Backend Programming

M. Yauri M. Attamimi

Upgrade your potential !

Foundation Class

- WEEK 1 (Session 2) -

Constants (I)

The term constant is used to denote some fixed values of character, string, boolean, and numeric.
consider the following code snippet:

```
var num = 50
var greeting = “Hi, Gopher !”
```

in the above code, `50` and `“Hi, Gopher !”` are constants being assigned to variable `num` and `greeting` respectively even though no keyword const were explicitly defined, they internally are constants in Go.

Constants (II)

A constant statement can appear anywhere as a var statement can.

e.g. :

```
const n = 1000000
const d = 3e20 / n
```

Constants as the name indicate cannot be reassigned again.

e.g. :

```
const num = 5
num = 7 //reassignment not allowed
```

Constants (III)

Value of constants should be known at compile time.

e.g. :

```
num1 := 5
num2 := 7
const maxnum = math.Max(num1, num2) // not allowed
```

we can’t assign maxnum with math.Max since the function call takes places at runtime, while constant value should be known already at compile time.

Constants (IV)

- A string constant such as `“Golang Foundation”` doesn’t have any type.
- Untyped constants have a default type associated with them, and Go will supply the type for us if and only if the line of code demands it (such as variable assignment, function call, or explicit type conversion) as shown on the various code snippets below:

```
// variable assignment
var className = “Golang Foundation”

// function call, math.Sin expect float64 for it’s argument
math.Sin(5000000)

// explicit type conversion
int64(25.0)
```

Constants (V)

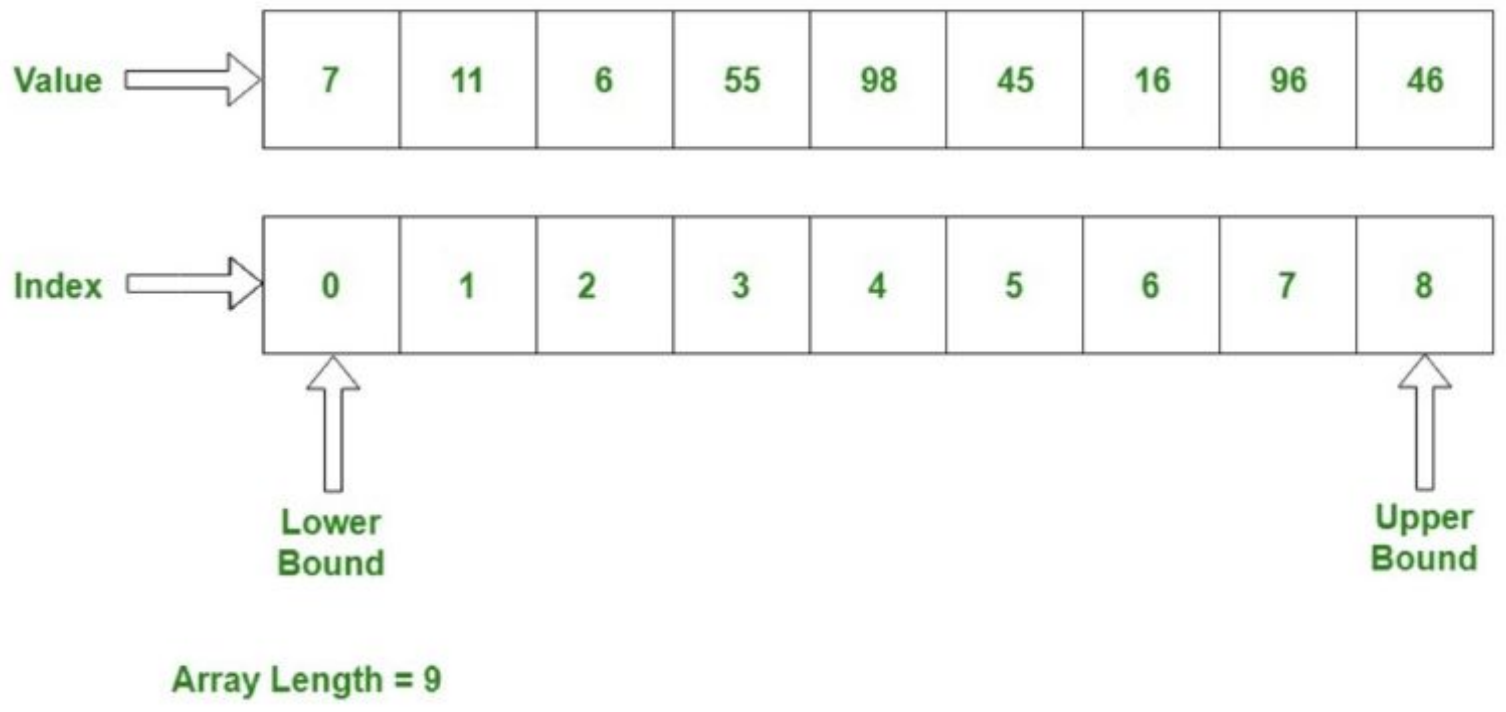
Could you guess what would happen with this following code snippet ?

```
...
const trueConst = true
type myBool bool
var defaultBool = trueConst
var customBool myBool = trueConst
defaultBool = customBool
...
```

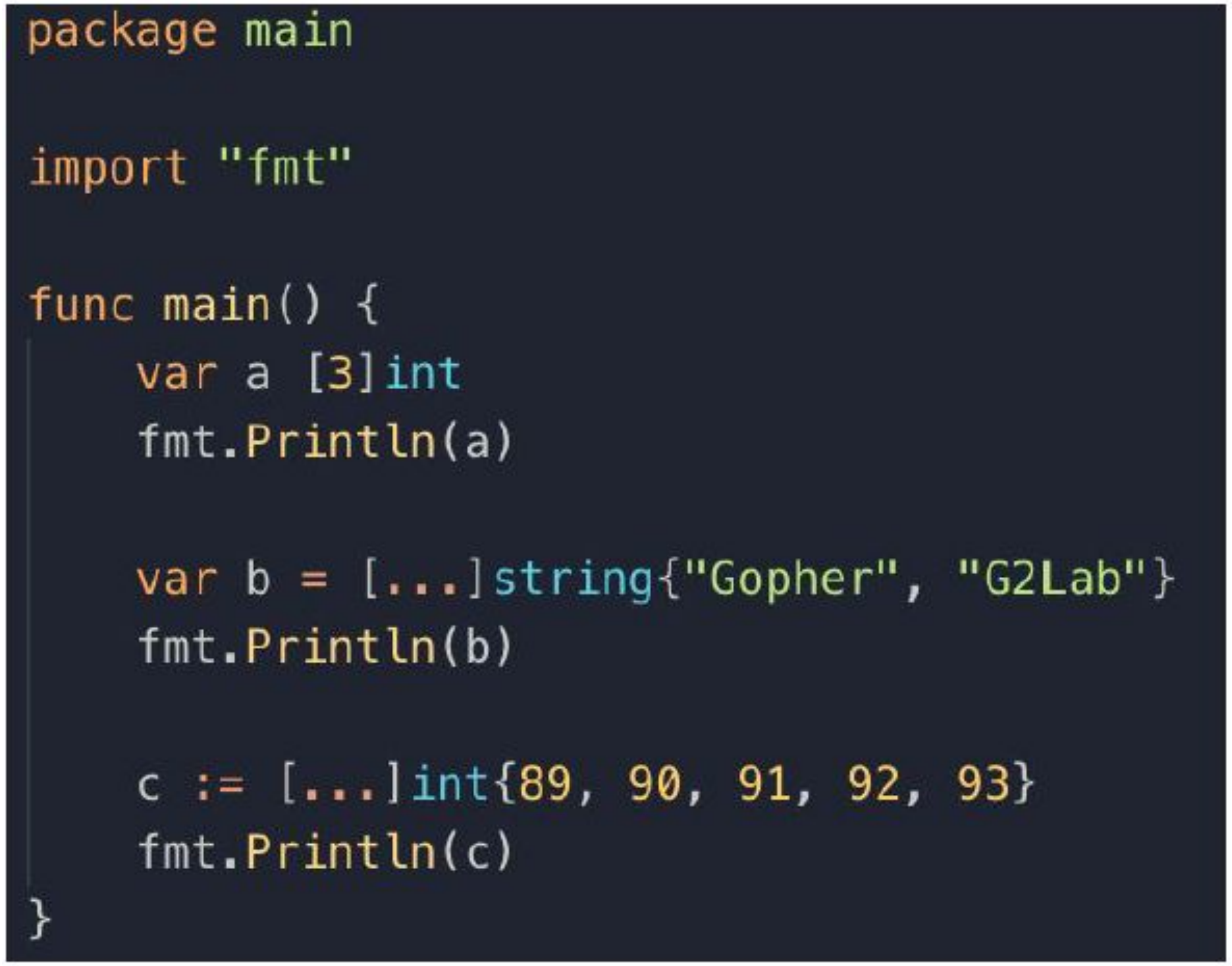
Array and Slices

Array

An array is a collection of elements that belong to the same type. For example the collection of integers 5, 8, 9, 79, 76 form an array. Mixing values of different types, for example an array that contains both strings and integers is not allowed in Go.



2 Dimensional Array

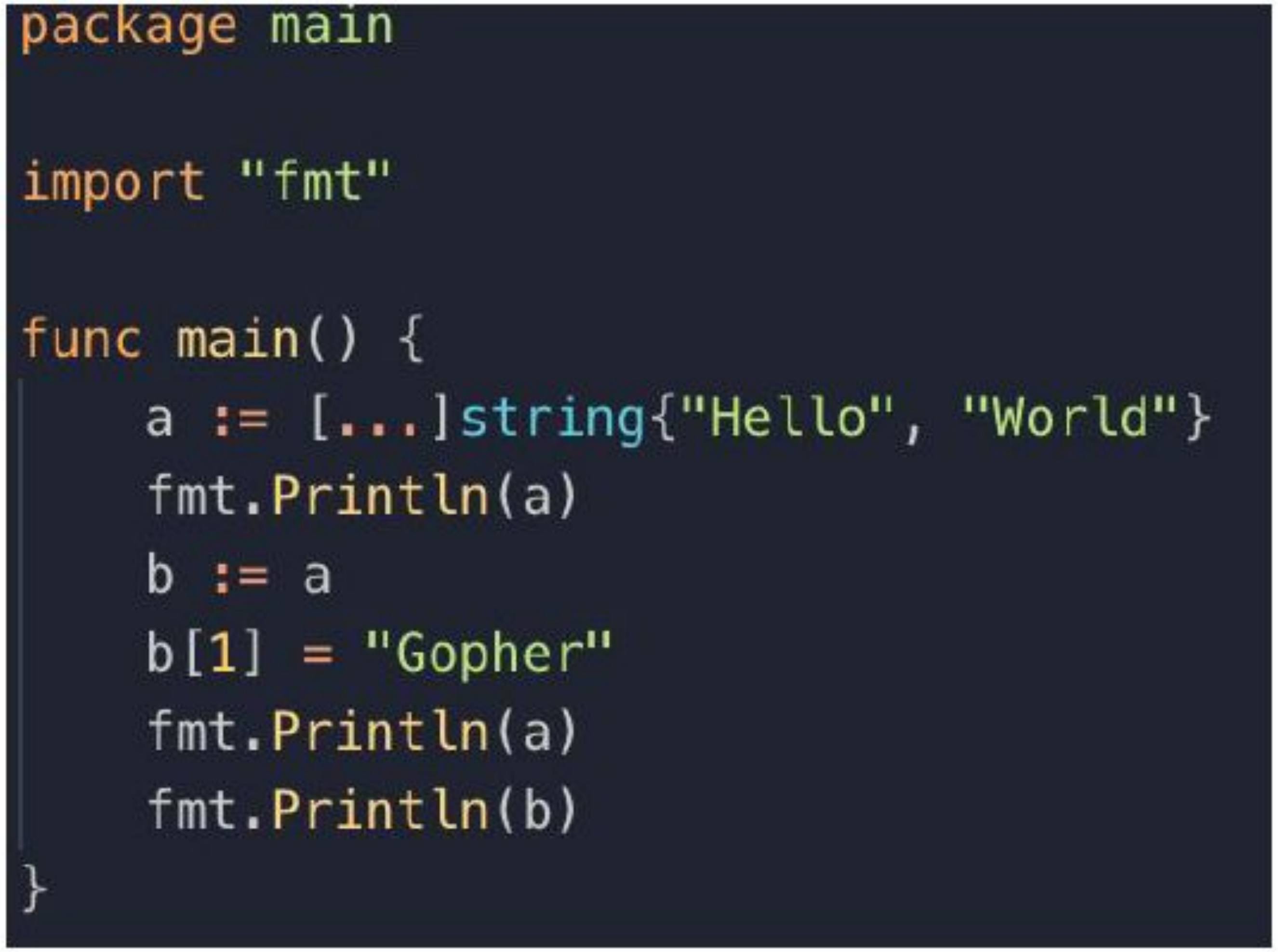


```
package main

import "fmt"

func main() {
    var a [3]int
    fmt.Println(a)

    var b = [...]string{"Gopher", "G2Lab"}
    fmt.Println(b)

    c := [...]int{89, 90, 91, 92, 93}
    fmt.Println(c)
}
```

How about multidimensional (>2) arrays?

