TODO

Return multiple value

Package

Package is collection of common source code files.

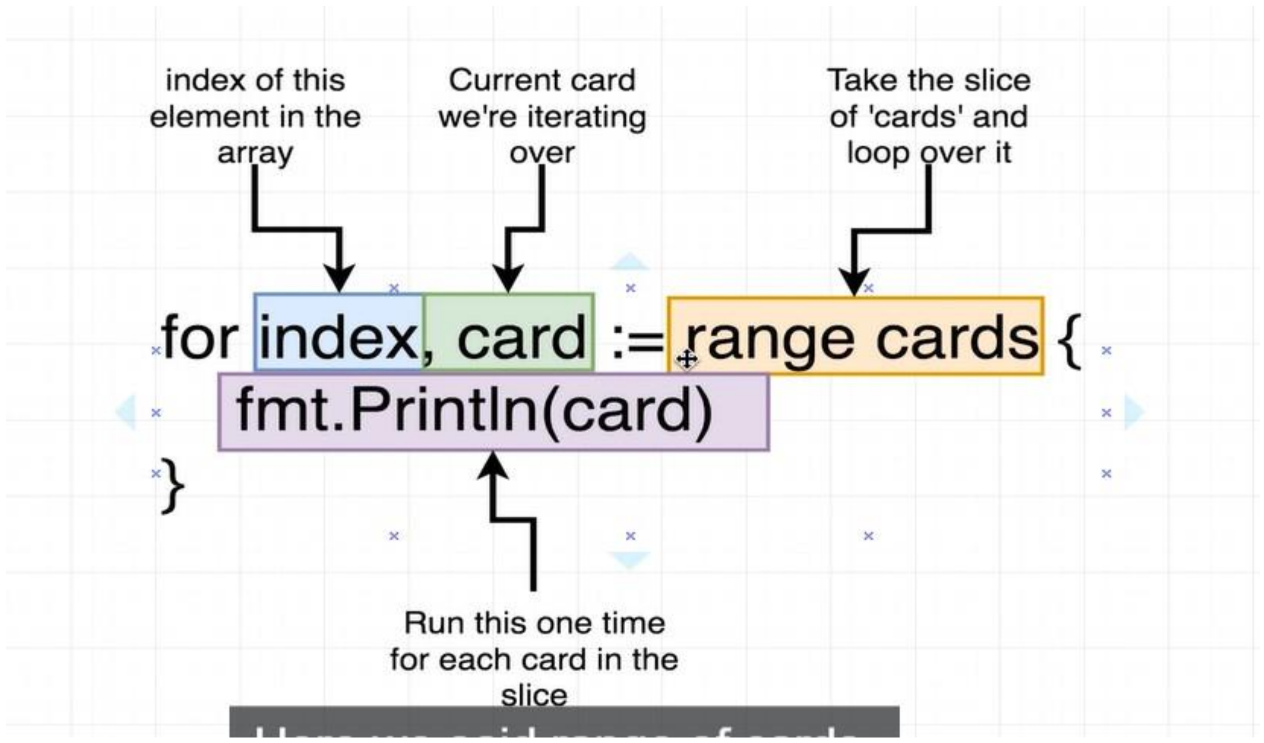
Package == project==workspace

"%+v"	Print struct value with its corresponding field .

Defer	delays the execution util function is over , it used file close, resource release it ensure that if there error, panic occurs it handle properly.
	It is LIFO (Last In First Out) .
	ex.
	`func foo() int {`
	`  defer fmt.Println("\n One")`
	`defer fmt.Println("\n Two")`
	`defer fmt.Println("\n Three")`
	`}`
	
	O/P => Three
	Two
	One

Panic	It is like ***throw*** in c++, after panic execution stop. panics are typically used for unrecoverable errors, so try to ovoid using panic.
	Catch exception like :
	`func foo() int {`
	`        defer fmt.Println("\n defer")`
	`        fmt.Println("inside foo")`
	`        panic(" foo throw")`
	`        fmt.Println("After foo")`
	`        return 10`
	`}`
	`func main() {`
	`        defer func() {`
	`                ret := recover()`
	`                if ret != nil {`
	`                        fmt.Println(" Recover ", ret)`
	`                }`
	`        }()`
	
