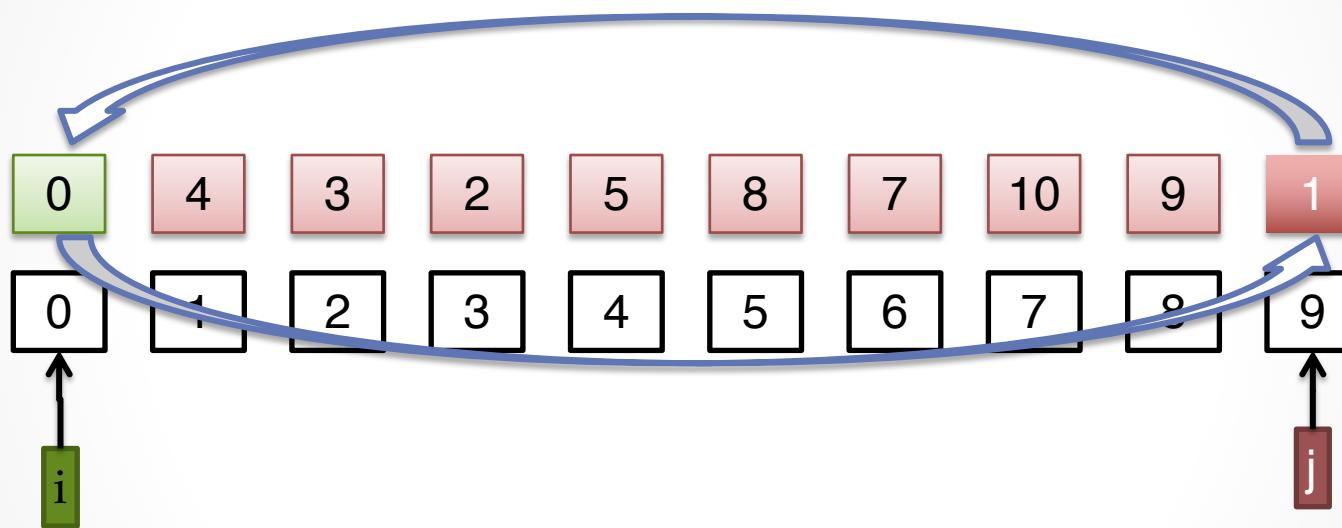
# Sorting Illustrated



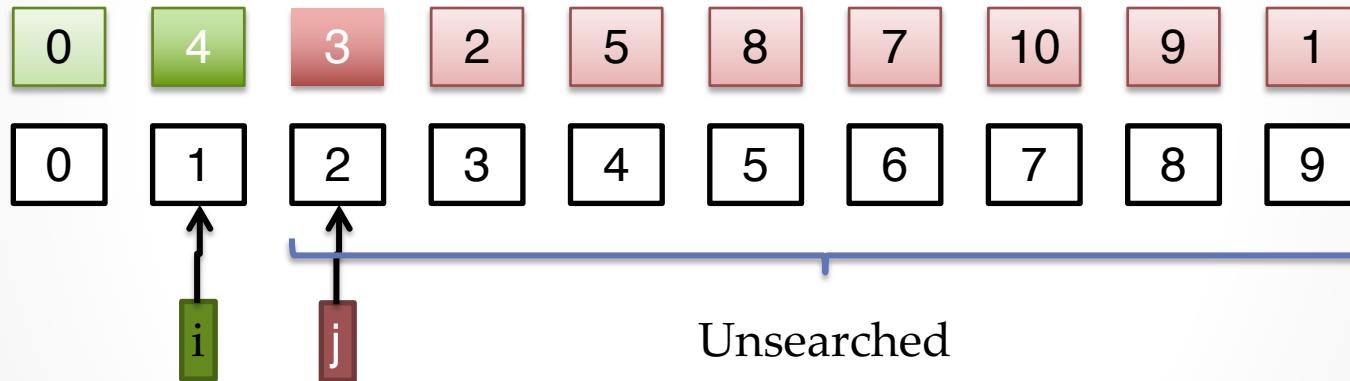
# Sorting Illustrated



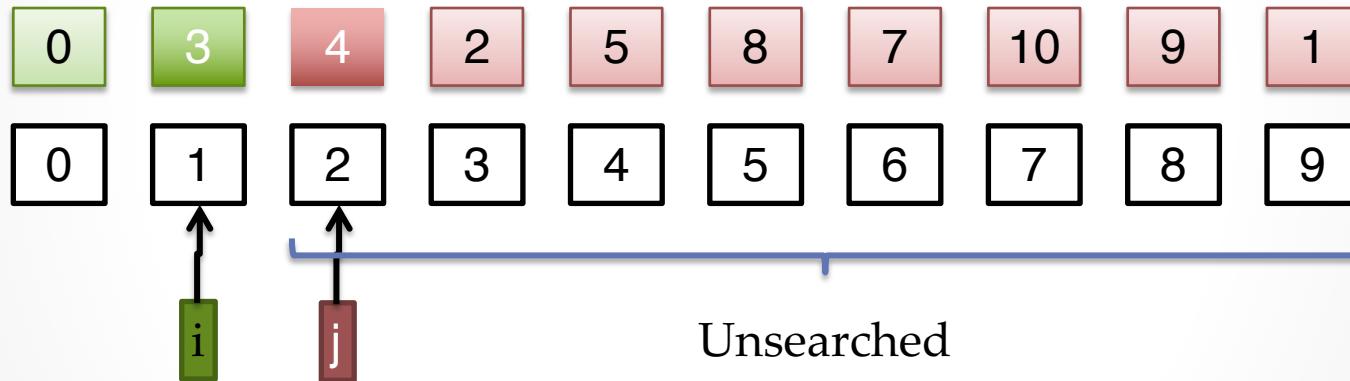
# Sorting Illustrated



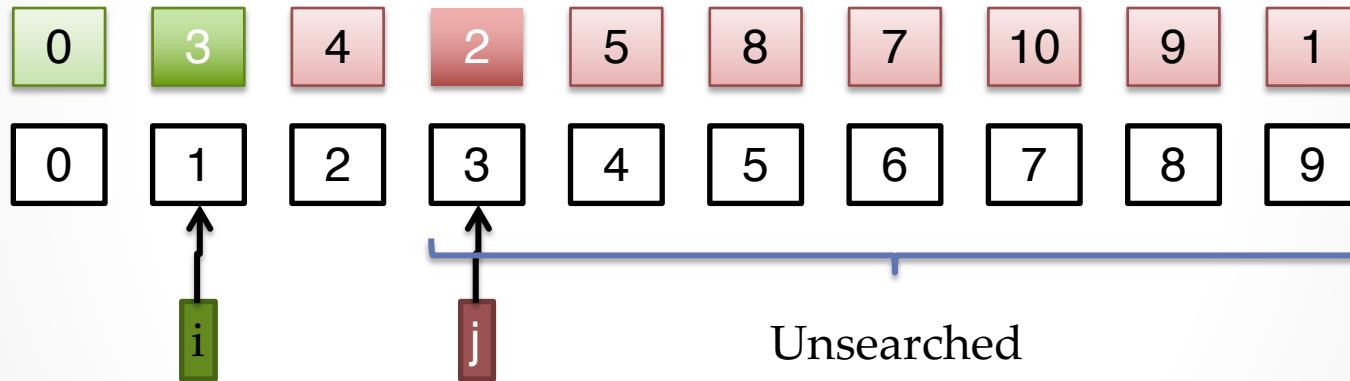
# Sorting Illustrated



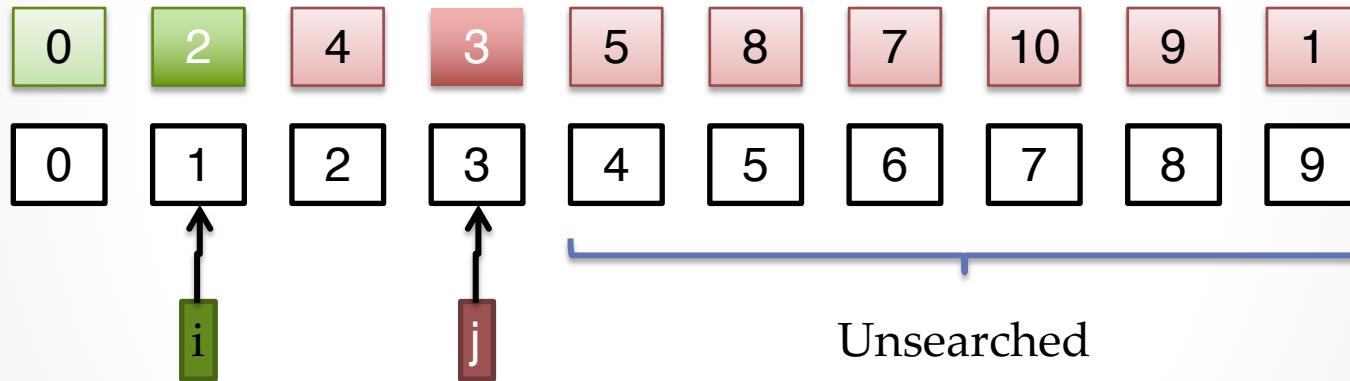
# Sorting Illustrated



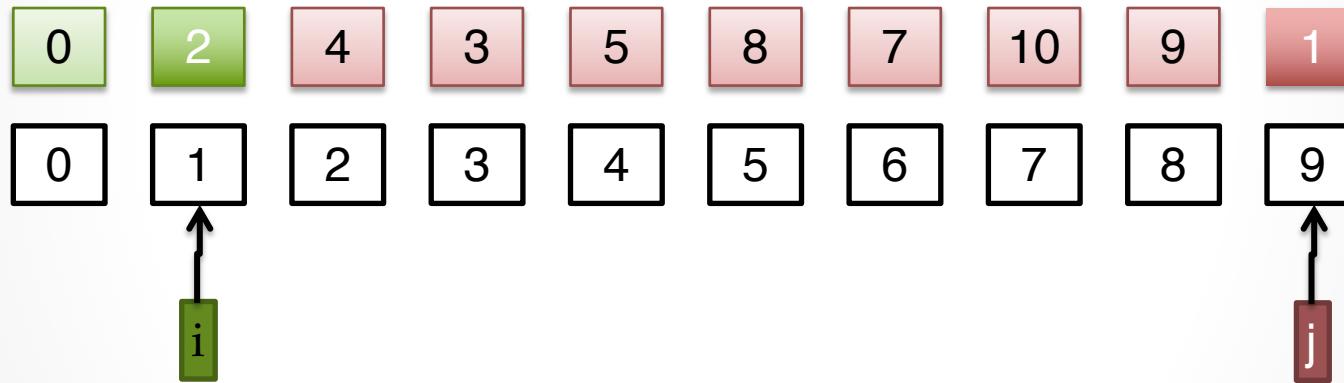
# Sorting Illustrated



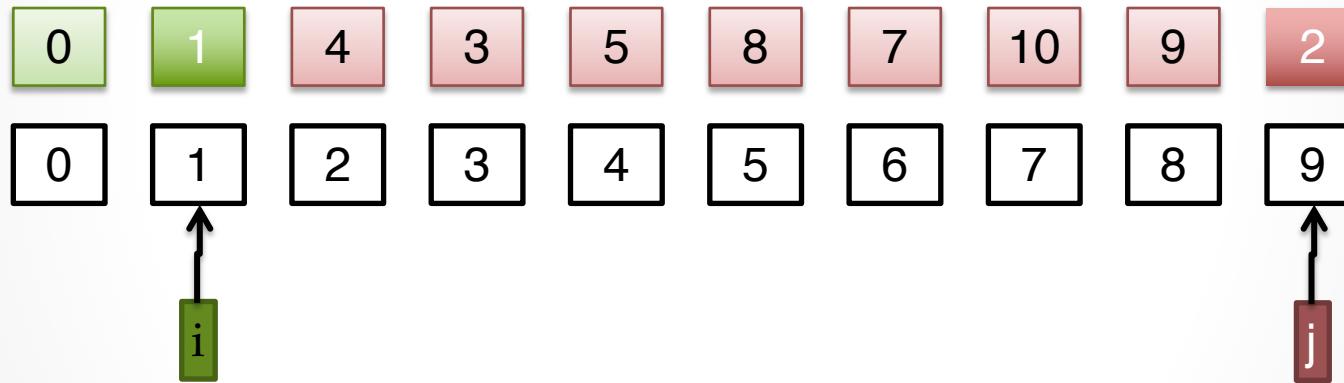
# Sorting Illustrated



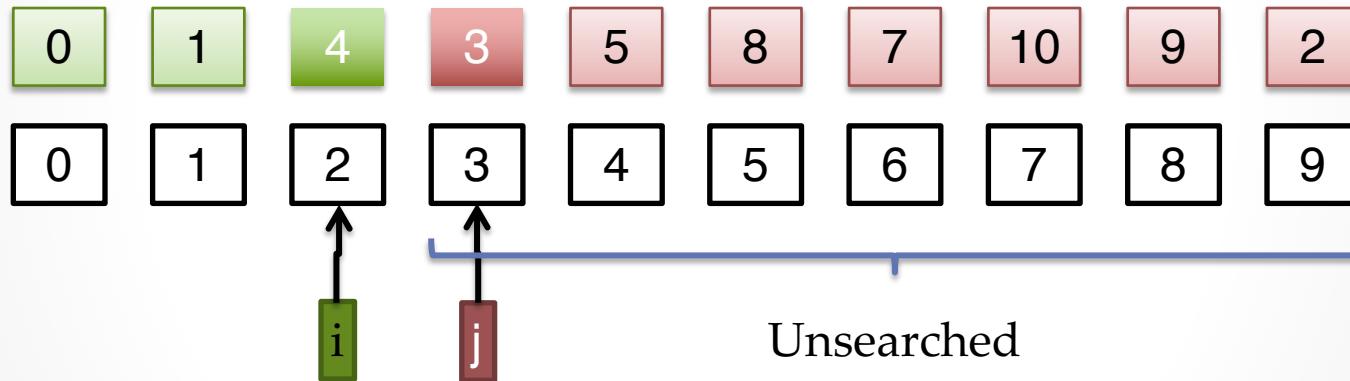
# Sorting Illustrated



# Sorting Illustrated



# Sorting Illustrated



# Review

- An Array in Go Lang is a **value**
  - Therefore, it gets **copied** when **assigned**
    - Hence, **pass-by-value**
- TIP:
  - Use either a *slice\** or *pointer+ to array* for Function parameter(s) or return value(s)
    - Prefer Slices instead of pointer to array\*

\* We cover slices in the next lecture

+ We cover pointers in Section 8



# Declaring and Using Slices

Section 3 – Lecture 3



# Topics

- What is an Slice?
- How to declare a Slice
- How to store and retrieve values from a Slice
- How calculate the length of a Slice
- Iterating over the values of a Slice



# What is a slice?

- Verrol's definition:
  - A ***slice*** is a subset or window into an ***array***



# Subset of an array

nums =>

|    |    |    |    |
|----|----|----|----|
| 12 | 53 | 86 | 94 |
| 0  | 1  | 2  | 3  |



# Subset of an array

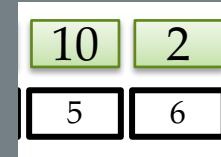
nums =>

|    |    |    |    |    |   |
|----|----|----|----|----|---|
| 53 | 86 | 94 | 75 | 10 | 2 |
| 1  | 2  | 3  | 4  | 5  | 6 |



# Subset of an array

nums =>



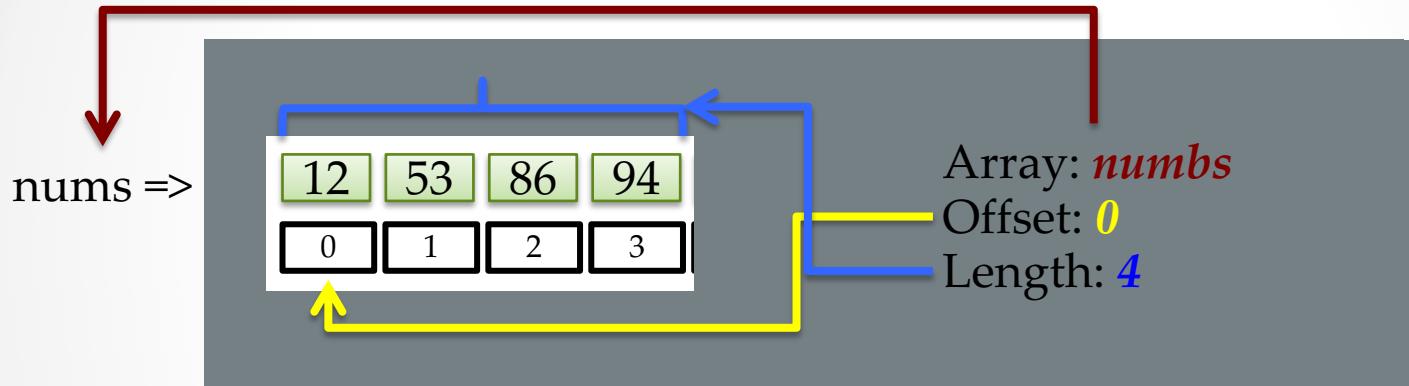
# Subset of an array

nums =>

|    |    |    |    |    |    |   |    |    |    |
|----|----|----|----|----|----|---|----|----|----|
| 12 | 53 | 86 | 94 | 75 | 10 | 2 | 15 | 37 | 40 |
| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  |



# Slice



# Slice

nums =>

|    |    |    |    |    |   |
|----|----|----|----|----|---|
| 53 | 86 | 94 | 75 | 10 | 2 |
| 1  | 2  | 3  | 4  | 5  | 6 |

Array: *nums*

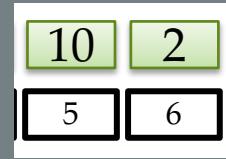
Offset: 1

Length: 6



# Slice

nums =>



Array: *nums*  
Offset: 5  
Length: 2



# Slice

Array: *nums*   Offset: 0   Length: 10

nums =>

|    |    |    |    |    |    |   |    |    |    |
|----|----|----|----|----|----|---|----|----|----|
| 12 | 53 | 86 | 94 | 75 | 10 | 2 | 15 | 37 | 40 |
| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  |



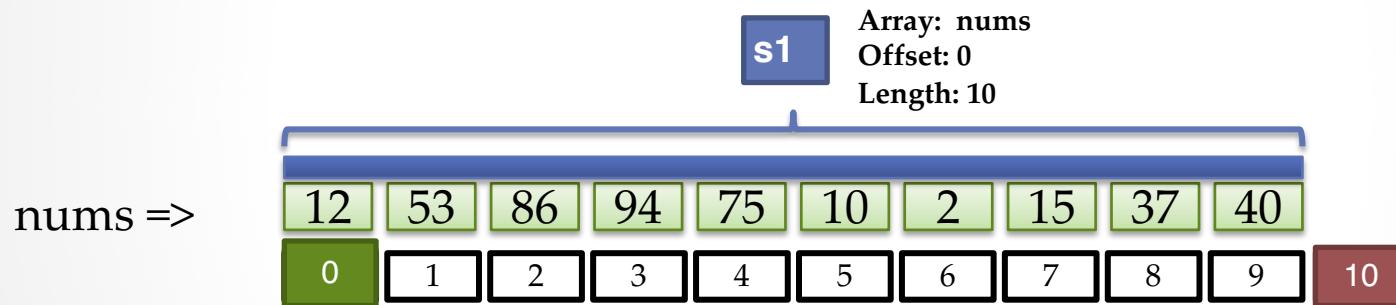
# Declaring slice variable

```
var s []int
```



# Slice Illustrated

```
nums := [10]int{12, 53, 86, 94, 75, 10, 2, 15, 37, 40}
```

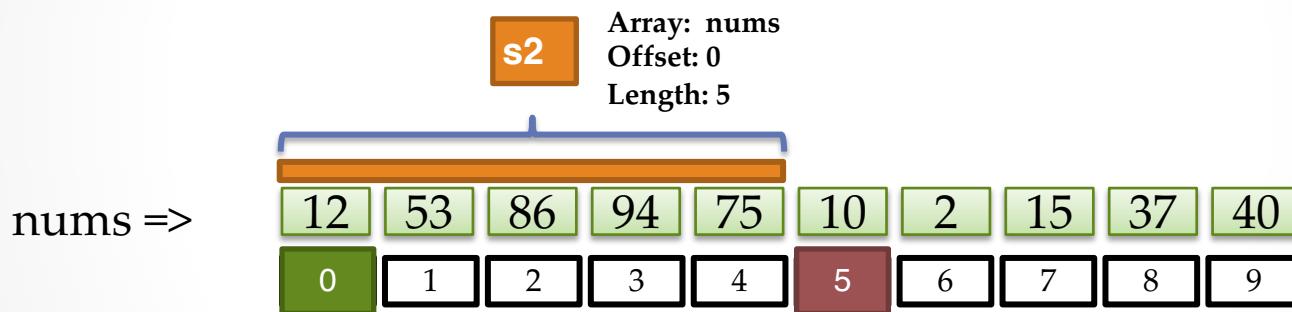


```
s1 := nums[:] // nums[0:len(nums)]
```



# Slice Illustrated

```
nums := [10]int{12, 53, 86, 94, 75, 10, 2, 15, 37, 40}
```

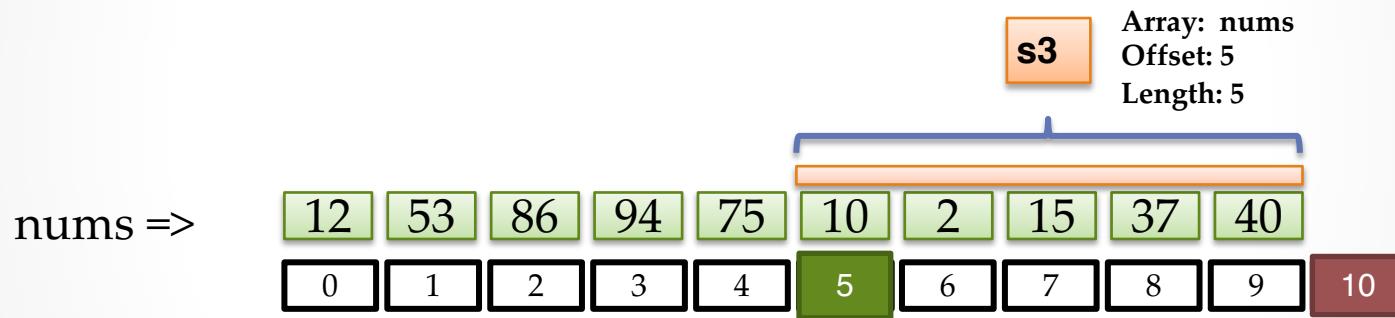


```
s2 := nums[:5] // nums[0:5]
```



# Slice Illustrated

```
nums := [10]int{12, 53, 86, 94, 75, 10, 2, 15, 37, 40}
```

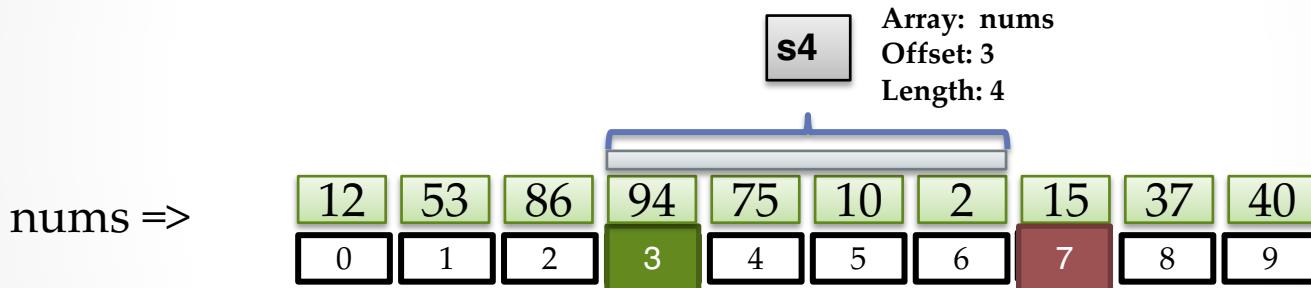


```
s3 := nums[5:] // nums[5:len(nums)]
```



# Slice Illustrated

```
nums := [10]int{12, 53, 86, 94, 75, 10, 2, 15, 37, 40}
```



```
s4 := nums[3:7]
```



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Slice\\_types](https://golang.org/ref/spec#Slice_types)
  - [https://golang.org/ref/spec#Slice\\_expressions](https://golang.org/ref/spec#Slice_expressions)
- Go Slices: Usage and Internals
  - <https://blog.golang.org/go-slices-usage-and-internals>



# Slices and Functions

Section 3 – Lecture 4



# Topics

- Slices and Functions
  1. Passing Slices as Function Parameter
  2. Slices as Function Return Types
  3. Comparing Arrays and Slices in the context of Functions
  4. Revisit variadic functions



# Review

- A Slice is an efficient abstraction built on an Array
  - Slices gives you **all** the benefits of an Array, with **none** of the drawbacks
  - Arrays and Slices can be passed as varadic parameters using the expand operator ‘...’