Array are "value types"

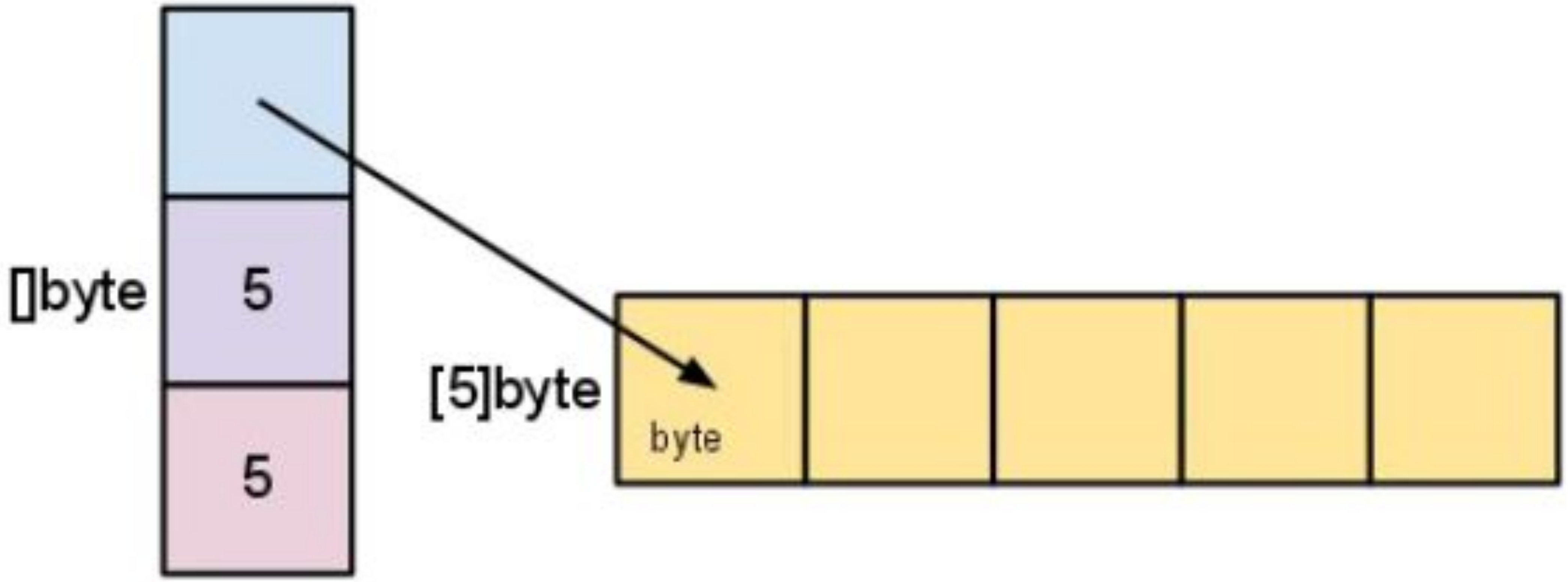
Guess what happens…



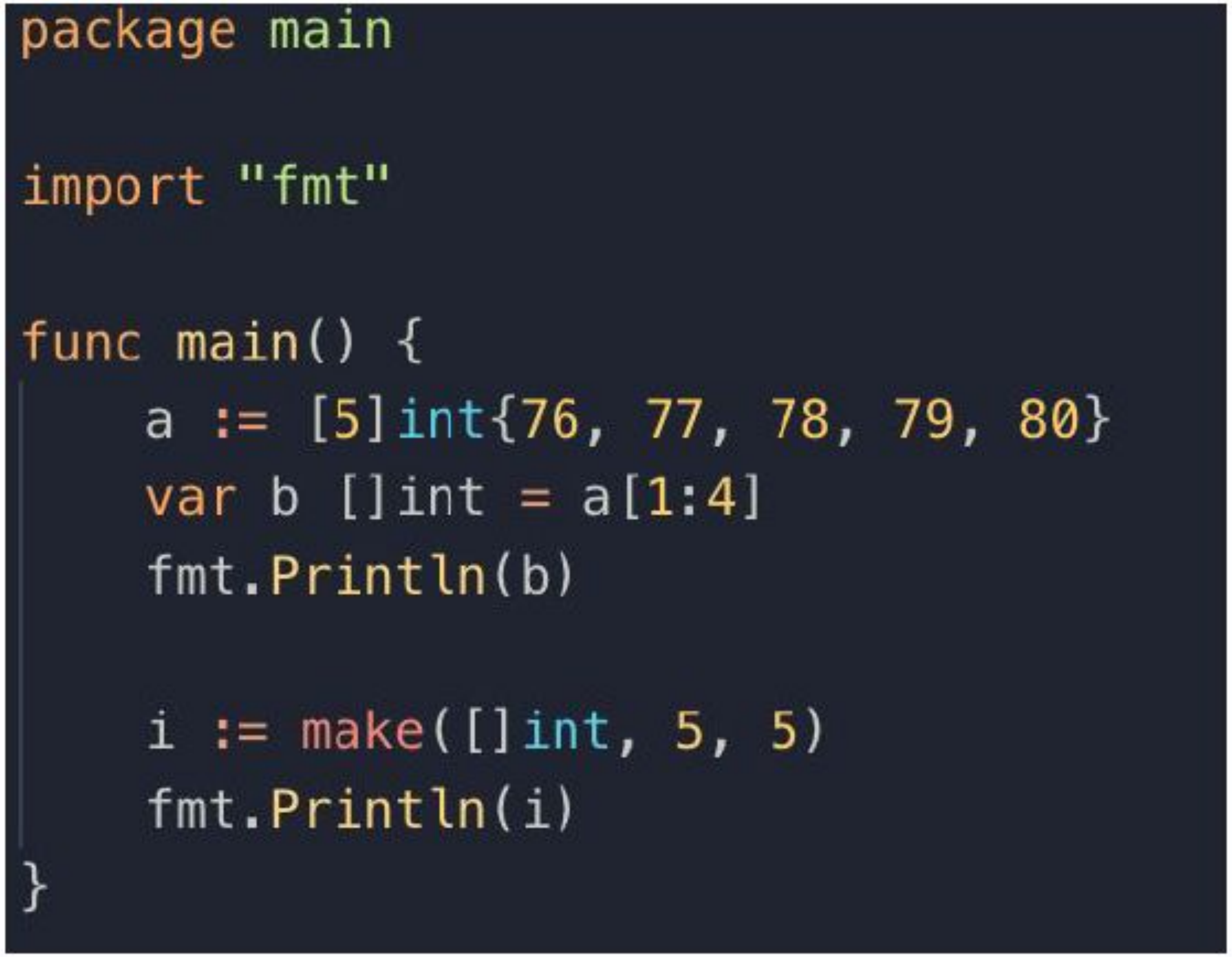
```
package main

import "fmt"

func main() {
	a := [...]string{"Hello", "World"}
	fmt.Println(a)
	b := a
	b[1] = "Gopher"
	fmt.Println(a)
	fmt.Println(b)
}
```

Slices

A slice is a convenient, flexible and powerful wrapper on top of an array. Slices do not own any data on their own. They are the just references to existing arrays.

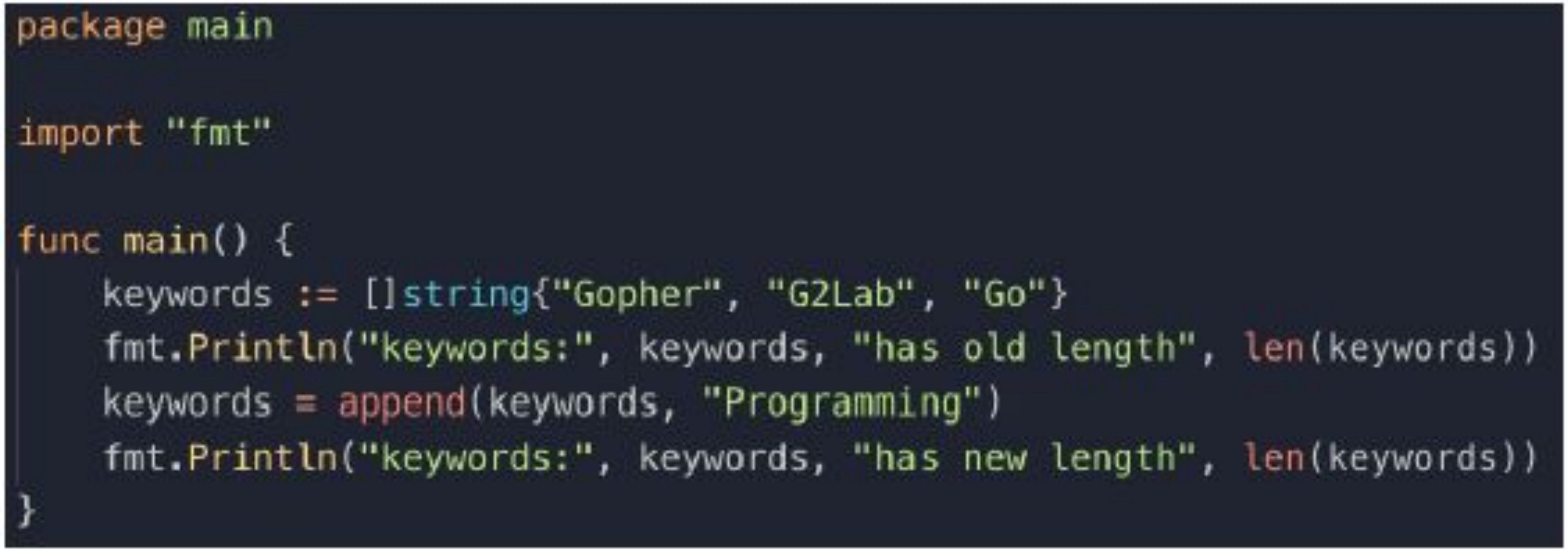


Example


```go
package main

import "fmt"

func main() {
    a := [5]int{76, 77, 78, 79, 80}
    var b []int = a[1:4]
    fmt.Println(b)

    i := make([]int, 5, 5)
    fmt.Println(i)
}
```

Appending to a Slice

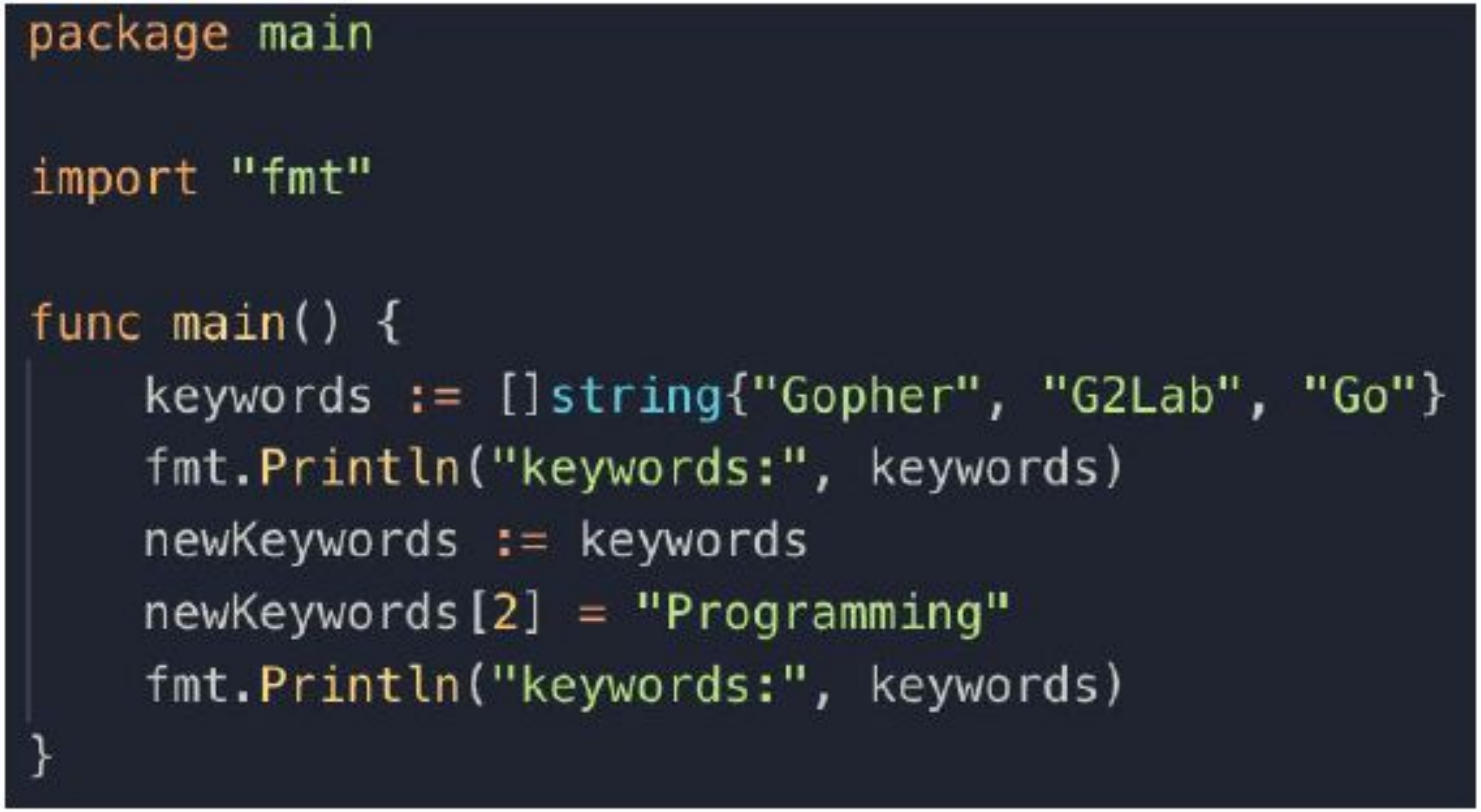
As we already know arrays are restricted to fixed length and their length cannot be increased. Slices are dynamic and new elements can be appended to the slice using `append` function. The definition of append function is
`func append(s []T, x ...T) []T`.



```go
package main

import "fmt"

func main() {
    keywords := []string{"Gopher", "G2Lab", "Go"}
    fmt.Println("keywords:", keywords, "has old length", len(keywords))
    keywords = append(keywords, "Programming")
    fmt.Println("keywords:", keywords, "has new length", len(keywords))
}
```

Slices are "reference types"