	`        fmt.Printf("%d", foo())`
	`        fmt.Println("Hello World")`
	`}`


```
For  index, value := range arr {
}
```


index of this element in the array
Current card we're iterating over
Take the slice of 'cards' and loop over it
Run this one time for each card in the slice

```
for index, card := range cards {
    fmt.Println(card)
}
```

```
var arr [5]int = [5]int{1, 2, 3, 4, 5} for i, v := range arr {
        //fmt.Printf("\nindex=%d", i, "value=%d", v) fmt.Printf("\nindex=%d, value=%d", i, v)
    }
```

```
ar := [5]int{10, 20, 30, 40, 50}
    for i, v := range ar {
        //fmt.Printf("\nindex=%d", i, "value=%d", v) fmt.Printf("\nindex=%d, value=%d", i, v)
    }
```

No.	Array	Slice
Size	Fixed.	Dynamic size can grow shrink like vector.Slices are built on top of arrays and provide a more flexible way to work with collections of data.
Declaration Syntax	var arr `[5]` int	var slice `[]` int, OR slice := make([]int, 0, 5)
Passing Argument	Array pass by value	Slice by reference.

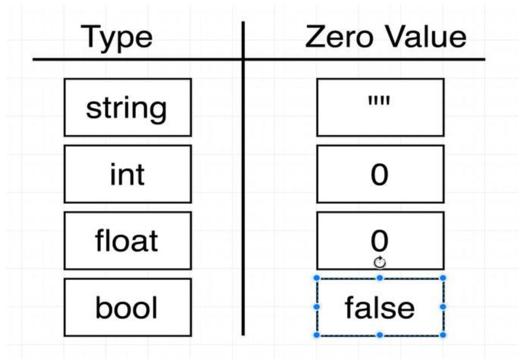
Usage	need a fixed-size collection of elements	more commonly used in Go because of their flexibility and dynamic nature. Support more operation like slicing ,appending

---

Struct

```
Import (
"fmt"
"unsafe")

Type Emp struct {
Id int
Name string
}

Func main() {
E:= Emp {id :1, Name:"Sagar")
tempid := unsafe.Sizeof(e)

fmt.Printf("Emp id=%d, Name=%s", e.id, e.name)

}
```

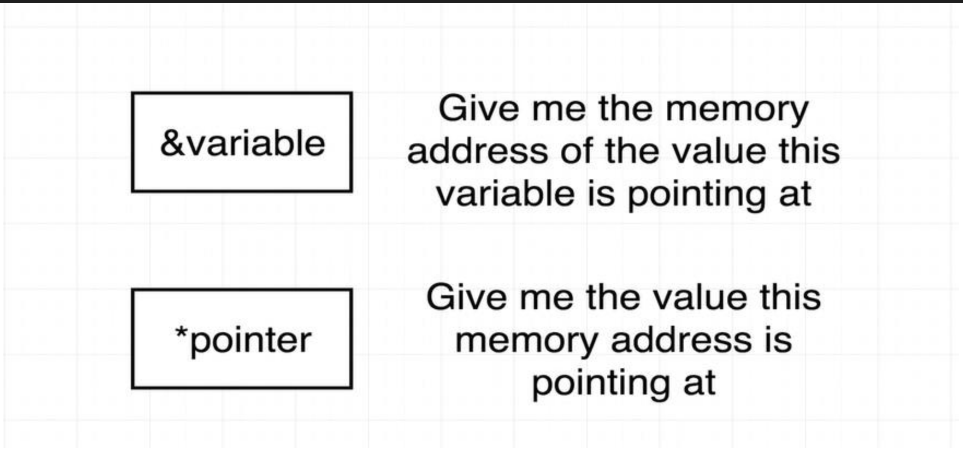
Note: - When we just declared struct NOT initialized then by default value is zero .