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Slice\\_types](https://golang.org/ref/spec#Slice_types)
  - [https://golang.org/ref/spec#Slice\\_expressions](https://golang.org/ref/spec#Slice_expressions)
- Go Slices: Usage and Internals
  - <https://blog.golang.org/go-slices-usage-and-internals>



# CECS Slices

**CECS (Create, Expand, Copy, and Shrink)**

Section 3 – Lecture 5

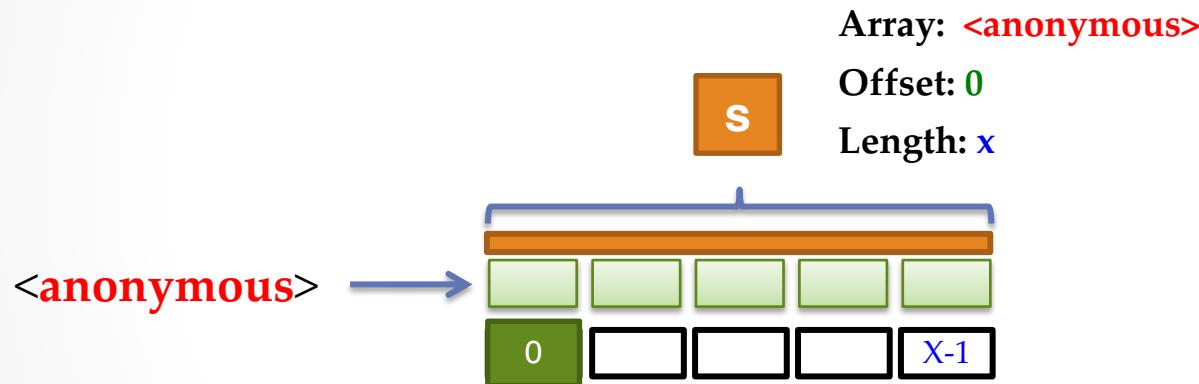


# Topics

- Creating Slices at runtime
- Expand/Grow Slices
- Copy
- Shrink Slices

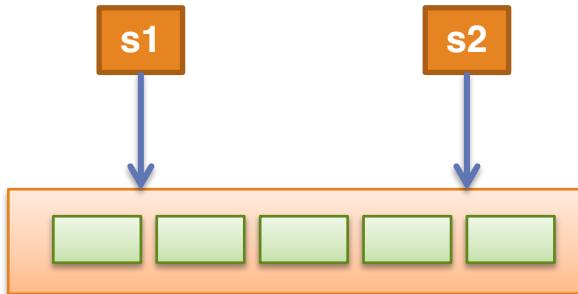


# Creating Slices At Runtime



# Slice Shallow Copy

```
var s1 = ar[:]
var s2 = s1
```



# Slice Deep Copy

```
var s1 = ar[:]
```



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Slice\\_types](https://golang.org/ref/spec#Slice_types)
  - [https://golang.org/ref/spec#Slice\\_expressions](https://golang.org/ref/spec#Slice_expressions)
  - [https://golang.org/ref/spec#Making\\_slices\\_maps\\_and\\_channels](https://golang.org/ref/spec#Making_slices_maps_and_channels)
  - [https://golang.org/ref/spec#Appending\\_and\\_copying\\_slices](https://golang.org/ref/spec#Appending_and_copying_slices)
- Go Slices: Usage and Internals
  - <https://blog.golang.org/go-slices-usage-and-internals>



# Section 3 Review

Section 3 – Lecture 6



# Topics

- Programs Arguments
  - ‘os.Args’ from Package ‘os’
- Arrays/slices of arrays/slices
  - Multi-dimensions arrays/slices
- Slice capacity
  - Re-slicing a slice
- String slices



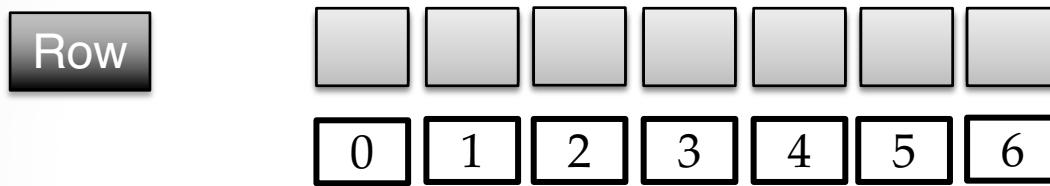
# Program Arguments

```
>>> go run main.go
```

Program Name      Program Arguments



# Multi-dimension Array/Slice

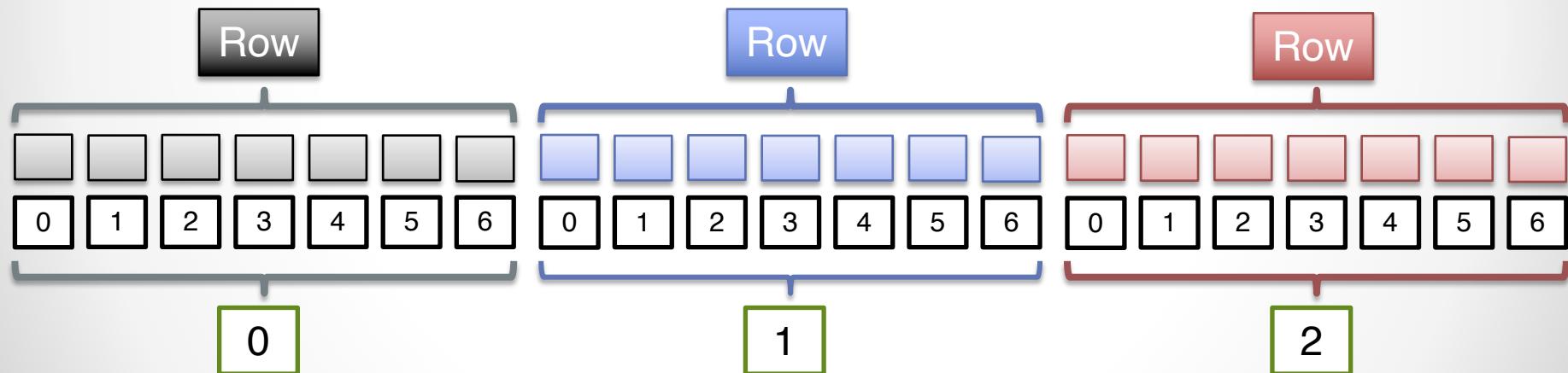


```
type Row [7]int
```



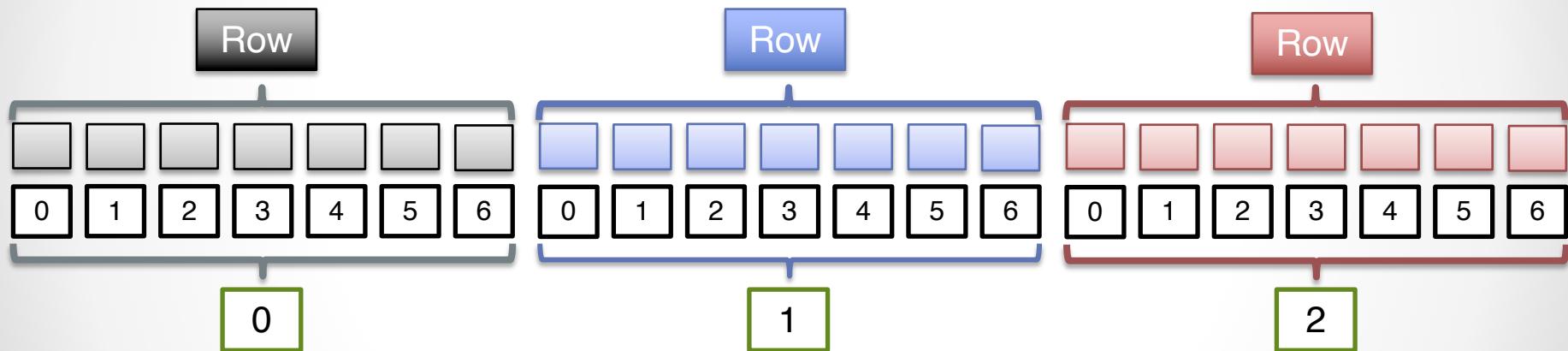
# Multi-dimension Array/Slice

```
var multipleRows [3]Row
```



# Multi-dimension Array/Slice

```
var multipleRows [3]Row
```

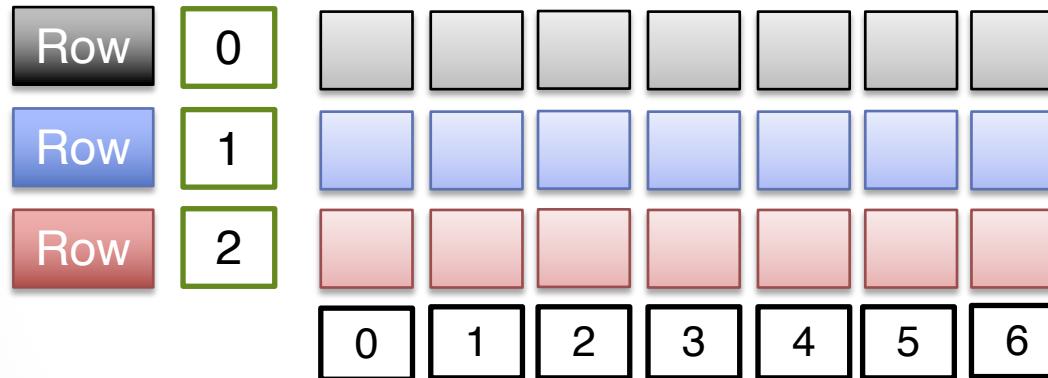


```
x := multipleRows[1][4]
```



# Multi-dimension Array/Slice

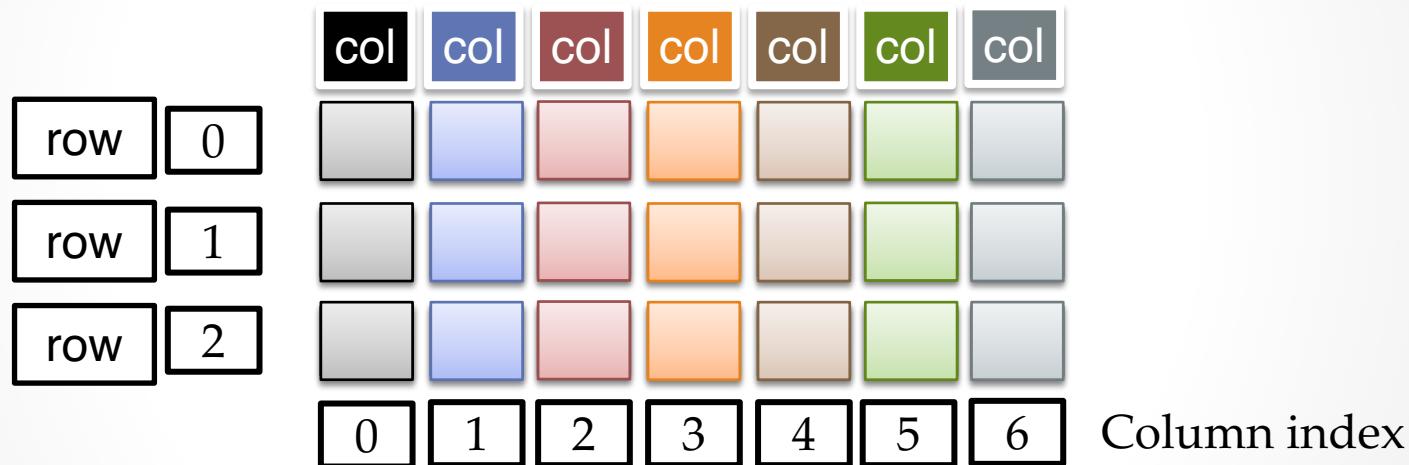
```
var multipleRows [3]Row
```



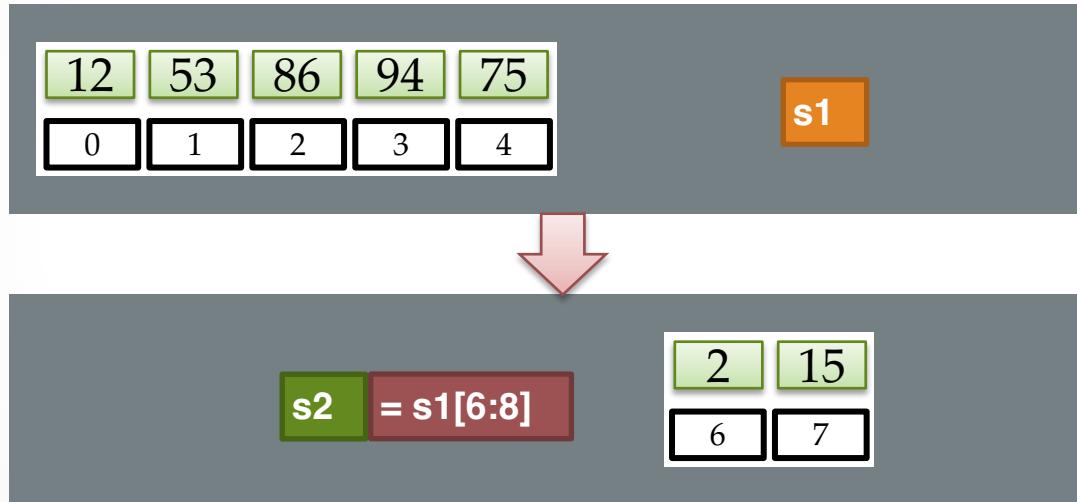
```
type Table [3]Row
```



# Multi-dimension Array/Slice



# Re-slice a Slice



# String Illustrated

“Hello”

H e l l o



# string as []byte

var s = "Hello, 世界"

|            |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|
| character  | H  | e  | l  | l  | o  | ,  |    | 世  | 界  |    |    |    |    |
| byte value | 48 | 65 | 6c | 6c | 6f | 2c | 20 | e4 | b8 | 96 | e7 | 95 | 8c |
| byte index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |



s[4] => 0x6F => 'o'



s[10] => 0xE7 => 'ç'



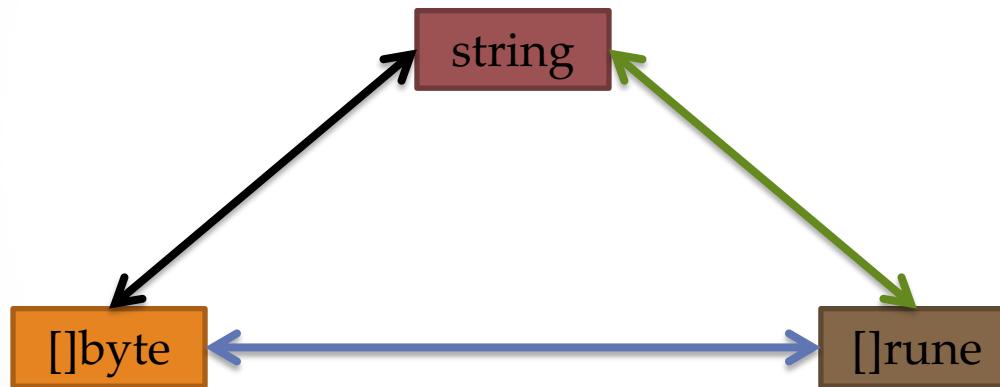
# Runes of a String

“Hello, 世界”

|            |    |    |    |    |    |    |    |      |    |      |    |    |    |
|------------|----|----|----|----|----|----|----|------|----|------|----|----|----|
| rune index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7    | 8  | 9    | 10 | 11 | 12 |
| rune value | 48 | 65 | 6c | 6c | 6f | 2c | 20 | 4e16 |    | 754c |    |    |    |
| character  | H  | e  | l  | l  | o  | ,  | 世  |      | 界  |      |    |    |    |
| byte value | 48 | 65 | 6c | 6c | 6f | 2c | 20 | e4   | b8 | 96   | e7 | 95 | 8c |
| byte index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7    | 8  | 9    | 10 | 11 | 12 |



# String – Triple Threat



# Resource

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
  - <http://bit.ly/2ibBNol>
- Strings, bytes, runes and characters in Go
  - <https://blog.golang.org/strings>
- Length and capacity
  - [https://golang.org/ref/spec#Length\\_and\\_capacity](https://golang.org/ref/spec#Length_and_capacity)



# Section 4 : map

- What is a map?
- How to declare a map
- How to create a map
- How to store and retrieve values using map
- How calculate the number of items in a map
- Map iteration
- Delete elements from a map



# maps

Section 4 – Lecture 1



# Array at a Glance

| Intent         | Code                  |
|----------------|-----------------------|
| Declare/Create | var a [I] <i>type</i> |
| Store          | a[i] = v              |
| Retrieve       | v = a[i]              |
| Iterate        | for i, v := range a   |
| Size           | len(a)                |

\* 'i' **must** be an integer value  $\geq 0$



# Slice at a Glance

| Intent   | Code                                                                              |
|----------|-----------------------------------------------------------------------------------|
| Declare  | <code>var s []<i>type</i></code>                                                  |
| Create   | <code>s = make([]<i>type</i>, i)</code> or <code>a[:]</code> or <code>s[:]</code> |
| Store    | <code>s[i] = v</code>                                                             |
| Retrieve | <code>v = s[i]</code>                                                             |
| Iterate  | <code>for i, v := range s</code>                                                  |
| Size     | <code>len(s)</code>                                                               |

\* 'i' **must** be an integer value  $\geq 0$



# What is a Map?

- Formal definition:
  - A **map** is an unordered group of elements of **one type**, called the **element type**, indexed by a set of **unique keys** of **another type**, called the **key type**.
    - The value of an uninitialized map is **nil**.

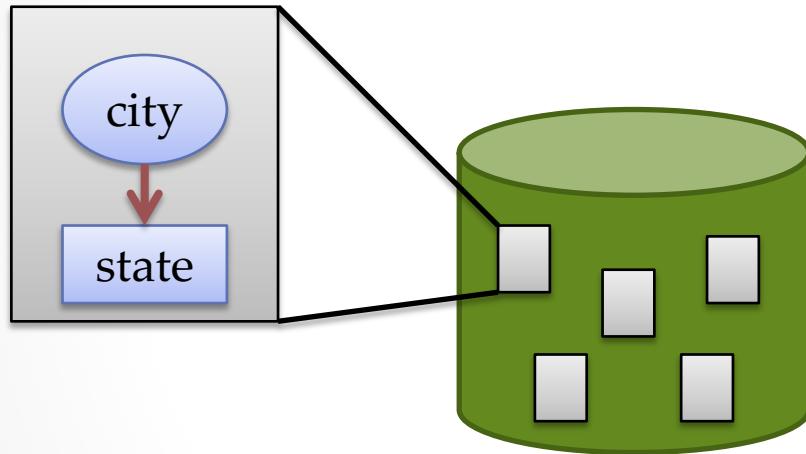


# Map Illustrated

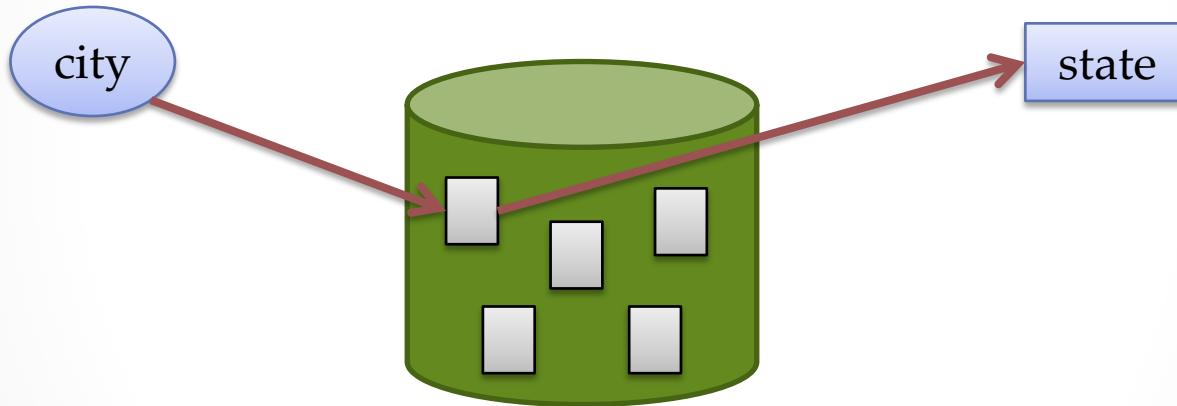
| City          | State          |
|---------------|----------------|
| Charlotte     | North Carolina |
| Brooklyn      | New York       |
| San Francisco | California     |
| Manhattan     | New York       |
| San Jose      | California     |



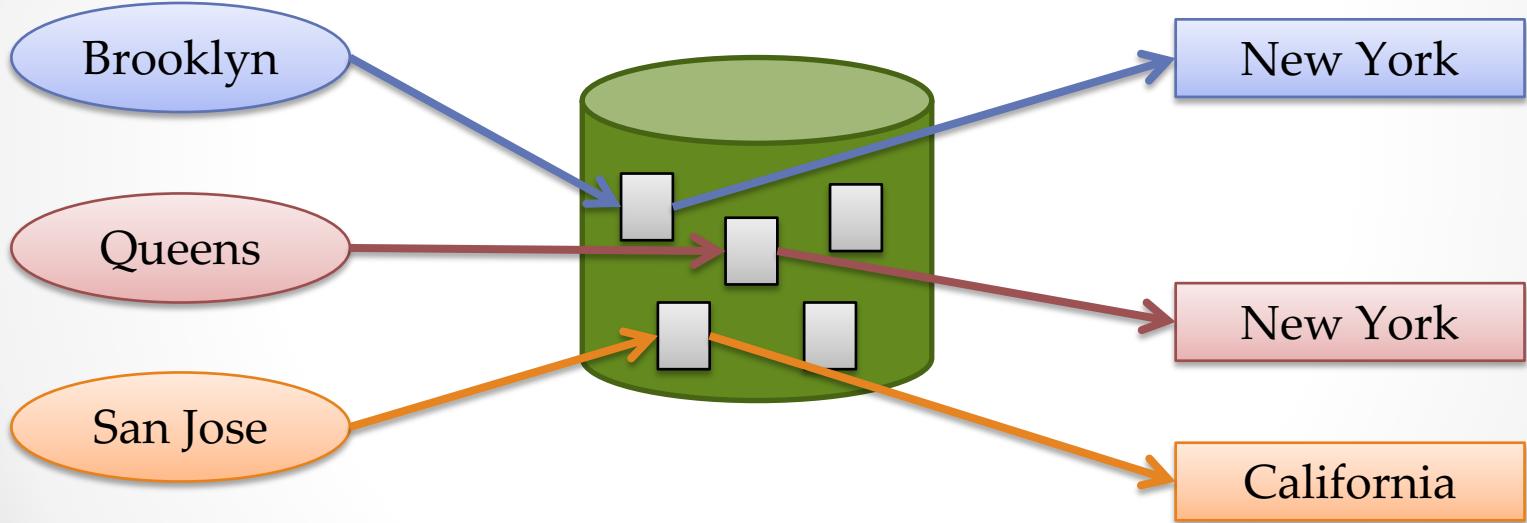
# Map Illustrated



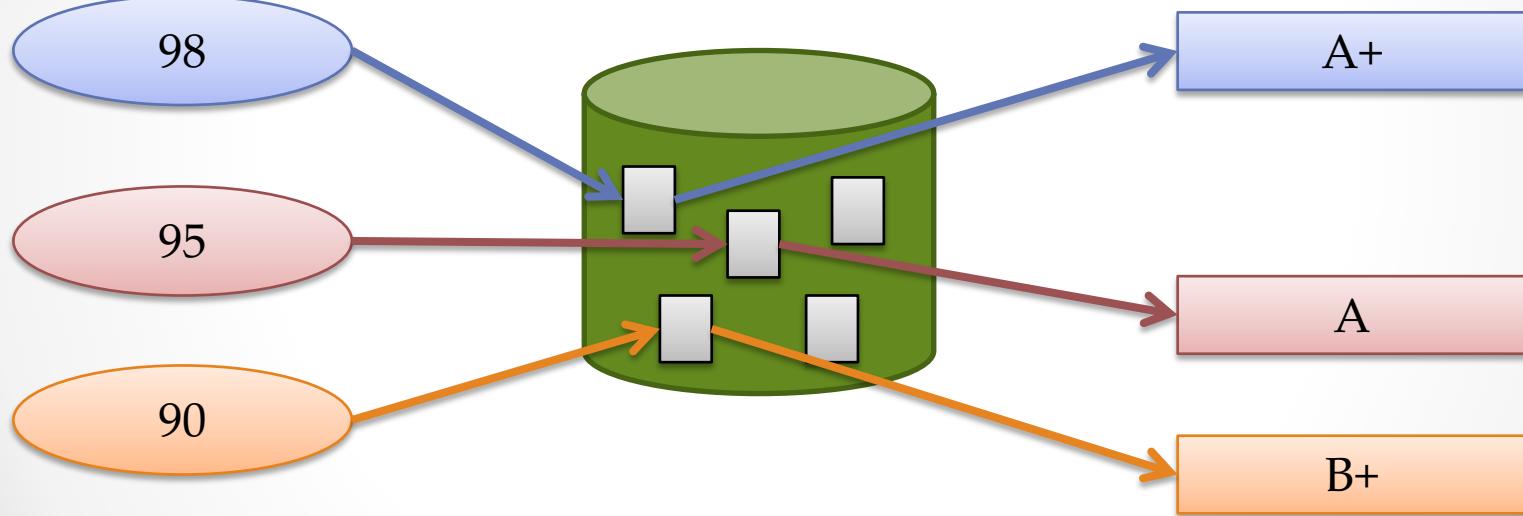
# Map Illustrated



# Map Illustrated



# Map Illustrated



# Map at a Glance

| Intent  | Code                                                             |
|---------|------------------------------------------------------------------|
| Declare | <code>var m map[<i>ktype</i>]<i>vtype</i></code>                 |
| Insert  | <code>m[<i>k</i>] = <i>v</i></code>                              |
| Lookup  | <code><i>v</i> = m[<i>k</i>] or <i>v,ok</i> = m[<i>k</i>]</code> |
| Delete  | <code>delete(m, <i>k</i>)</code>                                 |
| Iterate | <code>for <i>k, v</i> := range m</code>                          |
| Size    | <code>len(m)</code>                                              |

\* Key-type *must* have == operation (no map, slice, or func)



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Map\\_types](https://golang.org/ref/spec#Map_types)
- Go Maps in Action
  - <https://blog.golang.org/go-maps-in-action>



# Section 5 : structs

- Introduction
  - What is a struct?
  - Declaring and Using structs
- Advance Usage
  - Initialize
  - Anonymous Fields
  - Nested structs
- Misc
  - structs and Methods
  - structs and Functions
  - Field Visibility



# Introduction to structs

Section 5 – Lecture 1



# Topics

- What is a Struct?
- Arrays/Slices vs Maps vs Structs
- Simple examples of using a Struct

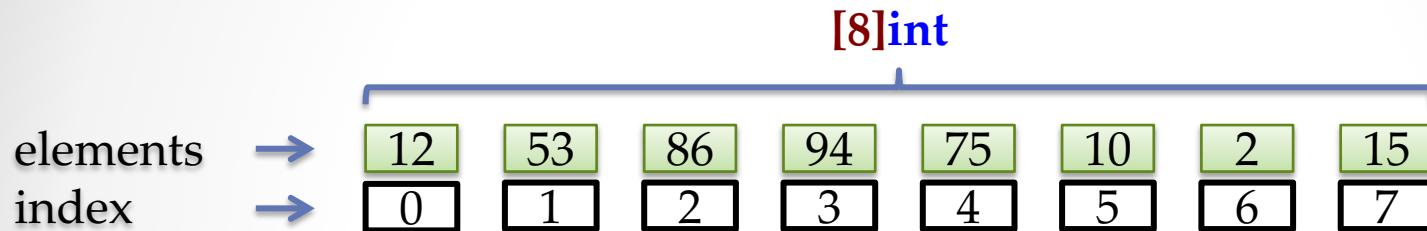


# What is a Struct?

- A **struct** is a **sequence of named elements**, called **fields**, each of which has a **name** and a **type**. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.