Guess what happens…



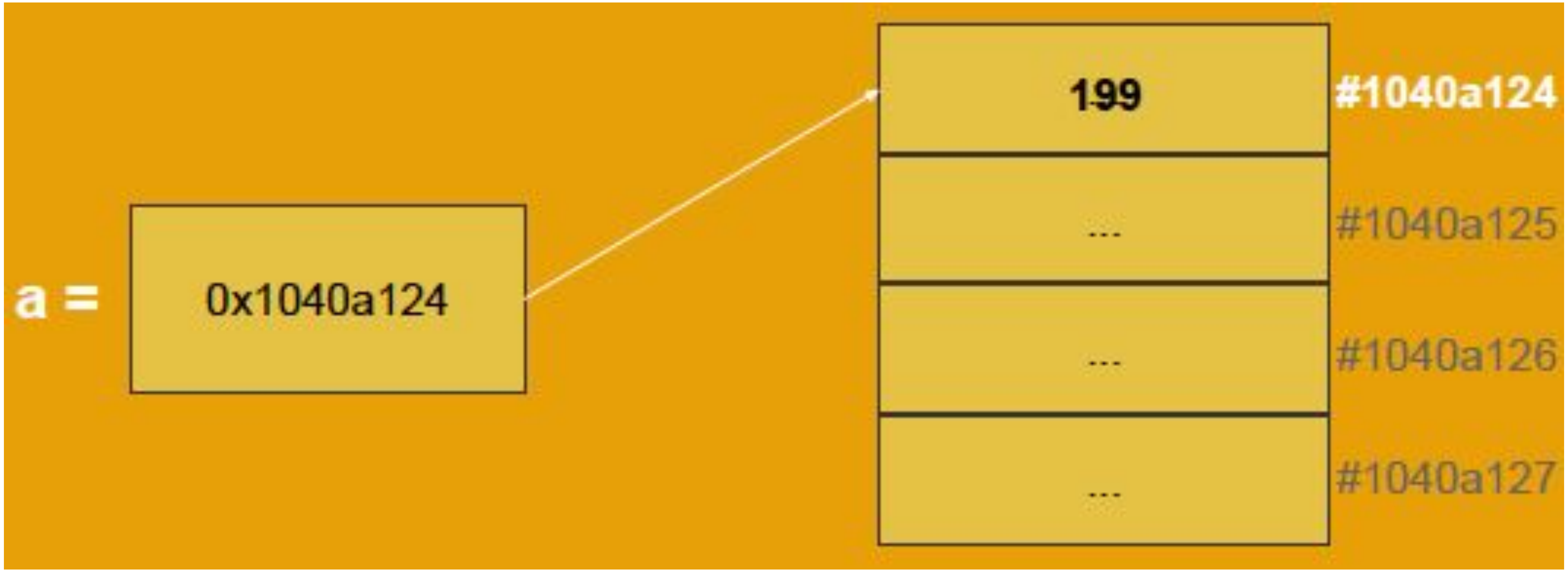
```go
package main

import "fmt"

func main() {
	keywords := []string{"Gopher", "G2Lab", "Go"}
	fmt.Println("keywords:", keywords)
	newKeywords := keywords
	newKeywords[2] = "Programming"
	fmt.Println("keywords:", keywords)
}
```

How to optimize slice's memory?

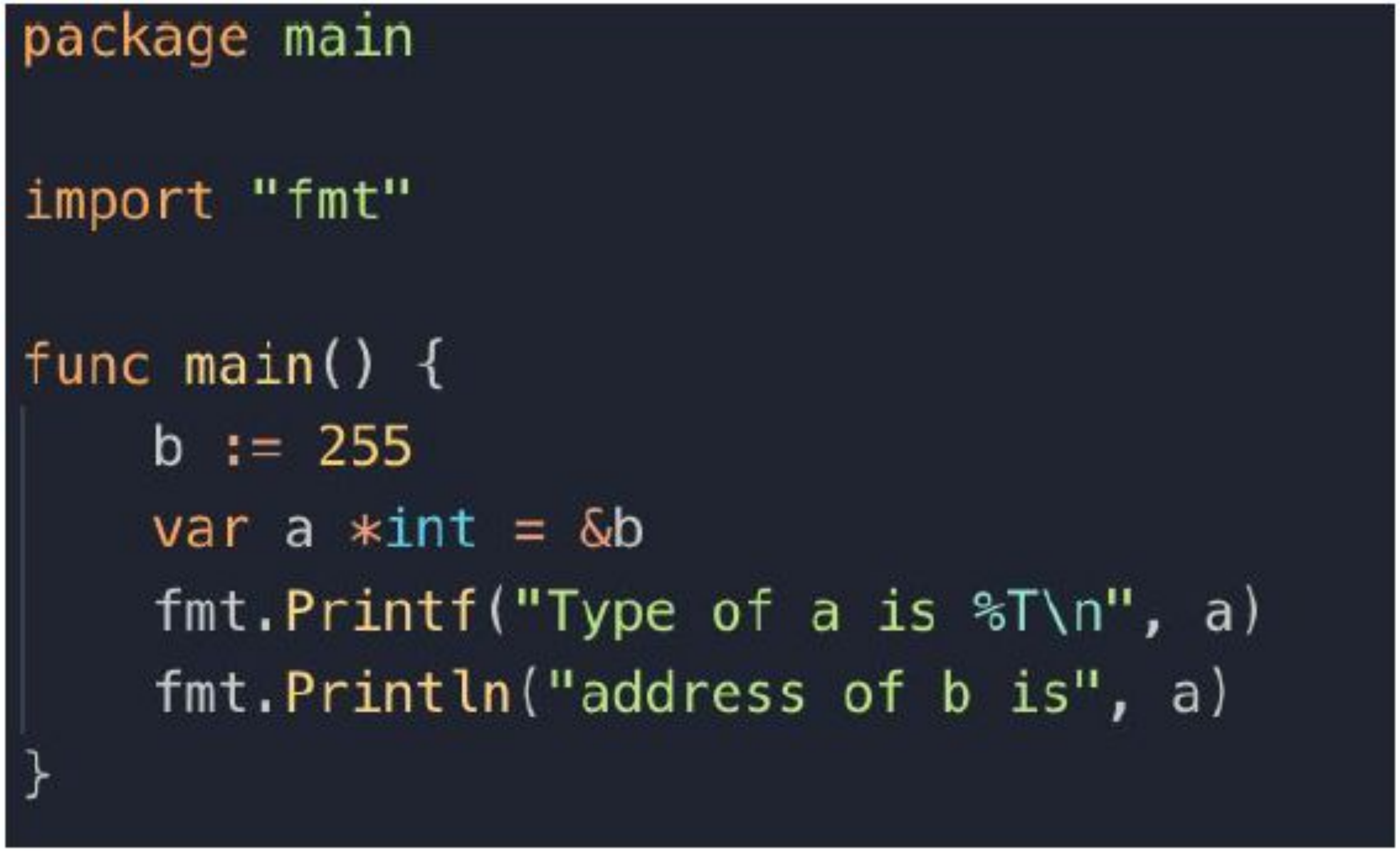
Reference Types; Pointers

What is a Pointer?

A “pointer” is a variable which stores the memory address of another variable.

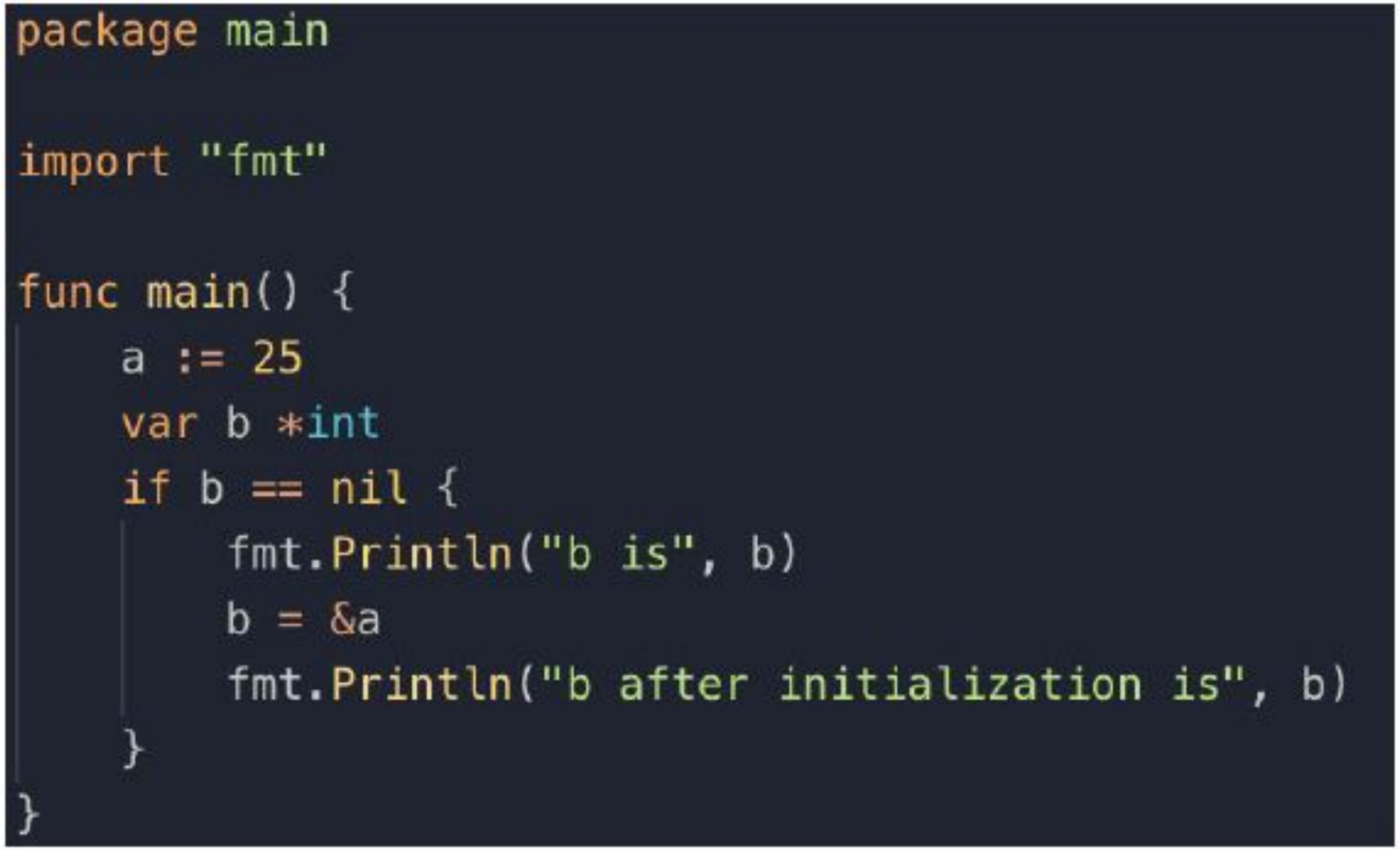


Declaring a Pointer



```go
package main

import "fmt"

func main() {
    b := 255
    var a *int = &b
    fmt.Printf("Type of a is %T\n", a)
    fmt.Println("address of b is", a)
}
```

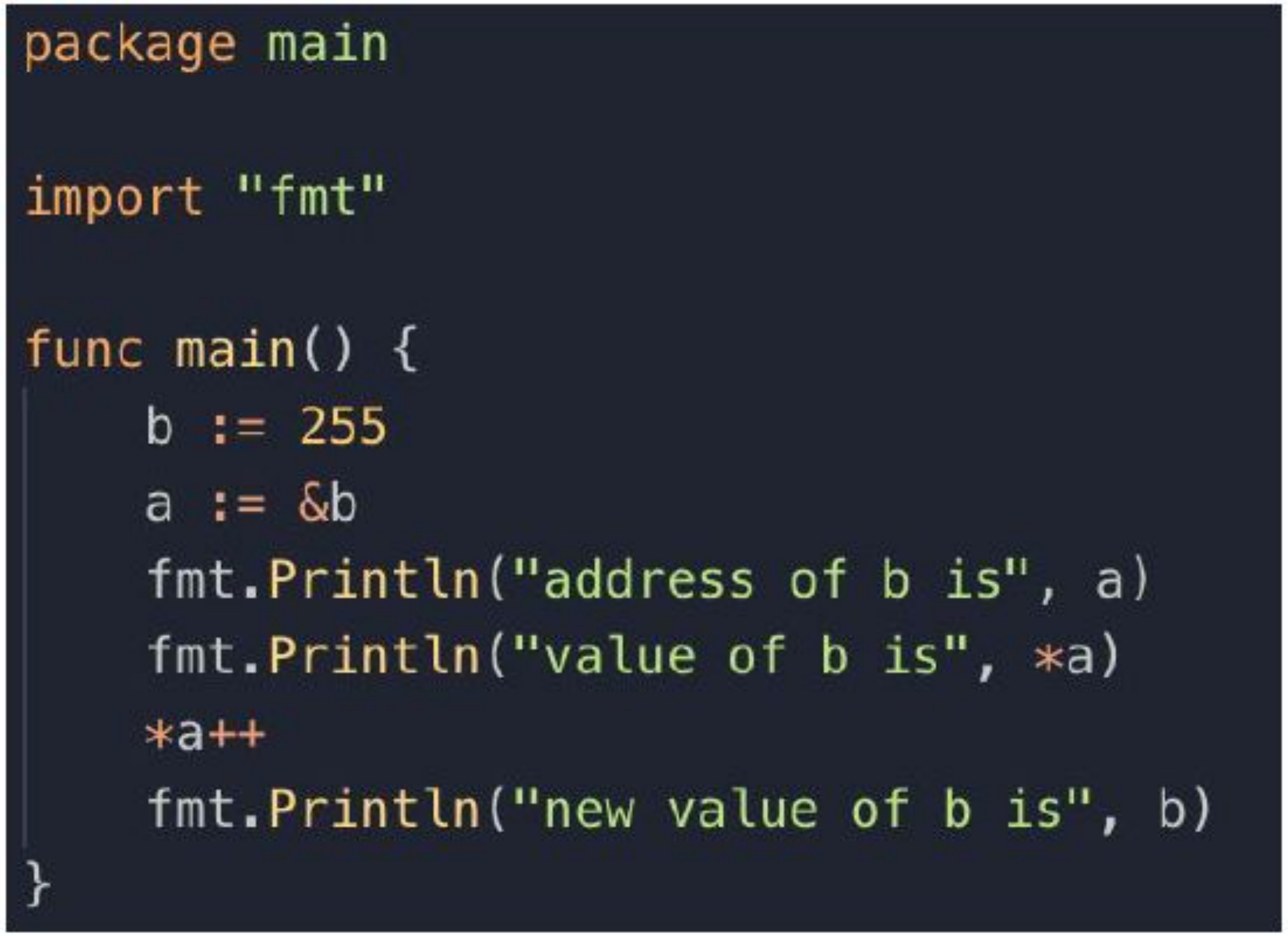
Pointer Zero Value is "nil"



```go
package main

import "fmt"

func main() {
    a := 25
    var b *int
    if b == nil {
        fmt.Println("b is", b)
        b = &a
        fmt.Println("b after initialization is", b)
    }
}
```