Type	Zero Value
string	""
int	0
float	0
bool	false

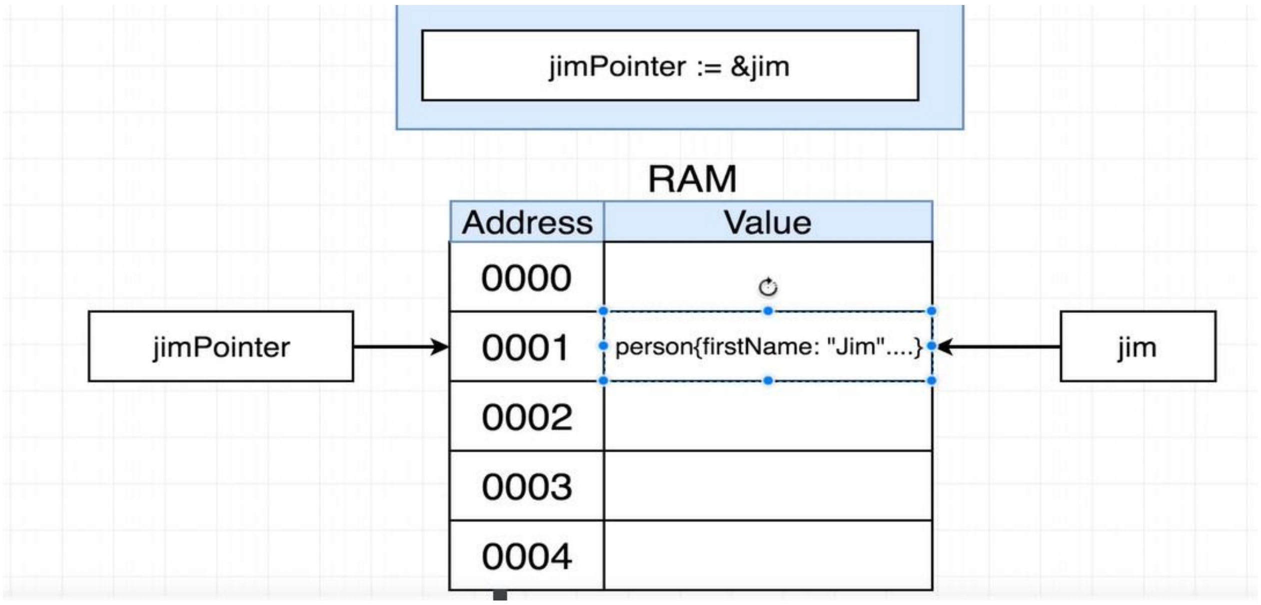
Struct using pointer, So its like reference pass to function.

