Chapter 3

Objective

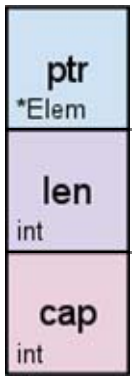
- Slices in Golang
- Slice Operator Sysntax
- Using slice **operator** (start:end)
- Golang make Function
- Difference between the make Function and new Function
- Creating Slice Using Built-In make Function
- Passing Slice as Function Argument
- Using append function
- Multi-Dimensional Slices
- Blank identifier (underscore)
- Range in Golang
- Maps in Golang
- Adding items to a map
- Retrieving Map Items
- Check For Map Key
- Iterating Map Elements
- Deleting Map Items
- Maps are Similar to Slice
- User Defined Data Structures
- Declaring a structure
- Creating named structures
- Creating anonymous structures
- Zero value of a structure
- Accessing individual fields of a struct

This page is left blank intentionally

Unit 1

Slices in Golang

- **Slice** gives a powerful interface to store data in **underlying** array.
- The slice is a **three-item** descriptor containing a **pointer** to the data (inside an array), the **length**, and the **capacity**.



- The slice is **nil** unless three-item descriptor is **initialized**
- Create a **nil** slice using following syntax, slice is un-initialized:

```
var aSlice []int  // not referencing underlying array
```

- The slice type is an abstraction built on top of Golang's **array type**
- Unlike an array type, a slice type has **no specified** length, `[]int`.
- The **empty slice** has an underlying array created.
- The slice descriptor pointer points to underlying array.
- Use a slice **composite literal** to create an empty slice of integers.

```
aSlice := []int{}  ({} is a literal)
```

- **Composite literal** consist of a **type** of the literal followed by a **brace**-bound list of elements.

```
elements := []int{1, 2, 3, 4}
```

- Declare and initialize a variable for slice in a single line using slice literal.

```
letters := []string{"g", "h", "i"}
```

- A slice literal is declared just like an array literal, except there is **no element count**

```
letters := []string{"a", "b", "c", "d"}

cityNames := []string{"San Jose","Santa Clara","Berkeley"}
```

```
 1 package main    // Example 3-1
 2
 3 import "fmt"
 4
 5 func main() {
 6    // Cannot create nil Slice using short notation
 7    // zipLine := []int // Error: expected expression
 8
 9    // nil Slice creation using long notation
10    var zipLineOne []int
11
12    // Slice creation using short notation
13    LineOne := []int{}
14
15    // Slice creation using long notation
16    var LineTwo []int = []int{1, 2}
17
18    fmt.Println("nil =", zipLineOne == nil, "zipLineOne = ",
zipLineOne)
19   fmt.Println("nil =", LineOne == nil, "LineOne = ", LineOne)
20   fmt.Println("nil =", LineTwo == nil, "LineTwo = ", LineTwo)
21 }
```

Output:

```
        nil = true zipLineOne =  []
        nil = false LineOne =  []
        nil = false LineTwo =  [1 2]
```

Unit 2

Slice Operator Sysntax

- Slices operator supports a "slice" **`mySlice[start:end]`**.
- Get a slice of the elements **`mySlice[2], mySlice[3], and mySlice[4].`**

```
mySlice := mySlice[2:5] // start:end, 3rd,4th & 5th element
```

- When slicing with operator [start:end], start >= 0; end <= capacity **.**
- Get a slice up to (but excluding) **`mySlice[5].`**