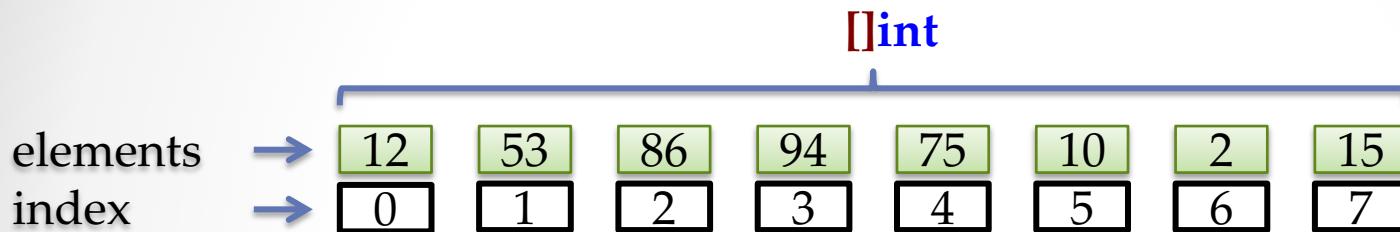
# Arrays



1. Flexible element type
  - o **All elements are of the same type**
2. Fixed boundaries
3. Integer indexing



# Slices

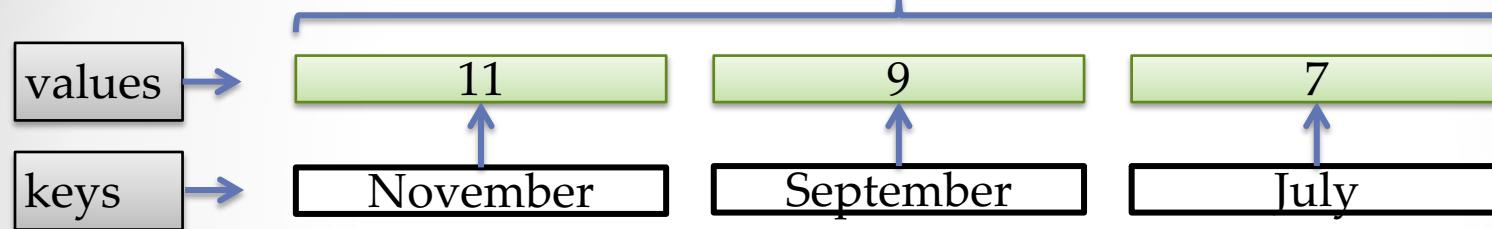


1. Flexible element type
  - o **All elements are of the same type**
2. **Resizable**
3. Integer indexing



# Map

`map[string]int`



1. Flexible element type
  - o **All values are of the same type**
2. Resizable
3. **Flexible indexing type**
  - o **All keys are of the same type**



# Example : People Database

| First Name | Last Name | SSN         | Age |
|------------|-----------|-------------|-----|
| Kermy      | Soro      | 703-59-1234 | 23  |
| Rochell    | Soro      | 568-61-1234 | 35  |
| Sheffy     | Soro      | 670-44-1234 | 21  |
| Devon      | Jones     | 343-78-1234 | 49  |
| Buddie     | Jones     | 176-64-1234 | 27  |
| Bevon      | Peters    | 741-37-1234 | 61  |
| Zara       | Peters    | 623-48-1234 | 19  |
| Eugenius   | Peters    | 754-90-1234 | 15  |
| Aloise     | James     | 614-27-1234 | 10  |
| Ellene     | Smith     | 277-25-1234 | 72  |



# Example : People Database

| First Name | Last Name | SSN         | Age   |
|------------|-----------|-------------|-------|
| Kermy      | Soro      | 703-59-1234 | 23    |
| Rochell    | Soro      | 568-61-1234 | 35    |
| Sheffy     | Soro      | 670-44-1234 | 21    |
| Devon      | Jones     | 343-78-1234 | 49    |
| Buddie     | Jones     | 176-64-1234 | 27    |
| Bevon      | Peters    | 741-37-1234 | 61    |
| Zara       | Peters    | 623-48-1234 | 19    |
| Eugenius   | Peters    | 754-90-1234 | 15    |
| Aloise     | James     | 614-27-1234 | 10    |
| Ellene     | Smith     | 277-25-1234 | 72    |
| string     | string    | string      | uint8 |



# Example : People Database

Diagram illustrating the structure of a People Database:

The database is represented as a 2D array:

```
[[{"First Name": "Kermy", "Last Name": "Soro", "SSN": "703-59-1234", "Age": 23}, {"First Name": "Rochell", "Last Name": "Soro", "SSN": "568-61-1234", "Age": 35}, {"First Name": "Sheffy", "Last Name": "Soro", "SSN": "670-44-1234", "Age": 21}, {"First Name": "Devon", "Last Name": "Jones", "SSN": "343-78-1234", "Age": 49}, {"First Name": "Buddie", "Last Name": "Jones", "SSN": "176-64-1234", "Age": 27}, {"First Name": "Bevon", "Last Name": "Peters", "SSN": "741-37-1234", "Age": 61}, {"First Name": "Zara", "Last Name": "Peters", "SSN": "623-48-1234", "Age": 19}, {"First Name": "Eugenius", "Last Name": "Peters", "SSN": "754-90-1234", "Age": 15}, {"First Name": "Aloise", "Last Name": "James", "SSN": "614-27-1234", "Age": 10}, {"First Name": "Ellene", "Last Name": "Smith", "SSN": "277-25-1234", "Age": 72}]]
```

The columns are labeled:

- First Name
- Last Name
- SSN
- Age

Annotations:

- A red bracket on the left indicates the type `[]string`.
- A blue bracket above the table indicates the type `[]string`.
- The bottom row of the table is highlighted in green, indicating the column types:
  - First Name: `string`
  - Last Name: `string`
  - SSN: `string`
  - Age: `uint8 -> string`

| First Name | Last Name | SSN         | Age             |
|------------|-----------|-------------|-----------------|
| Kermy      | Soro      | 703-59-1234 | 23              |
| Rochell    | Soro      | 568-61-1234 | 35              |
| Sheffy     | Soro      | 670-44-1234 | 21              |
| Devon      | Jones     | 343-78-1234 | 49              |
| Buddie     | Jones     | 176-64-1234 | 27              |
| Bevon      | Peters    | 741-37-1234 | 61              |
| Zara       | Peters    | 623-48-1234 | 19              |
| Eugenius   | Peters    | 754-90-1234 | 15              |
| Aloise     | James     | 614-27-1234 | 10              |
| Ellene     | Smith     | 277-25-1234 | 72              |
| string     | string    | string      | uint8 -> string |



# Example : People Database

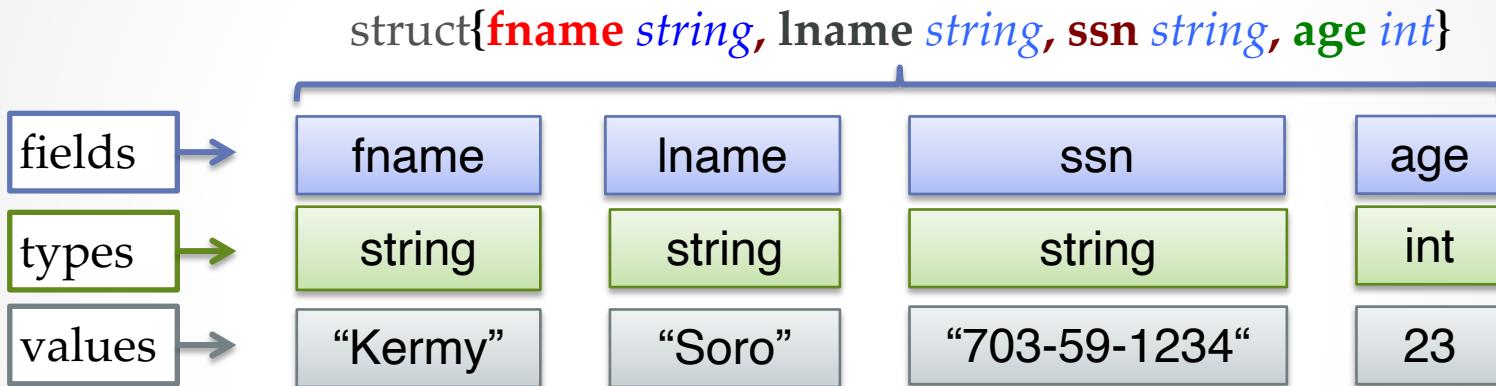
type Person []string

[:]Person

| First Name | Last Name | SSN         | Age             |
|------------|-----------|-------------|-----------------|
| Kermy      | Soro      | 703-59-1234 | 23              |
| Rochell    | Soro      | 568-61-1234 | 35              |
| Sheffy     | Soro      | 670-44-1234 | 21              |
| Devon      | Jones     | 343-78-1234 | 49              |
| Buddie     | Jones     | 176-64-1234 | 27              |
| Bevon      | Peters    | 741-37-1234 | 61              |
| Zara       | Peters    | 623-48-1234 | 19              |
| Eugenius   | Peters    | 754-90-1234 | 15              |
| Aloise     | James     | 614-27-1234 | 10              |
| Ellene     | Smith     | 277-25-1234 | 72              |
| string     | string    | string      | uint8 -> string |



# Structs Illustrated



- What is a struct?
  - A **struct** is a **sequence of named elements**, called **fields**, each of which has a **name** and a **type**...



# Review: Struct at a Glance

| Intent  | Code                                        |
|---------|---------------------------------------------|
| Declare | <code>var s struct{field0 type, ...}</code> |
| Assign  | <code>s.fieldX = v</code>                   |
| Lookup  | <code>v = s.fieldX</code>                   |



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Struct\\_types](https://golang.org/ref/spec#Struct_types)



# Advance structs

# Usage

Section 5 – Lecture 2



# Topics

- Initialization
- Nesting Structs
- Anonymous Fields
  - Basic types
  - Embedded Structs
    - Accessing fields in an anonymous Struct
    - Dealing with ambiguity



# Review

Section 5 – Lecture 3



# Topics

- Adding Methods to structs
- structs and Functions
  - To copy or not to copy?
- Field Visibility
  - Private vs Public



# Section 6 : goroutine

- Basics
  - What is a goroutine?
  - Creating goroutines
- Advance Usage
  - Waiting and synchronization
  - Pitfalls



# Introduction to goroutines

Section 6 – Lecture 1

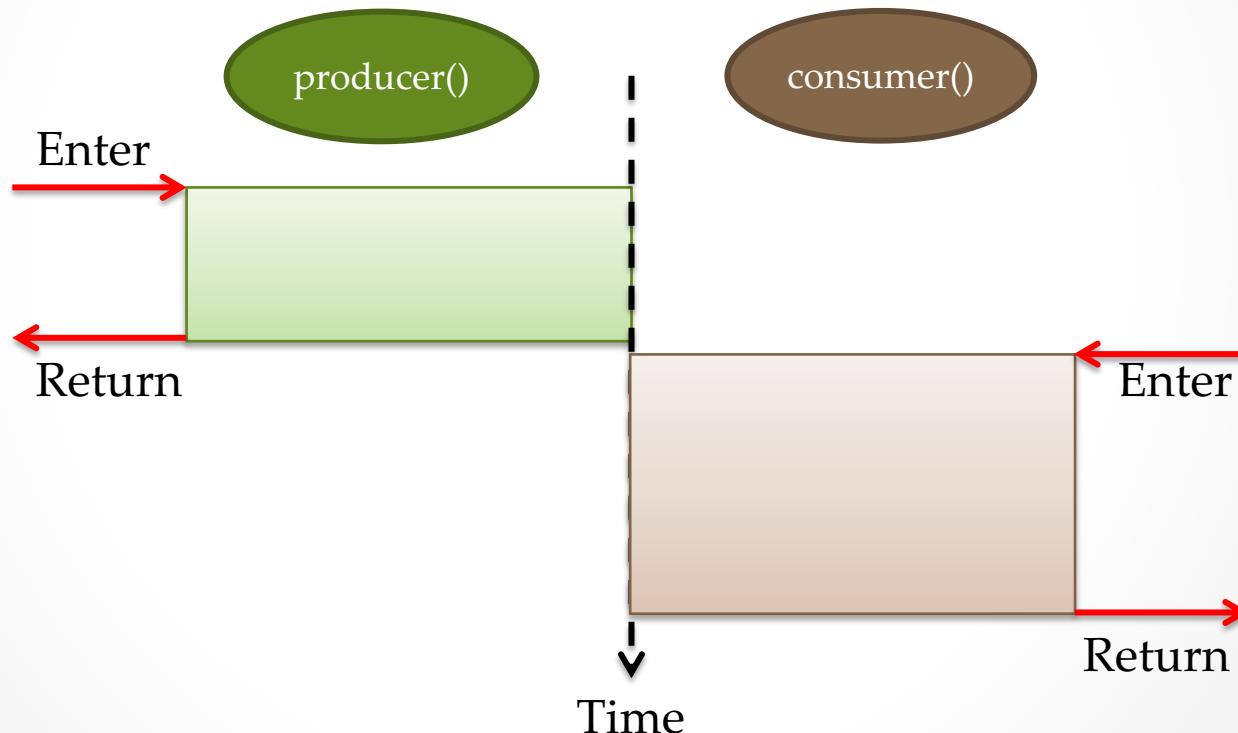


# Topics

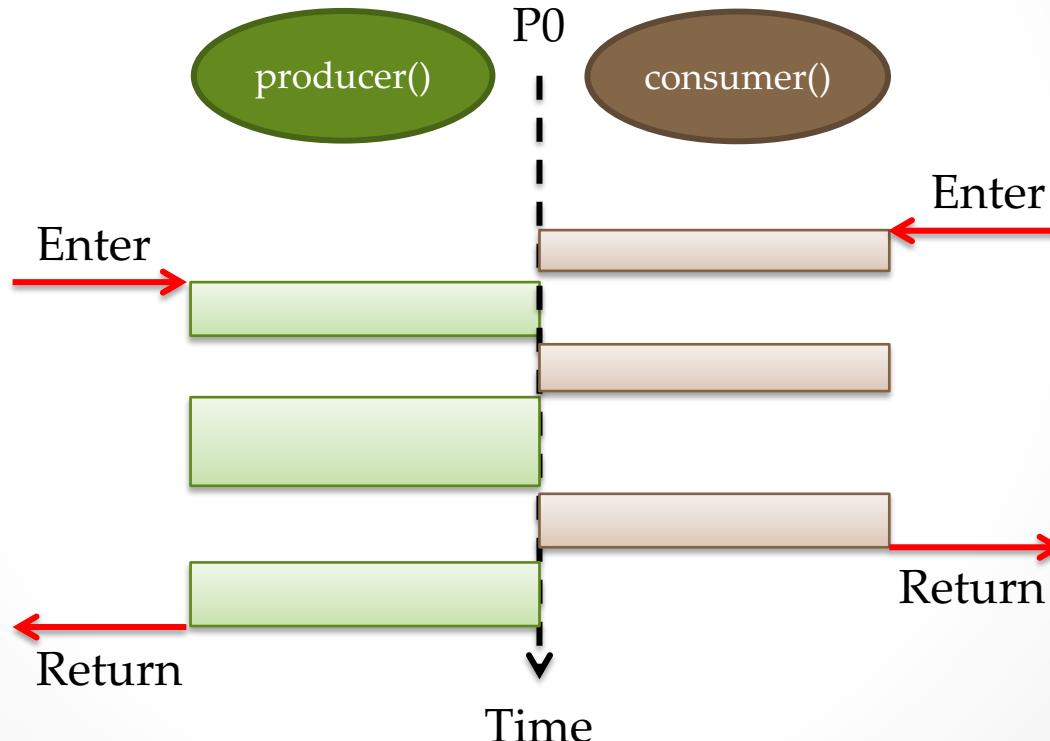
- Creating a ‘goroutine’
  - Using named function
  - Using anonymous function
- What is a ‘goroutine’?
  - What is the Responsibility of a goroutine?
- When does my program end?



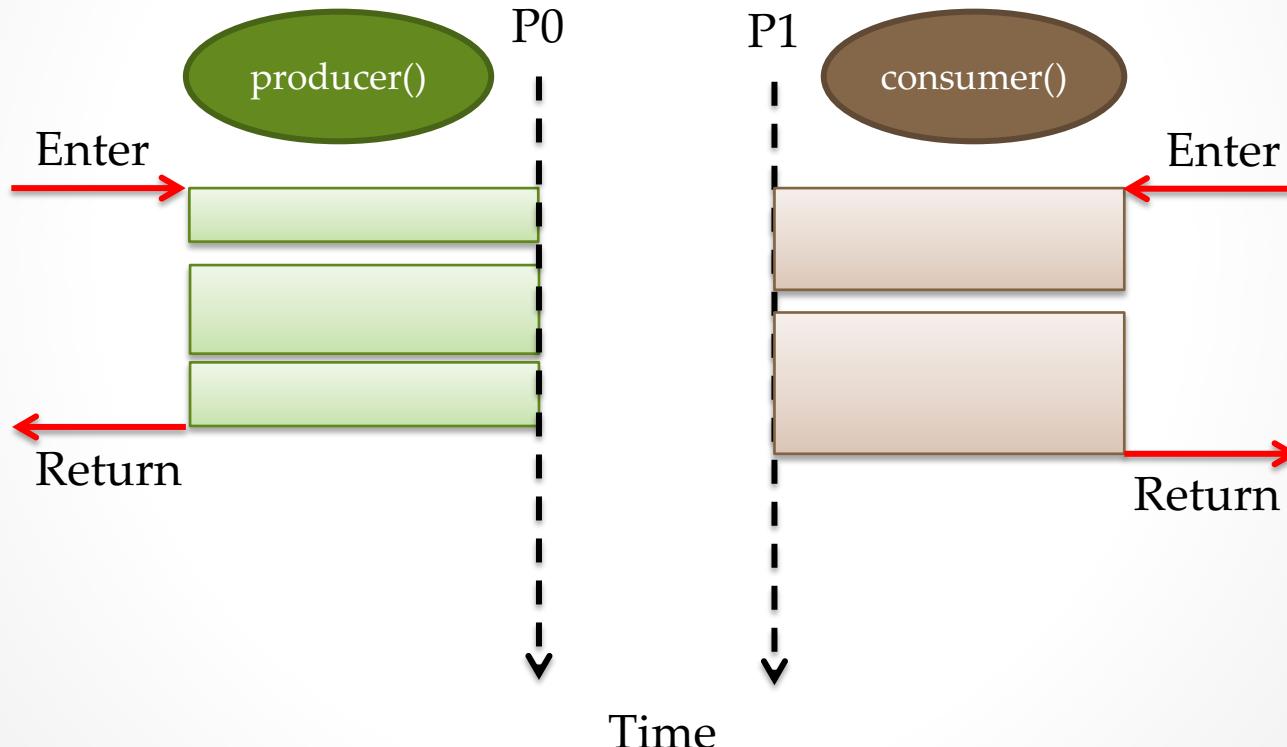
# Sequential Functions



# Concurrent Functions



# Parallelism



# Concurrency vs. Parallelism

- Concurrency **is not** Parallelism
- Concurrency is a way of writing code
- Parallelism is how the code execute
- You get Parallelism **for free** using Concurrent programming



# Concurrency is not parallelism

- “when people hear the word concurrency they often think of parallelism, a related but quite distinct concept. In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.”[1]



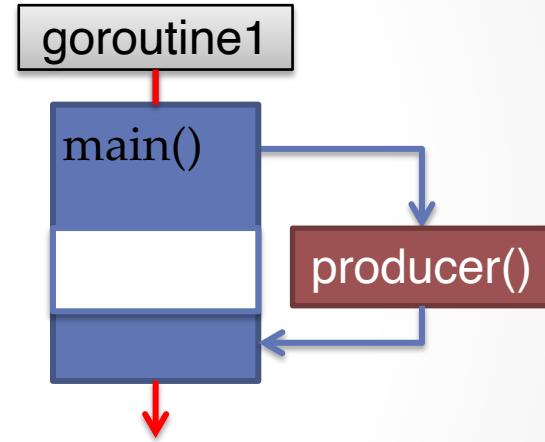
# What is a ‘goroutine’?

- Responsibility of a ‘goroutine’
  - A *goroutine* is **responsible** for managing the **execution context** of a **function**.



# goroutine!

```
1 package main
2
3 func main() {}
4
5
```

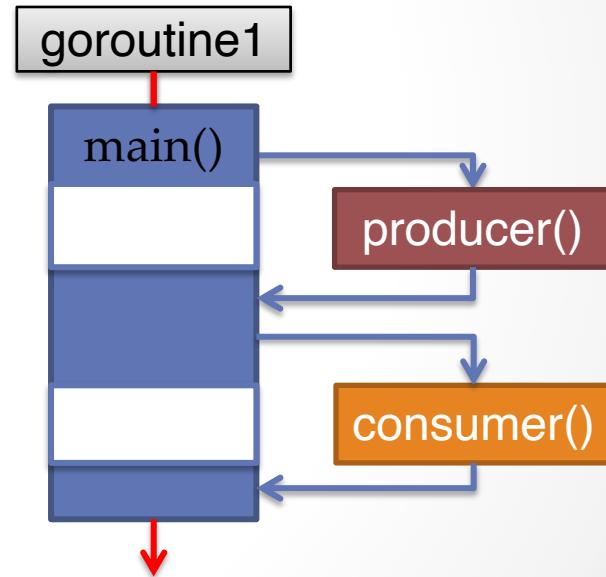


A *goroutine* is **responsible** for managing the *execution context* of a function.



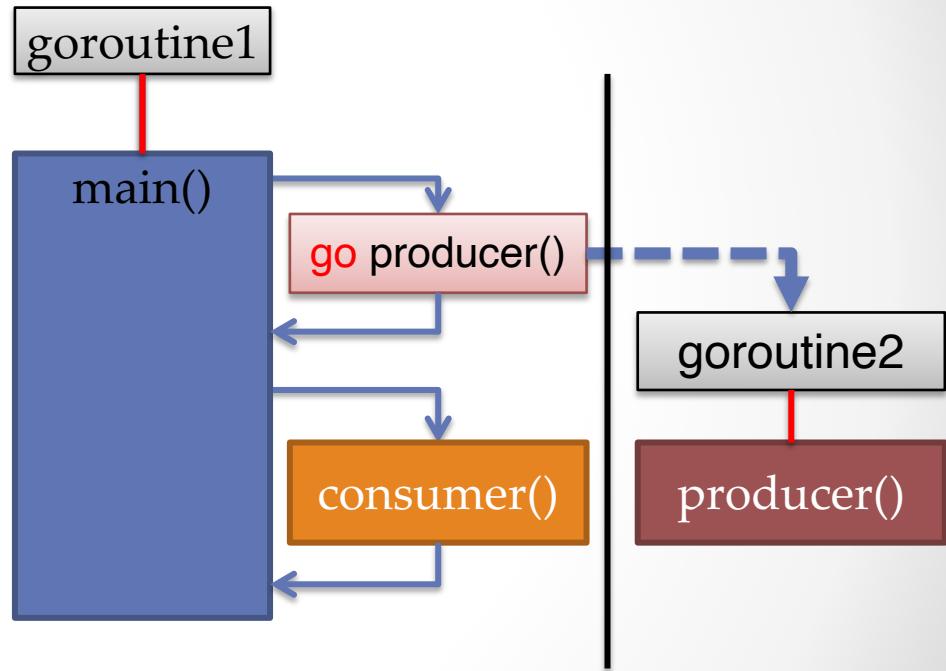
# goroutine!

```
3 func main() {
4 producer()
5 consumer()
6 }
7
8 func consumer() {
9 }
10
11 func producer() {
12 }
```



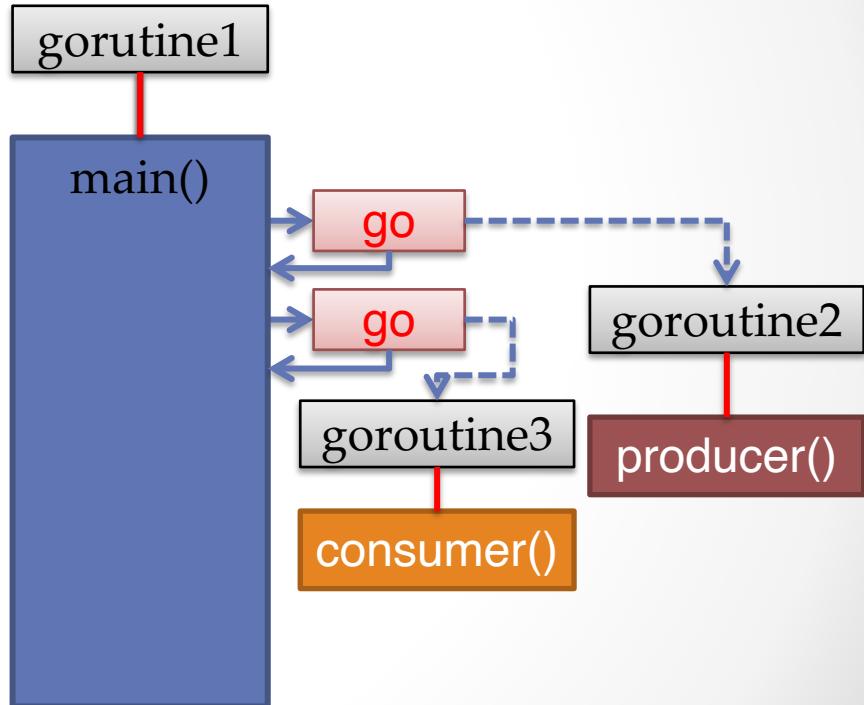
# goroutine!

```
3 func main() {
4 go producer()
5 consumer()
6 }
7
8 func consumer() {
9 }
10
11 func producer() {
12 }
13 }
```



# goroutine!

```
3 func main() {
4 go producer()
5 go consumer()
6 }
7
8 func consumer() {
9 }
10
11 func producer() {
12 }
```



## The End of 'main()'

- Your Program ends, when '**main()**' ends or completes.
- Which also means that **all** goroutines also **ends** or gets **killed**.



# Additional Resource

- Go Statement
  - [https://golang.org/ref/spec#Go\\_statements](https://golang.org/ref/spec#Go_statements)
- [1] - Rob Pike – ‘Concurrency Is Not Parallelism’
  - [https://www.youtube.com/watch?v=cN\\_DpYBzKso](https://www.youtube.com/watch?v=cN_DpYBzKso)



# Waiting and Synchronization

Section 6 – Lecture 2

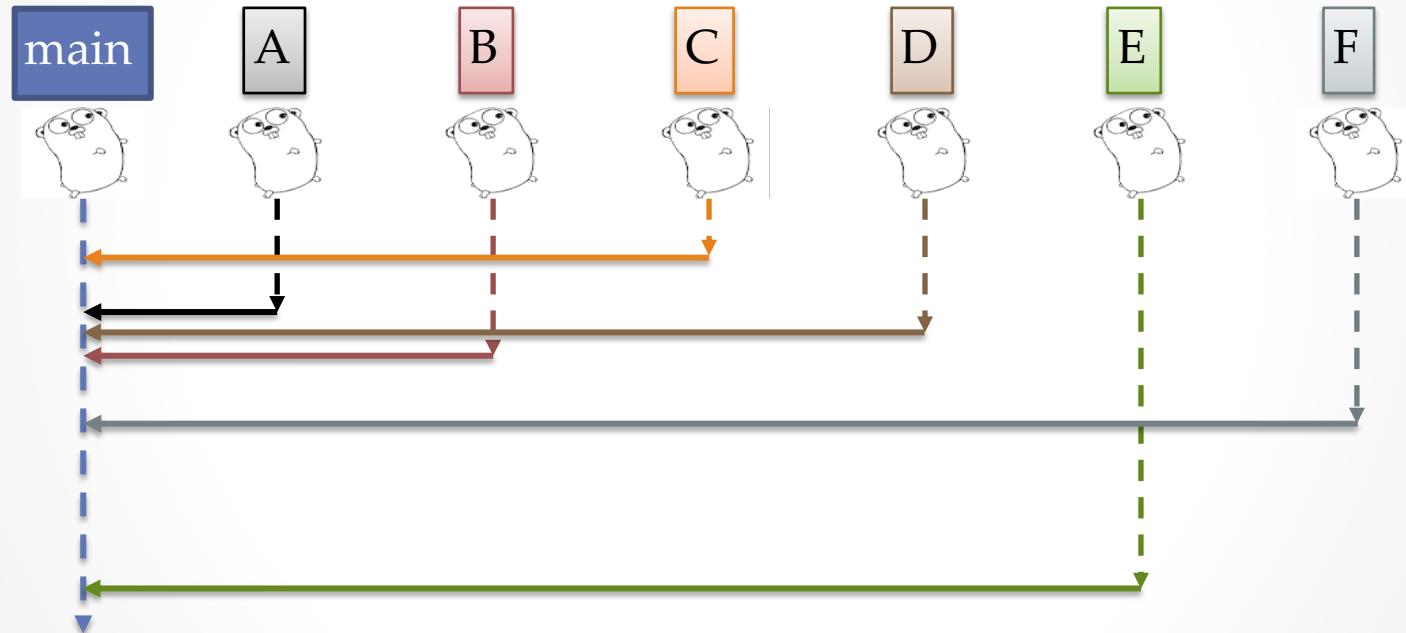


# Topics

- Waiting for goroutines
  - Non-ideal
  - Ideal
- Synchronization



# Waiting for Goroutines To Complete



# Additional Resource

- Go Statement
  - [https://golang.org/ref/spec#Go\\_statements](https://golang.org/ref/spec#Go_statements)
- sync.WaitGroup
  - <https://golang.org/pkg/sync/#WaitGroup>



# goroutine pitfalls

Section 6 – Lecture 3



# Topics

- Beware of when goroutines start
  - Give time for goroutine to start before your program ends
- Beware of goroutines launch as closures
  - Closures referencing '**the same variables**'
- Beware of sync.WaitGroup
  - Don't copy sync.WaitGroup object
  - Incorrect initialization or improper usage



# Additional Resource

- Go Statement
  - [https://golang.org/ref/spec#Go\\_statements](https://golang.org/ref/spec#Go_statements)
- sync.WaitGroup
  - <https://golang.org/pkg/sync/#WaitGroup>



# Review

Section 6 – Lecture 4



# Topics

- How many goroutines per application?
  - goroutines vs. OS threads
- Preventing concurrent access
  - Using sync.Mutex



# Goroutines vs OS Threads

- Go runtimes manages multiple **goroutines** per **OS thread**
  - You can have more goroutines than OS Threads
  - You can runs 100s of thousands to millions of goroutines
    - Whereas, you can only run a few thousands OS Threads
- OS Threads takes longer to start, hence heavyweight



# Key Take Away

- Goroutines are light weight
- All goroutines are killed when main() ends
- An application can create 100s of thousands or even millions of goroutines
- A goroutine manages function execution
- Goroutines makes it easier to think and program with concurrency patterns



# Additional Resource

- sync.WaitGroup
  - <https://golang.org/pkg/sync/#WaitGroup>
- sync.Mutex
  - <https://golang.org/pkg/sync/#Mutex>
- Strings Pacakge
  - <https://golang.org/pkg/strings/>
- RegExp Package
  - <https://golang.org/pkg/regexp/>



# Lab

- Write a Go Language program which will print out the number of occurrences of each word.
  - Word Count (iterative)
  - Word Count (concurrent)
- Reading files
  - Using ‘shared.input.FileReader’ package
- Measuring time



# Example

To be or not to be, that is the question.



| Word      | Count |
|-----------|-------|
| To        | 1     |
| be        | 1     |
| or        | 1     |
| not       | 1     |
| to        | 1     |
| be,       | 1     |
| that      | 1     |
| is        | 1     |
| the       | 1     |
| question. | 1     |



# Example

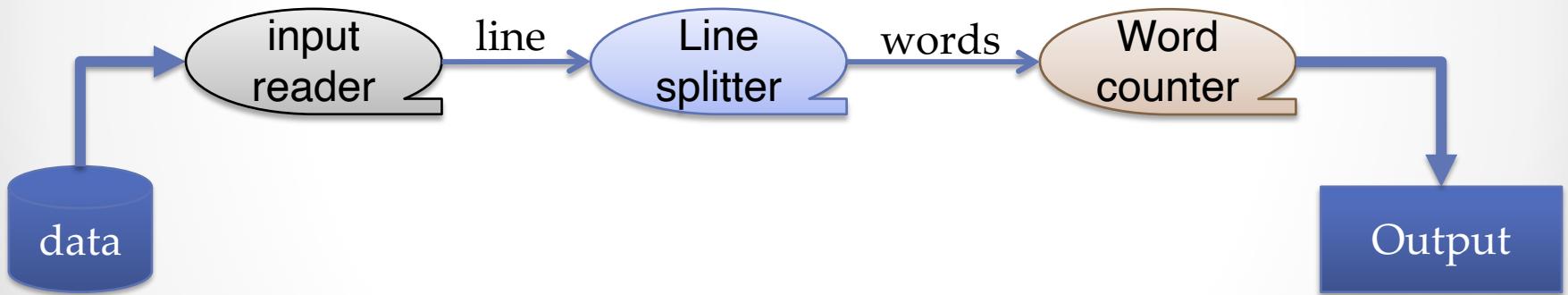
To be or not to be, that is the question.