Dereferencing a Pointer

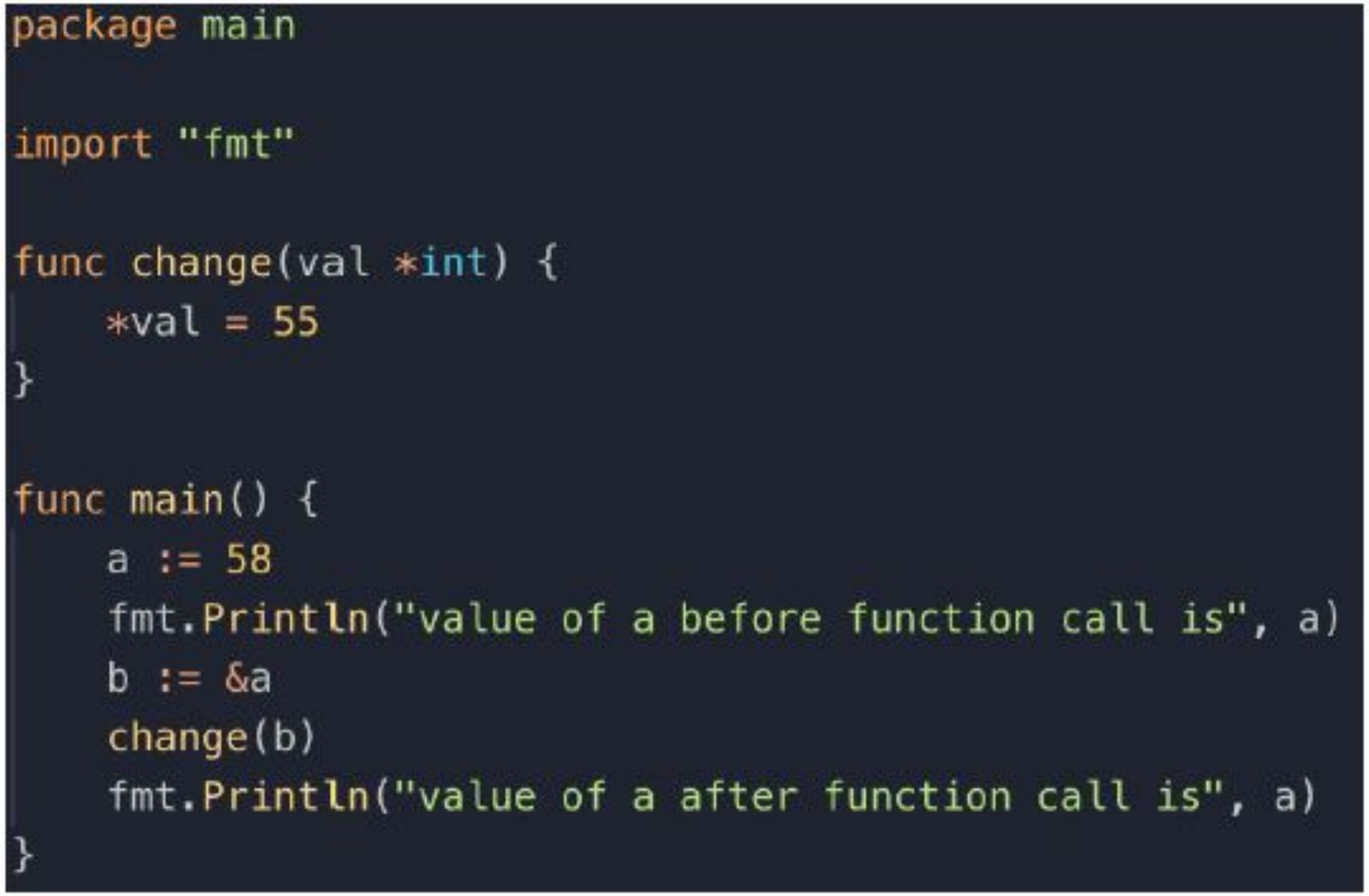
→ means accessing the value of the variable which the pointer points to.



```go
package main

import "fmt"

func main() {
	b := 255
	a := &b
	fmt.Println("address of b is", a)
	fmt.Println("value of b is", *a)
	*a++
	fmt.Println("new value of b is", b)
}
```

Passing Pointer to a Function

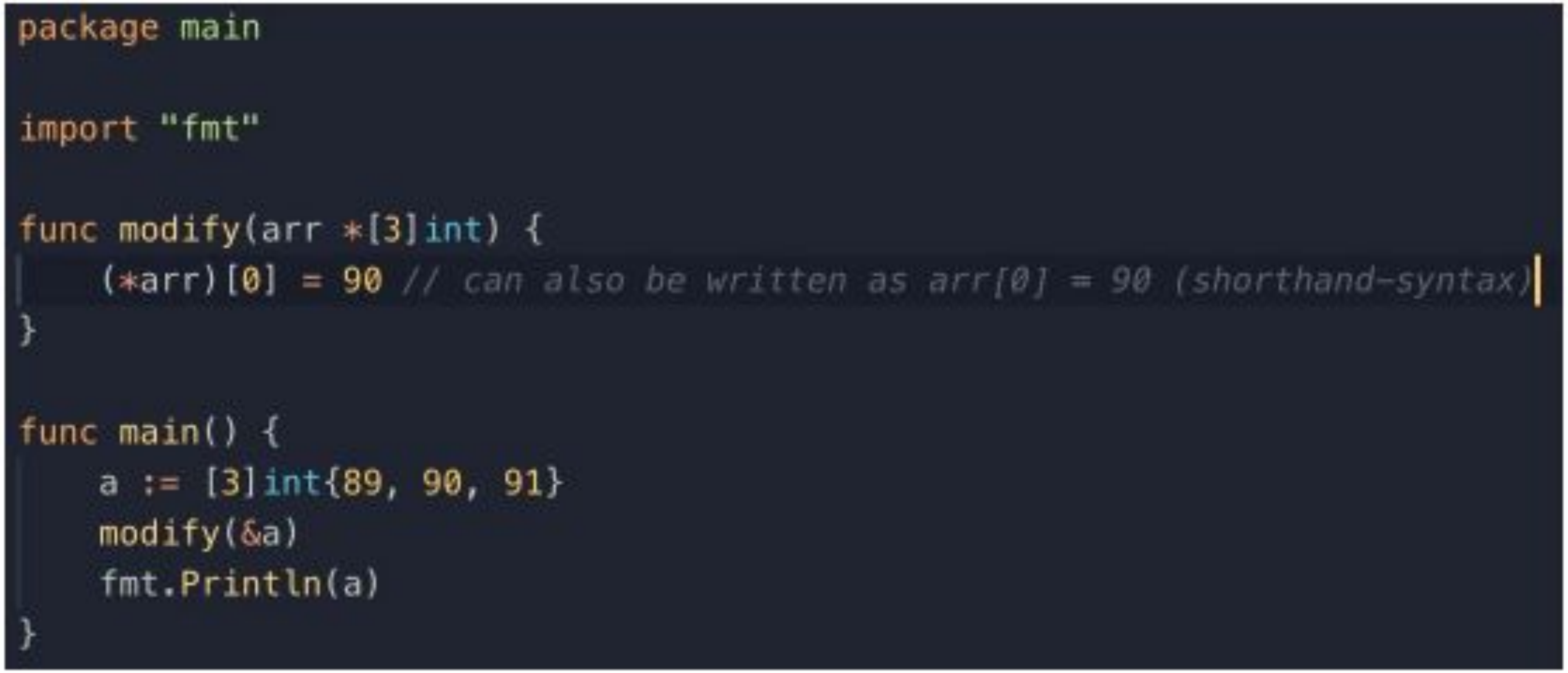
Guess what happens…



```
package main

import "fmt"

func change(val *int) {
    *val = 55
}

func main() {
    a := 58
    fmt.Println("value of a before function call is", a)
    b := &a
    change(b)
    fmt.Println("value of a after function call is", a)
}
```

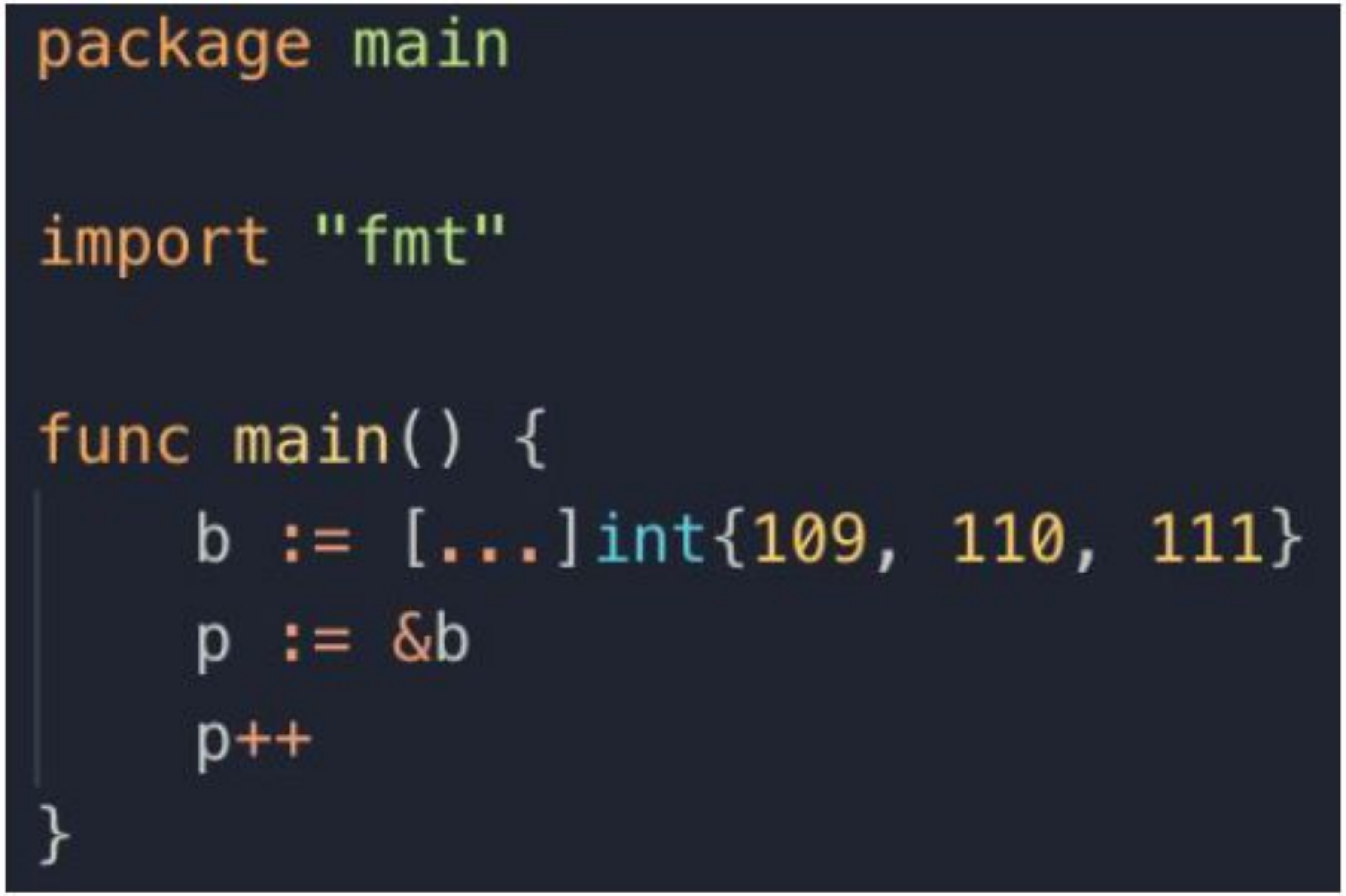
Passing Array by Reference

Could you explain these following codes?