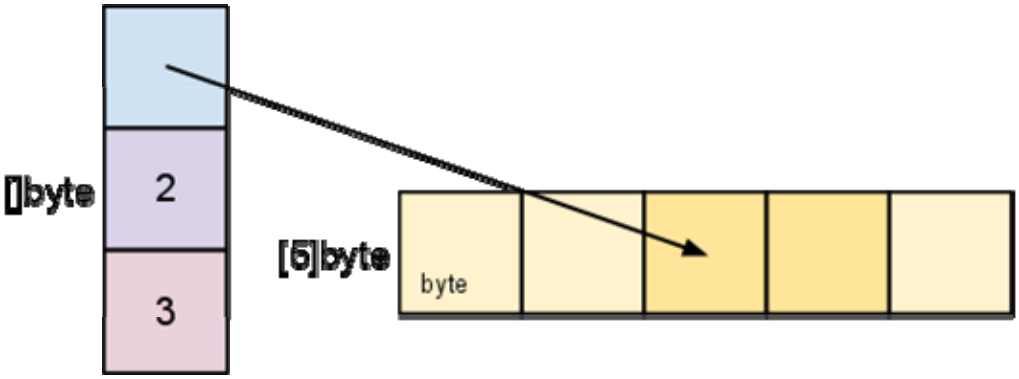
```
mySlice = mySlice[:5]
```

- Get a slice up from (and including) mySlice[2].

```
mySlice = mySlice[2:]
```

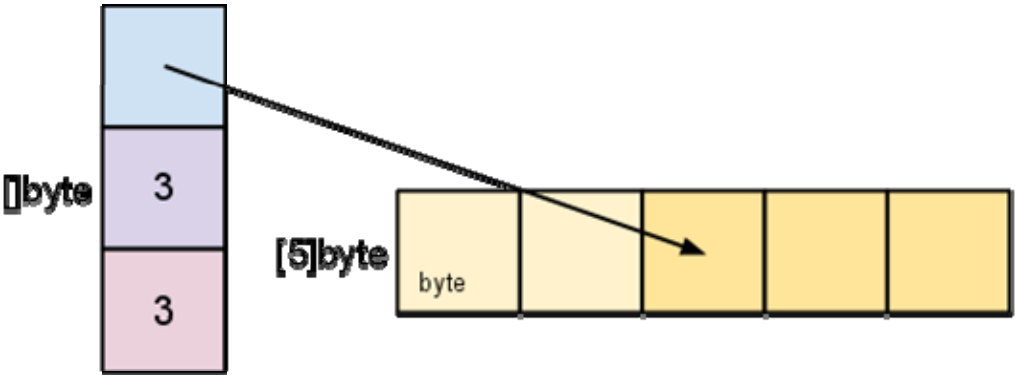
- Slicing does not copy the slice's data but it creates a new slice value that points to the original underling array.

```
mySlice = mySlice[2:3] // 3rd element
```



- Grow mySlice to its capacity by slicing it again:

```
mySlice = mySlice[:cap(mySlice)]
```



Using slice operator (start:end)

- Show the usage of slice operator

```
 1 package main    // Example 3-2
 2
 3 import "fmt"
 4
 5 func main() {
 6    intArray := [5]int{12, 34, 55, 66, 43}
 7
 8    var slice []int // Create a nil slice
 9    fmt.Println("Slice: ", slice == nil)
10
11    slice = intArray[:] // slice of all array elements
12
13   fmt.Printf("the len is %d and cap is %d \n", len(slice), cap(slice))
14    fmt.Printf("Address of slice %p\n", &slice[0])
15    fmt.Printf("Address of Array %p\n", &intArray)
16    fmt.Println("Slice Data: ", slice)
17    fmt.Println("Array Data: ", intArray)
18
19 // slice = slice[2:1] // invalid slice index: 2 > 1
20 // slice = slice[2:6] // panic: runtime error: slice bounds out of range
21 // slice = slice[2:2] //  []  (empty slice)
22   slice = slice[2:3] // [55] (only one element in a slice, up to 3rd element)
23    fmt.Println("Slice Data: ", slice)
24 }
```

Output:

```
Slice:  true
the len is 5 and cap is 5
Address of slice 0xc000018240
Address of Array 0xc000018240
Slice Data:  [12 34 55 66 43]
Array Data:  [12 34 55 66 43]
Slice Data:  [55]
```

