1. **Maps in Go**

Maps are Go’s built-in associative data type (sometimes called hashes or dicts in other languages). To create an empty map, use the built-in make: make(map[key-type]val-type) like **var mymap map[int]string**.

// An Empty map

map[Key\_Type]Value\_Type{}

// Map with key-value pair

map[Key\_Type]Value\_Type{key1: value1, ..., keyN: valueN}

**Example # 1**

package main

import "fmt"

func main() {

var map\_1 map[int]int

if map\_1 == nil {

fmt.Println("True")

} else {

fmt.Println("False")

}

map\_2 := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

fmt.Println("Map-2: ", map\_2)

}

**Example # 2**

package main

import "fmt"

func main() {

var My\_map = make(map[float64]string)

fmt.Println(My\_map)

My\_map[1.3] = "Rohit"

My\_map[1.5] = "Sumit"

fmt.Println(My\_map)

}

**Example # 3**

Package main

Import “fmt”

func main() {

var mymap = make(map[string]string)

mymap["1"] = "1"

mymap["2"] = "2"

mymap["3"] = "3"

d, e := mymap["4"] // If key doesnot exist then it will give a response

if e != true {

fmt.Println("Key Doesnot Exist")

}

fmt.Println(d)

1. **Range in Go**

Range iterates over elements in a variety of data structures. Let’s see how to use range with some data structures we’ve already learned. The range on arrays and slices provides both the index and value for each entry. Above we didn’t need the index, so we ignored it with the blank identifier \_. Sometimes we want the indexes, though. The range on the map iterates over key/value pairs.

package main

import "fmt"

func main() {

nums := []int{2, 3, 4}

sum := 0

for \_, num := range nums {

sum += num

}

fmt.Println("sum:", sum)

for i, num := range nums {

if num == 3 {

fmt.Println("index:", i)

}

}

kvs := map[string]string{"a": "apple", "b": "banana"}

for k, v := range kvs {

fmt.Printf("%s -> %s\n", k, v)

}

for k := range kvs {

fmt.Println("key:", k)

}

for i, c := range "go" {

fmt.Println(i, c)

}

}

1. **Recursion in Go**

Go supports recursive functions. This fact function calls itself until it reaches the base case of fact(0). Closures can also be recursive, but this requires the closure to be declared with a typed var explicitly before it’s defined. Since fib was previously declared in main, Go knows which function to call with fib here.

package main

import "fmt"

func fact(n int) int {

if n == 0 {

return 1

}

return n \* fact(n-1)

}

func main() {

fmt.Println(fact(7))

}

Pointers in Go

Go support pointers, allowing you to pass references to values and records within your program. We’ll show how pointers work in contrast to values with two functions: zeroval and zeroptr. zeroval has an int parameter so that arguments will be passed to it by value. zeroval will get a copy of ival distinct from the one in the calling function. zeroptr in contrast, has an \*int parameter, meaning that it takes an int pointer. The \*iptr code in the function body then dereferences the pointer from its memory address to the current value at that address. Assigning a value to a dereferenced pointer changes the value at the referenced address. The &i syntax gives the memory address of i, i.e. a pointer to i. zeroval doesn’t change the i in main, but zeroptr does because it has a reference to the memory address for that variable.

**Example#1**

package main

import "fmt"

func zeroval(ival int) {

ival = 0

}

func zeroptr(iptr \*int) {

\*iptr = 0

}

func main() {

i := 1

fmt.Println("initial:", i)

zeroval(i)

fmt.Println("zeroval:", i)

zeroptr(&i)

fmt.Println("zeroptr:", i)

fmt.Println("pointer:", &i)

}

Output of Pointers

initial: 1

zeroval: 1

zeroptr: 0

pointer: 0x42131100

1. **Strings and Runes in Go**

A Go string is a read-only slice of bytes. The language and the standard library treat strings especially - as containers of text encoded in UTF-8. In other languages, strings are made of “characters”. In Go, the concept of a character is called a rune - it’s an integer that represents a Unicode code point. s is a string assigned a literal value representing the word “hello” in the Thai language. Go string literals are UTF-8 encoded text. Since strings are equivalent to []byte, this will produce the length of the raw bytes stored within. Since strings are equivalent to []byte, this will produce the length of the raw bytes stored within. Indexing into a string produces the raw byte values at each index. This loop generates the hex values of all the bytes that constitute the code points in s. To count how many runes are in a string, we can use the utf8 package. Note that the run-time of RuneCountInString depends on the size of the string, because it has to decode each UTF-8 rune sequentially.

**Example#1**

package main

import (

"fmt"

"unicode/utf8"

)

func main() {

const s = "สวัสดี"

fmt.Println("Len:", len(s))

for i := 0; i < len(s); i++ {

fmt.Printf("%x ", s[i])

}

fmt.Println()

fmt.Println("Rune count:", utf8.RuneCountInString(s))

for idx, runeValue := range s {

fmt.Printf("%#U starts at %d\n", runeValue, idx)

}

fmt.Println("\nUsing DecodeRuneInString")

for i, w := 0, 0; i < len(s); i += w {

runeValue, width := utf8.DecodeRuneInString(s[i:])

fmt.Printf("%#U starts at %d\n", runeValue, i)

w = width

examineRune(runeValue)

}

}

func examineRune(r rune) {

if r == 't' {

fmt.Println("found tee")

} else if r == 'ส' {

fmt.Println("found so sua")

}

}

Output of Strings

Len: 18

e0 b8 aa e0 b8 a7 e0 b8 b1 e0 b8 aa e0 b8 94 e0 b8 b5

Rune count: 6

U+0E2A 'ส' starts at 0

U+0E27 'ว' starts at 3

U+0E31 'ั' starts at 6

U+0E2A 'ส' starts at 9

U+0E14 'ด' starts at 12

U+0E35 'ี' starts at 15

Using DecodeRuneInString

U+0E2A 'ส' starts at 0

found so sua

U+0E27 'ว' starts at 3

U+0E31 'ั' starts at 6

U+0E2A 'ส' starts at 9

found so sua

U+0E14 'ด' starts at 12

U+0E35 'ี' starts at 15

1. **Structs in Go**

Go’s structs are typed collections of fields. They’re useful for grouping data together to form records. This person struct type has name and age fields. newPerson constructs a new person struct with the given name. You can safely return a pointer to a local variable as a local variable will survive the scope of the function.

**Example#1**

package main

import "fmt"

type person struct {

name string

age int

}

func newPerson(name string) \*person {

p := person{name: name}

p.age = 42

return &p

}

func main() {

fmt.Println(person{"Bob", 20})

fmt.Println(person{name: "Alice", age: 30})

fmt.Println(person{name: "Fred"})

fmt.Println(&person{name: "Ann", age: 40})

fmt.Println(newPerson("Jon"))

s := person{name: "Sean", age: 50}

fmt.Println(s.name)

sp := &s

fmt.Println(sp.age)

sp.age = 51

fmt.Println(sp.age)

}

Output of Structus

{Bob 20}

{Alice 30}

{Fred 0}

&{Ann 40}

&{Jon 42}

Sean

50

51