to be or not to be that is the question

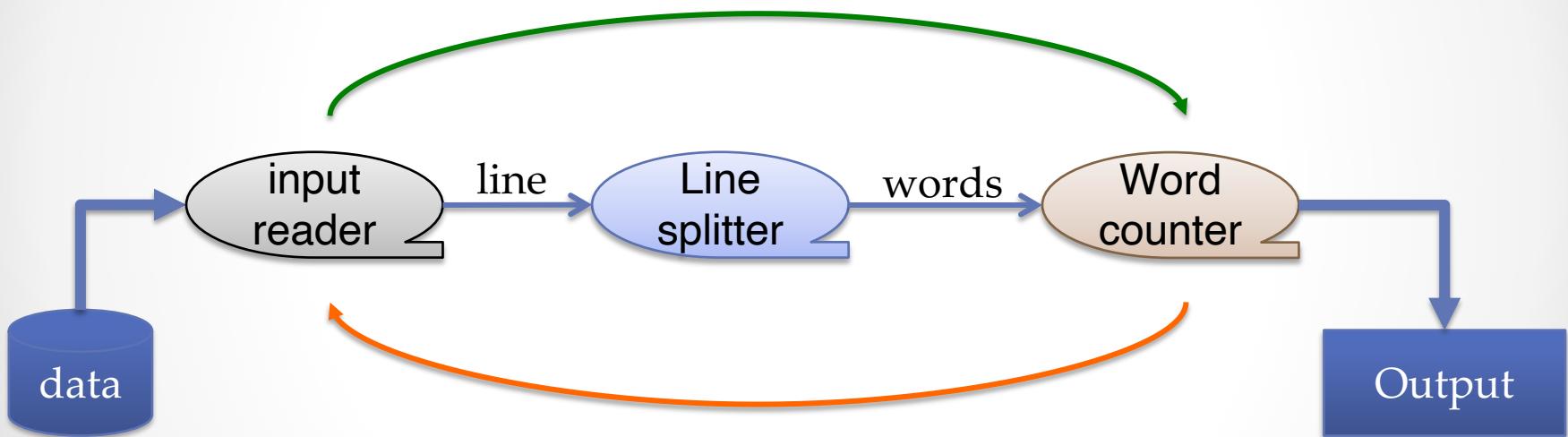


| Word     | Count |
|----------|-------|
| to       | 2     |
| be       | 2     |
| or       | 1     |
| not      | 1     |
| that     | 1     |
| is       | 1     |
| the      | 1     |
| question | 1     |

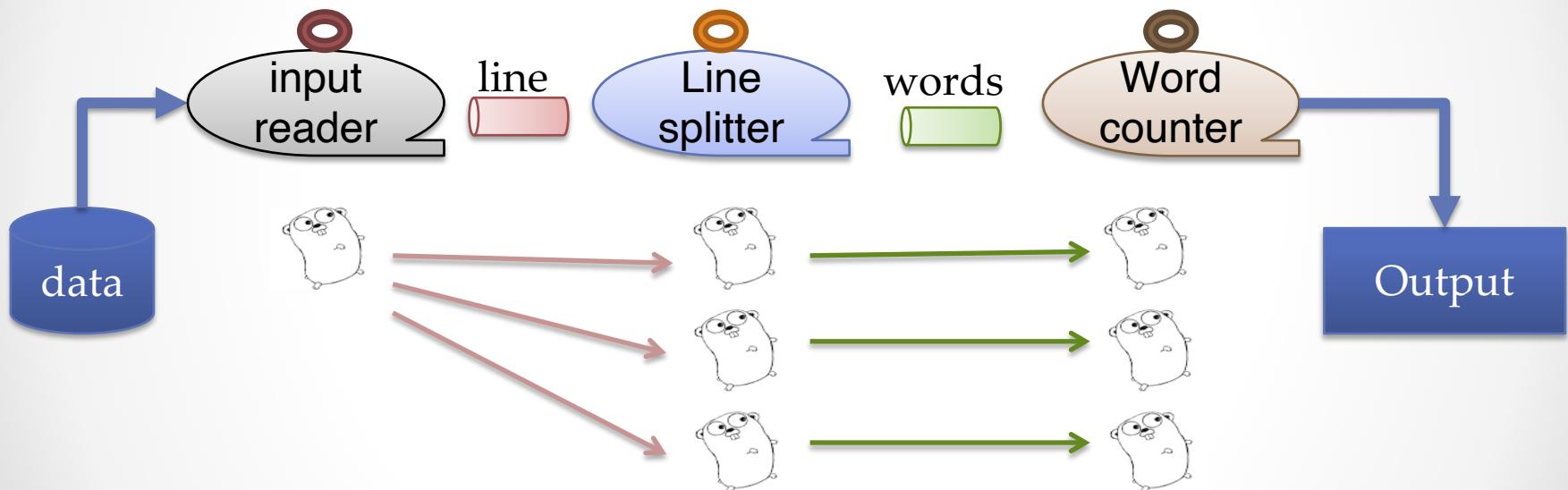
# Program Flow



# Lab 1: Iterative Solution



# Lab 2: Concurrent Solution



# Section 7 : channel

- Basics
  - What is a channel?
  - Declaring and Using channels
  - Functions and channels
- Advance Usage
  - goroutines and channels
    - Channel selection
    - Concurrency patterns
  - Read-only and Write-only Channel
  - Channel of channel



# Introduction to channels

Section 7 – Lecture 1



# Topics

- What is a Channel?
- Simple channel usage
  - Sending
  - Receiving
- Buffered and unbuffered channels

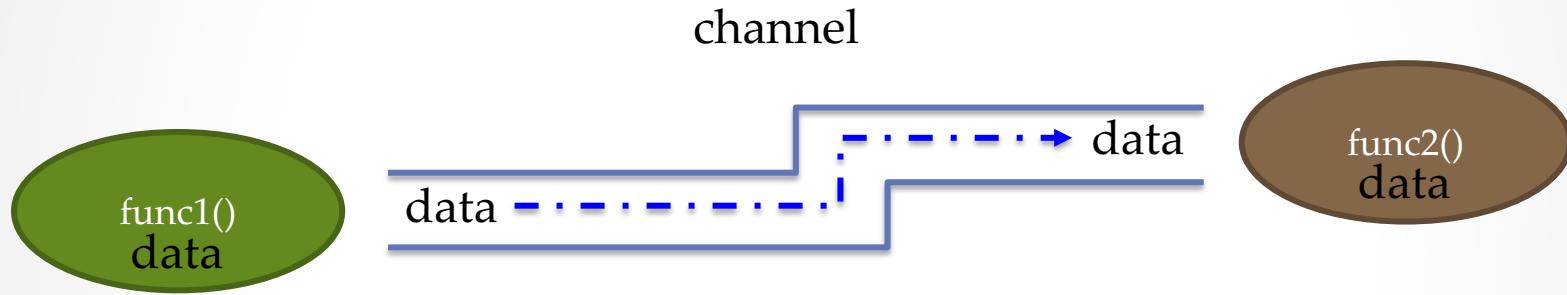


# What is a Channel?

- A **channel** provides a mechanism for **concurrently executing functions** to **communicate** by **sending** and **receiving** values of a **specified** element type. The value of an uninitialized channel is *nil*.



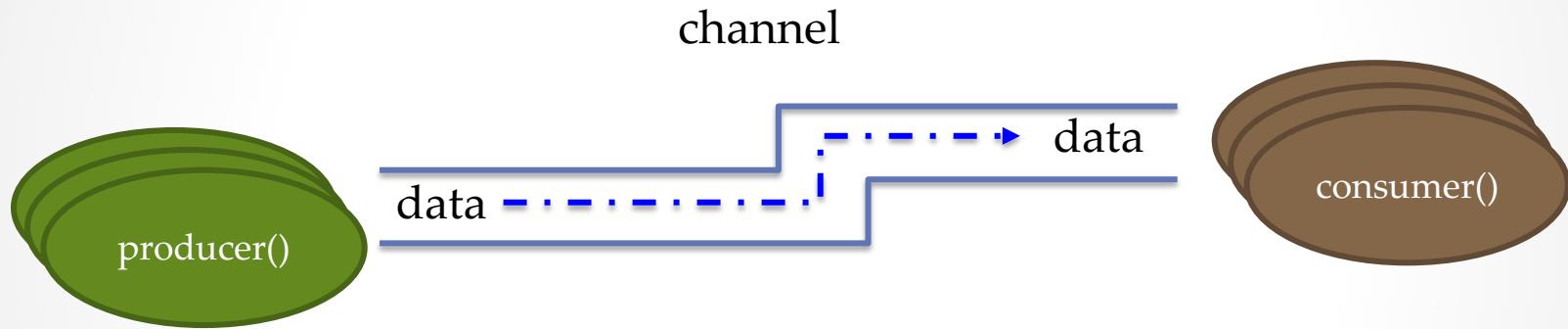
# Decoupling Send & Receiver



**FIFO : First In First Out**



# Producer/Consumer

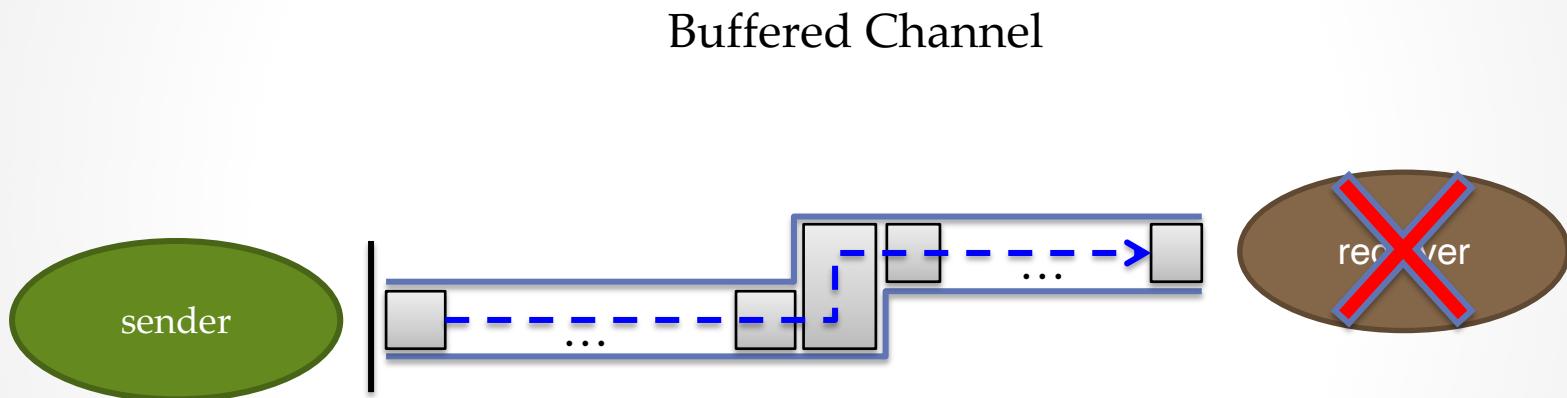


# Review

Section 7 – Lecture 1

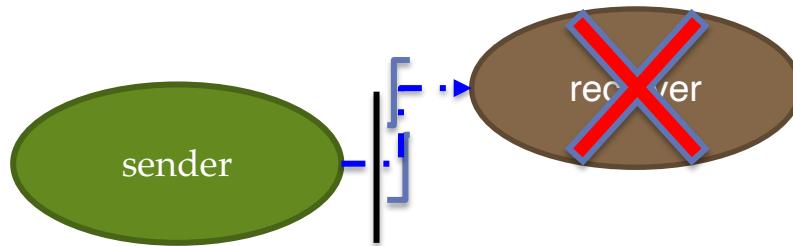


# Sender Blocked: Buffered Channel



# Sender Blocked: Unbuffered Channel

Unbuffered Channel



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types)



# Working with Closed channels

Section 7 – Lecture 2

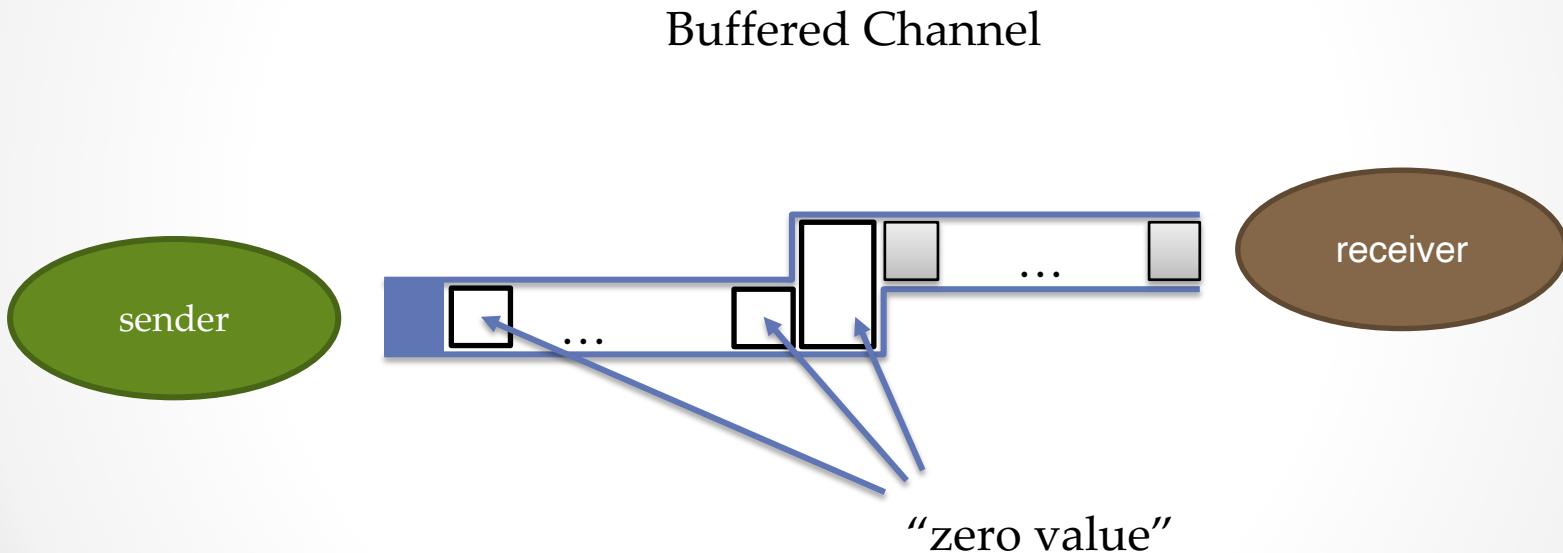


# Topics

- o Working with a '*closed*' channel
  - Testing for closed channel
- o Range operator



# Sender Blocked: Buffered Channel



# Channels & Functions

Section 7 – Lecture 3



# Topics

- Passing channels to functions
- Returning channels from functions



# Channels and Goroutines

Section 7 – Lecture 4

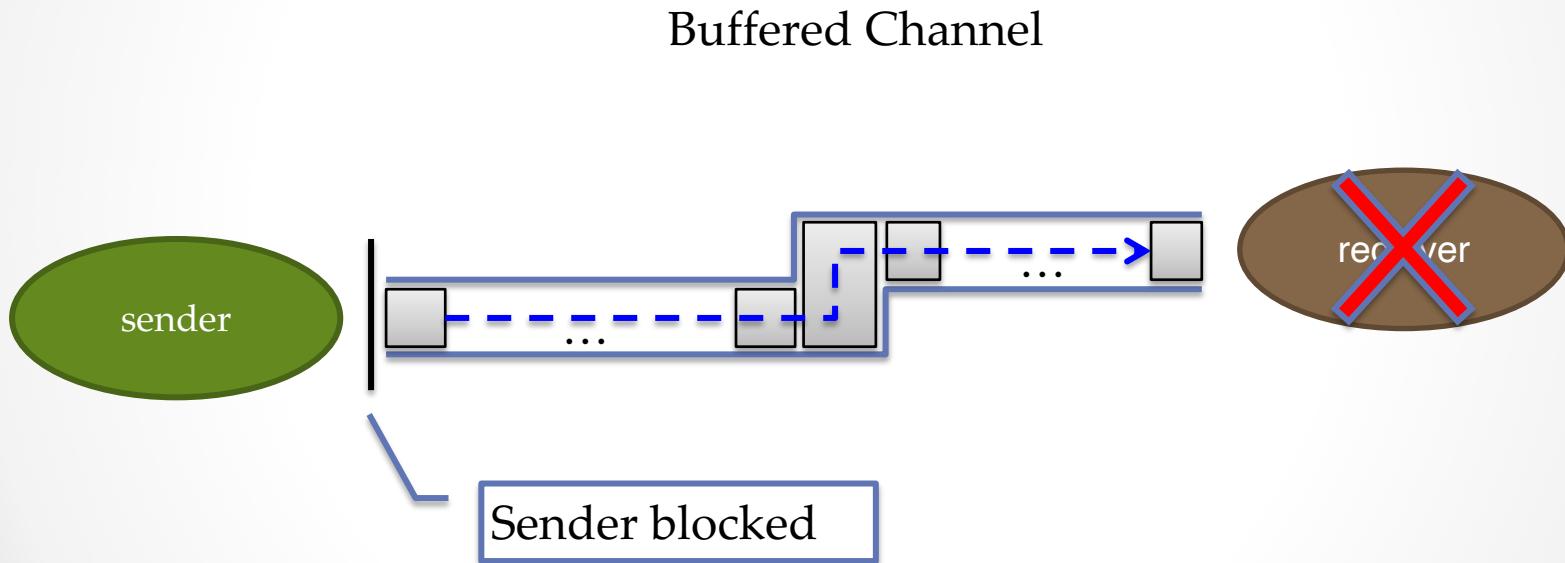


# Topics

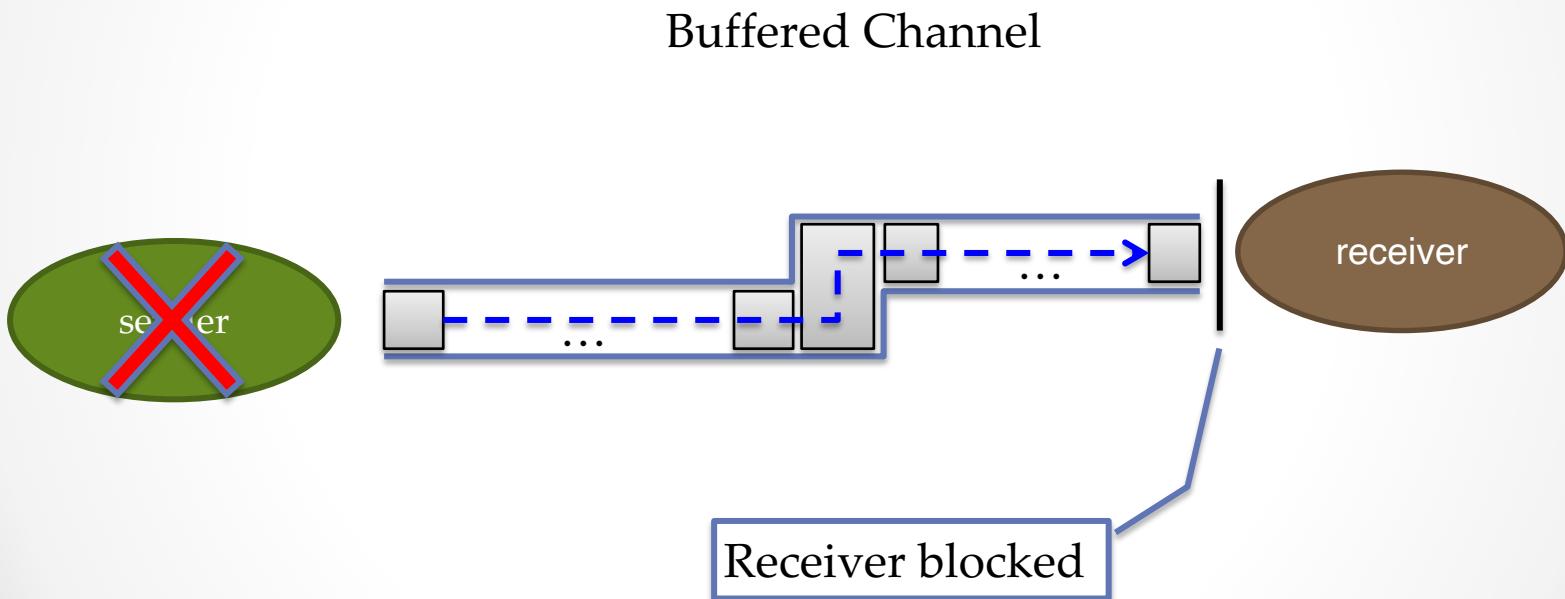
- Using channels to communicating between goroutines
- Deadlock
  - Blocked goroutines
    - Sending
    - Receiving