- Create slice from underlying array using slice operator

```
 1 package main   // Example 3-3
 2 import "fmt"
 3
 4 func main() {
 5    var aSlice []int // slice is nil, no underlying array
 6
 7    fmt.Println("aSlice == nil", aSlice == nil)
 8
 9    // create an array of int
10    intArray := [9]int{10,20,30,40,50,60,70,80,90}
11    //     index value 0  1  2  3  4  5  6  7  8
11
12    // get slice from intArray
13    // start:end, 3rd,4th & 5th element [30,40,50]
14   aSlice = intArray[2:5] // slice operator (start:end)
15   fmt.Println("aSlice == nil", aSlice == nil, "and aSlice = ", aSlice)
16   fmt.Println("Address of aSlice[0]: ", &aSlice[0])
17   fmt.Println("Address of intArray[2]: ", &intArray[2])
18   fmt.Println("Value at aSlice[0]: ", aSlice[0])
19   fmt.Println("Value at intArray[2]: ", intArray[2])
20 }
21
22
```

```
Output:
        aSlice == nil true
        aSlice == nil false and aSlice =  [30 40 50]
        Address of aSlice[0]:  0xc000020100
        Address of intArray[2]:  0xc000020100
        Value at aSlice[0]:  30
        Value at intArray[2]:  30
```

Unit 3

Golang make Function

- The **make()** function is a special built-in function that is used for initializing **slices, maps, and channels**.

Difference between the make Function and new Function

- Unlike the **new()** function, **make()** function does not return a **pointer**
- The **built-in** function **make** (T, args) serves a purpose different from new(T) function.
- The make function creates **slices, maps, and channels** only, and it returns an initialized (not zeroed) value of type T (not *T).
- The built-in new(T) function allocates "zeroed" storage for a new item of type T.
- After allocation of storage, **new** function **returns its address**, a value of type ***T**
- The new function returns a pointer to a newly allocated zero value of type T.

Creating Slice Using Built-In make Function

- Slices, maps, and channels can also be initialized using a **composite literal** expressions, as well as with **make()** function.
- Create slice using **composite literal**

```
elements := []int{1, 2, 3, 4}
```

- A slice can be created with the **built-in make() function**, which has the signature:

```
func make([]T, len, cap) []T
```

- T stands for the element type of the slice to be created
- The **make** function takes a type, a length, and an optional capacity.
- To create an empty slice with non-zero length, use the built-in **make**.

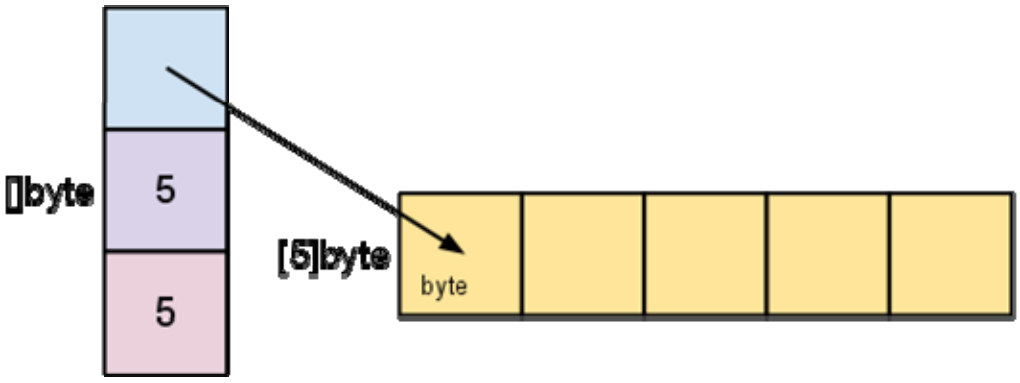
```
mySlice := make([]string, 3)
```

- When make is called it **allocates an array** and returns a slice that refers to that array.

```
var mySlice []byte // mySlice is nil, empty descriptor
mySlice = make([]byte, 5, 5)
```

- When the capacity argument is omitted, it defaults to the specified length. Here's a more compact version of the same code:

```
mySlice := make([]byte, 5)
```



- The length and capacity of a slice can be inspected using the built-in **len** and **cap** functions.

```
len(mySlice) == 5
cap(mySlice) == 5
```

- The length is the number of elements referred to by the slice.
- The capacity is the number of elements in the underlying array (beginning at the element referred to by the slice pointer).
- A slice cannot be grown **beyond** its **capacity**, it will cause a runtime panic
- When indexing **outside** the bounds of a slice or array, it will also cause a runtime panic
- Slices cannot be **re-sliced** below zero to access earlier elements in the array.

- Unlike arrays, slices are typed only by the elements they contain (not the number of elements).