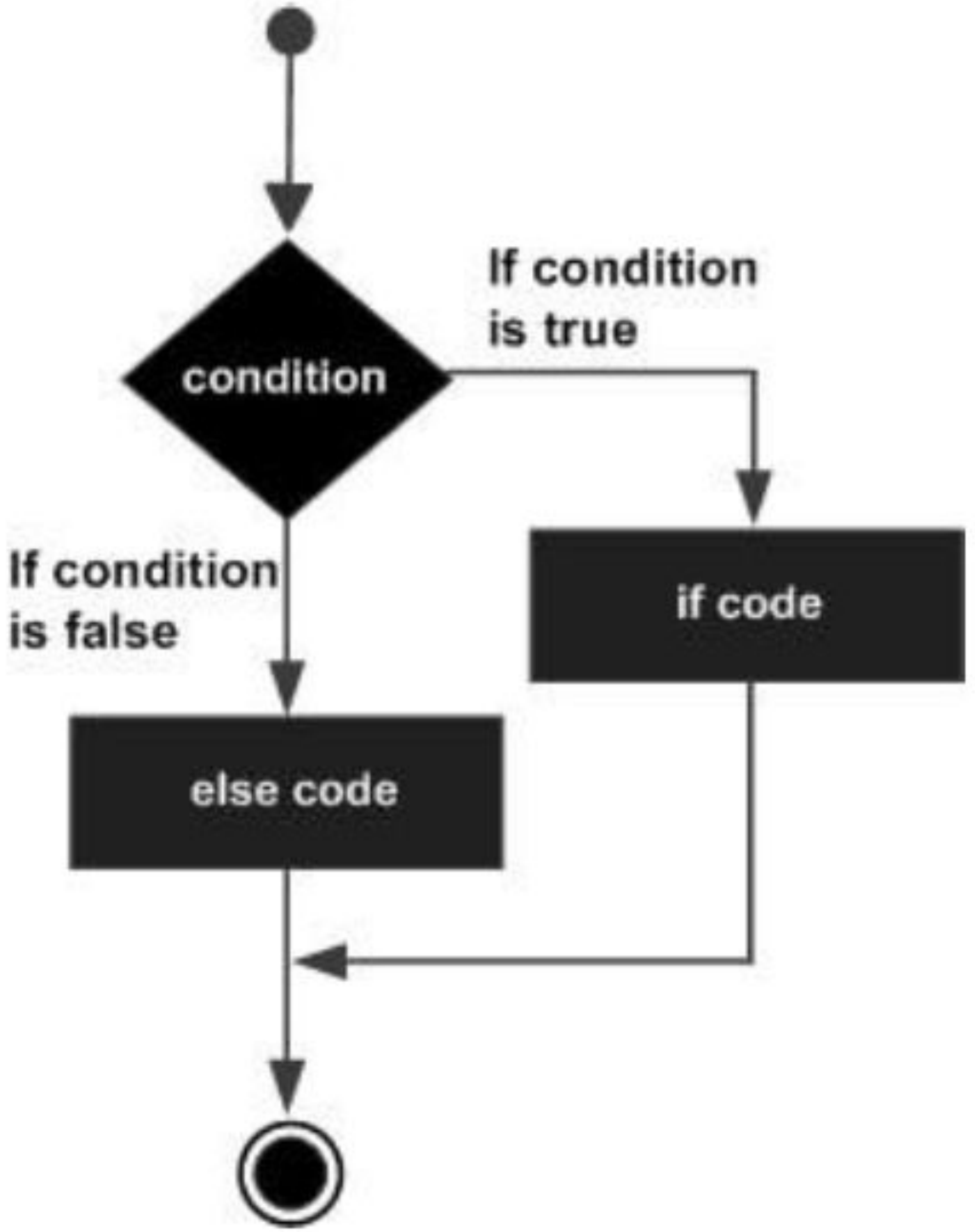
```go
package main

import "fmt"

func modify(arr *[3]int) {
    (*arr)[0] = 90 // can also be written as arr[0] = 90 (shorthand-syntax)
}

func main() {
    a := [3]int{89, 90, 91}
    modify(&a)
    fmt.Println(a)
}
```

Go Doesn’t Support Pointer Arithmetic



```
package main

import "fmt"

func main() {
    b := [...]int{109, 110, 111}
    p := &b
    p++
}
```

If Statement


condition
If condition is true
If condition is false
if code
else code

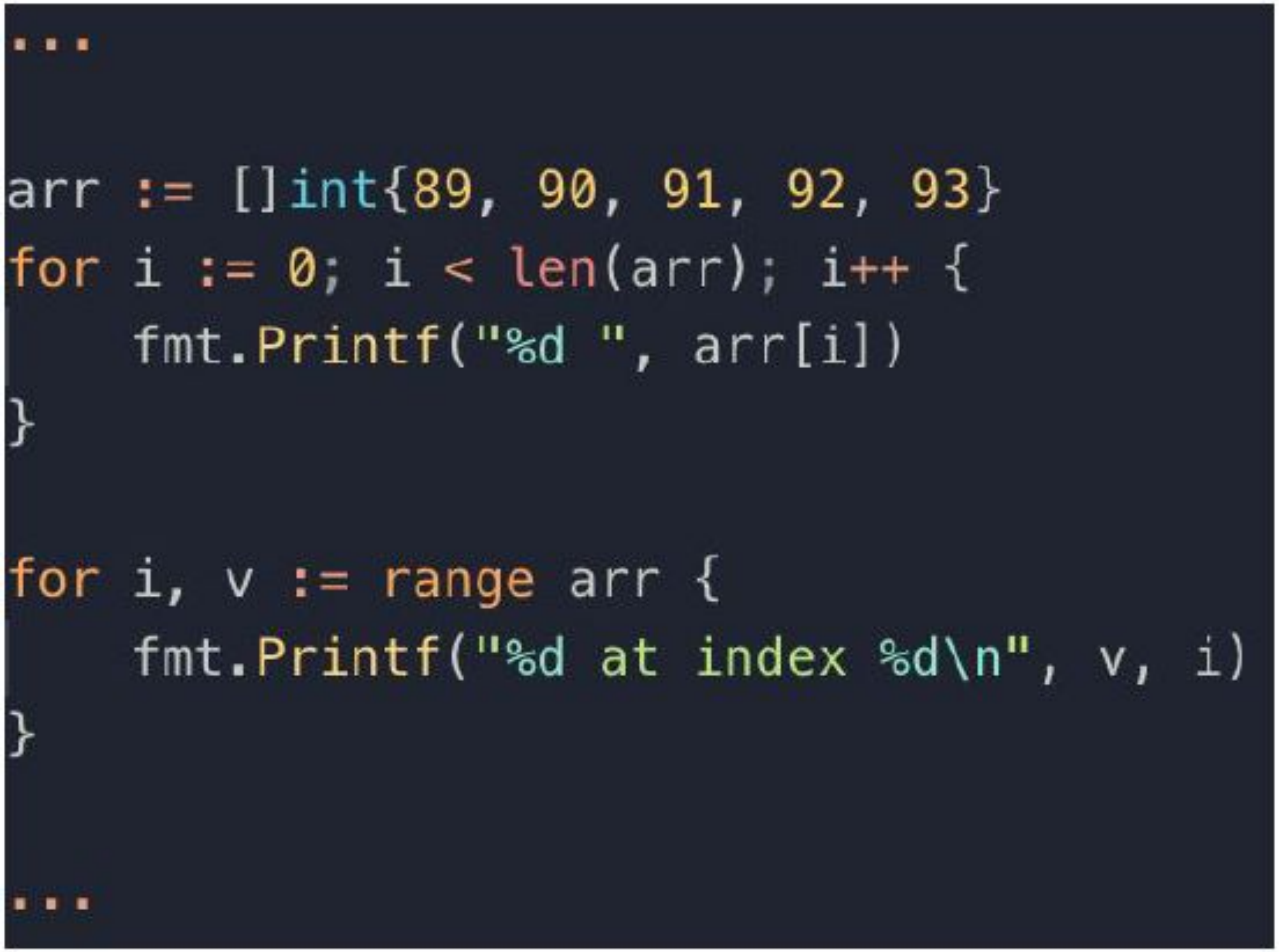
Basic Syntax

```
if condition {
}
```

```
if condition {
} else if condition {
} else {
}
```

Switch Statement

Loop Statements

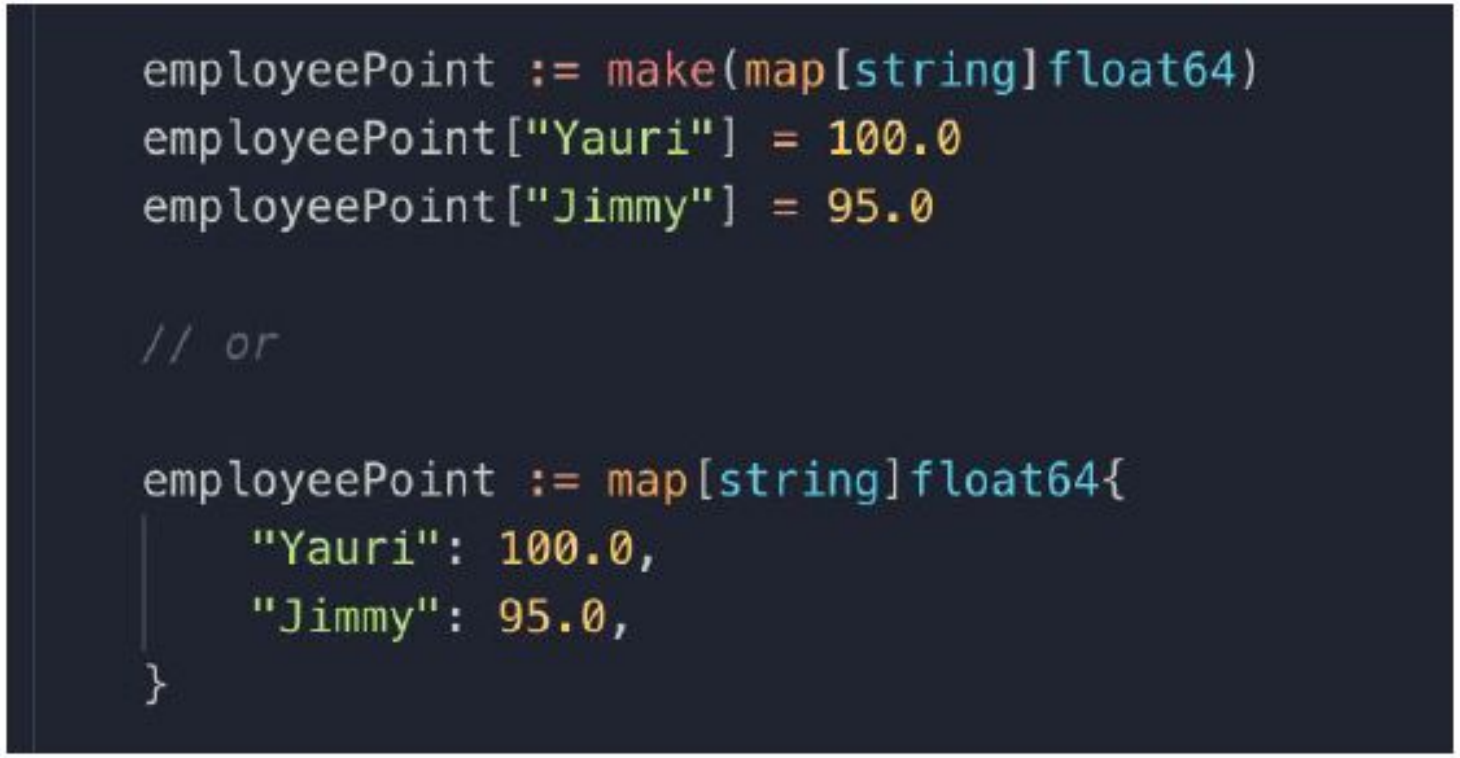


```go
...

arr := []int{89, 90, 91, 92, 93}
for i := 0; i < len(arr); i++ {
    fmt.Printf("%d ", arr[i])
}

for i, v := range arr {
    fmt.Printf("%d at index %d\n", v, i)
}


...
```

Map

Map Definition

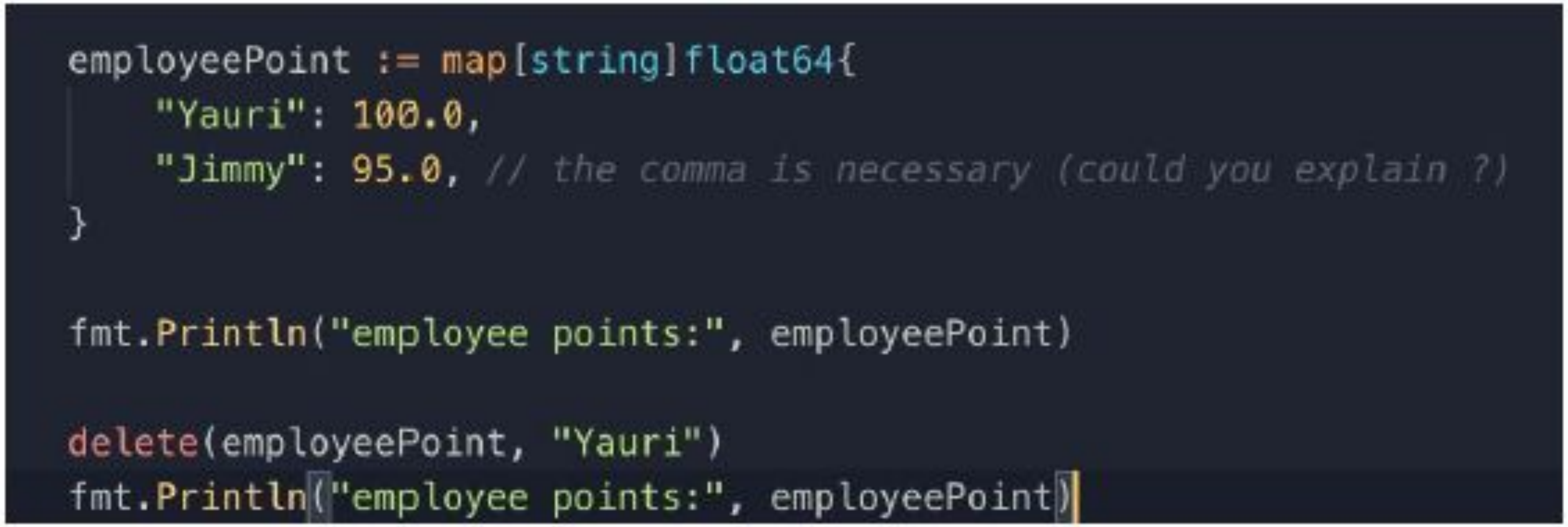
⇒ Go builtin type which associates a value to a key (a pair of key and value).



```go
employeePoint := make(map[string]float64)
employeePoint["Yauri"] = 100.0
employeePoint["Jimmy"] = 95.0

// or

employeePoint := map[string]float64{
    "Yauri": 100.0,
    "Jimmy": 95.0,
}
```