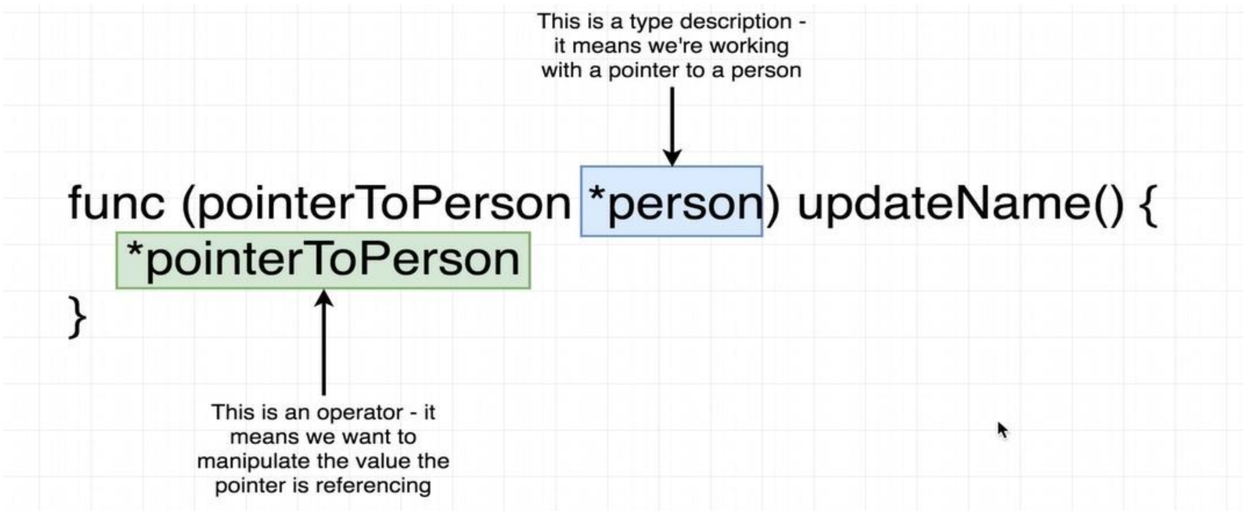
```
type Emp struct {
        id          int
        name string
}

/*func (e Emp) update() { e.id = 201
        e.name = "Sagar"
}*/

func (e *Emp) update() { (*e).id = 201
        (*e).name = "Sam"
}

func main() {

        e := Emp{id: 101, name: "Sagar"} eptr := &e
        fmt.Printf("\n Emp value id=%d, name=%s ", e.id, e.name)
        //e.update()
        fmt.Printf("\nAfter update Emp value id=%d, name=%s ", e.id, e.name) eptr.update()
        fmt.Printf("\nAfter pointer update Emp value id=%d, name=%s ", e.id, e.name)
}
```



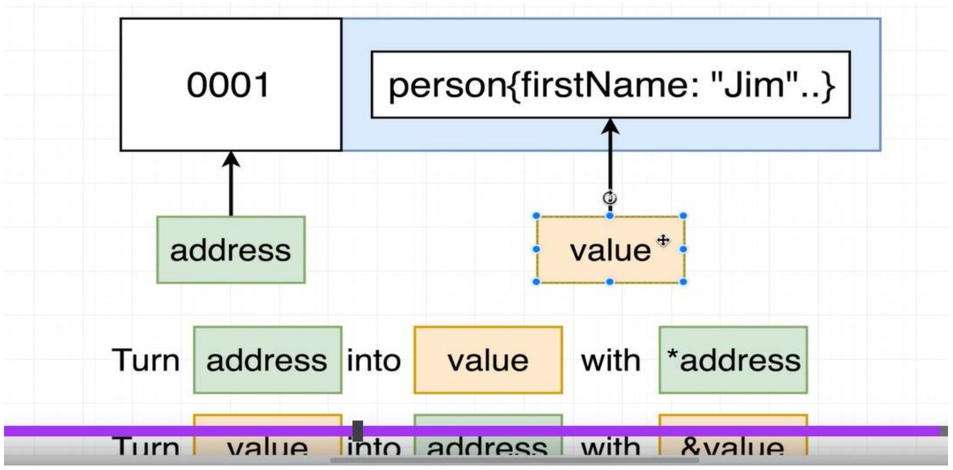
&variable — Give me the memory address of the value this variable is pointing at

*pointer — Give me the value this memory address is pointing at




This is a type description - it means we're working with a pointer to a person
This is an operator - it means we want to manipulate the value the pointer is referencing

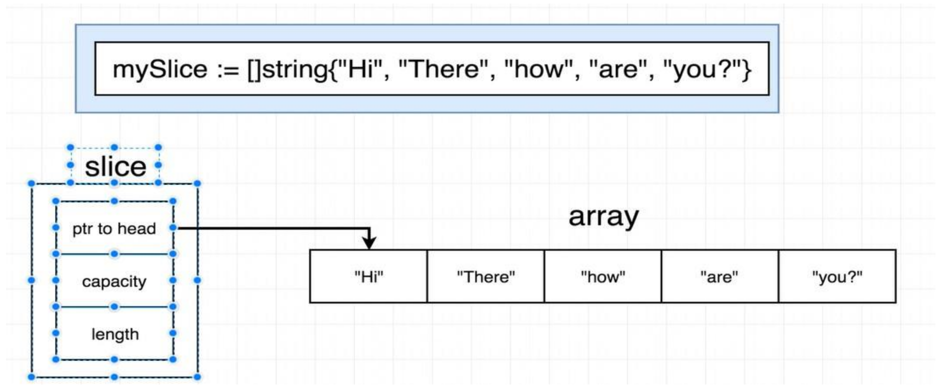
```
func (pointerToPerson *person) updateName() {
    *pointerToPerson
}
```