- Slices support several more operations that make them richer than arrays.

- The built-in **append function**, which returns a slice containing one or more new values.

```
mySlice = append(mySlice, "d")
mySlice = append(mySlice, "e", "f")
```

- Slices can also be copied.
- Create an empty slice **newSlice** of the same length as mySlice and use built-in **copy** function.

```
newSlice := make([]string, len(mySlice))
copy(newSlice, mySlice) // mySlice copied into newSlice
```

```
 1 package main  // Example 3-4
 2 import "fmt"
 3
 4 func main() {
 5    // make a slice of length 3 (initially zero-valued).
 6     mySlice := make([]string, 3) // creates slice descriptor
 7      fmt.Println("mySlice empty:", mySlice)
 8
 9    // Set and get just like with arrays.
10    mySlice[0] = "a"
11    mySlice[1] = "b"
12    mySlice[2] = "c"
13    fmt.Println("mySlice Values:", mySlice)
14    fmt.Println("mySlice[2] Value:", mySlice[2])
15
16    // len returns the length of the slice as expected.
17
18    fmt.Println("mySlice length: ", len(mySlice))
19    fmt.Println("mySlice Capacity: ", cap(mySlice))
20    mySlice = append(mySlice, "d")
21    fmt.Println("mySlice length: ", len(mySlice))
22    fmt.Println("mySlice Capacity: ", cap(mySlice))
23
24    newSlice := make([]string, len(mySlice))
25    copy(newSlice, mySlice)
26    fmt.Println("newSlice length: ", len(newSlice))
27    fmt.Println("newSlice Capacity: ", cap(newSlice))
28 }
29
```

Output:

```
mySlice empty: [  ]
mySlice Values: [a b c]
mySlice[2] Value: c
mySlice length:  3
mySlice Capacity:  3
mySlice length:  4
mySlice Capacity:  6
newSlice length:  4
newSlice Capacity:  4
```

Unit 4

Passing Slice as Function Argument

- In golang slice is pass by value to a function
- Function gets the copy of the slice, but descriptor still will be pointing to underlying array
- Use slice to modify underlying array elements inside the function

```
 1 package main  // Example 3-5
 2 import "fmt"
 3
 4 func modifySlice(aSlice []int) {
 5     aSlice[0] = 90
 6 }
 7
 8 func main() {
 9     intArray := [3]int{89, 90, 91}
10
11     // Pass slice as function argument
12     modifySlice(intArray[:])// using slice operator (:)
13     fmt.Println(intArray)
14 }
15
```

```
Output:
[90 90 91]
```

- Slice pass by value to function **SlicePassByValue()**

```
 1 package main  // Example 3-6
 2 import "fmt"
 3
 4 func SlicePassByValue(sliceFruit []string) {
 5    sliceFruit[0] = "banana"
 6    sliceFruit[1] = "peach"
 7
 8    fmt.Println("Within Function Body:", sliceFruit)
 9   fmt.Printf("Slice address %p within function:\n", &sliceFruit)
10 }
11
12 func main() {
13    mySlice := []string{"apple", "orange"}
14
15    fmt.Printf("Slice address %p within main:\n", &mySlice)
16    fmt.Println("Before Function Call:", mySlice)
17
18    SlicePassByValue(mySlice) // Pass Slice by Value
19
20    fmt.Println("After Function Call:", mySlice)
21 }
```

Output:

```
Slice address 0xc00008a000 within main:
Before Function Call: [apple orange]
Within Function Body: [banana peach]
Slice address 0xc00008a060 within function:
After Function Call: [banana peach]
```