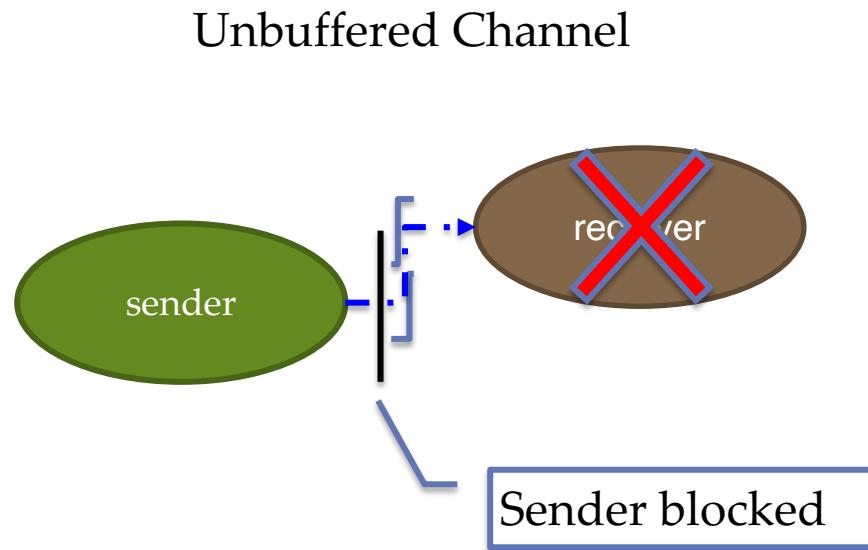
# Sender Blocked: Buffered Channel



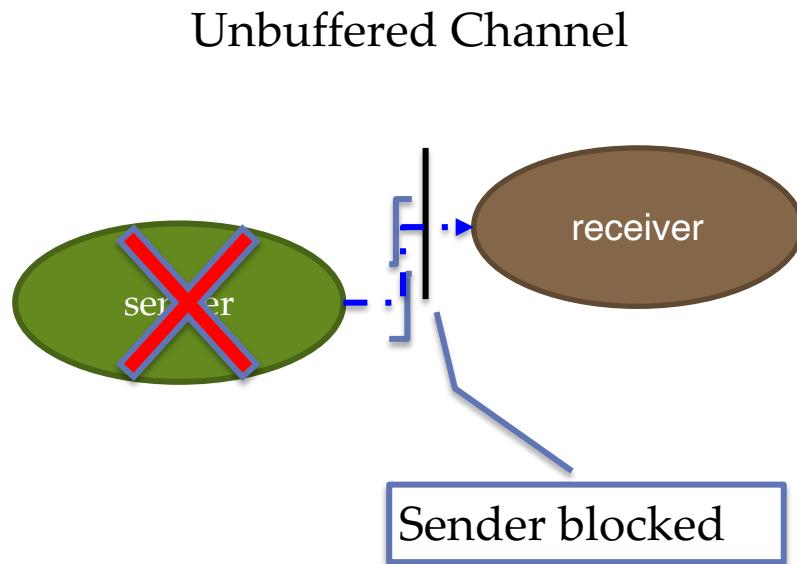
# Receiver Blocked: Buffered Channel



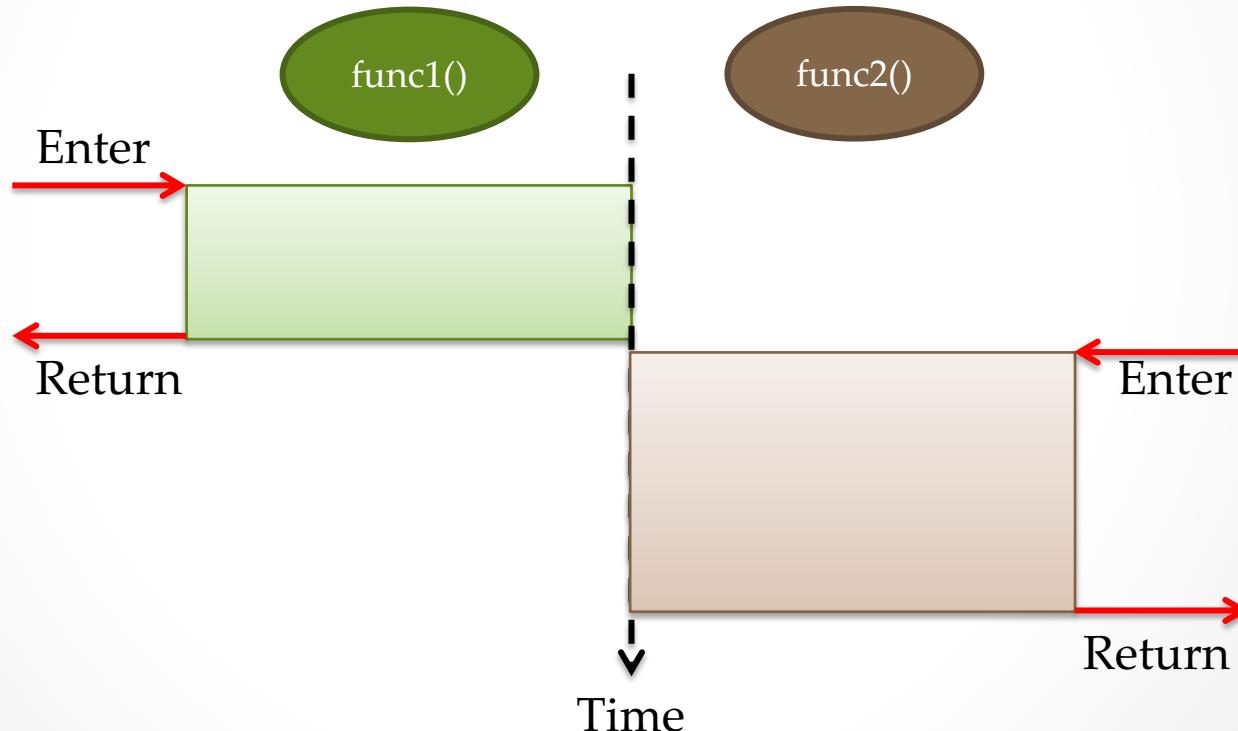
# Sender Blocked: Unbuffered Channel



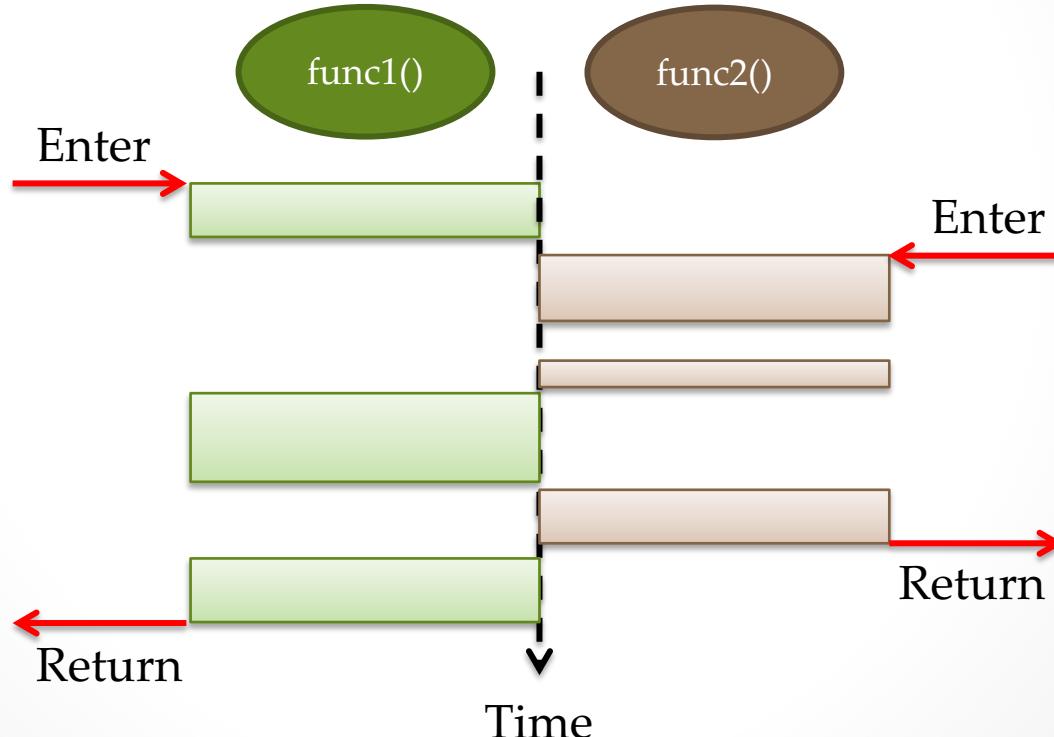
# Receiver Blocked: Unbuffered Channel



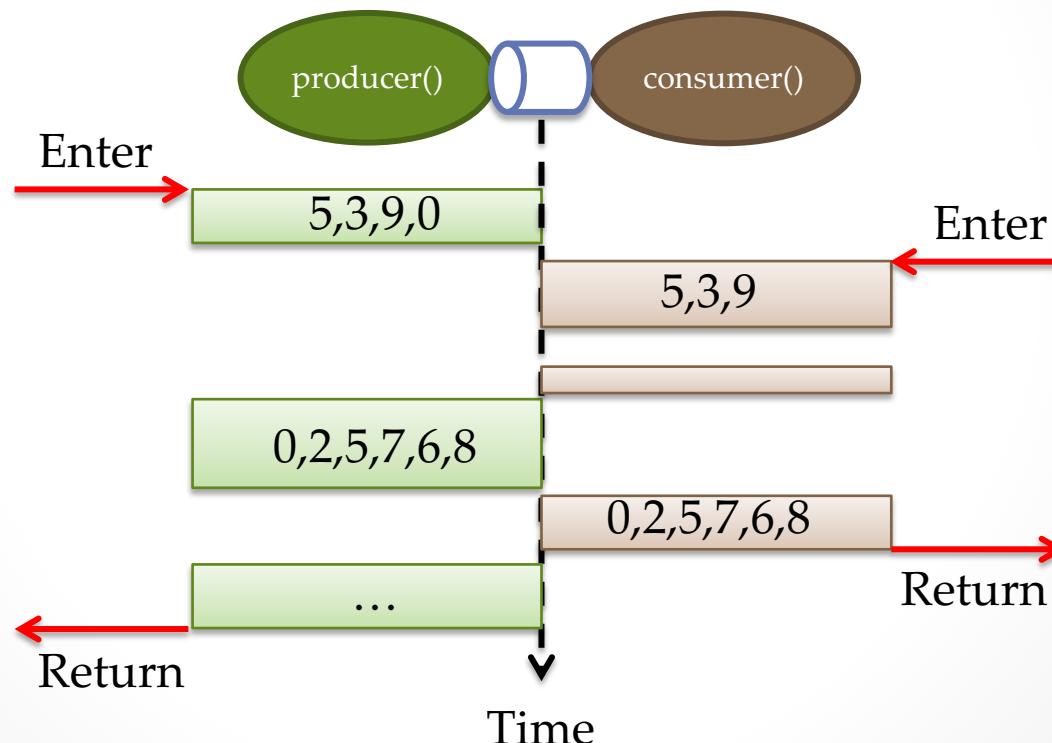
# Serial Function Call



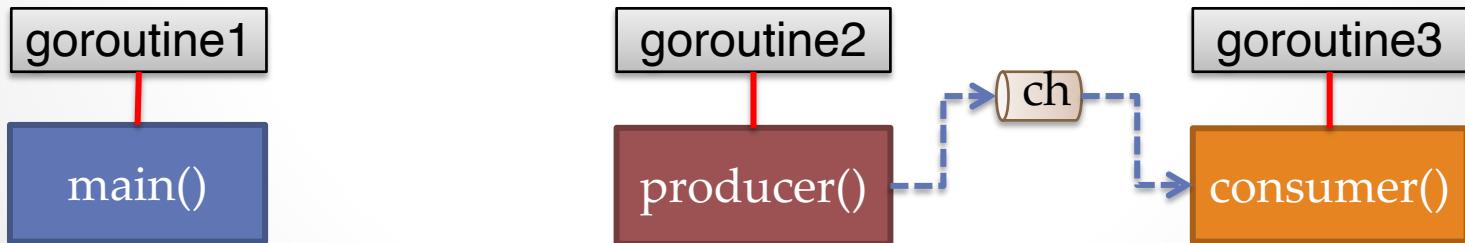
# Concurrency at Glance



# Concurrency with Channel



```
5 func main() {
6 ch := make(chan int)
7 go producer(ch)
8 go consumer(ch)
9 }
10 func consumer(in chan int) {
11 }
12 func producer(out chan int) {
13 }
```



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types)



# Channel Selection

Section 7 – Lecture 5



# Topics

- Inserting delays
  - standard ‘time’ package
- Using the ‘select’ keyword
  - How to choose a channel?
  - Using ‘default’ case
- Timing out



# Using a Single Channel

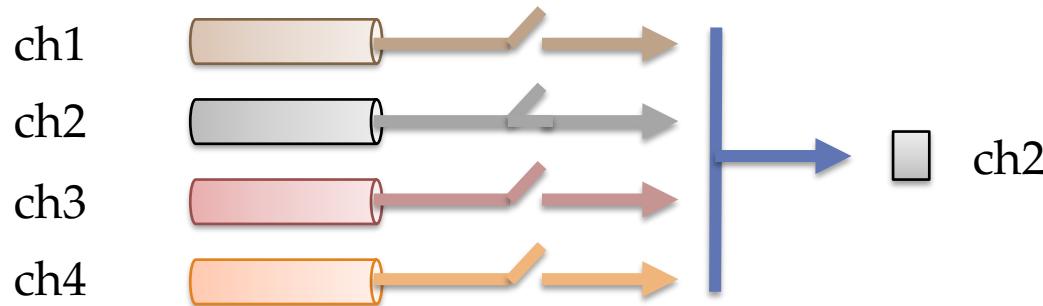
Send



Receive



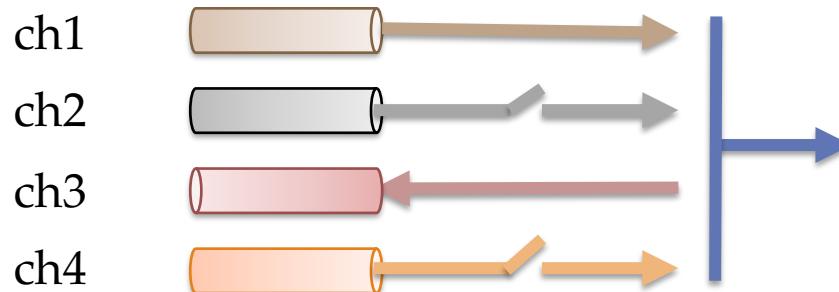
# Channel Selection



*'select'* uniformly picks one of the channels that is *ready*



# Selection



Receive from ch1?  
or  
Send to ch3?

*'select'* uniformly picks one of the channels that is *ready*



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types)
  - [https://golang.org/ref/spec#Select\\_statements](https://golang.org/ref/spec#Select_statements)



# Concurrency Patterns

Section 7 – Lecture 6



# Topics

- Simple Patterns
  - Generator
  - Sink
  - Processor
  - Fan-Out
  - Fan-In

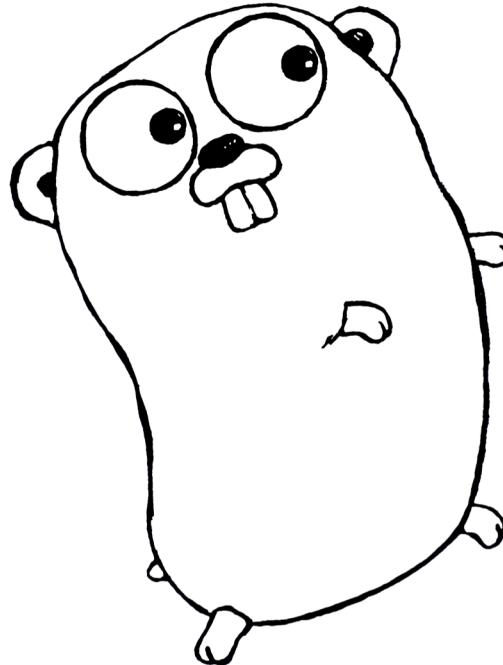


# What is a Design Pattern

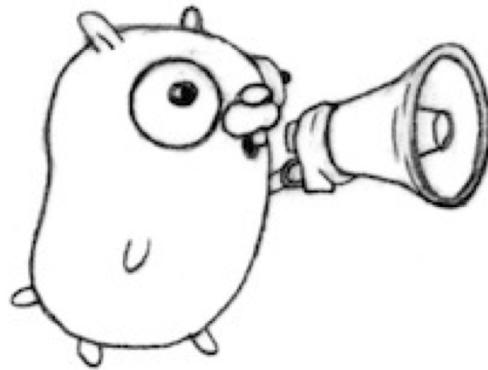
- From Wikipedia:
  - It is a *description* or *template* for how to *solve* a *problem* that can be used *in many different situations*. *Design patterns* are *formalized best practices* that the programmer can use to solve *common problems* when designing an application or system.
    - [https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)



# Go (golang) Mascot : goroutine

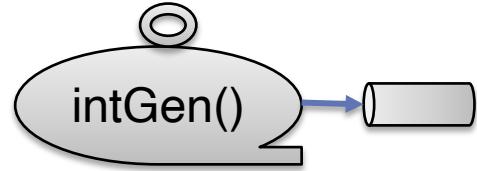


# Generator

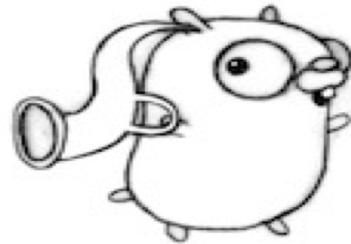


A **function** that launches a **goroutine** to *produce data* on a **channel** it **returns**

# Example 1



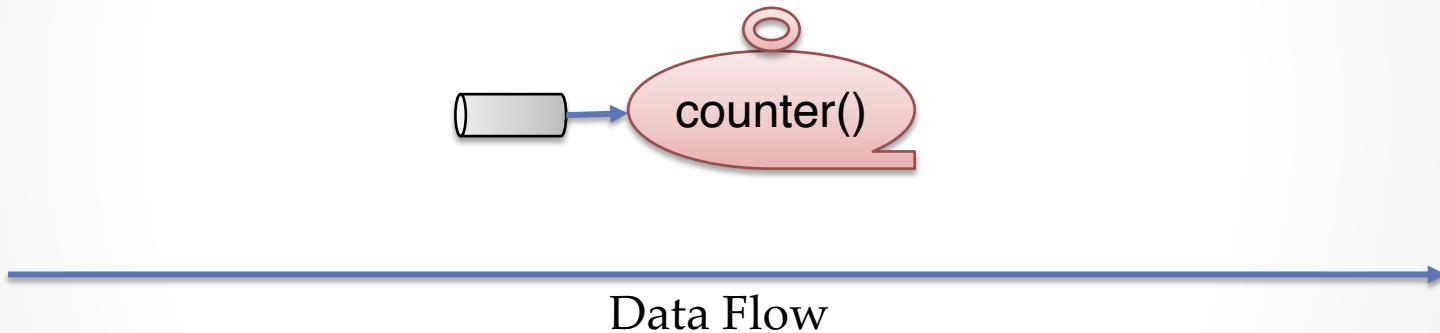
# Sink



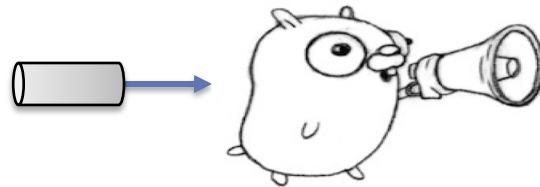
A **function** that launches a **goroutine** to *consume data* on a **channel**



# Example 2 & 3



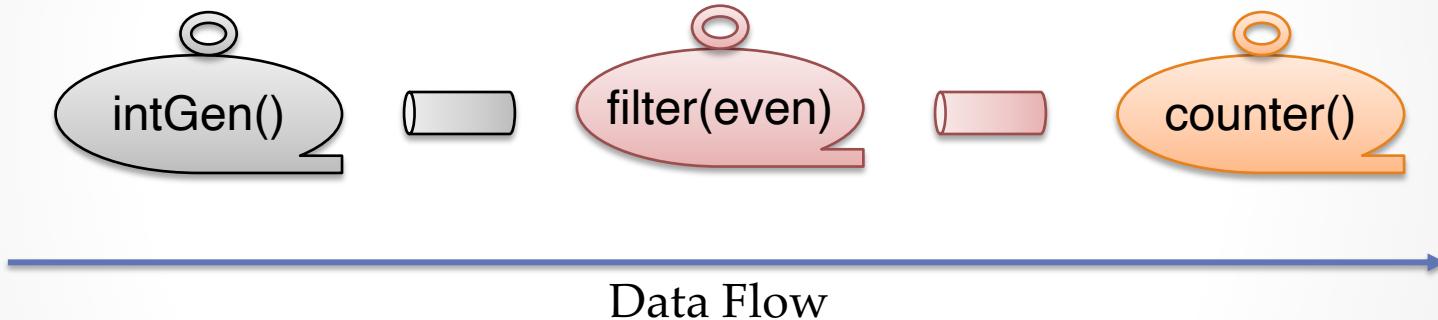
# Processor



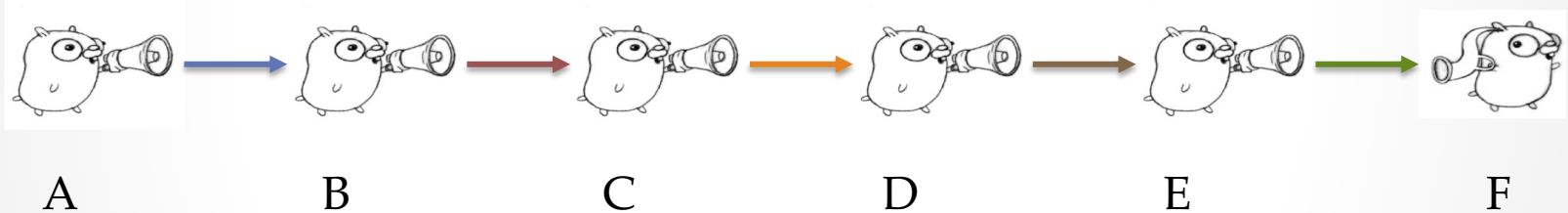
A *function* that launches a *goroutine* to *consume & produce data*



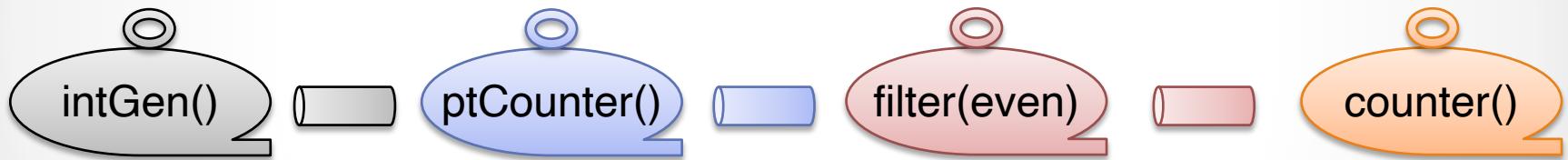
# Example 4



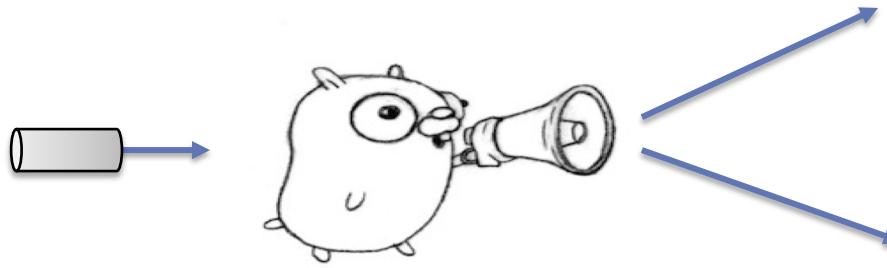
# Daisy-Chain/Pipeline



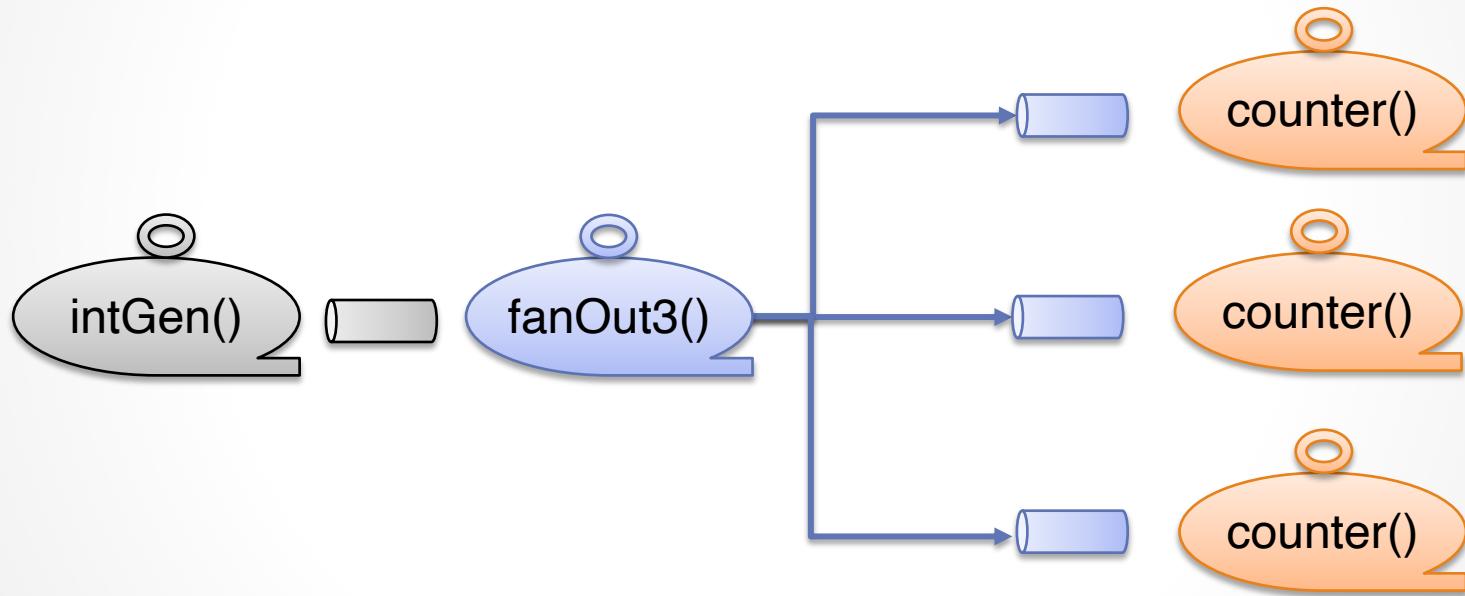
# Example 5



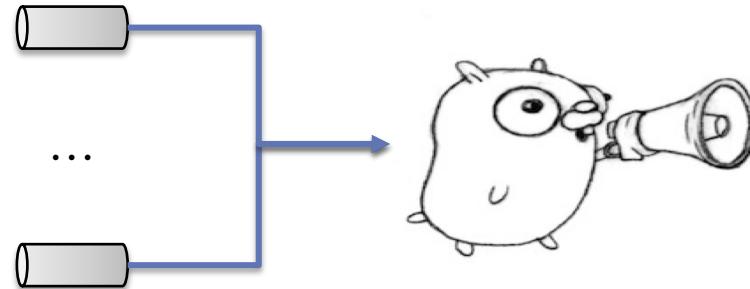
# Fan-Out



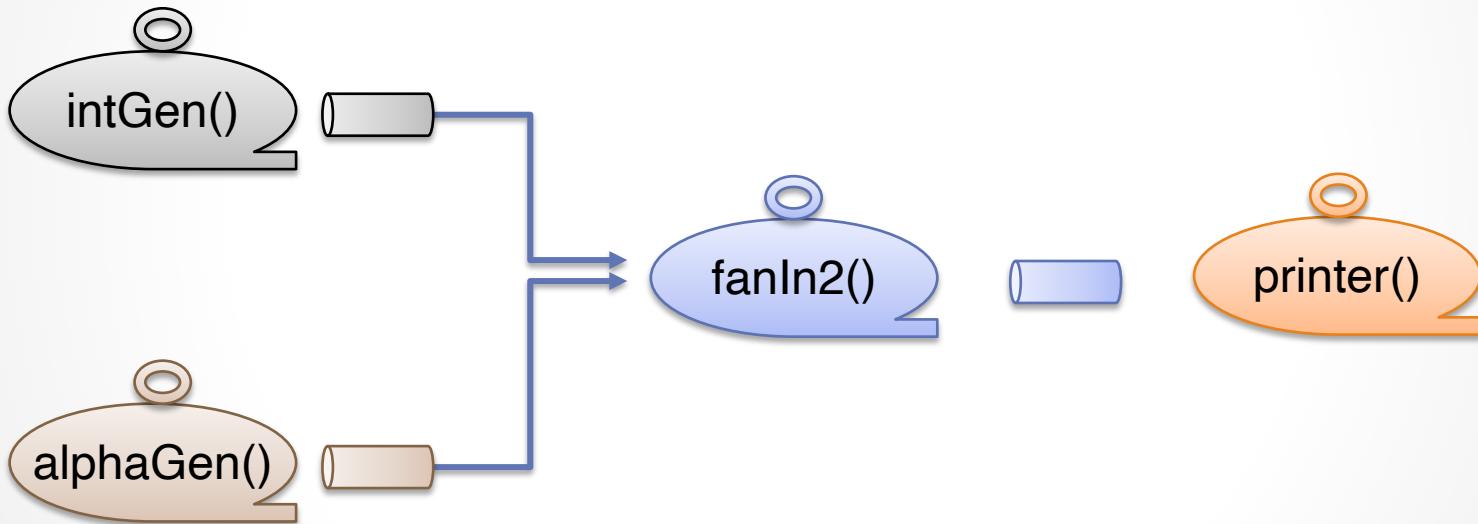
# Example 6



# Fan-In



# Example 7



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types)
- Advance Go Concurrency Patterns
  - <https://blog.golang.org/advanced-go-concurrency-patterns>
  - <https://www.youtube.com/watch?v=QDDwwewePbDtW>



# Review & Misc

Section 7 – Review



# Topics

- Receive-only Channels
- Send-only Channels
- Channel of channels
- Labs
  - Word Count (revisited)

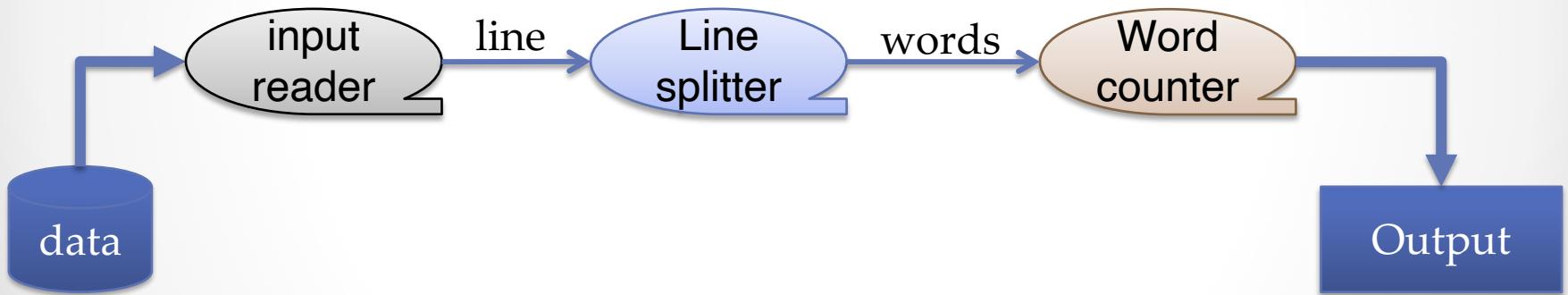


# Labs

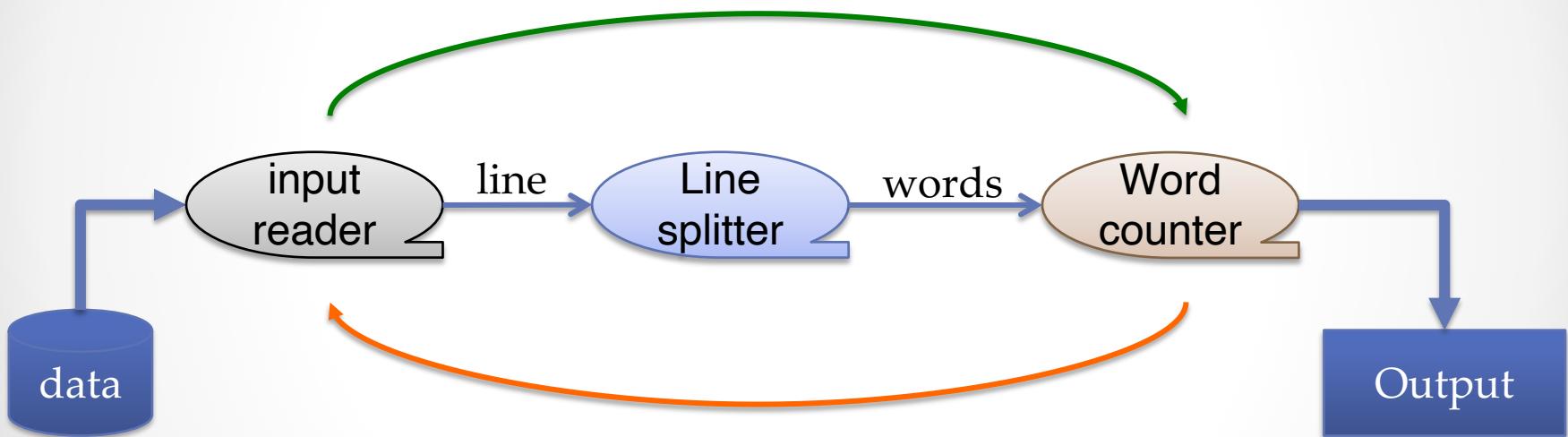
- Word Count
  - Write a Go Language program which will print out the number of occurrences of each word.
    - The program read text data from one or more files.
  - os.Args values