0001
person{firstName: "Jim"..}
address
value
Turn address into value with *address
Turn value into address with &value

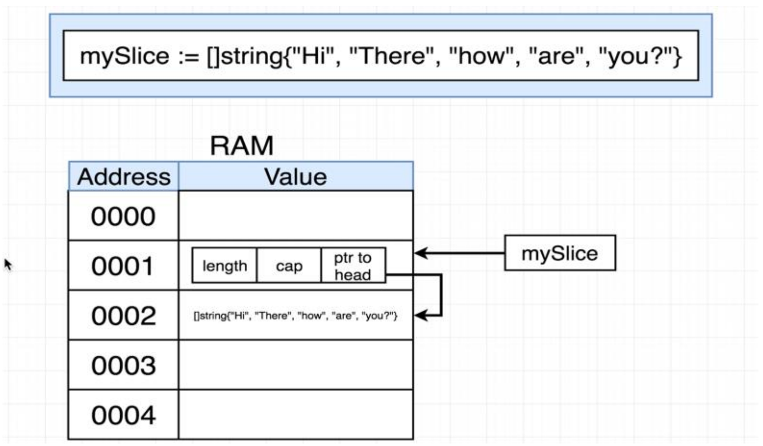
IMP : Structure can pass as value OR it just pass with/Without pointer but receiver you have used pointer at receiver then it become pointer.

e :=Emp{id:111,name:"Sagar"} //initialize e object **e.update()** **func (epointer *emp)update() {}**	e :=Emp{id:111,name:"Sagar"} //initialize e object eptr = &e **eptr.update()** **func (epointer *emp)update() {}**
Above both type work	

Difference between slice and struct



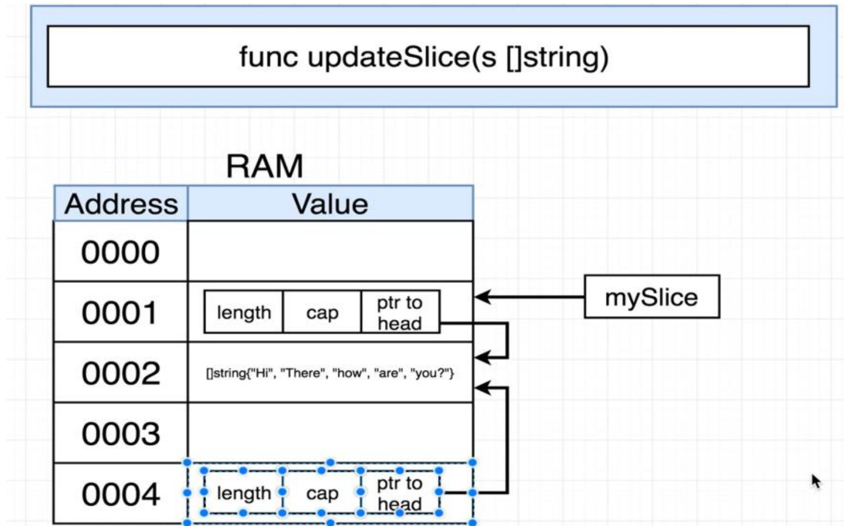
```
mySlice := []string{"Hi", "There", "how", "are", "you?"}
```