- Slice pass by **Pointer** to function **SlicePassByValue()**

```
 1 package main  // Example 3-7
 2 import "fmt"
 3
 4 // Slice is pass by value to nilSliceFunc
 5 func nilSliceFunc(nilSlice []int) {
 6
 7    fmt.Printf("nilSliceFunc Address %p \n", &nilSlice)
 8 }
 9
10 // Slice is pass by pointer to SLicePassByPointer
11 func SlicePassByPointer(sliceFruit *[]string) {
12    (*sliceFruit)[0] = "banana"
13    (*sliceFruit)[1] = "peach"
14    fmt.Println("Within Function Body:", *sliceFruit)
15   fmt.Printf("Slice address %p within function Body:\n", &sliceFruit)
16 }
17
18 func main() {
19    var nilSlice []int  // create nil slice
20
21    nilSliceFunc(nilSlice)  // pass nilSlice to function
22
23   fmt.Printf("nilSlice Address %p within main\n", &nilSlice)
24
25    mySlice := []string{"apple", "orange"}
26
27    fmt.Printf("mySlice address %p within main:\n", &mySlice)
28    fmt.Println("Before Function Call:", mySlice)
29
30    // pass mySlice by pointer to function
31    SlicePassByPointer(&mySlice)
32    fmt.Println("After Function Call:", mySlice)
33 }
34
```

Output:

```
nilSliceFunc Address 0xc00000c080
nilSlice Address 0xc00000c060 within main
mySlice address 0xc00000c0a0 within main:
Before Function Call: [apple orange]
Within Function Body: [banana peach]
Slice address 0xc00000e030 within function Body:
After Function Call: [banana peach]
```

Using append function

- Go provides a built-in append function to add data to a slice.
- Its signature is: **`func append(s []T, x ...T) []T`**
- The append function appends the elements to the end of the slice and grows the slice if a greater capacity is needed.
- To use append function within the user function the arguments of the function must be pointer to slice, otherwise change will not be seen by the caller.

```
 1 package main  // Example 3-8
 2 import "fmt"
 3
 4 // function argument pass by value
 5 func modifySlice(aSlice []int) {
 6     aSlice[0] = 290
 7     aSlice = append(aSlice, 75)
 8 }
 9
10 func main() {
11     intSlice := []int{89, 90, 91} // Composite literal
12
13     // Pass intSlice to function as argument
14     modifySlice(intSlice) // Pass Slice a value
15     fmt.Println(intSlice)
16     intSlice = append(intSlice, 95)
17     fmt.Println(intSlice)
18 }
19
```

Output:

```
[290 90 91]
[290 90 91 95]
```

- The **append** element within the function **modifySlice** is not available to a intSlice in the **main** function.

```
 1 package main  // Example 3-9
 2 import "fmt"
 3
 4 // function argument pass by pointer to slice
 5 func modifySlice(aSlice *[]int) {
 6     (*aSlice)[0] = 290  // de-reference slice pointer
 7     *aSlice = append(*aSlice, 75)
 8 }
 9
10 func main() {
11     intSlice := []int{89, 90, 91}
12
13     // Pass intSlice to function as argument
14     modifySlice(&intSlice) // Pass Slice address
15     fmt.Println(intSlice)
16     intSlice = append(intSlice, 95)
17     fmt.Println("After append from main:", intSlice)
18 }
19
```

Output:

```
[290 90 91 75]
After append from main: [290 90 91 75 95]
```

- The **append** element within the function **modifySlice** is available to a intSlice in the **main** function.

Unit 5

Multi-Dimensional Slices

- A multi-dimensional array is an array of arrays; similarly multi-dimensional slice is a slice of slices.
- Slices can be composed into multi-dimensional data structures in golang.
- A slice points to an underlying array and is internally represented by a slice descriptor.
- The syntax to declare a two dimensional slice would be:

```
twoDSlice = make([][]int, 2)
```

- Above declaration means that we want to create a slice of 2 slices.
- The length of each of the inner 2 slices has to be explicitly initialized like

```
for row := range twoDSlice {
    twoDSlice[row] = make([]int, 3)
}
```

- **OR** create inner slice following way using composite literal:

```
twoDSlice[0] = []int{1, 2, 3}  // row one
twoDSlice[1] = []int{4, 5, 6}  // row two
```

```
 1 package main // Example 3-10
 2 import "fmt"
 3
 4 func main() {
 5    twoDSlice := make([][]int, 2) // 2 rows
 6    twoDSlice[0] = []int{1, 2, 3} // row one
 7    twoDSlice[1] = []int{4, 5, 6} // row two
 8    fmt.Println()
 9    fmt.Printf("rows in slice: %d\n", len(twoDSlice))
10    fmt.Printf("columns : %d\n", len(twoDSlice[0]))
11   fmt.Printf("Total elements: %d\n", len(twoDSlice)*len(twoDSlice[0]))
12    fmt.Println("Row One", twoDSlice[0])
13    fmt.Println("Row two", twoDSlice[1])
14
15    for i, row := range twoDSlice {
16       for j, _ := range row {
17          fmt.Print(twoDSlice[i][j], " ")
18       }
19
20       fmt.Println(" ")
21    }
22 }
```