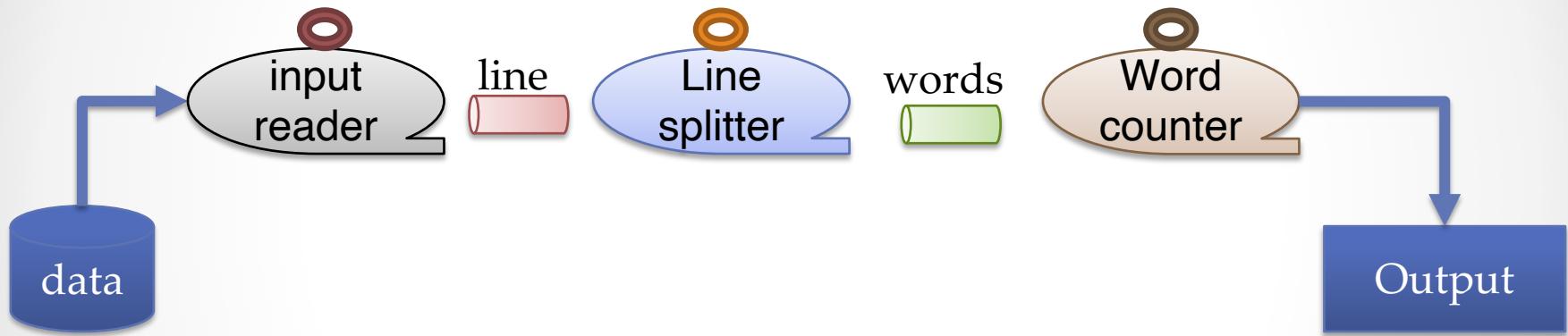
# Program Flow



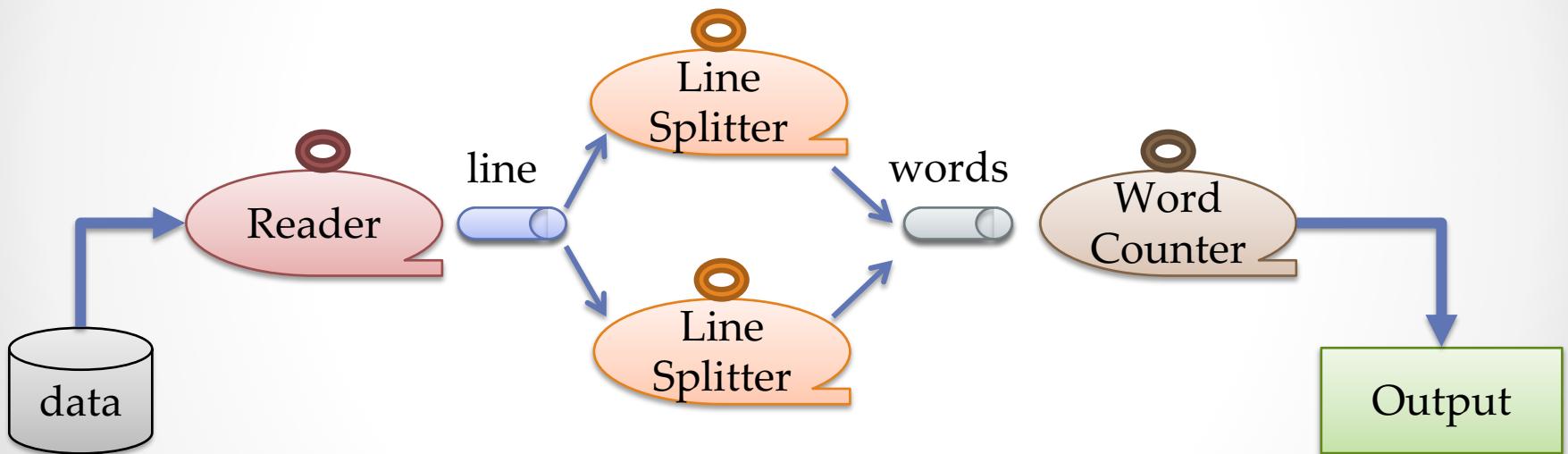
# Iterative Design



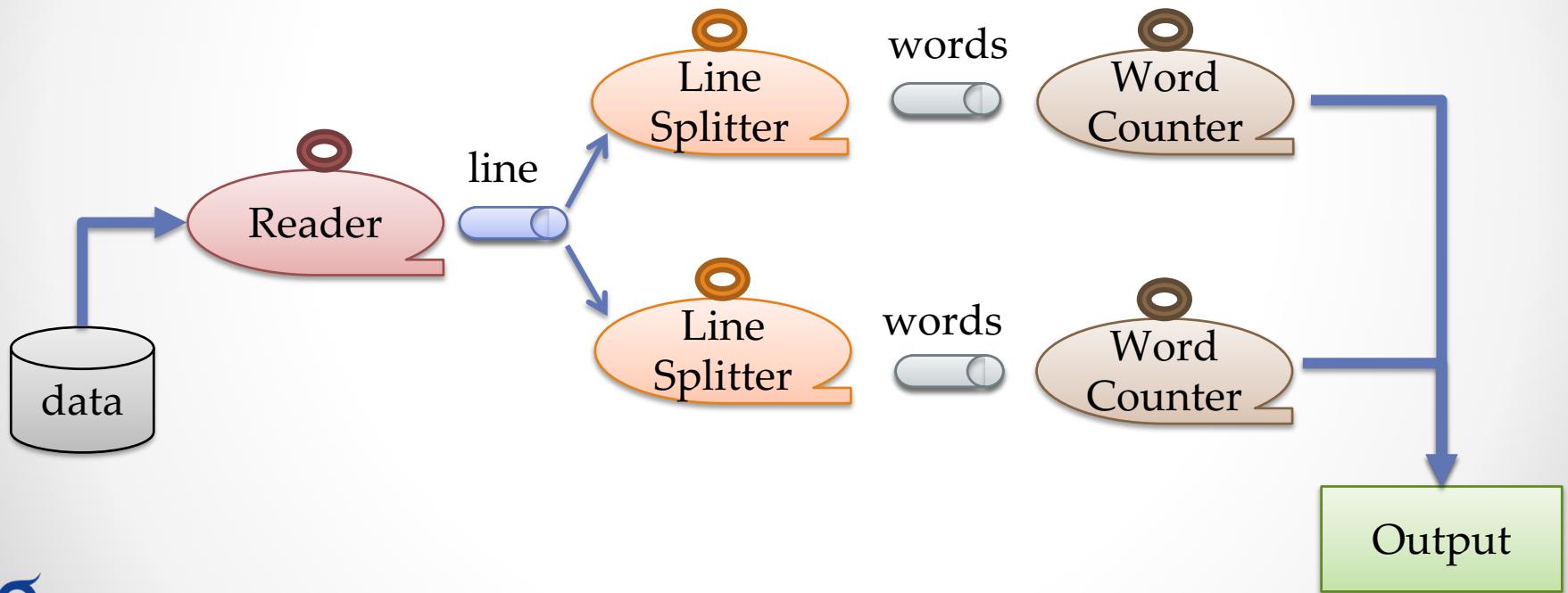
# Concurrent Design - 01



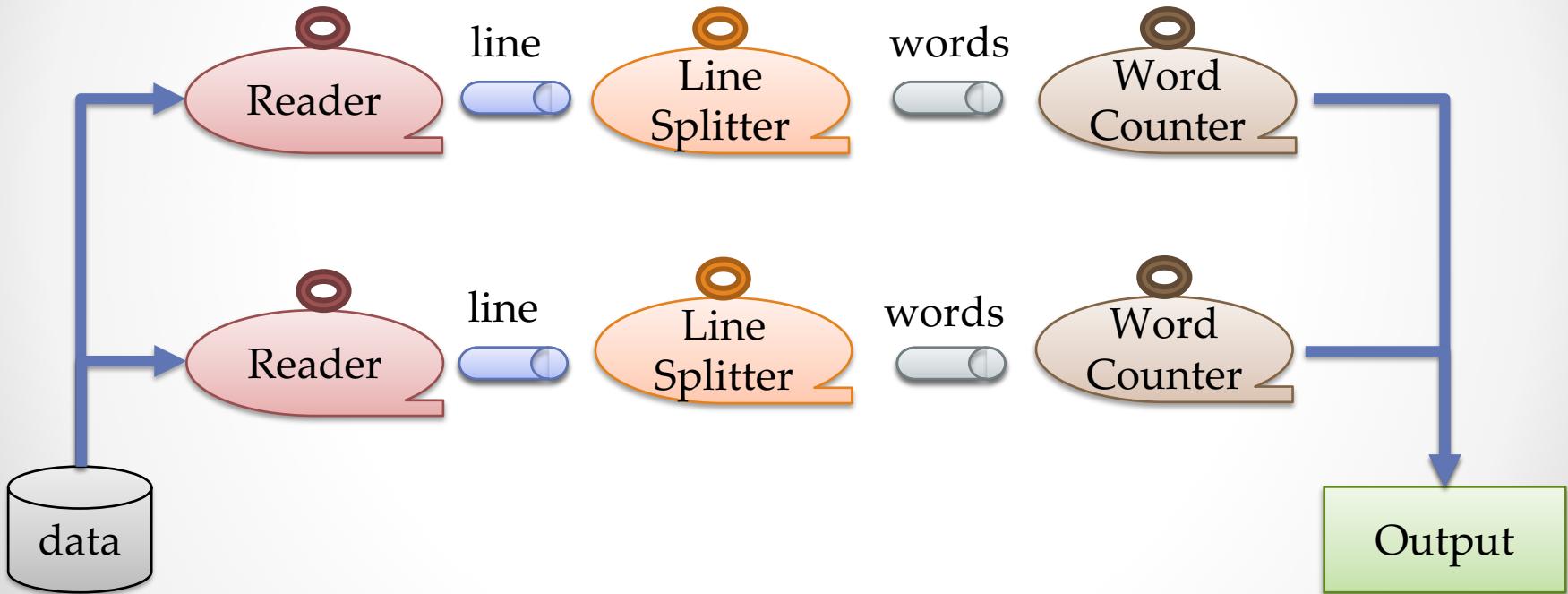
# Concurrent Design - 02



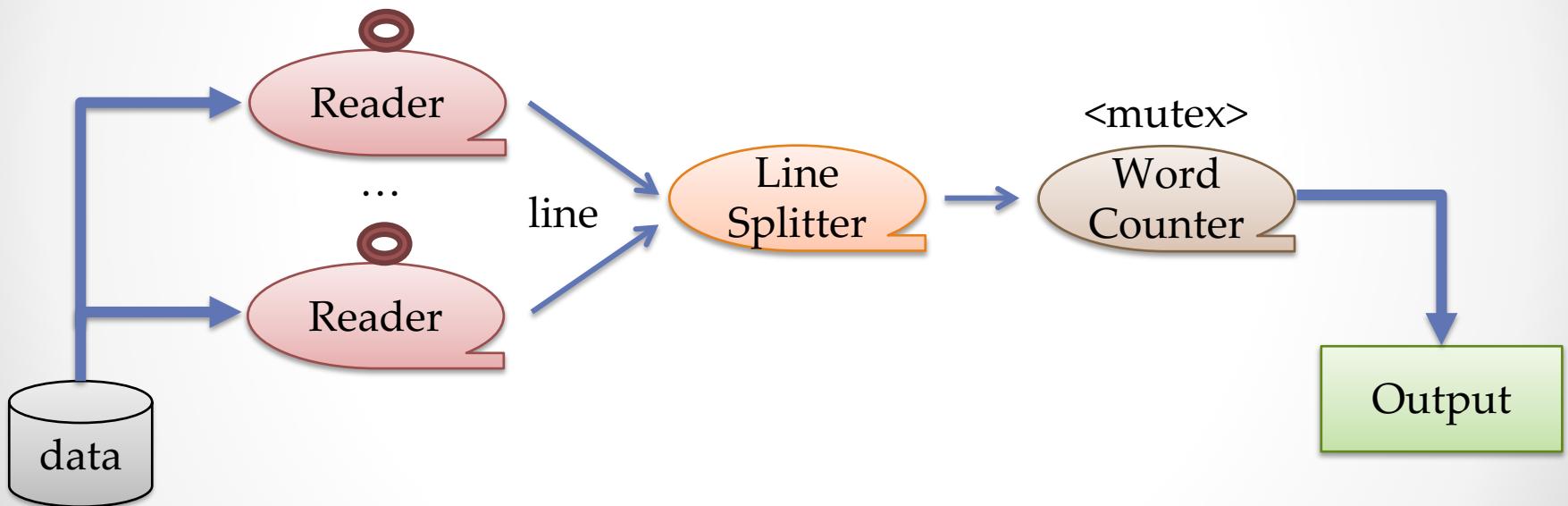
# Concurrent Design - 03



# Concurrent Design - 04



# Section 06 – Lab 02



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Channel\\_types](https://golang.org/ref/spec#Channel_types)
- Advance Go Concurrency Patterns
  - <https://blog.golang.org/advanced-go-concurrency-patterns>
  - <https://www.youtube.com/watch?v=QDDwwewePbDtW>



# Section 8: Pointers

- Basic
  - What is a Pointer?
  - Nil pointer value
  - Using pointers
    - How to creating pointers
    - Pointer deference
- Advance
  - Pointers and functions
  - Pointers to Pointers



# What is a Pointers

Section 8 – Lecture 1

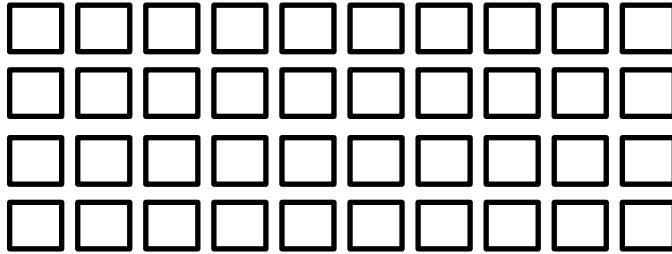


# Topics

- What is a ‘pointer’?
- Declaring a pointer
  - ‘nil’ pointer value



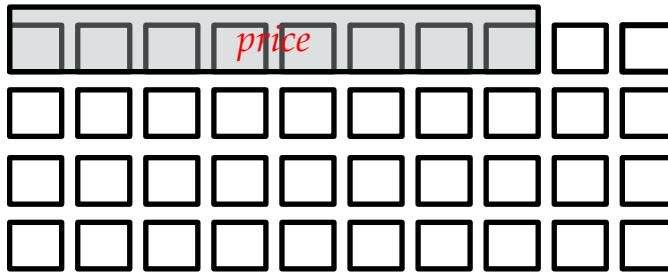
# Variables in Memory



$\square$  = 1 bytes = 8 bits



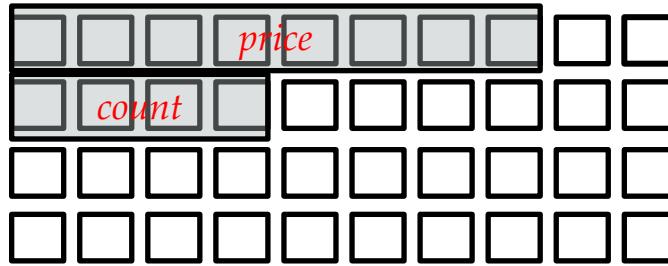
# Variables in Memory



| Variable Name | Type    | # bytes per type |
|---------------|---------|------------------|
| price         | float64 | 8                |



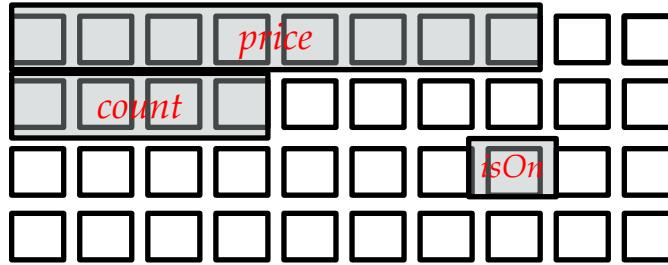
# Variables in Memory



| Variable Name | Type    | # bytes per type |
|---------------|---------|------------------|
| price         | float64 | 8                |
| count         | int32   | 4                |



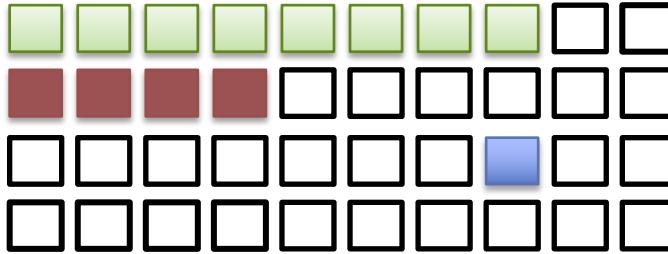
# Variables in Memory



| Variable Name | Type    | # bytes per type |
|---------------|---------|------------------|
| price         | float64 | 8                |
| count         | int32   | 4                |
| isOn          | bool    | 1                |



# Variables in Memory



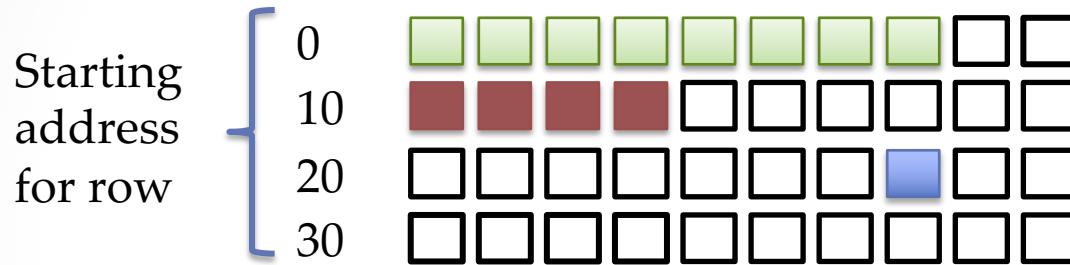
| Variable Name | Type    | # bytes per type |
|---------------|---------|------------------|
| price         | float64 | 8                |
| count         | Int32   | 4                |
| isOn          | bool    | 1                |

# Variables in Memory

|                                                                                   |                                                                                   |                                                                                   |                                                                                   |                                                                                   |                                                                                    |                                                                                     |                                                                                     |                                                                                     |                                                                                     |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |  |  |  |  |  |  |  |
| 0                                                                                 | 1                                                                                 | 2                                                                                 | 3                                                                                 | 4                                                                                 | 5                                                                                  | 6                                                                                   | 7                                                                                   | 8                                                                                   | 9                                                                                   |
|  |  |  |  |  |  |  |  |  |  |
| 10                                                                                | 11                                                                                | 12                                                                                | 13                                                                                | 14                                                                                | 15                                                                                 | 16                                                                                  | 17                                                                                  | 18                                                                                  | 19                                                                                  |
|  |  |  |  |  |  |  |  |  |  |
| 20                                                                                | 21                                                                                | 22                                                                                | 23                                                                                | 24                                                                                | 25                                                                                 | 26                                                                                  | 27                                                                                  | 28                                                                                  | 29                                                                                  |
|  |  |  |  |  |  |  |  |  |  |



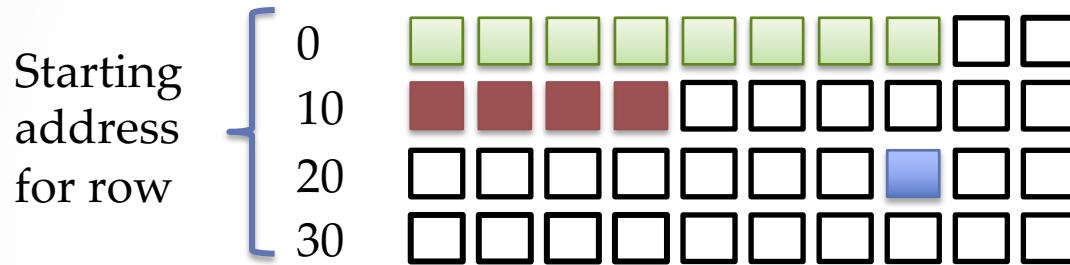
# Variables in Memory



| Variable Name | Type    | # bytes per type | Memory location (start, end) |
|---------------|---------|------------------|------------------------------|
| price         | float64 | 8                | 0, 7                         |
| count         | Int32   | 4                | 10, 13                       |
| isOn          | bool    | 1                | 27                           |



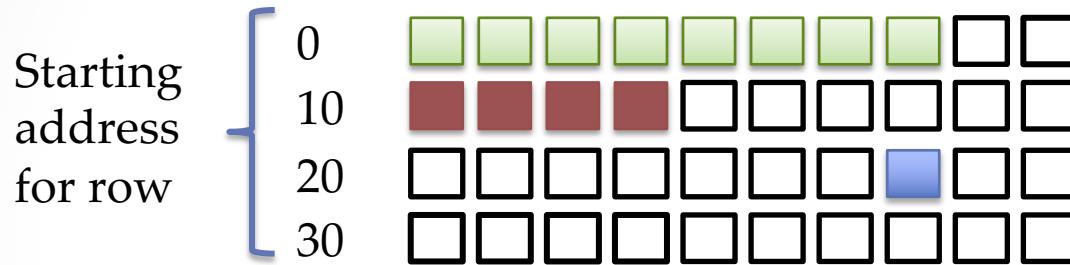
# Variables in Memory



| Variable Name | Type    | # bytes per type | Memory location (start) |
|---------------|---------|------------------|-------------------------|
| price         | float64 | 8                | 0                       |
| count         | Int32   | 4                | 10                      |
| isOn          | bool    | 1                | 27                      |



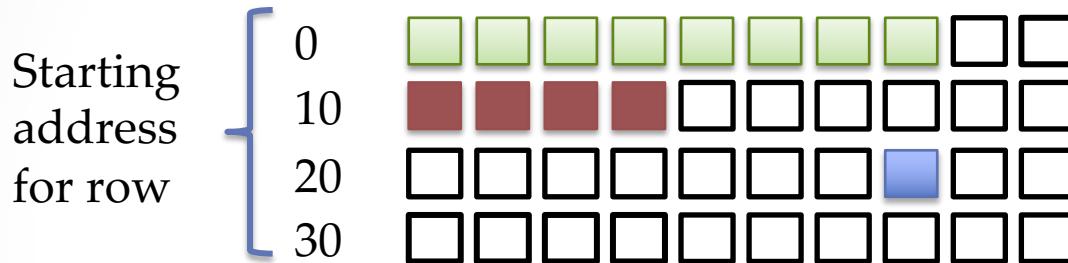
# Variables in Memory



| Variable Name | Type    | # bytes per type | Address in memory |
|---------------|---------|------------------|-------------------|
| price         | float64 | 8                | 0                 |
| count         | Int32   | 4                | 10                |
| isOn          | bool    | 1                | 27                |



# Pointers to variables in Memory



| Address in memory | Type of value in memory |    | Pointer  |
|-------------------|-------------------------|----|----------|
| 0                 | float64                 | => | *float64 |
| 10                | int32                   | => | *int32   |
| 27                | bool                    | => | *bool    |



# What is a Pointer?

- A pointer is an appropriate **type** that can hold the '*numeric value*', representing the **location** of a value of the **type** in memory.
  - In other words, a pointer of type '*int*', can hold the number value representing *the memory location* of an '*int*' value.
  - **NOTE:** We can only get *the memory location (address)*, of a non-const value stored in *memory*.



# Understanding Pointers

- We have a value **1737**

| Value |
|-------|
| 1737  |



# Understanding Pointers

- But we want to store our value **1737** in **memory**
- So we need a **variable**

| Variable | Value |
|----------|-------|
| count    | 1737  |



# Understanding Pointers

- Our **value**'s location in **memory**, i.e. it's **address**

| Variable | Value | Address |
|----------|-------|---------|
| count    | 1737  | 156     |

address = memory location



# Understanding Pointers

- Both **value** and **address** are just numbers
- Since an **address** in memory is just a number, we can store an address in yet another **variable**.

| Variable | Value | Address | Type    |
|----------|-------|---------|---------|
| count    | 1737  | 156     | int32   |
| pCount   | 156   | 500     | *int32  |
| ppCount  | 500   | 720     | **int32 |

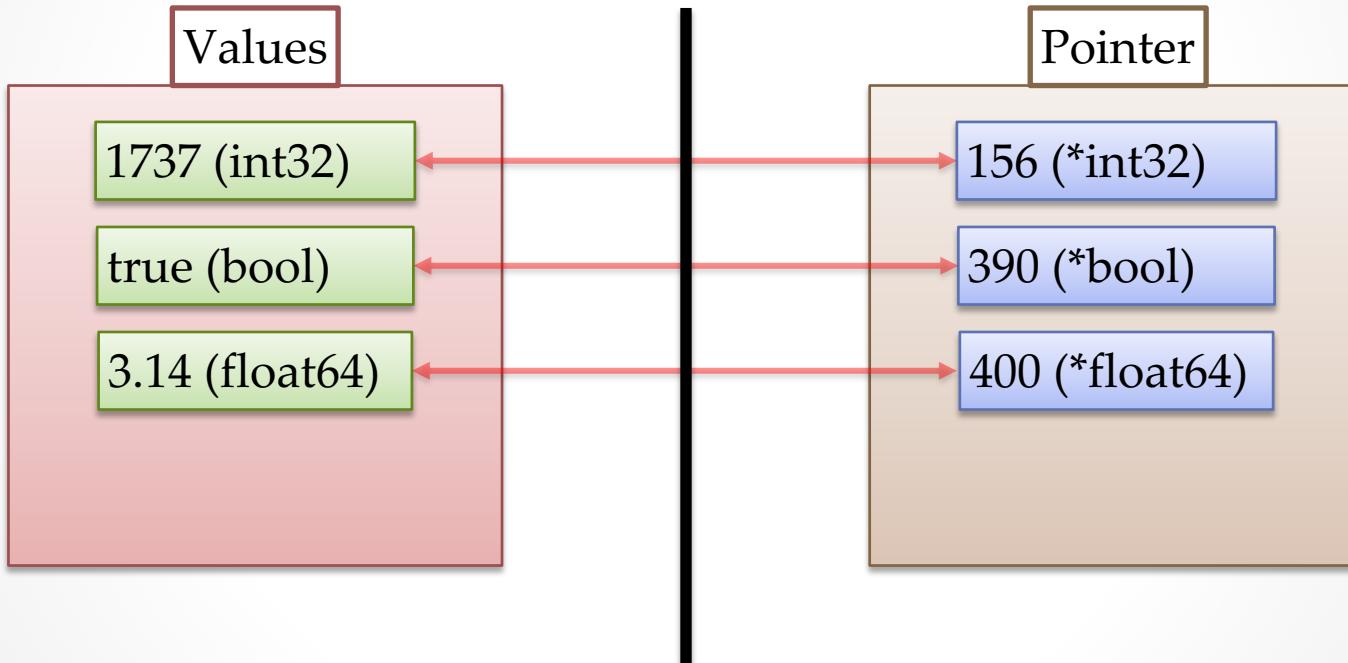


# Review

Section 8 – Lecture 1



# Two Worlds



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Pointer\\_types](https://golang.org/ref/spec#Pointer_types)



# Creating Pointers

Section 8 – Lecture 2



# Topics

- Creating a pointer value
  - Using '**addressof**' operator
  - Using '**new**' function



# Making a Pointer

- The Address-Of Operator
  - When applied to a variable or value, returns a pointer
    - **NOTE:** Not applicable
      - Const values
      - Numeric or String literals
  - The '**address of**' operator is '**&**'
    - Eg:

```
var count = 1737
```

```
var pCount = &count
```



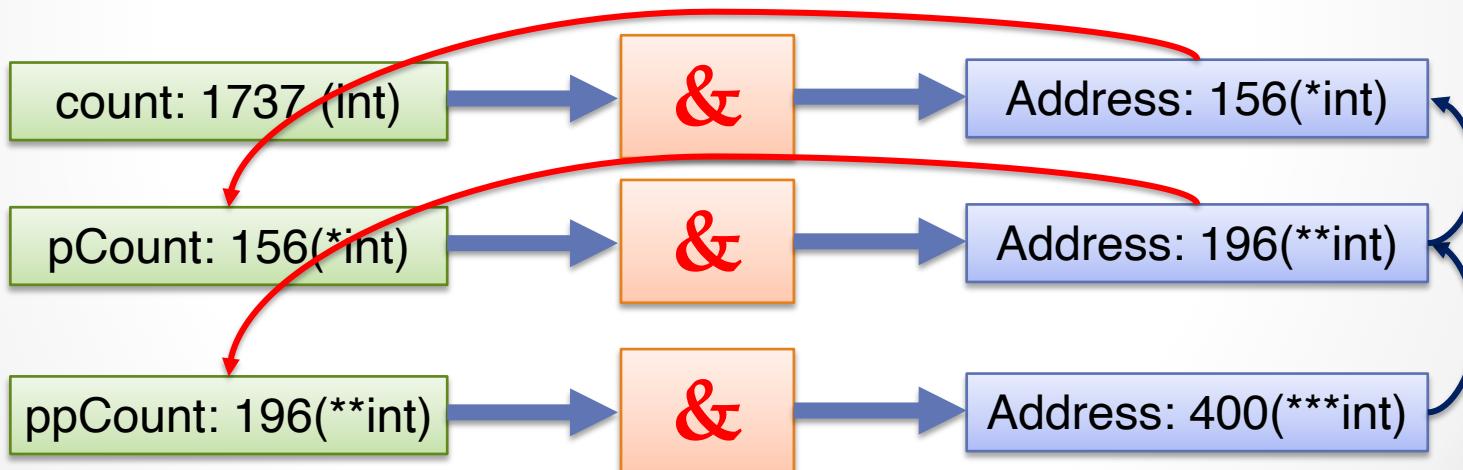
# 'Address Of' Operator : &

- Use the '**address of**' operator to get the **address** of a **variable**.

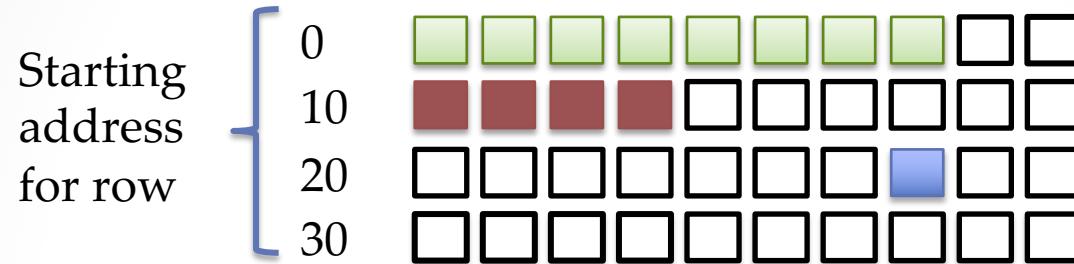


# 'Address Of' Operator : &

- Use the '**address of**' operator to get the **address** of **ANY variable** or **non-numeric/string literal value**.