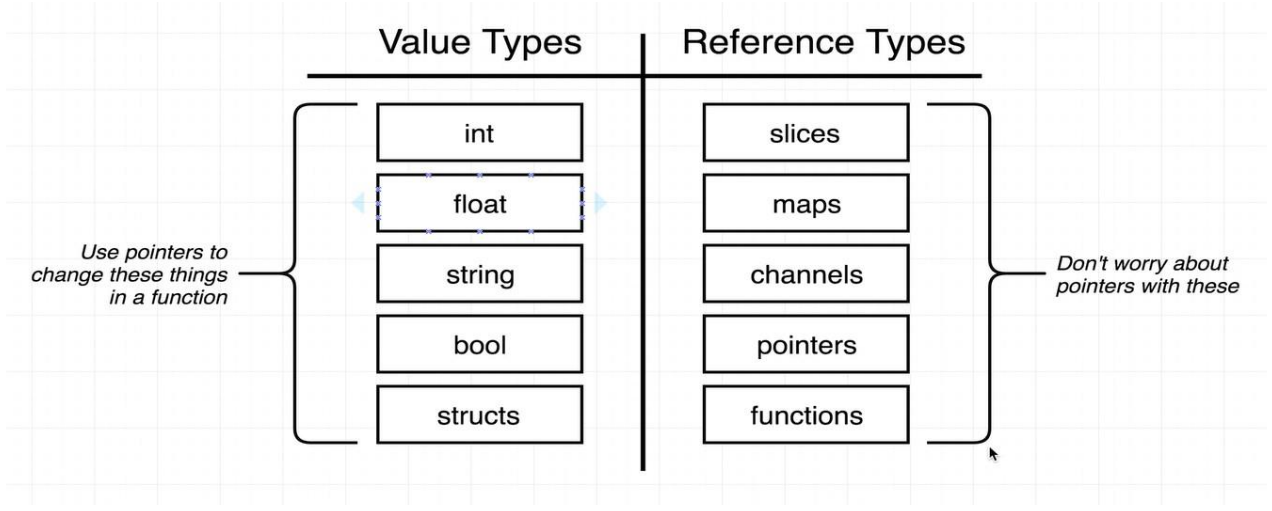
```
mySlice := []string{"Hi", "There", "how", "are", "you?"}
```

Address	Value
0000	
0001	length cap ptr to head
0002	[]string{"Hi", "There", "how", "are", "you?"}
0003	
0004	

Note: Go is pass by value language



Here When pass slice as argument then slice will copy its value as shown above.


Value Types
Reference Types
int
float
string
bool
structs
slices
maps
channels
pointers
functions
Use pointers to change these things in a function
Don't worry about pointers with these

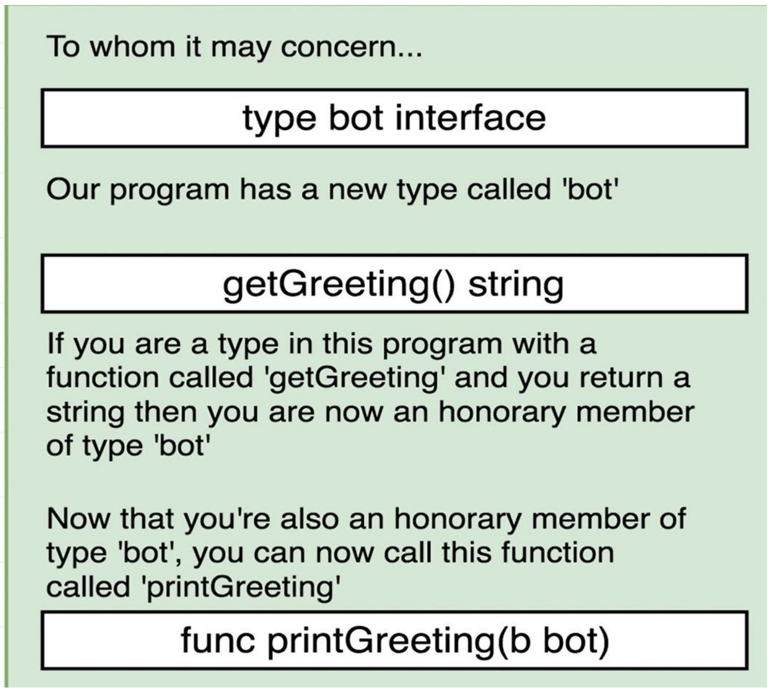
MAP

Mapname := map[key]value
myMap :=map[int]string
mymap :=make(map[int]string)
Maps are unordered collections, meaning that the order of key-value pairs is not guaranteed.