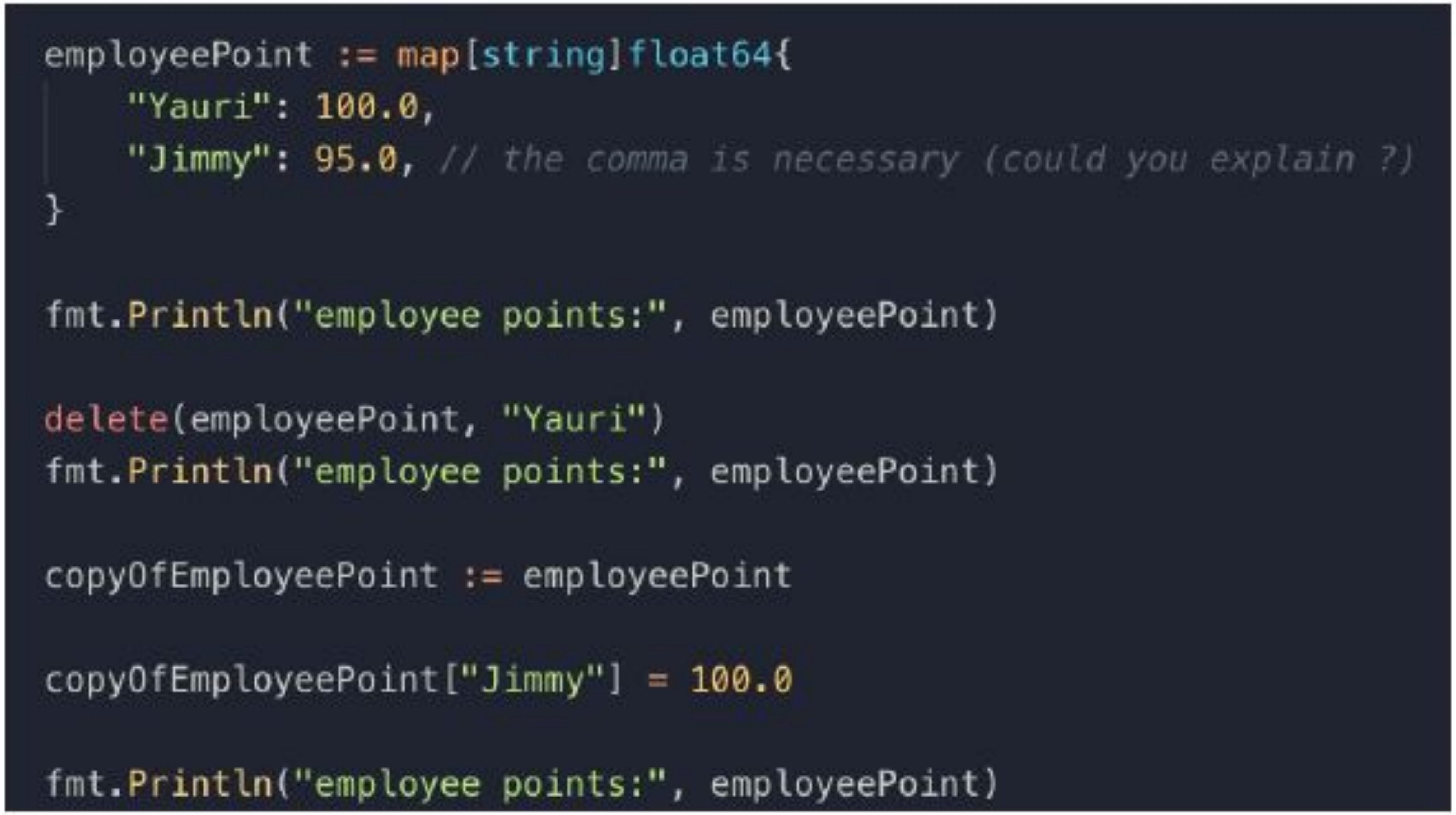
Check if a Value exists or not

```
value, ok := map[key]
```



```go
employeePoint := map[string]float64{
    "Yauri": 100.0,
    "Jimmy": 95.0, // the comma is necessary (could you explain ?)
}

fmt.Println("employee points:", employeePoint)

delete(employeePoint, "Yauri")
fmt.Println("employee points:", employeePoint)
```

Could you explain this code snippet?



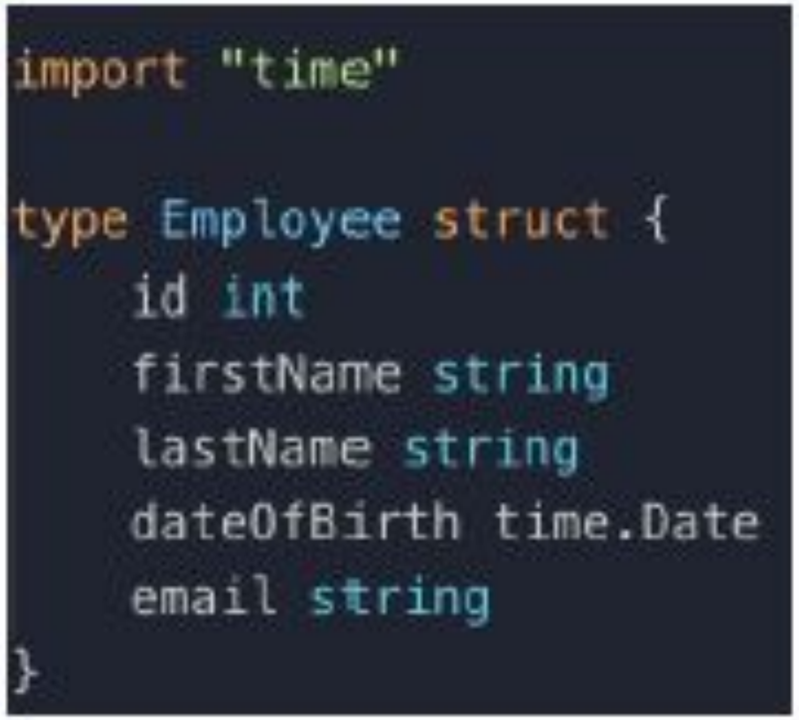
```go
employeePoint := map[string]float64{
    "Yauri": 100.0,
    "Jimmy": 95.0, // the comma is necessary (could you explain ?)
}

fmt.Println("employee points:", employeePoint)

delete(employeePoint, "Yauri")
fmt.Println("employee points:", employeePoint)

copyOfEmployeePoint := employeePoint

copyOfEmployeePoint["Jimmy"] = 100.0

fmt.Println("employee points:", employeePoint)
```

Range

Struct

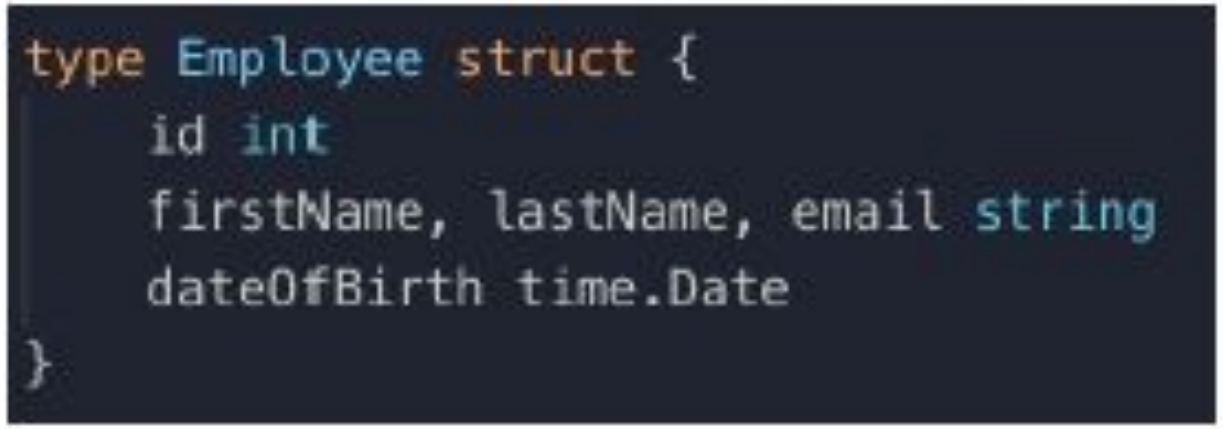
Struct Definition

⇒ a user defined type which represents a collection of fields (like a record in the database table)



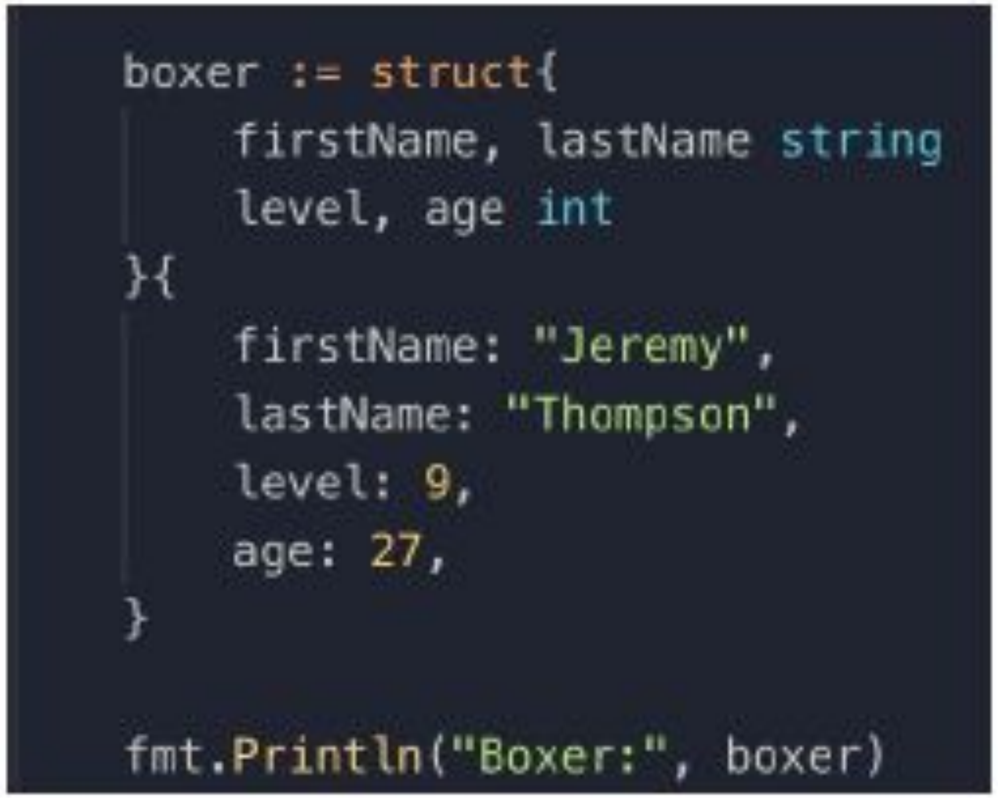
```go
import "time"

type Employee struct {
    id int
    firstName string
    lastName string
    dateOfBirth time.Date
    email string
}
```

OR

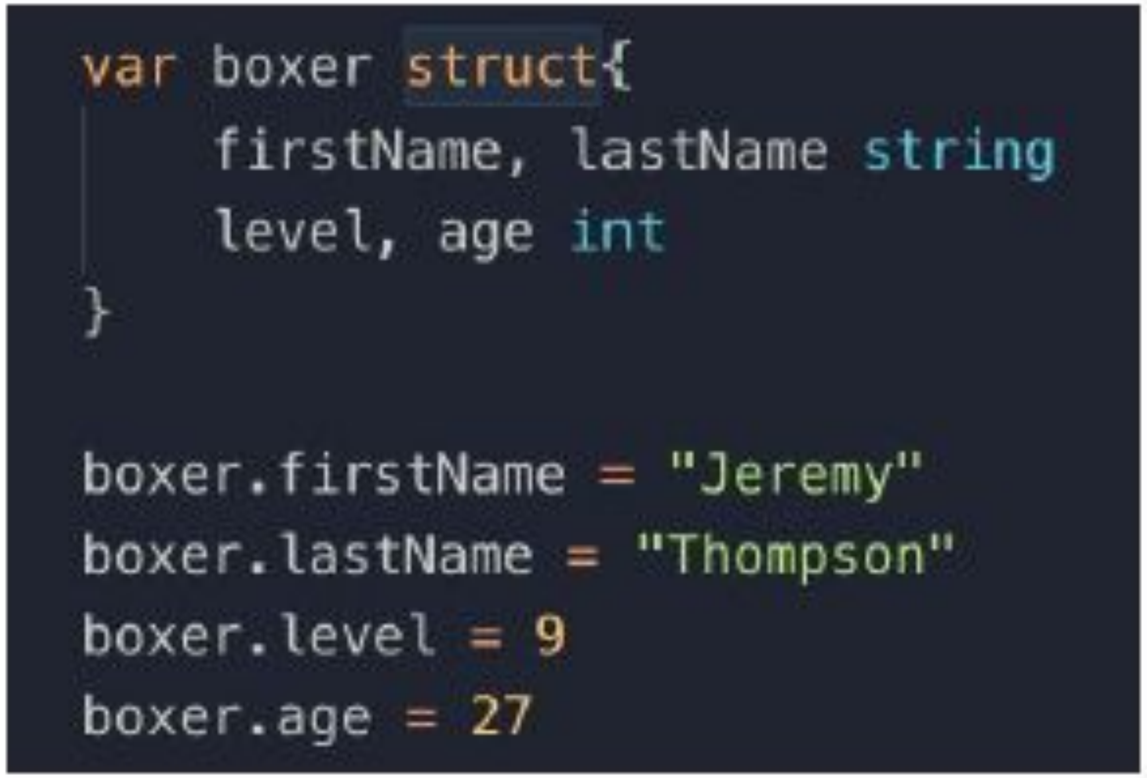


```go
type Employee struct {
    id int
    firstName, lastName, email string
    dateOfBirth time.Date
}
```