# Pointers to *variables* in Memory

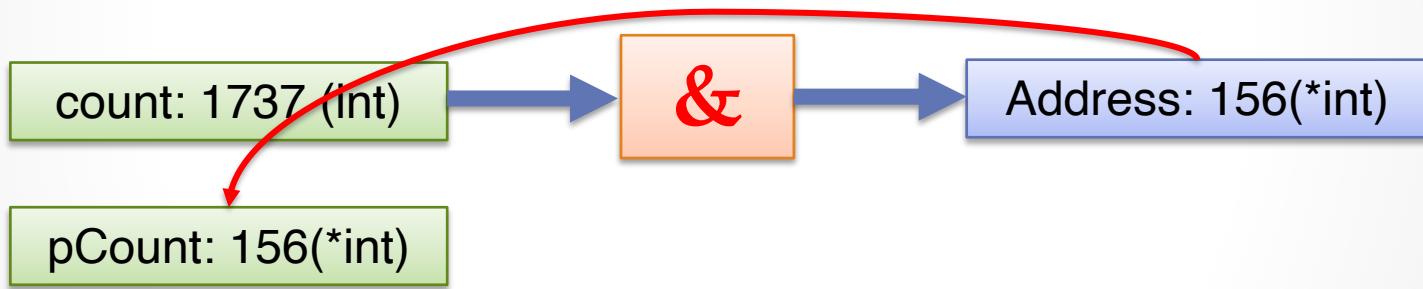


| Address in memory | Type of value in memory |    | Pointer  |
|-------------------|-------------------------|----|----------|
| 0                 | float64                 | => | *float64 |
| 10                | int32                   | => | *int32   |
| 27                | bool                    | => | *bool    |
| 30                | string                  | => | *string  |



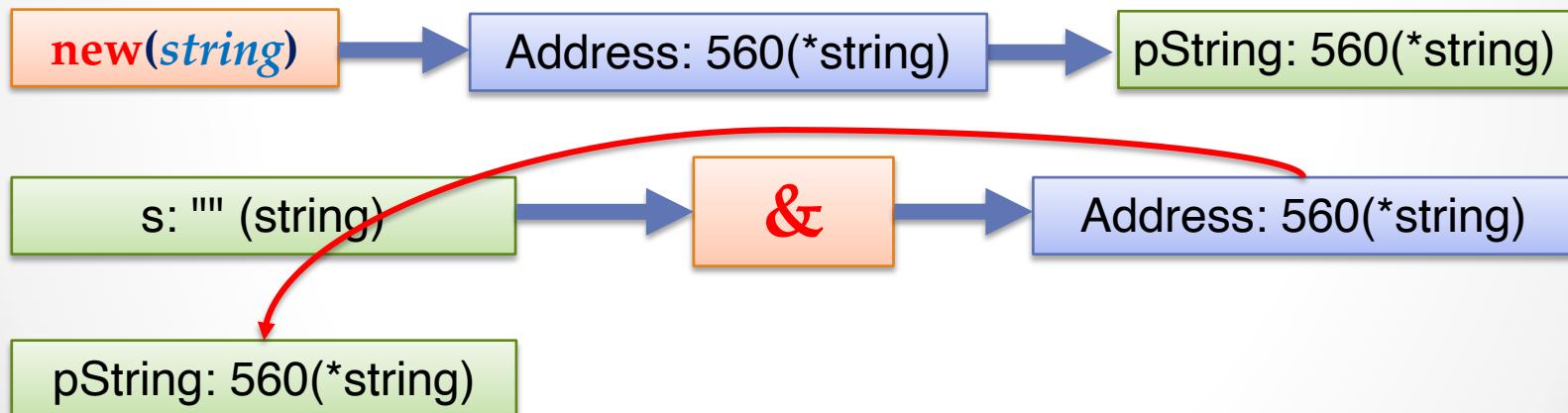
# 'new()' function

- Use the '**new()**' to get the **address** of memory allocated for the requested **type**.



# 'new()' function

- Use the '**new()**' to get the **address** of memory allocated for the requested **type**.

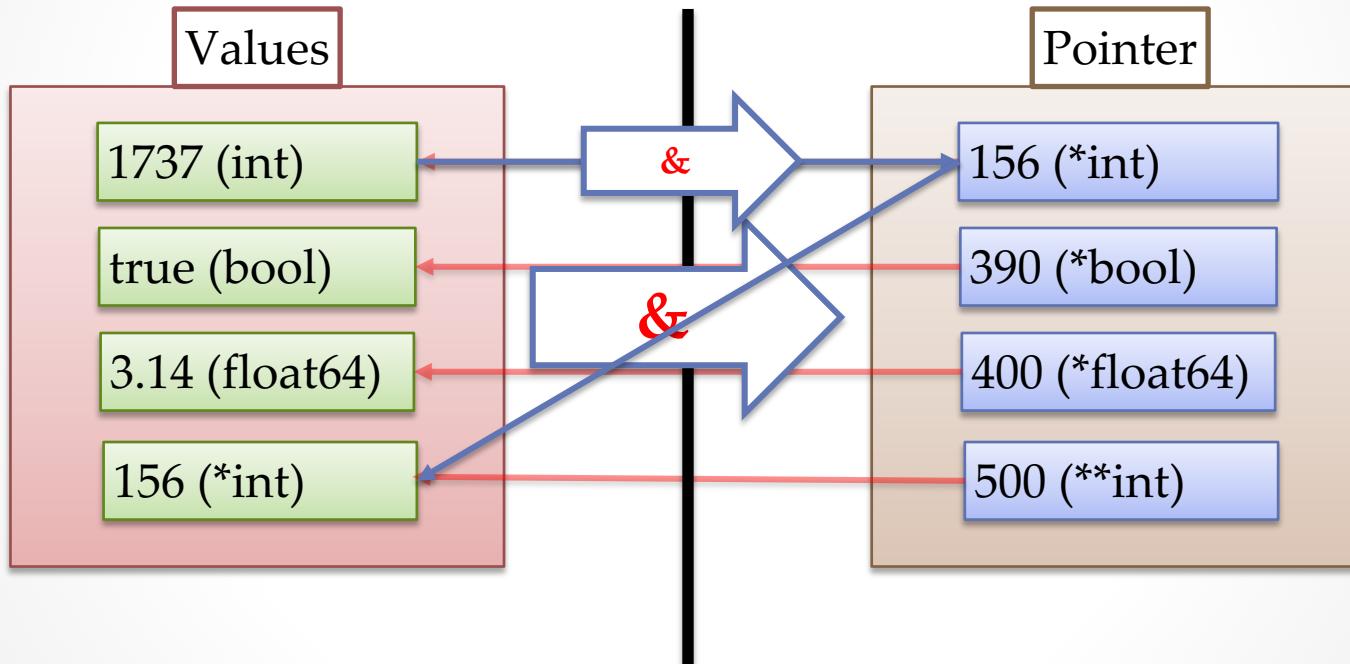


# Review

Section 8 – Lecture 2



# Two Worlds



# Using Pointers

Section 8 – Lecture 3



# Topics

- Dereferencing a pointer
  - Using '**dereference**' operator
  - Pitfalls when using pointers



# Using a Pointer

- Using the Dereference Operators
  - Dereference:
    - means to *obtain* the value *referenced* by a pointer
    - The '**dereference**' operator is '**\***'
      - Eg:

`var count = 1737`

`var pCount = &count`

`var count2 = *pCount`



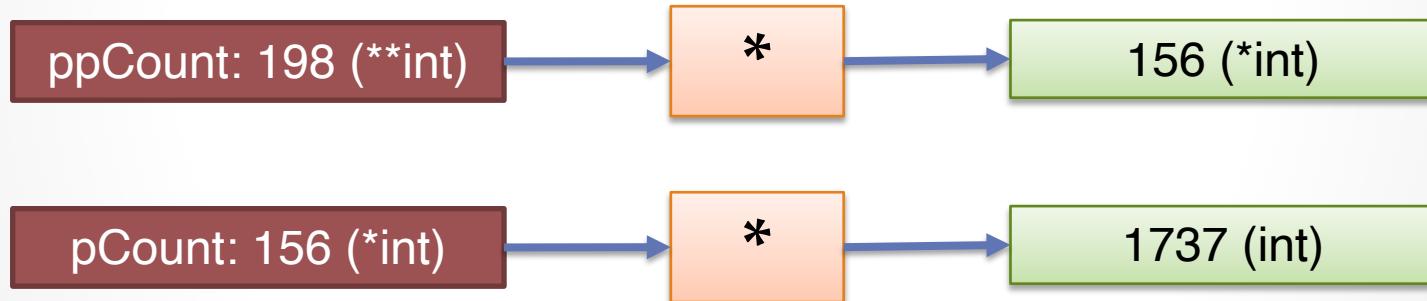
# Dereference Operator: \*

- Use the '**dereference**' operator to get the **value** stored at the memory location indicated by the **pointer**.
  - **NOTE:** A '**pointer**' is a **memory location** with a **type**.



# Dereference Operator: \*

- Use the '**dereference**' operator to get the **value** pointed to by **ANY pointer**.

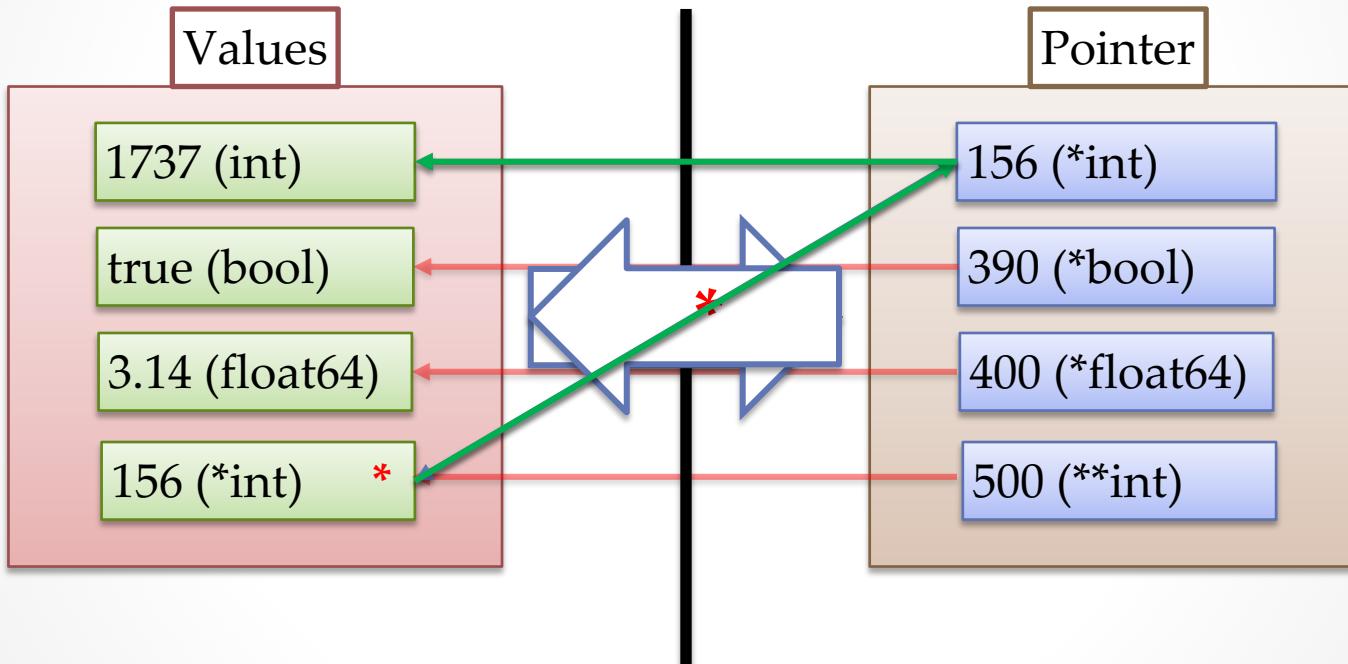


# Review

Section 8 – Lecture 3



# Two Worlds



# Pointers & Functions

Section 8 – Lecture 4



# Topics

- Copying Pointers
  - Passing Pointers to functions
  - Returning Pointers from functions
- The case for pointers



# Review

Section 8



# The case for pointers

- Pointers are cool
- Not as scary as they may first seem
- Provides indirection
  - Which *can be* useful in manipulating large memory objects
- Pitfalls
  - Always check return value of `new()`
  - Always check pointers for *nil* before **dereferencing**



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Pointer\\_types](https://golang.org/ref/spec#Pointer_types)
- Go Data Structures
  - <https://research.swtch.com/godata>



# Section 9: Interfaces

- Basic
  - What is an Interface?
  - Declaring a new interfaces
    - Interface vs. Struct
  - Implementing an interface
- Advance
  - Interfaces and functions
  - Method Sets
    - T vs \*T



# What is an Interface?

Section 9 – Lecture 1



# Topics

- What is a ‘interface’?
- Difference between ‘interface’ and ‘struct’?
- Declaring an interface



# What are 'interfaces' used for?

- An '*interface*' defines the **set** of **operations/methods** available.
  - A 'type' can implement or provide operations for several interfaces.
    - Example:

```
type Worker interface {
 Work() // worker can do work
}
```



# Interface vs. Struct

```
type Person struct {
 Name string
 Salary float32
 HireDate time.Time
}
```

```
type Worker interface {
 Work() // worker can do work
}
```

| Struct                                                                                 | Interface                                                                          |
|----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Defines the memory layout of a type                                                    | Defines which operations are available                                             |
| How should a value of this type be stored in memory, the memory footprint so to speak. | What actions can be taken in manipulating a value which implements this interface. |



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Interface\\_types](https://golang.org/ref/spec#Interface_types)



# Implementing an Interface

Section 9 – Lecture 2



# Topics

- Implementing an interface
  - Using type T
  - Using type \*T
- Why use T vs \*T



# Interface vs. Struct

```
type Person struct {
 Name string
 Salary float32
 HireDate time.Time
}
```

```
type Worker interface {
 Work() // worker can do work
}
```

| Struct                                                                                 | Interface                                                                          |
|----------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Defines the memory layout of a type                                                    | Defines which operations are available                                             |
| How should a value of this type be stored in memory, the memory footprint so to speak. | What actions can be taken in manipulating a value which implements this interface. |



# Review Method

- What is a method?
  - A **function** that takes a **receiver**

```
type Currency float64

func (c Currency) String() string{
 return fmt.Sprintf("$%.2f", float64(c))
}

var c Currency = 11.04
fmt.Println(c. String()) // $11.04
```

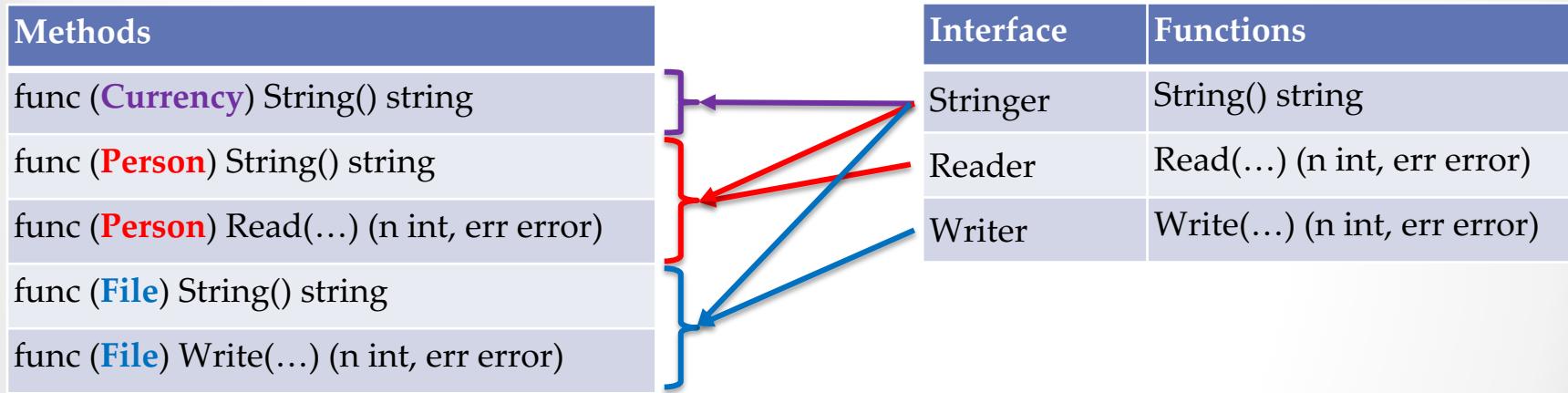


# Example of Interfaces

| Function                                | Interface |
|-----------------------------------------|-----------|
| func Quack()                            | Duck      |
| func Waddle()                           |           |
| func String() string                    | Stringer  |
| func Read(b []byte) (n int, err error)  | Reader    |
| func Write(b []byte) (n int, err error) | Writer    |



# Implementing Interfaces



IMPORTANT: A *type* implements an *interface*, if it provides function definitions for **ALL** methods of the interface.



# Implementing Interfaces

| Type     | Methods                                             | Implemented Interface    |
|----------|-----------------------------------------------------|--------------------------|
| Currency | func ( <b>Currency</b> ) String() string            | Stringer                 |
| Person   | func ( <b>Person</b> ) String() string              | Stringer, Reader         |
|          | func ( <b>Person</b> ) Read(...) (n int, err error) |                          |
| File     | func ( <b>File</b> ) String() string                | Stringer, Reader, Writer |
|          | func ( <b>File</b> ) Read(...) (n int, err error)   |                          |
|          | func ( <b>File</b> ) Write(...) (n int, err error)  |                          |



# Method Set

Section 9 – Lecture 3



# Topics

- What is a Method Set?
- Method Set T vs \*T



# What is a Method Set?

- The set of methods that can be selected for a value or interface variable.
  - If **v** is a *value* of type **T** or **\*T**, **v**'s method set is all the methods **callable** by a selector on **v**, such as **v.M()**.
  - If **i** is an *interface* variable of *interface* type **I**, **i**'s method set is all methods defined for interface **I**, such as **i.M()**.



# Language Spec: Method Sets

- A type may have a *method set* associated with it.
- The method set of an interface type is its interface.
- The method set of any type T consists of all methods declared with receiver type T.
- The method set of the corresponding pointer type \*T is the set of methods declared with receiver \*T or T
  - That is, it also contains the method set of T
- Any other type has an empty method set.



# Method Set for T and \*T

Type

```
type Person struct {}
```

Methods

T = Person

```
func (p Person) Name() string
```

\*T = \*Person

```
func (p *Person) SetAge(a uint8)
```

Method Sets

T = Person

```
func (p Person) Name() string
```

\*T = \*Person

```
func (p *Person) SetAge(a uint8)
```

```
func (p Person) Name() string
```



# Resource

- Lang Specification
  - [https://golang.org/ref/spec#Interface\\_types](https://golang.org/ref/spec#Interface_types)
- Method Set
  - [https://golang.org/ref/spec#Method\\_sets](https://golang.org/ref/spec#Method_sets)
- Assignability
  - <https://golang.org/ref/spec#Assignability>



# Interfaces and Functions

Section 9 – Lecture 4



# Topics

- *Interface* type as function parameter
- Function returning an *interface* type



# Interface Variable

- Given an *interface*:
  - We can create *interface variables*
  - An *interface variable* has two (2) hidden fields
    - **<value>**
      - holds a copy of the assigned *value*
    - **<type>**
      - Holds information about the *value's type*

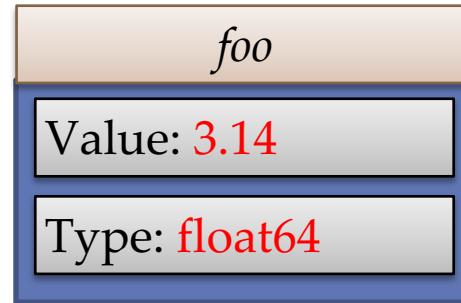
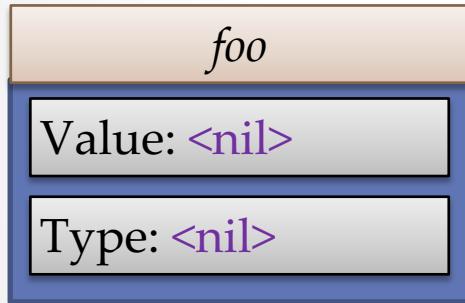


# Illustration: interface variable

`var foo interface{}`

`foo = 3.14`

`foo = &Person{}`



# Type Assertion

Section 9 – Lecture 5



# Topics

- Type assertion
  - Mechanism to reveal dynamic type
- Type Switching
  - Using type as switch case condition

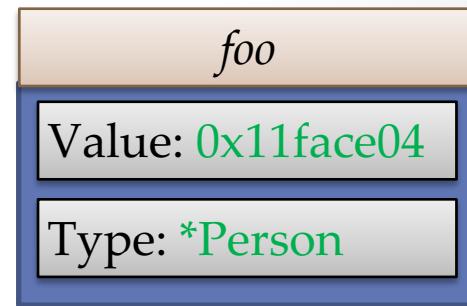
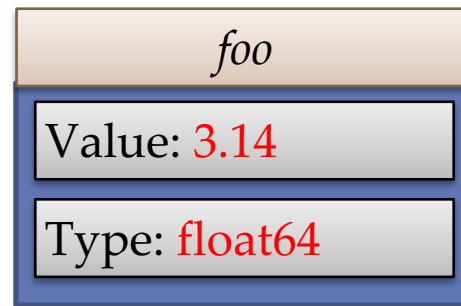
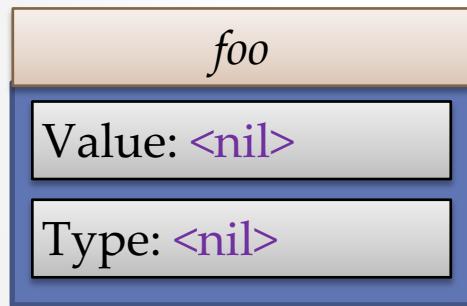


# Illustration: interface variable

`var foo interface{}`

`foo = 3.14`

`foo = &Person{}`

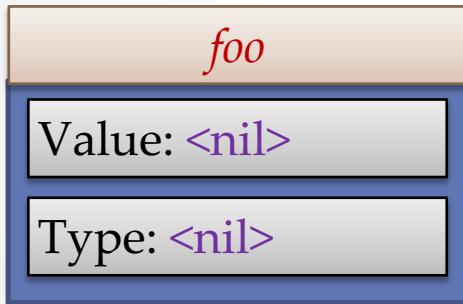


# Dynamic vs. static type

`var foo interface{}`

*foo* = 3.14

*foo* = &Person{}



1. What is '*foo*'s type?

`var goo = foo`

2. What is '*goo*' type?

`var hoo = foo`

3. What is '*hoo*' type?



# Dynamic vs. static type

```
var foo interface{}
```

static type?

*foo = 3.14*



*foo = &Person{}*



# Assignability

Section 9 – Lecture 6



# Topics

- When are types assignable
- When are Named vs. unnamed types
  - the same
  - different



# Review

Section 9



# Topics

- Interfaces vs structs
  - Interfaces define functions
  - Structs define memory layout
- Method Set
  - T vs \*T
- Implementing interfaces
- Type Assertion
  - x.(type)
- Assignability
  - When are Named vs. unnamed types



# Resource

- Lang Specification
  - <https://golang.org/ref/spec#Types>
- Type Identity
  - [https://golang.org/ref/spec#Type\\_identity](https://golang.org/ref/spec#Type_identity)
- Assignability
  - <https://golang.org/ref/spec#Assignability>



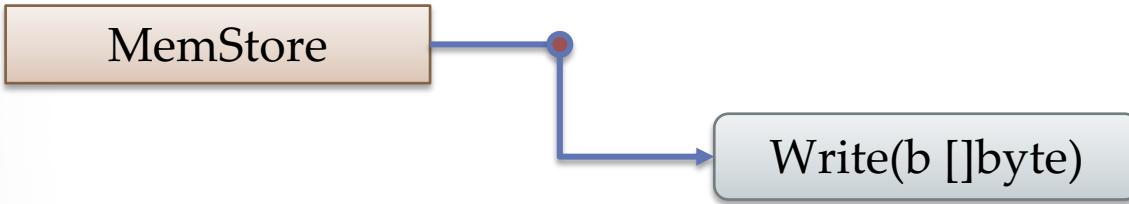
# Labs

Section 9



# In-Memory Store

- Writing data

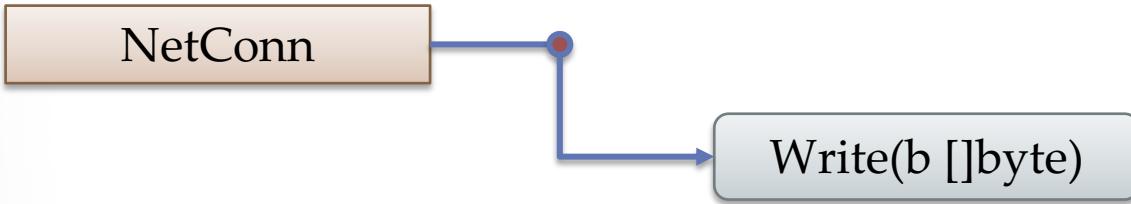


```
func (m *MemStore) Write(b []byte) (n int, err error) {
 ...
 // add bytes in 'b' to m.data
}
```



# Network Connection

- Sending data

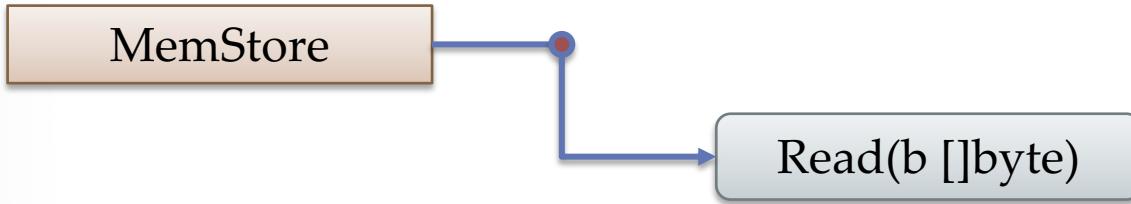


```
func (nc *NetConn) Write(b []byte) (n int, err error) {
 ...
 // copy bytes in 'b' to nc.device
}
```



# In-Memory Store

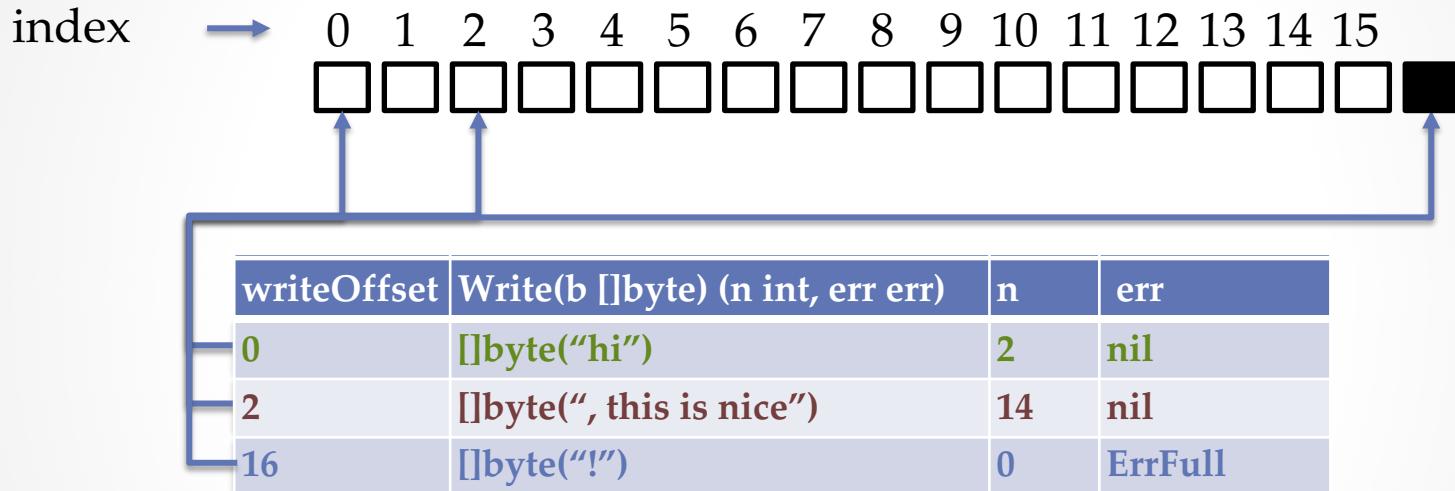
- Reading data



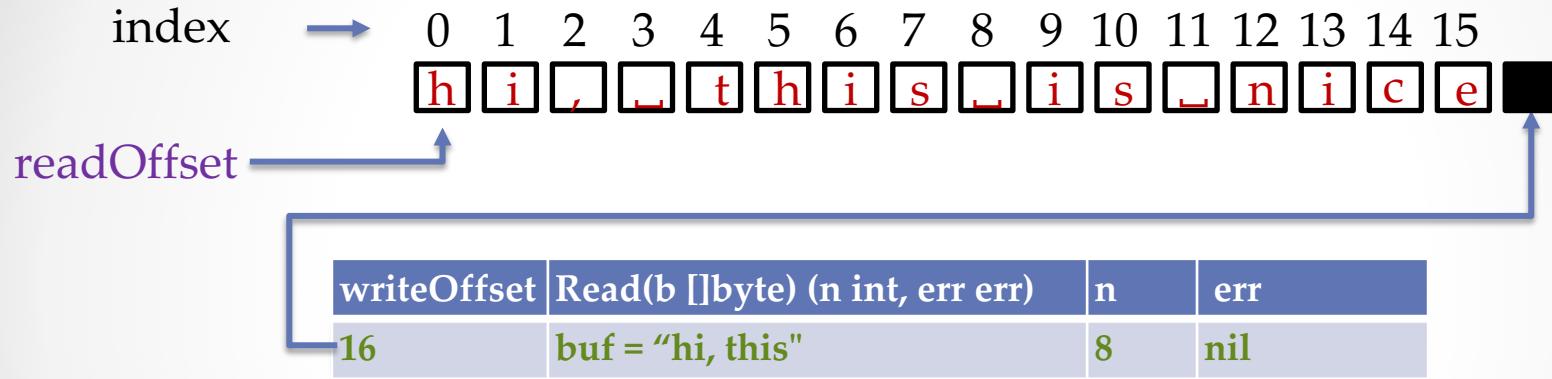
```
func (m *MemStore) Read(b []byte) (n int, err error) {
 ...
 // copy bytes from m.data to 'b'
}
```