Output:

```
rows in slice: 2
columns : 3
Total elements: 6
Row One [1 2 3]
Row two [4 5 6]
1 2 3
4 5 6
```

Unit 6

Blank identifier (underscore)

- The blank identifier _ is an anonymous placeholder.
- It may be used like any other identifier in a declaration, but it does not introduce a binding.
- The blank identifier provides a way to ignore left-hand side values in an assignment.
- It can also be used to import a package solely for its side effects.

```
import _ "image/png" // init png decoder function
```

- Blank identifier is used during development to avoid compiler errors about unused imports and variables in a half-written program

Range in Golang

- The range keyword is used in for loop to iterate over items of an array, slice, channel or map.
- With array and slices, **range** returns the index of the item as integer.
- With maps, **range** returns the key of the next key-value pair.

```
 1 package main  // Example 3-11
 2 import "fmt"
 3
 4 func main() {
 5
 6    // Use range to sum the numbers in a slice.
 7    // Arrays work like this too.
 8
 9    sum := 0
10    numSlice := []int{2, 3, 4}
11
12    for _, eachNum := range numSlice {
13       sum += eachNum
14    }
15
16    fmt.Println("Total:", sum)
17 }
18

Output:
Total: 9
```

- The range on arrays and slices provides both the index and value for each entry.
- Above example does not need the index, ignored it with the **blank identifier** _.
- If index is needed use for loop index variable.

```
 1 package main  // Example 3-12
 2 import "fmt"
 3
 4 func main() {
 5    numSlice := []int{2, 3, 4}
 6
 7    for index, eachNum := range numSlice {
 8       if eachNum == 3 {
 9          fmt.Println("index:", index)
10       }
11    }
12 }
13
```

```
Output:
index: 1
```

Unit 7

Maps in Golang

- Go provides a built-in **map type** that implements a **hash table**.
- Maps in Go associates a **value** to a **key-type**
- The value can be retrieved using the corresponding key-type
- Map can be created using **composite literal** or using **make** function.

```
keys := map[int]string{100:"Jim",200:"Joe",300:"Kim"}
```

- A map can be created by passing the type of key and value to the **make function**.

- **make**(map[**type of key**]**type of value**) is the syntax to create a map.

```
personSalary := make(map[string]int)
```

- The above line of code creates a map named **personSalary** which has string keys and int values.

- It's not necessary that only string types should be keys.

- All comparable types such as boolean, integer, float, complex, string, ... can also be keys.

- The **zero value** of a map is **nil**.

```
var m map[string]string
```

- Items **cannot** be added to **nil** map, it will cause **run time panic**.
- Use **make** function to initialize the zero value map OR use **composite literal**

```
m = make(map[string]string)
OR  m = map[string]string{}
```

- Two maps **cannot** be compared using the == operator.

- The == can be only used to check if a map is **nil**.

- Length of the map can be determined using the **len** function.

```
 1 package main  // Example 3-13
 2 import "fmt"
 3
 4 func main() {
 5    var personSalary map[string]int //personSalary is nil map
 6     if personSalary == nil {
 7        fmt.Println("map is nil. Going to make one.")
 8
 9        // initialized using the make function.
10        personSalary = make(map[string]int)
11     }
12
13     // len(personSalary) determines the length of the map.
14     fmt.Println("length is", len(personSalary))
15 }
16
```

```
Output:
        map is nil. Going to make one.
        length is 0
```

Adding items to a map

- The syntax for adding new items to a map is the same as that of arrays.

```
 1 package main  // Example 3-14
 2 import "fmt"
 3
 4 func main() {
       // Create and initialize personSalary map
 5     personSalary := make(map[string]int)
 6     personSalary["steve"] = 12000
 7     personSalary["jamie"] = 15000
 8     personSalary["mike"] = 9000
 9     fmt.Println("personSalary map contents:", personSalary)
10 }
11
```

```
Output:
personSalary map contents: map[jamie:15000 mike:9000 steve:12000]
```