Anonymous Structure

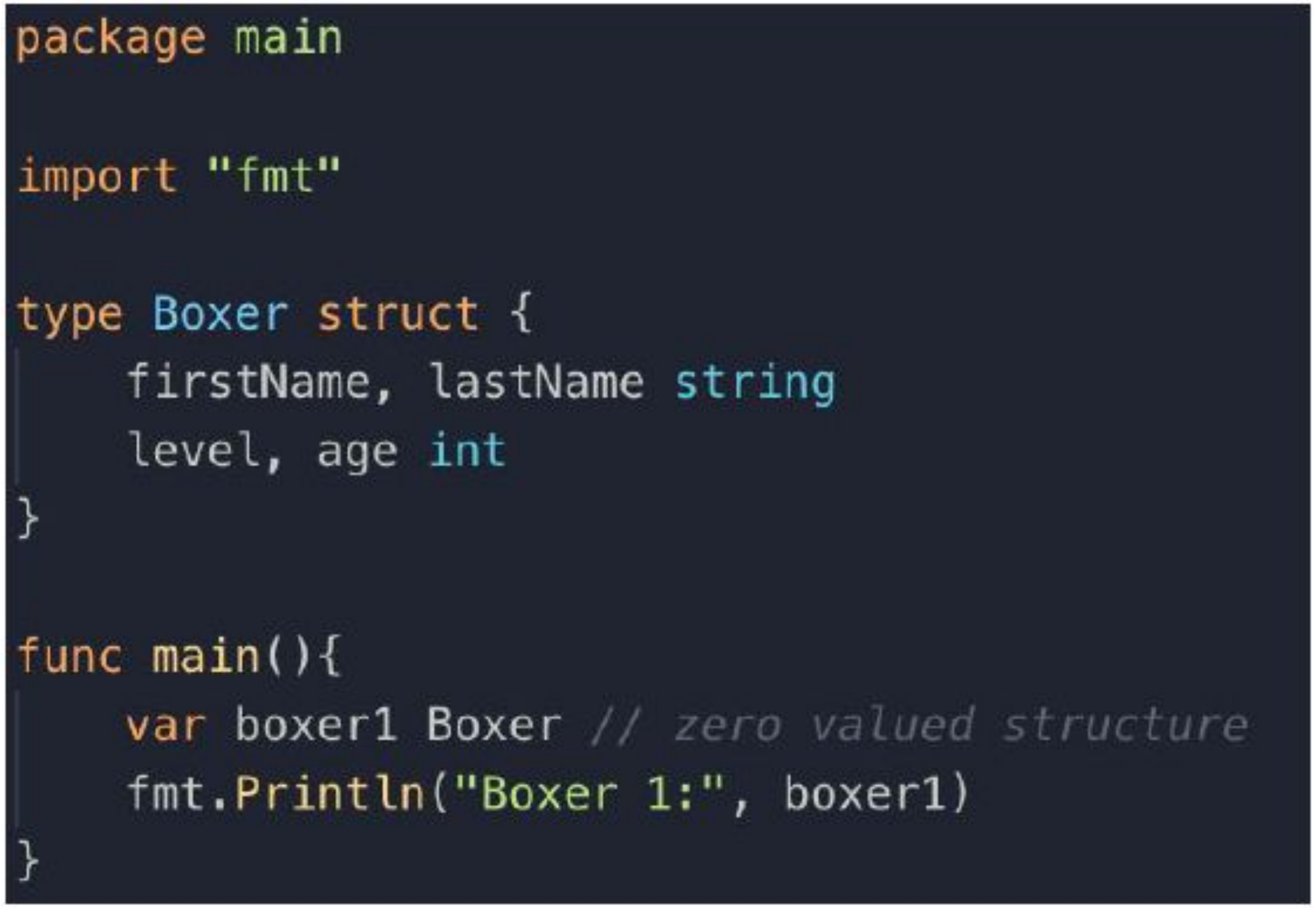


```go
boxer := struct{
    firstName, lastName string
    level, age int
}{
    firstName: "Jeremy",
    lastName: "Thompson",
    level: 9,
    age: 27,
}

fmt.Println("Boxer:", boxer)
```



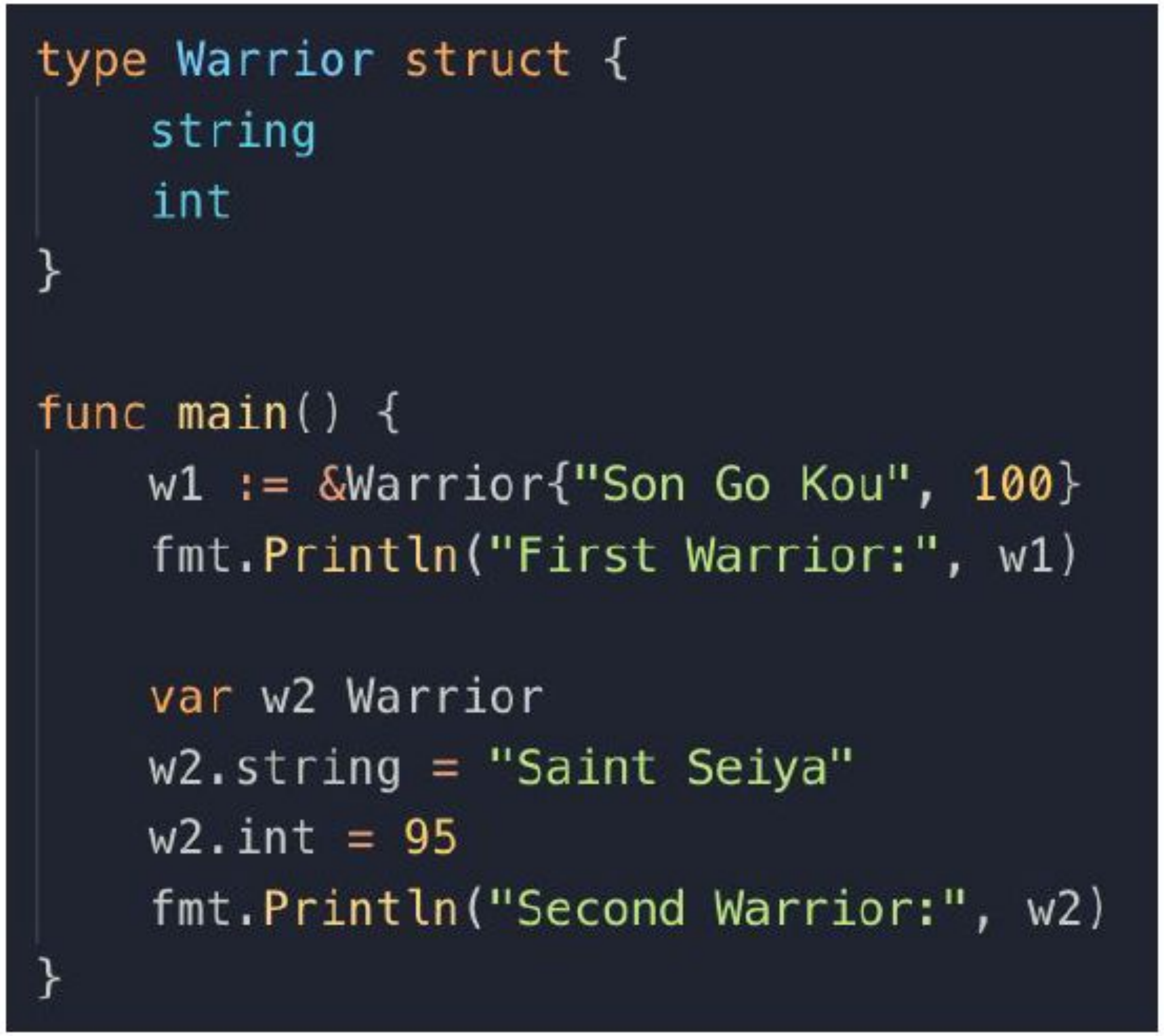
```go
var boxer struct{
    firstName, lastName string
    level, age int
}

boxer.firstName = "Jeremy"
boxer.lastName = "Thompson"
boxer.level = 9
boxer.age = 27
```

Zero Valued Structure



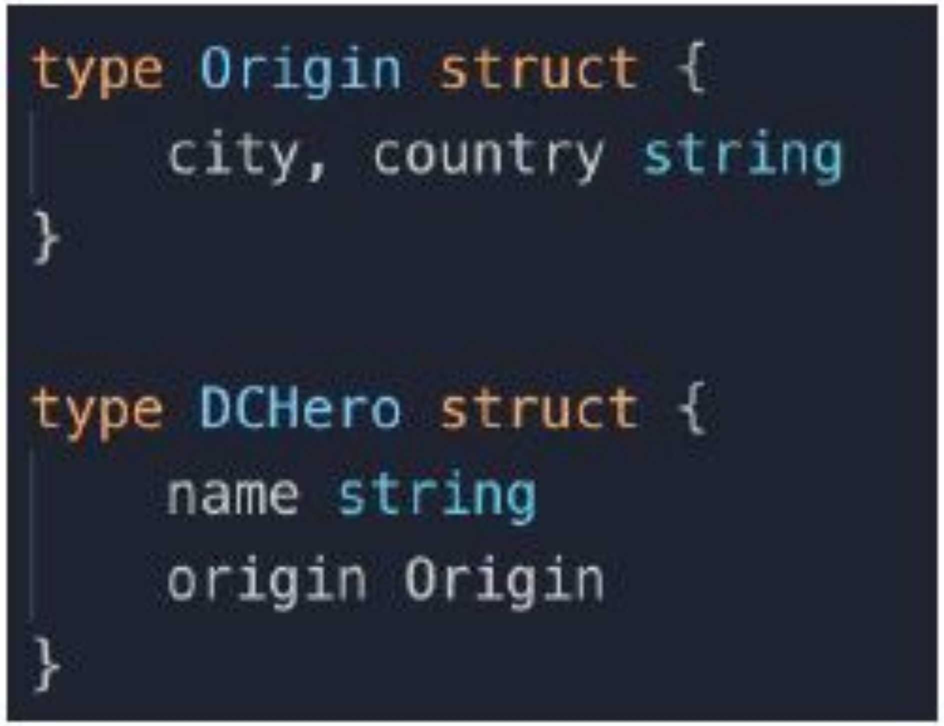
```go
package main

import "fmt"

type Boxer struct {
    firstName, lastName string
    level, age int
}

func main(){
    var boxer1 Boxer // zero valued structure
    fmt.Println("Boxer 1:", boxer1)
}
```

Anonymous Struct Fields

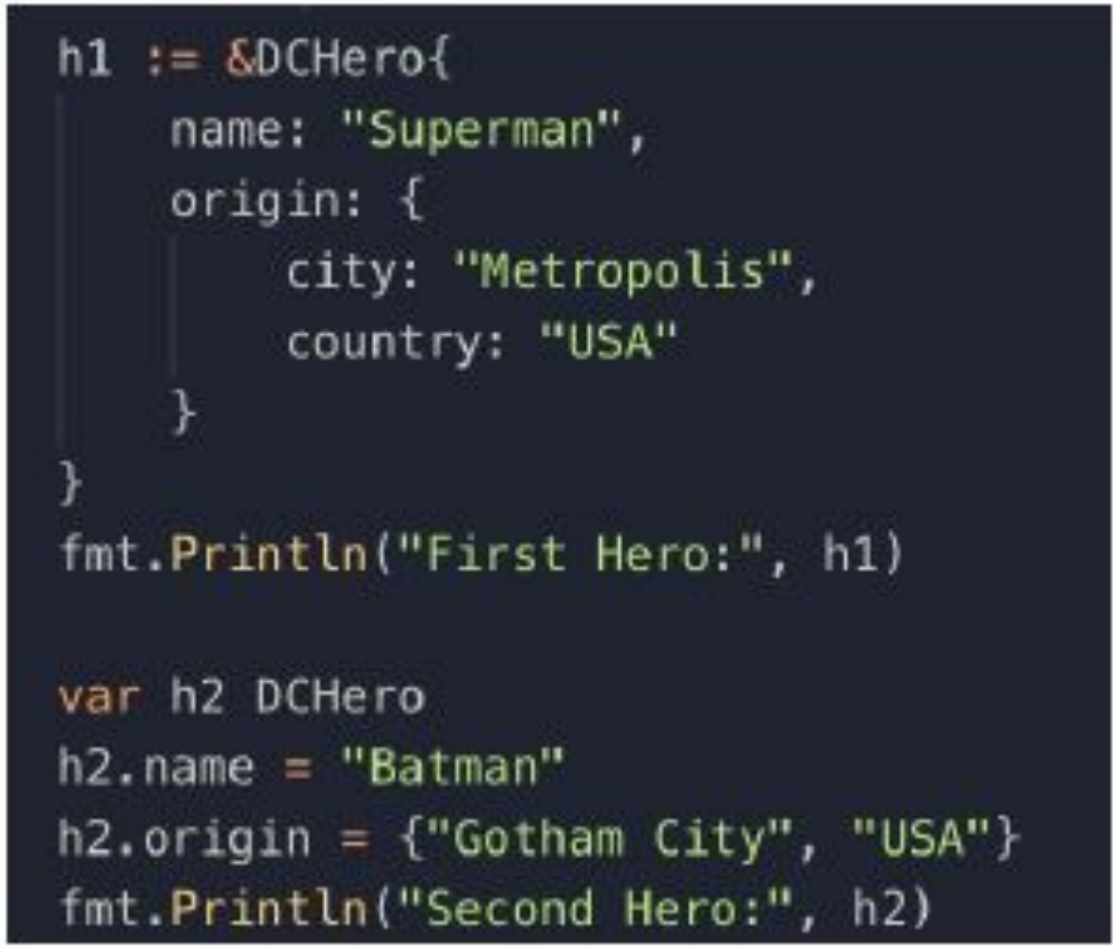


```go
type Warrior struct {
    string
    int
}

func main() {
    w1 := &Warrior{"Son Go Kou", 100}
    fmt.Println("First Warrior:", w1)

    var w2 Warrior
    w2.string = "Saint Seiya"
    w2.int = 95
    fmt.Println("Second Warrior:", w2)
}
```