# Memory Store Writes



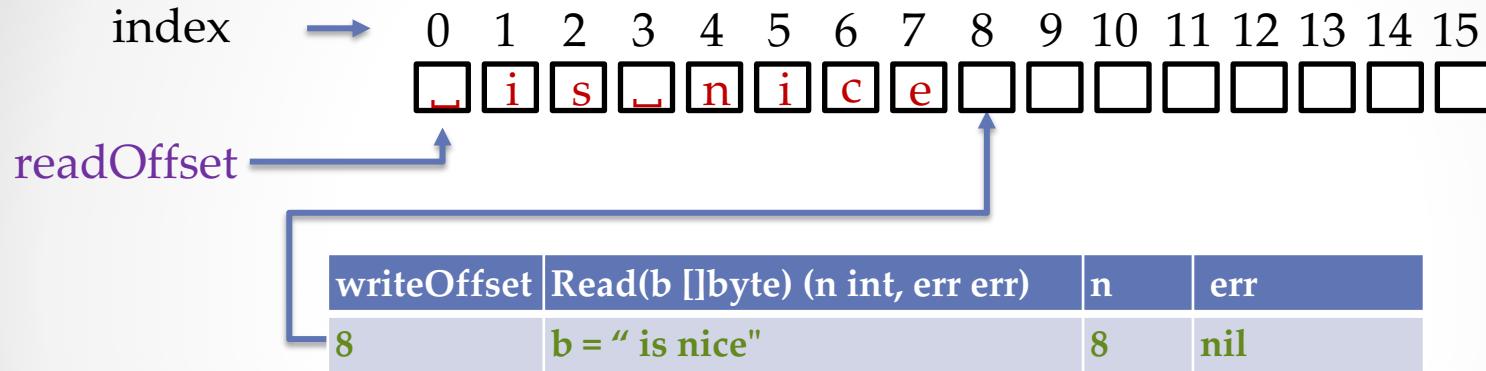
# Memory Store Reads



buf := make([]byte, 8, 8)



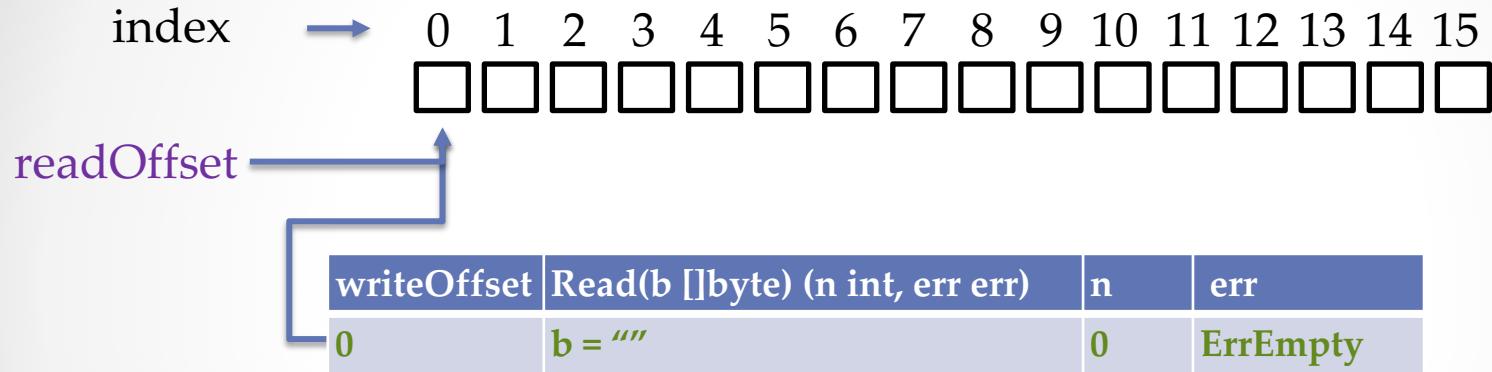
# Memory Store Reads



buf := make([]byte, 8, 8)



# Memory Store Reads



buf := make([]byte, 8, 8)



# Section 10: Standard Packages

- `io`
  - **Types:**
    - `Interfaces`
    - `Constants`
  - **Functions**
- `os`
  - **Types**
  - **Functions**
- `fmt`
  - **Types**
  - **Functions**



# io.Writer and io.Reader

Section 10 – Lecture 1



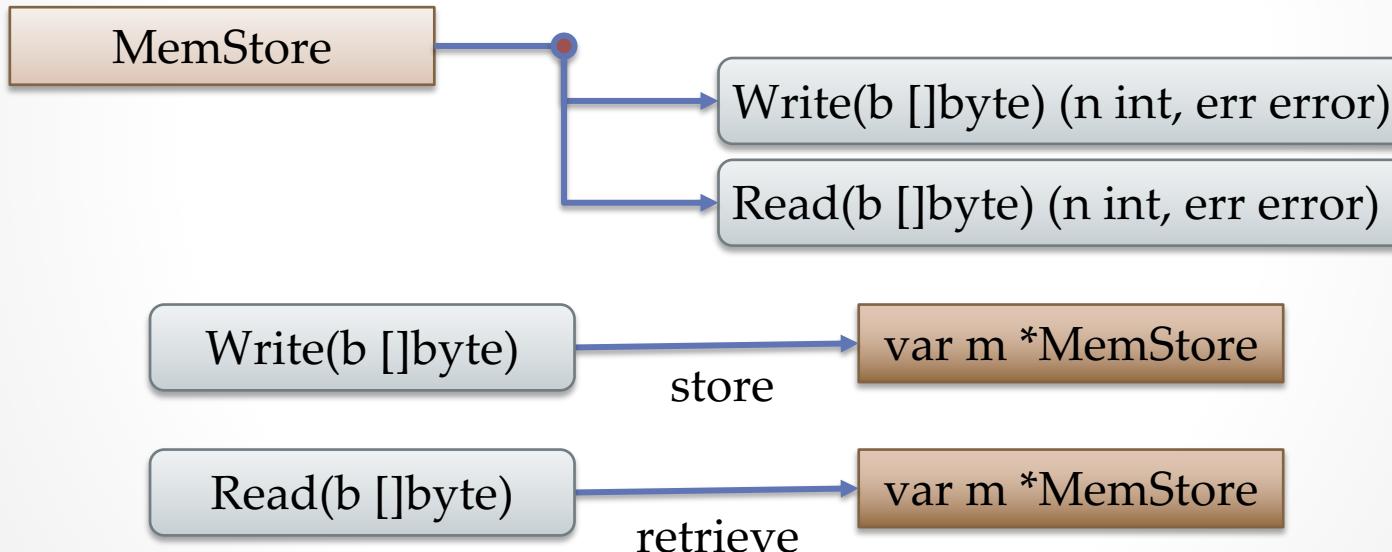
# Topics

- The io.Writer interface
- The io.Reader interface
- Implementing io.Writer & io.Reader
  - io errors
    - io.EOF



# Memory Store

- Writing and reading data



# Resource

- Package Documentation
  - <https://golang.org/pkg/io>
  - <https://golang.org/pkg/os>
  - <https://golang.org/pkg/fmt>



# io Functions

Section 10 – Lecture 2



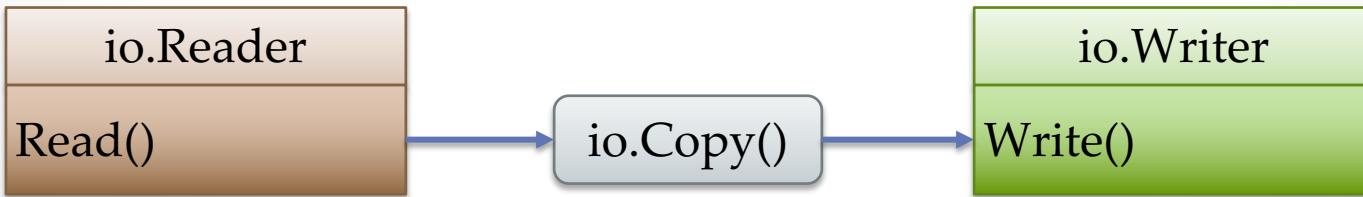
# Topics

- Convenient functions
  - `WriteString()`
  - `Copy()`
  - `Pipe()`



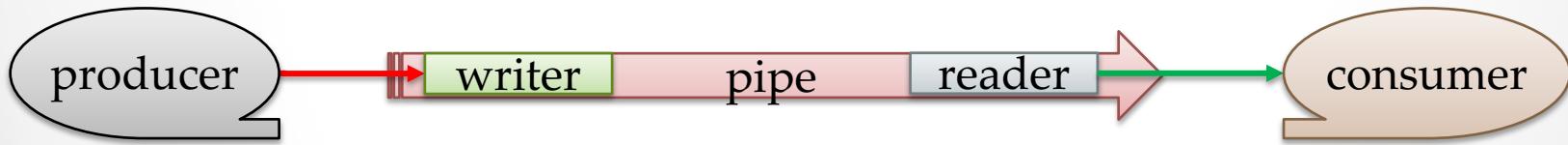
# io.Copy()

- Copy bytes from io.Reader value to io.Writer value
  - Error on read or write stops copy
    - io.EOF on io.Reader value is not consider an error



# io.Pipe()

- Creates an ‘in-memory’ reader and writer
  - Returns connected ‘reader’ and ‘writer’



# Resource

- Package Documentation
  - <https://golang.org/pkg/io>
  - <https://golang.org/pkg/os>
  - <https://golang.org/pkg/fmt>



# os.File

Section 10 – Lecture 3



# Topics

- OS Package types and functions
  - File I/O (input/output) related
    - Creating files
    - Writing data to files
    - Opening files
    - Reading data from files
  - General OS package offerings



# Desired File Operations

- What are some appropriate file operations?
  - Create one if it does not exist
  - Open an existing file
  - Store data, or write
  - Retrieve data, or read
  - Close for to signal no more updates
  - Rename
  - Delete
  - File permission
  - Etc.

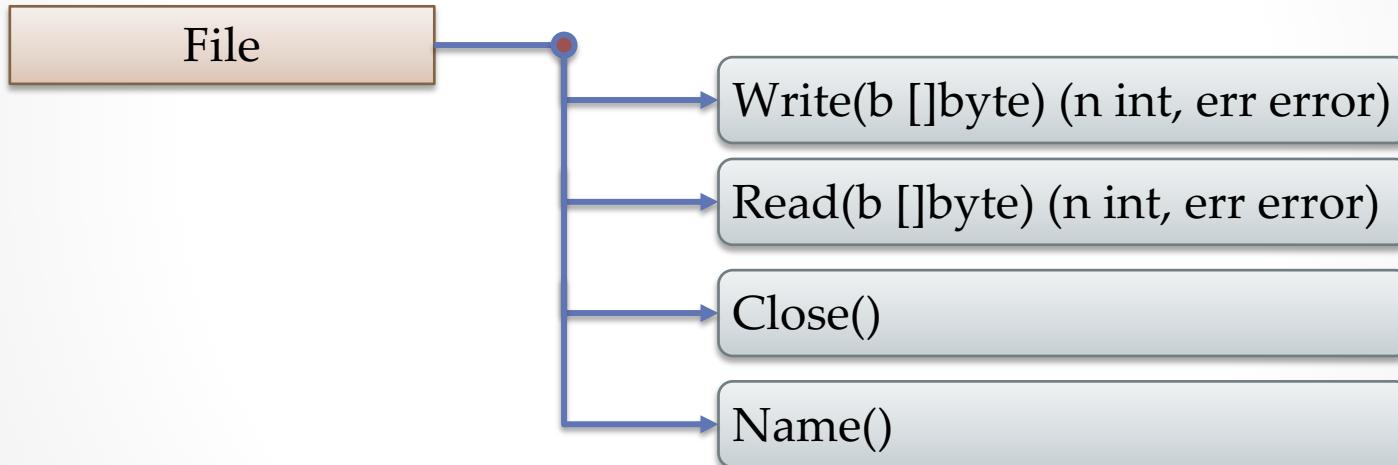


# Os vs. File Operations

- OS responsibilities
  - File creation
  - Open an existing file
  - Rename
  - Delete
  - Etc.
- File operations
  - Write
  - Read
  - Close
  - File permission
  - Etc.



# File Operations



# Review

- Other useful things to do with `os.File`:
  - Size, Mode, Permission, Check if exist, etc.
  - Get directory listing
- Many other things in `os` package
  - Args
  - User info
  - Host info
  - Process creation
  - Etc.



# Formatted Output

Section 10 – Lecture 4



# Topics

- Formatted output
  - Write to standard out (***stdout***)
    - *fmt.Print()*, *fmt.Println()*, and *fmt.Sprintf()*
  - Write to string, return string
    - *fmt.Sprint()*, *fmt.Sprintln()*, and *fmt.Sprintf()*
  - Write to specified ***file (io.Writer)***
    - *fmt.Fprint()*, *fmt.Fprintln()*, and *fmt.Fprintf()*



# Resource

- Package Documentation
  - <https://golang.org/pkg/io>
  - <https://golang.org/pkg/os>
  - <https://golang.org/pkg/fmt>



# Formatted Input

Section 10 – Lecture 5



# Topics

- Formatted input
  - Read from standard in (***stdin***)
    - *fmt.Scan()*, *fmt.Scanln()*, and *fmt.Scanf()*
  - Read from string
    - *fmt.Sscanf()*, *fmt.Sscanfln()*, and *fmt.Sscanff()*
  - Read from specified ***file (io.Reader)***
    - *fmt.Fscan()*, *fmt.Fscanfln()*, and *fmt.Fscanff()*



# Resource

- Package Documentation
  - <https://golang.org/pkg/io>
  - <https://golang.org/pkg/os>
  - <https://golang.org/pkg/fmt>



# Review

Section 10 – Lecture 6



# Topics

- Pitfalls of using and implementing io.Reader
  - Remember io errors
    - Handling Read errors:
      - Especially io.EOF
        - **Consumed the bytes before failing on error**
  - io.Reader/io.Writer
    - is **\*not\*** appropriate for every type
  - Formatted output
    - Don't forget fmt.FprintX() and fmt.SprintX()
      - Use fmt.FprintfX() for formatted output to io.Writer implementations
  - There are many other standard packages
  - Miscellaneous



# Resource

- Package Documentation
  - <https://golang.org/pkg/io>
  - <https://golang.org/pkg/os>
  - <https://golang.org/pkg/fmt>



# Labs

Section 10



# Command Runner

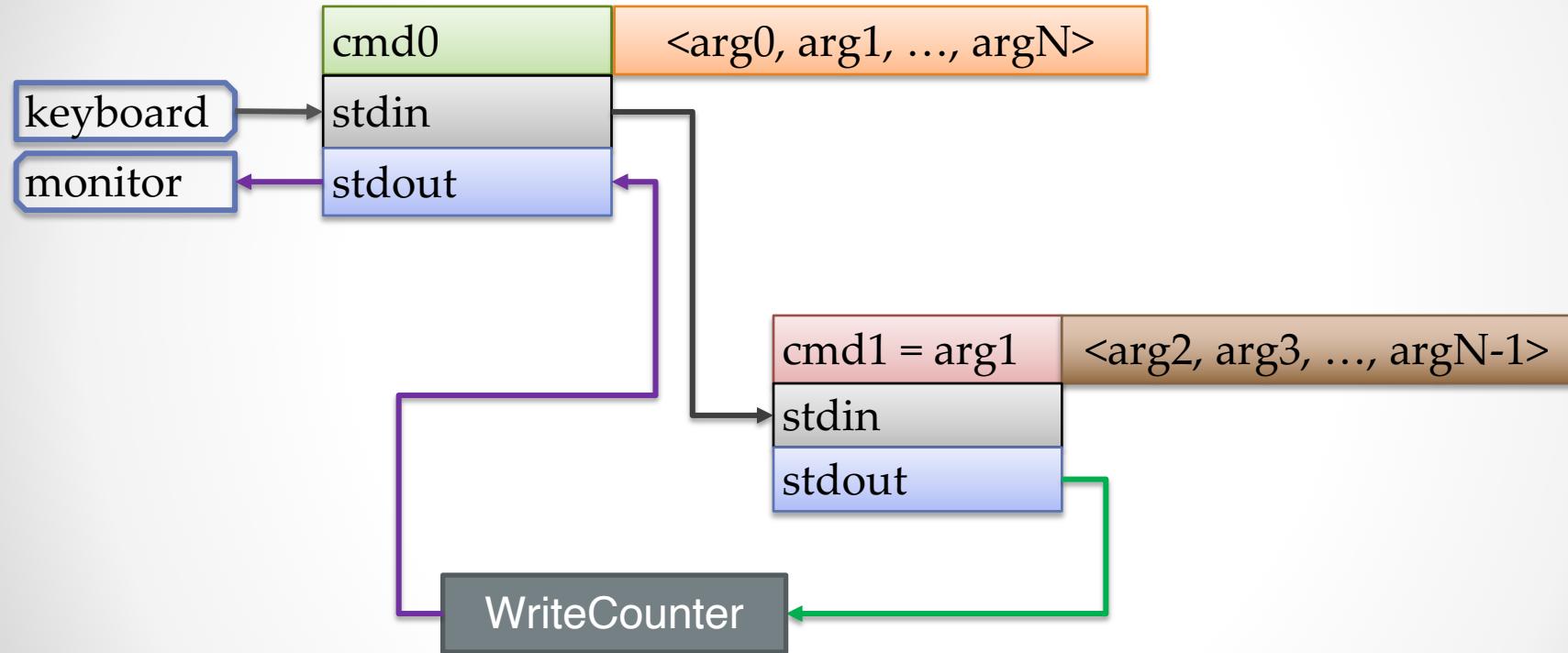
Section 10 – Lab 1



# Topics

- Implement a ‘command runner’ program:
  - Requirements
    - Run a given command and its argument
    - Monitor the output the command such that:
      - Total bytes produced it counted
      - Number of times it wrote output





# Section 11: net Package

- `net/http`
  - `HTTP Client`
    - `Sending Requests`
  - `HTTP Server`
    - `Handling Requests`
- `net`
  - `TCP/IP Client`
  - `TCP/IP Server`



# Simple HTTP Server & Client

Section 11 – Lecture 1

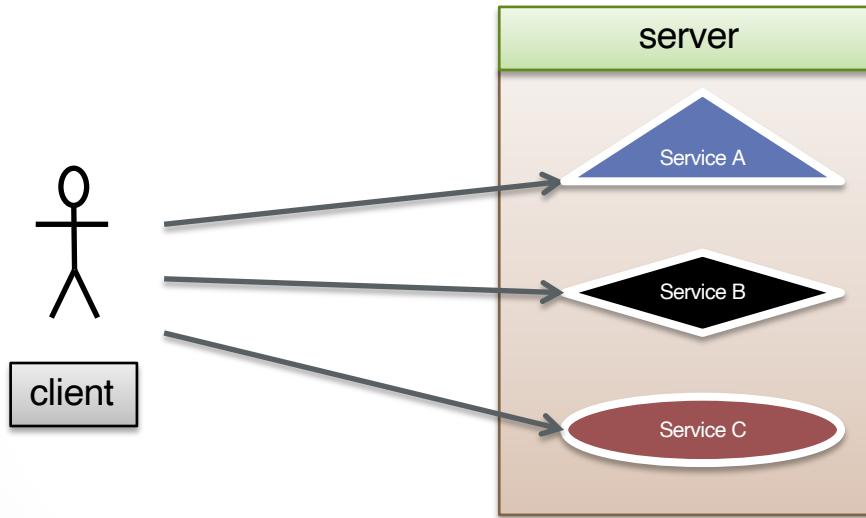


# Topics

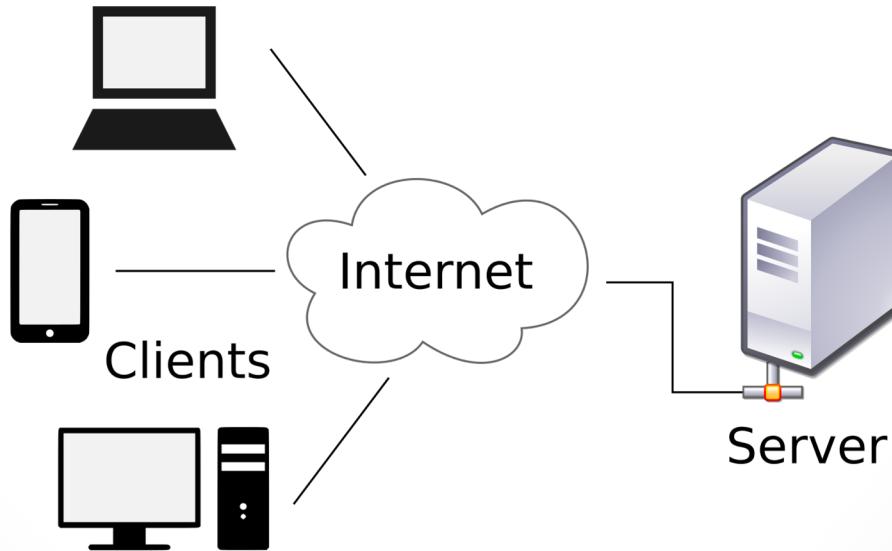
- Creating an HTTP Client
  - GET request
- Creating an HTTP Server
  - Handling requests



# Server & Client Interaction



# Computer Server & Client



# HTTP: Communication

- HTTP – Hypertext Transport Protocol
- Protocol – The rules for communication
  - The two parties communicating are:
    - Client : Usually a web browser (or HTTP client)
    - Server: The web server (or HTTP server)



# Why HTTP?

- HTTP '**fairly**' is simple
  - Made up of REQUEST/RESPONSE
    - Client makes REQUEST
    - Server sends RESPONSE to client REQUEST
      - NOTE: The sever '**never**' initiates an exchange



# Resource

- Package Documentation
  - <https://golang.org/pkg/net/http>



# HTTP Server v2

Section 11 – Lecture 2



# Topics

- Sending dynamic data
- Handling multiple
- Retrieving client info
- Reading request body



# http.Handler

Section 11 – Lecture 3



# Topics

- Registering handlers
  - `http.HandleFunc()` vs `http.Handle()`
- Implementing `http.Handler` interface



# Generic TCP/IP Server & Client

Section 11 – Lecture 4



# Topics

- Generic TCP/IP Client
  - Making connection using `net.Dial()`
- Generic TCP/IP Server
  - Listening for connections using `net.Listen()`
  - Handling connections from `Listener.Accept()`



# Labs

Section 11



# Remote Command Runner

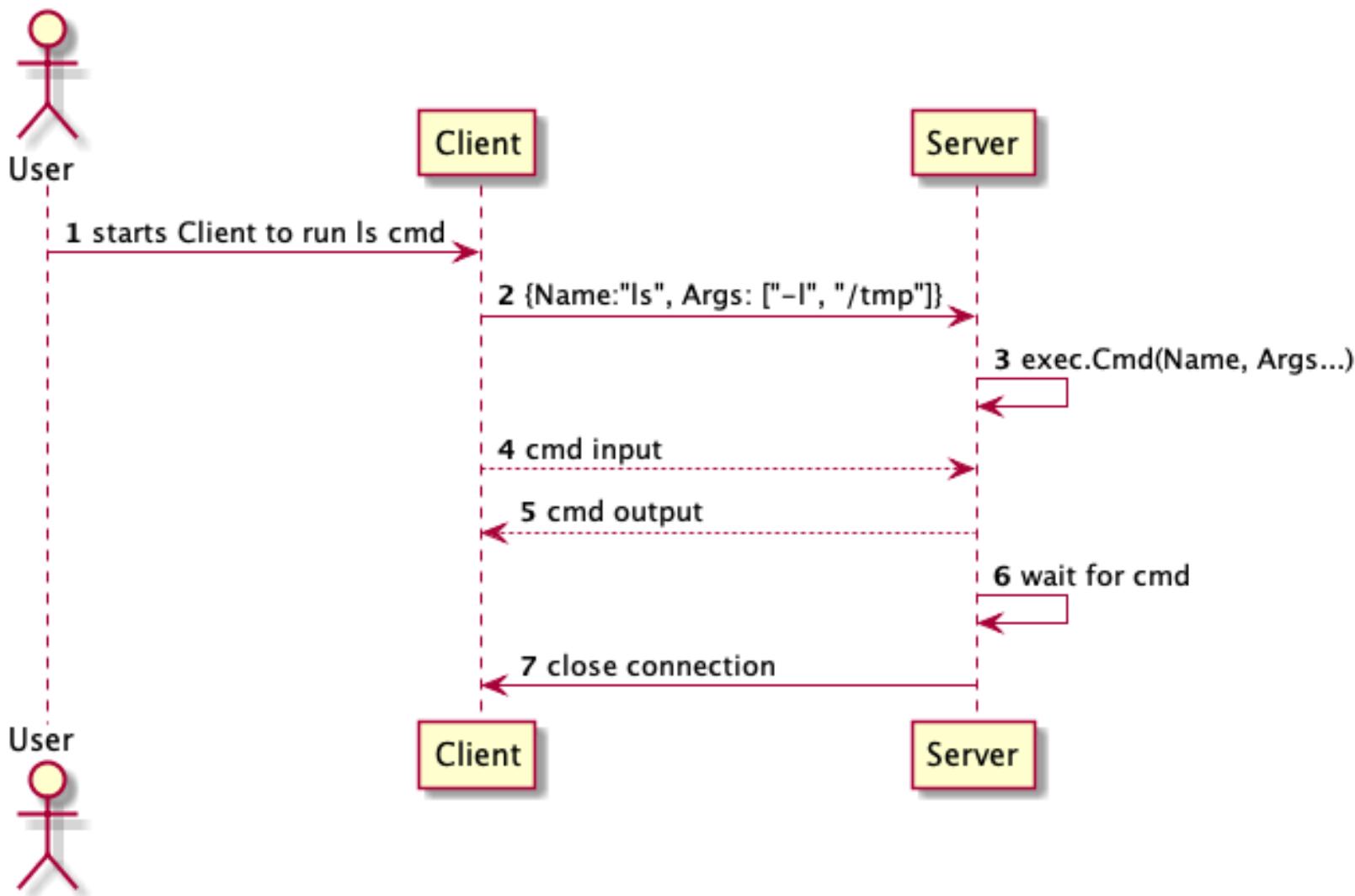
Section 11 – Lab 1



# Topics

- Implement a ‘remote command runner’ program:
  - Requirements
    - Server
      - Run a given command and its argument
      - Connections command stdin, stdour, and stderr to remote client
    - Client
      - Connects to server
      - Issue command to run
      - Monitor the output form the command such that:
        - Total bytes produced it counted
        - Count number of times output was written





# Course Review

GO Language for Tourist



# Course Outline

- Section 1: Installation and Setup
- Section 2: Basic Concepts
- Section 3: Arrays & Slices
- Section 4: Maps
- Section 5: Structs
- Section 6: Go-routines
- Section 7: Channels
- Section 8: Pointers
- Section 9: Interfaces
- Section 10: Standard I/O & File I/O
- Section 11: Networking



# Thank you!