- Initialize a map during declaration using composite literal
- Create personSalary and add two elements to it during declaration to itself.

```
 1 package main  // Example 3-15
 2 import "fmt"
 3
 4 func main() {
 5     //  using composite literal to create and initialize personSalary map
 6     personSalary := map[string]int {
 7         "steve": 12000,
 8         "jamie": 15000,
 9     }
10
11     fmt.Println("personSalary map contents before:", personSalary)
12      personSalary["mike"] = 9000
13     fmt.Println("personSalary map contents after:", personSalary)
14 }
15
```

```
Output:
personSalary map contents before: map[jamie:15000 steve:12000]
personSalary map contents after: map[jamie:15000 mike:9000
steve:12000]
```

Retrieving Map Items

- **map[key]** is the syntax to retrieve elements of a map.
- If element is not present the map will return zero value of the type of that element.

```
 1 package main  // Example 3-16
 2 import "fmt"
 3
 4 func main() {
 5     personSalary := map[string]int{
 6         "steve": 12000,
 7         "jamie": 15000,
 8     }
 9
10     personSalary["mike"] = 9000
11     employee := "jamie"

12     fmt.Println("Salary of", employee, "is", personSalary[employee])
13
14     fmt.Println("Salary of joe is", personSalary["joe"])
15 // joe is not the element, so return zero
16 }
17
```

```
Output:
        Salary of jamie is 15000
        Salary of joe is 0
```

Check For Map Key

- To find whether a key is present in a map or not, use following syntax

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

- If ok is true, then the key is present and its value is present in the variable value, else the key is absent.

```
 1 package main  // Example 3-17
 2 import "fmt"
 3
 4 func main() {
 5     personSalary := map[string]int{
 6         "steve": 12000,
 7         "jamie": 15000,
 8     }
 9     personSalary["mike"] = 9000
10     newEmp := "joe"
11
12     value, ok := personSalary[newEmp]
13     if ok == true { // ok will be false
14         fmt.Println("Salary of", newEmp, "is", value)
15     } else {
16         fmt.Println(newEmp,"not found")
17     }
18
19 }
20
```

Output:

```
joe not found
```

Iterating Map Elements

- Use **range** form of the **for loop**
- The order of the retrieval of values from a map using **for range** is not guaranteed to be the same for each execution of the loop.

```
 1 package main  // Example 3-18
 2
 3 import "fmt"
 4
 5 func main() {
 6
 7  /* create a map using composite literal in a short variable declaration*/
 8    countryCapitalMap := map[string]string{
 9       "France": "Paris", "Italy": "Rome", "Japan": "Tokyo"}
10
11    /* print map using keys*/
12    for country := range countryCapitalMap {
13   fmt.Println("Capital of", country, "is", countryCapitalMap[country])
14    }
15
16    /* print map using key-value*/
17    for country, capital := range countryCapitalMap {
18       fmt.Println("Capital of", country, "is", capital)
19    }
20 }
```

```
Output:
        Capital of France is Paris
        Capital of Italy is Rome
        Capital of Japan is Tokyo
        Capital of France is Paris
        Capital of Italy is Rome
        Capital of Japan is Tokyo
```

Deleting Map Items

- The delete(map, key) is the syntax to delete key from a map
- The delete function does no return any value.

```
 1 package main  // Example 3-19
 2 import "fmt"
 3
 4 func main() {
 5     personSalary := map[string]int{
 6         "steve": 12000,
 7         "jamie": 15000,
 8     }
 9
10     personSalary["mike"] = 9000
11     fmt.Println("map before deletion", personSalary)
12
13     delete(personSalary, "steve")
14     fmt.Println("map after deletion", personSalary)
15
16 }
17
```

```
Output:
map before deletion map[jamie:15000 mike:9000 steve:12000]
map after deletion map[jamie:15000 mike:9000]
```