Nested Struct

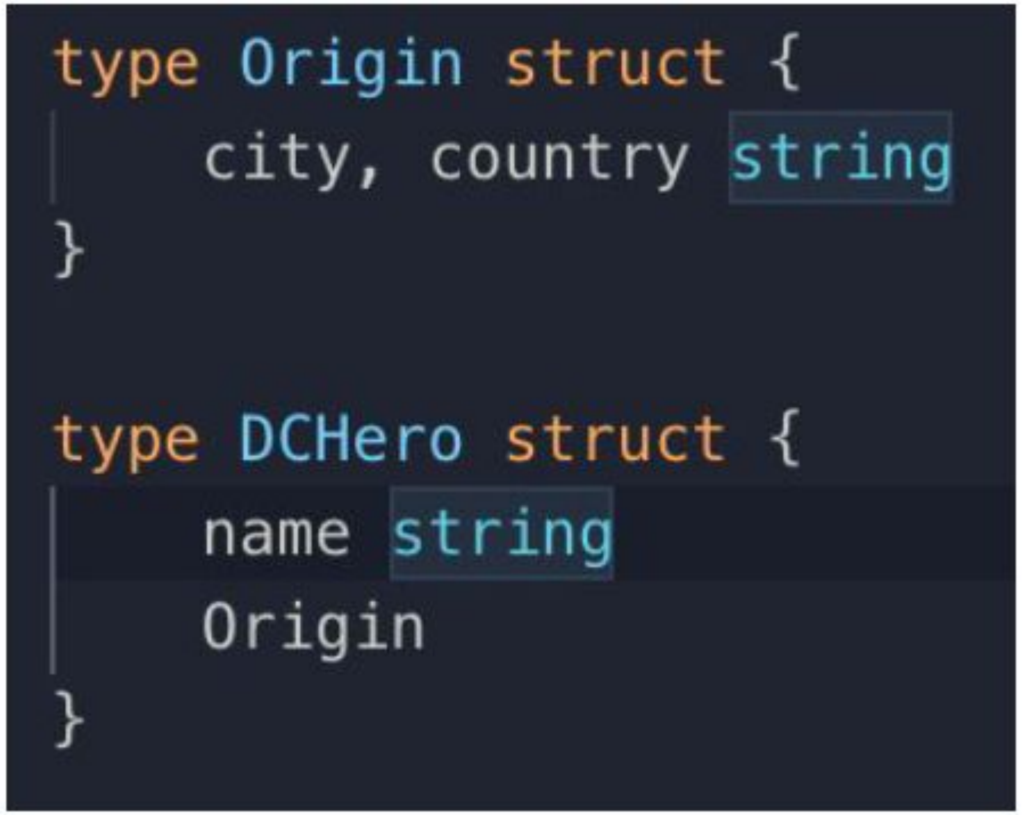


```go
type Origin struct {
    city, country string
}

type DCHero struct {
    name string
    origin Origin
}
```

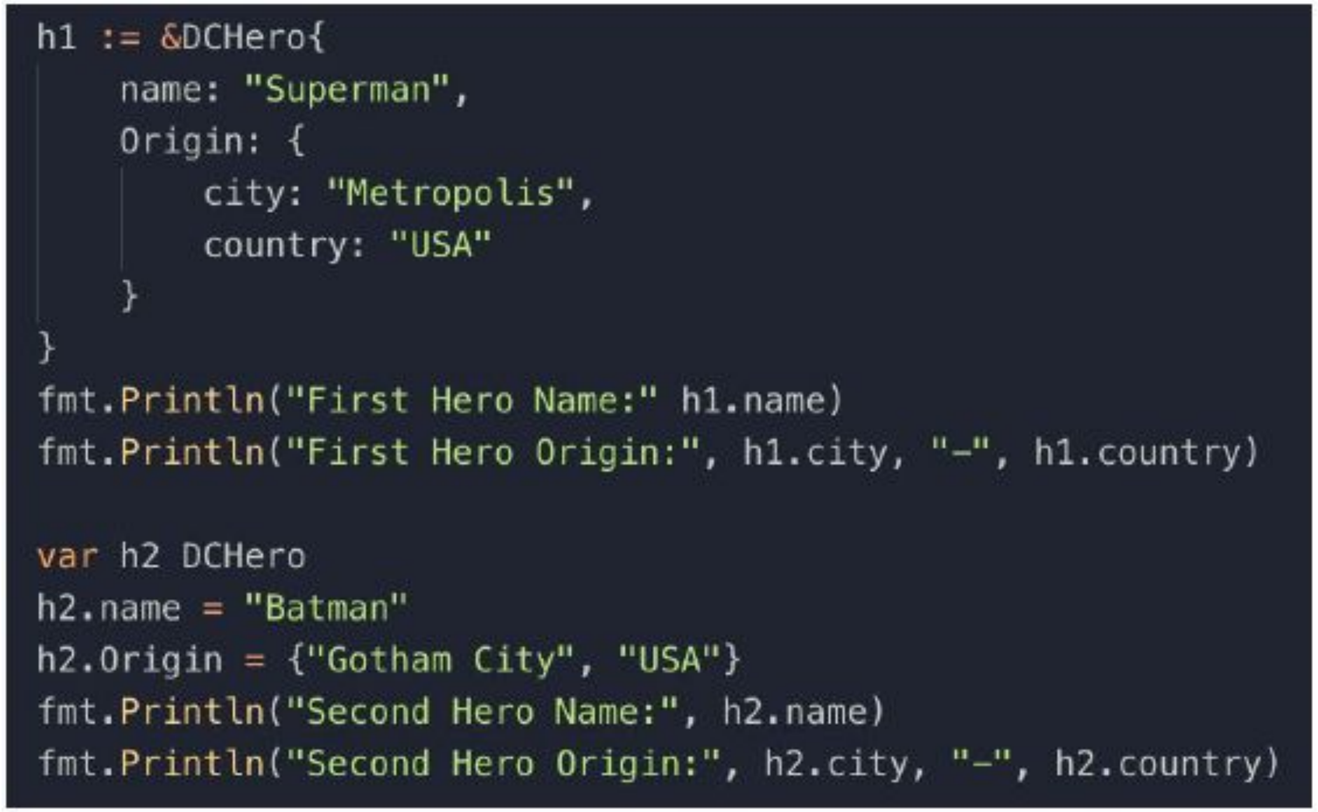


```go
h1 := &DCHero{
    name: "Superman",
    origin: {
        city: "Metropolis",
        country: "USA"
    }
}
fmt.Println("First Hero:", h1)

var h2 DCHero
h2.name = "Batman"
h2.origin = {"Gotham City", "USA"}
fmt.Println("Second Hero:", h2)
```

Promoted Fields



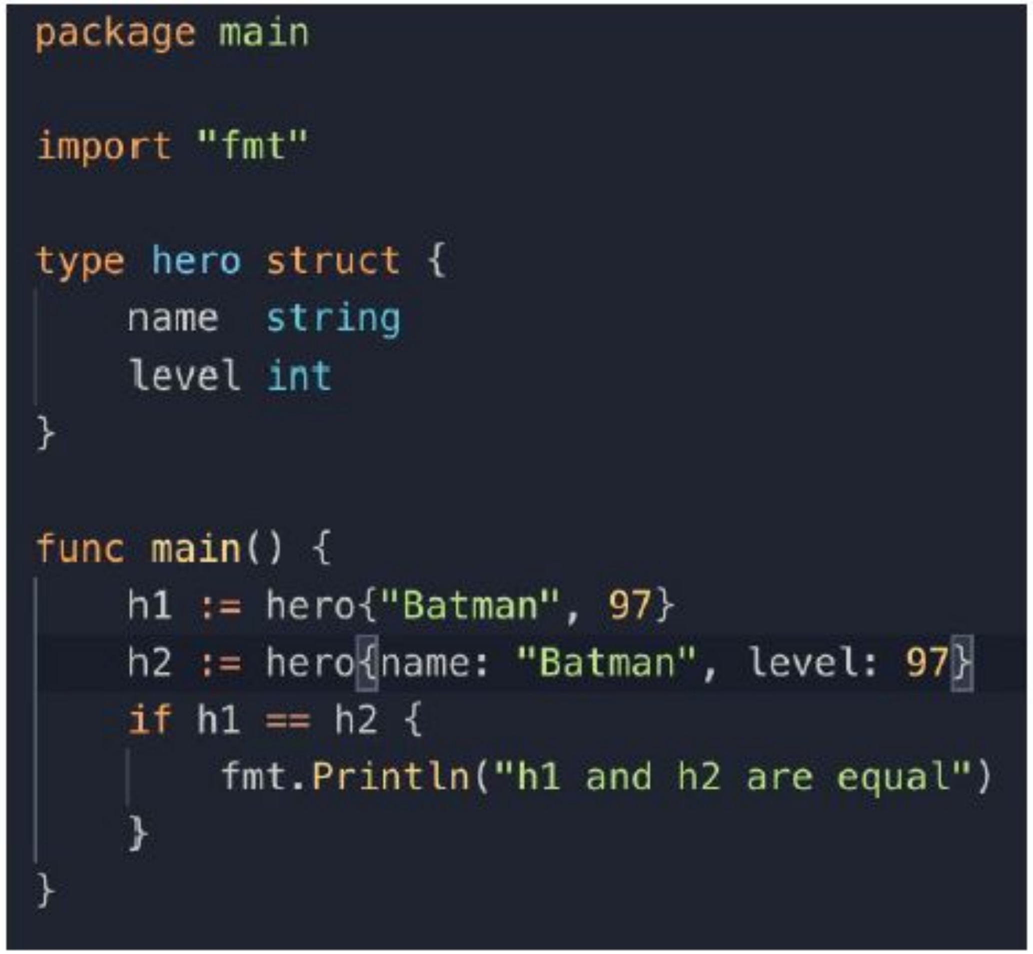
```go
type Origin struct {
    city, country string
}

type DCHero struct {
    name string
    Origin
}
```



```go
h1 := &DCHero{
    name: "Superman",
    Origin: {
        city: "Metropolis",
        country: "USA"
    }
}
fmt.Println("First Hero Name:" h1.name)
fmt.Println("First Hero Origin:", h1.city, "-", h1.country)

var h2 DCHero
h2.name = "Batman"
h2.Origin = {"Gotham City", "USA"}
fmt.Println("Second Hero Name:", h2.name)
fmt.Println("Second Hero Origin:", h2.city, "-", h2.country)
```

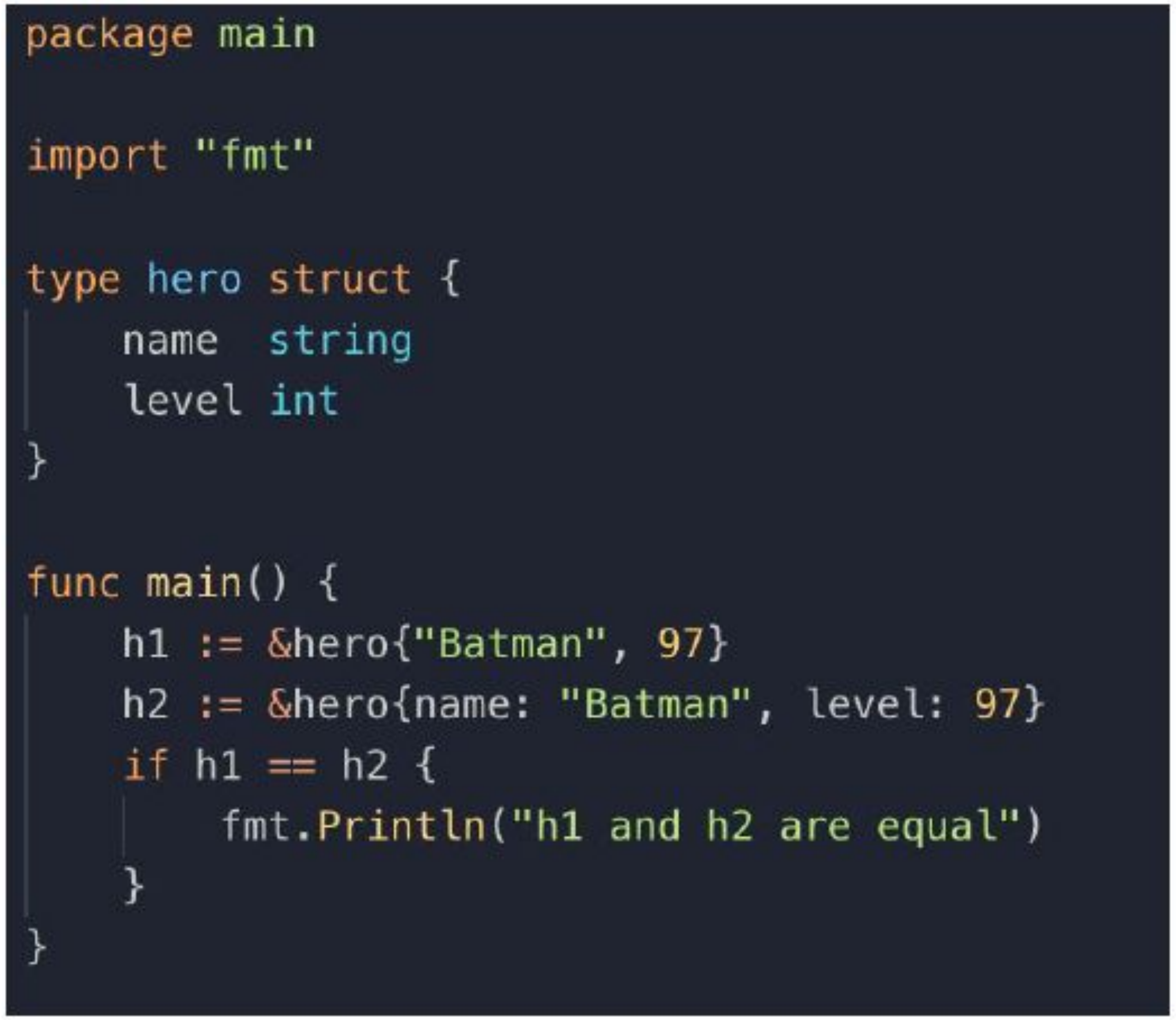
Structs Equality

⇒ 2 structs are comparable if and only if all of their fields have comparable types!!!



```go
package main

import "fmt"

type hero struct {
    name  string
    level int
}

func main() {
    h1 := hero{"Batman", 97}
    h2 := hero{name: "Batman", level: 97}
    if h1 == h2 {
        fmt.Println("h1 and h2 are equal")
    }
}
```

Could you guess the output? (please explain)



```go
package main

import "fmt"

type hero struct {
    name  string
    level int
}

func main() {
    h1 := &hero{"Batman", 97}
    h2 := &hero{name: "Batman", level: 97}
    if h1 == h2 {
        fmt.Println("h1 and h2 are equal")
    }
}
```

Function

Check out
https://exercism.io

SUMMARY

You have studied various built-in basic and composite data types, control statements, function, and how to use them all together in order to solve your problem with Go.

Thank You