Interface

you can’t overload same function , that why interface is introduce.

```
package main

import (
	"fmt"
)

type Bot interface {
	getGreeting() string
}

type Englishbot struct {
}

func (Englishbot) getGreeting() string { // This is member method of that struct return "English Hello" // So
	same name is allowed .
}

type Spanishbot struct {
}

func (Spanishbot) getGreeting() string {// This is member method of that struct return "Spanish Hola" // So
	same name is allowed .
}

func printGreeting(b Bot) {
	fmt.Println(b.getGreeting())
}

func main() {
	fmt.Printf("")

	e := Englishbot{} s :=
	Spanishbot{}
	printGreeting(e)
	printGreeting(s)
```



To whom it may concern...

type bot interface

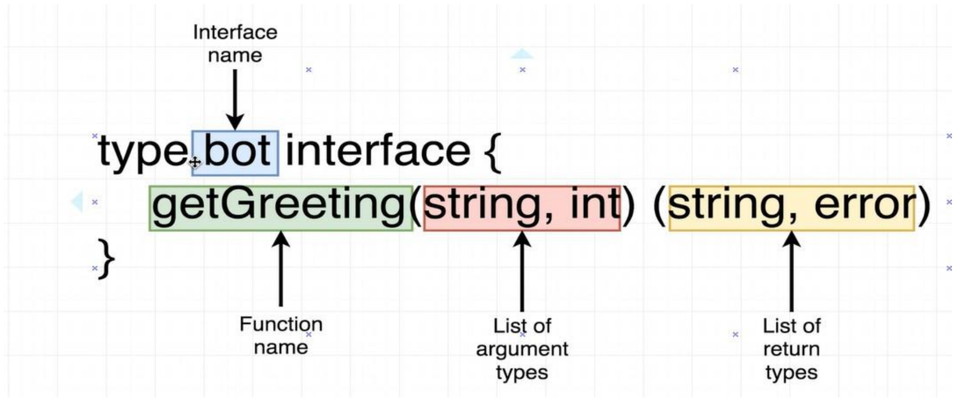
Our program has a new type called 'bot'

getGreeting() string

If you are a type in this program with a function called 'getGreeting' and you return a string then you are now an honorary member of type 'bot'

Now that you're also an honorary member of type 'bot', you can now call this function called 'printGreeting'

func printGreeting(b bot)



```
type bot interface {
    getGreeting(string, int) (string, error)
}
```

Interface automatically link with function . Q. How?

GoRoutine and channel

1. Goroutines are light weight thread.
2. They are functions that run concurrently with other goroutines within the same address space.

Q => what is mean by within same address space .

3. It is very cheap for switch overhead and memory .

Go Anonymous function/function literals

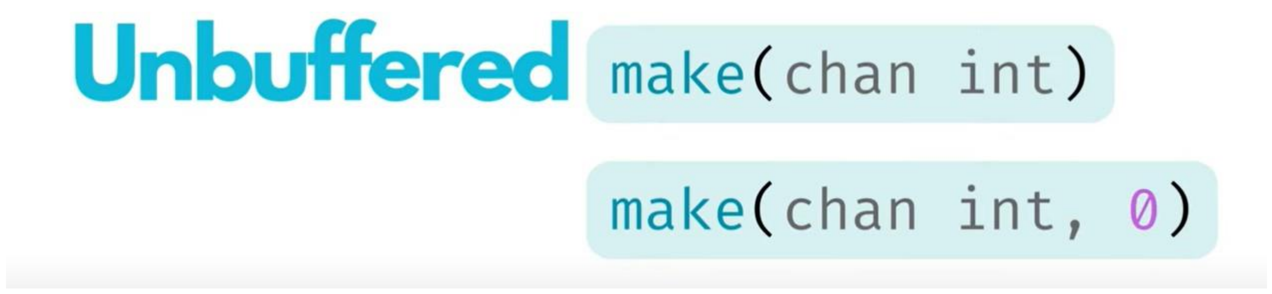
1. No function name
2. Useful for define inline function
3. Ex

```
func(parameter_list)(return_type){
// code..

// Use return statement if return_type are given
// if return_type is not given, then do not
// use return statement return
}()
```

```
func main() {
    fmt.Println("Hello, World!") f := func()
    {
        fmt.Println(" Anonamous function with variable called")
    }
    f()

    func() {
        fmt.Println(" Anonamous function only")
    }()

}
```

What is Buffered & unbuffered channel?

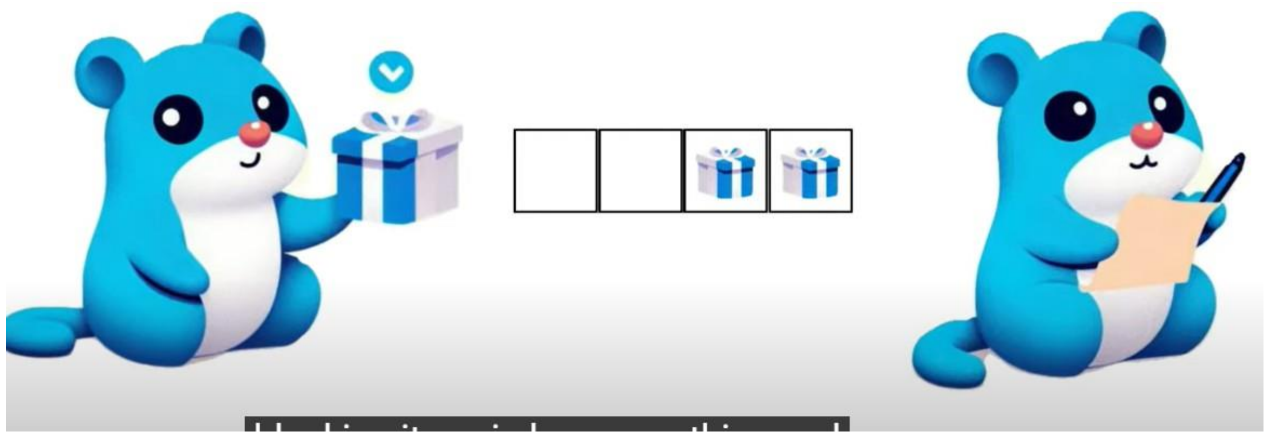
on the channel but the other goroutine
is busy so goroutine 1 Waits

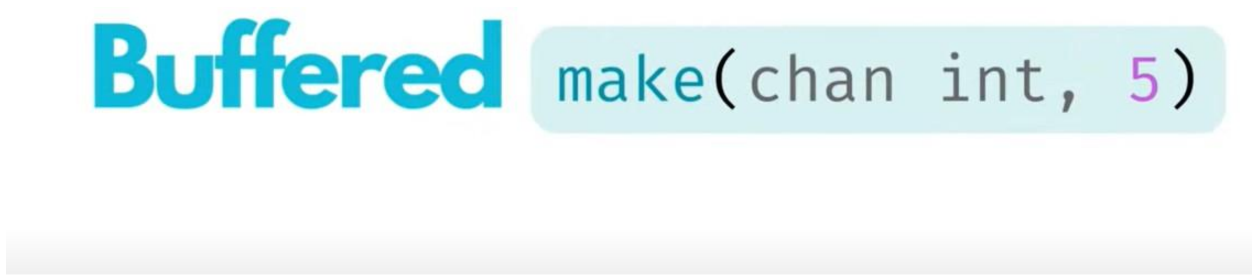

Unbuffered

```
make(chan int)
make(chan int, 0)
```

It is blocking until data is received.

Buffer channel




Buffered

```
make(chan int, 5)
```