Maps are Similar to Slice

- Slices have **descriptor** with three types; pointer to the underlying array, length of the slice, and capacity for the slice
- When creating a variable of type map using the "**make()**" function, under the hood, it calls on makemap() which returns ***hmap** (that is a pointer).
- When maps are passed as parameters to functions, changes inside the function are **visible** to the caller.
- When a map is assigned to a new variable, they both point to the same internal data structure.
- Therefore changes made in one map variable will reflect in the other map variable.

```
 1 package main  // Example 3-20
 2 import "fmt"
 3
 4 func main() {
 5     personSalary := map[string]int{
 6         "steve": 12000,
 7         "jamie": 15000,
 8     }
 9
10     personSalary["mike"] = 9000
11     fmt.Println("Original person salary", personSalary)
12
13     newPersonSalary := personSalary
14     newPersonSalary["mike"] = 18000
15     fmt.Println("Person salary changed", personSalary)
16
17 }
18
```

```
Output:
Original person salary map[jamie:15000 mike:9000 steve:12000]
Person salary changed map[jamie:15000 mike:18000 steve:12000]
```

- The map passed as function parameters, changes inside the function are also **visible** to the caller.

```
 1 package main  // Example 3-21
 2 import "fmt"
 3
 4 func addName(item map[string]int) {
 5     item["jill"] = 52000
 6 }
 7
 8 func main() {
 9    // Create and initialize personSalary map
10    personSalary := make(map[string]int)
11    personSalary["steve"] = 12000
12    personSalary["jamie"] = 15000
13    personSalary["mike"] = 9000
14    fmt.Println("Before addName Call:", personSalary)
15    addName(personSalary)
16    fmt.Println("After addName Call:", personSalary)
17 }
18
19
```

Output:

```
Before addName Call: map[jamie:15000 mike:9000 steve:12000]
After addName Call:  map[jamie:15000 jill:52000 mike:9000 steve:12000]
```

Unit 8

User Defined Data Structures

- A structure is a user defined type which represents a collection of fields.
- User defined data structures should be created for grouping the data into a single unit.
- It make sense to create a structure for employee firstName, lastName and age as one group.

Declaring a structure

```
type Employee struct {
    firstName string
    lastName  string
    age       int
}
```

OR

```
type Employee struct {
    firstName, lastName string
    age, salary         int
}
```

- The above snippet declares a structure type Employee which has fields firstName, lastName and age.
- The above structure creates a new **type** called Employee

Creating named structures

```
 1 package main  // Example 3-22
 2 import "fmt"
 3
 4 type Employee struct {
 5     firstName, lastName string
 6     age, salary         int
 7 }
 8
 9 func main() {
10
11    //creating structure using field names
12    // not necessary that the order of the field names should
13    //be same as that while declaring the structure type.
14    emp1 := Employee{
15         firstName: "Sam",
16         age:       25,
17         salary:    500,
18         lastName:  "Anderson",
19     }
20
21     //creating structure without using field names
22     // maintain the order of fields to be the same as
23     //specified in the structure declaration.
24     emp2 := Employee{"Thomas", "Paul", 29, 800}
25
26     fmt.Println("Employee 1", emp1)
27     fmt.Println("Employee 2", emp2)
28 }
29
```

Output:

```
Employee 1 {Sam Anderson 25 500}
Employee 2 {Thomas Paul 29 800}
```

Creating Anonymous Structures

- The anonymous structures can be created without declaring a new type.

```
var employee struct {
        firstName, lastName string
        age int
}
```

```
 1 package main  // Example 3-23
 2 import "fmt"
 3
 4 func main() {
 5
 6 // creates new struct variable emp3, does not define new struct type
 7     emp3 := struct {  // Anonymous Structure
 8         firstName, lastName string
 9         age, salary         int
10     }{
11         firstName: "Andreah",
12         lastName:  "Nikola",
13         age:       31,
14         salary:    5000,
15     }
16
17     fmt.Println("Employee 3", emp3)
18 }
19
```

```
Output:
            Employee 3 {Andreah Nikola 31 5000}
```

Zero value of a structure

- When a **struct** is defined and it is not explicitly initialized with any value
- The fields of the **struct** are assigned their zero values by default.

```
 1 package main  // Example 3-24
 2 import "fmt"
 3
 4 type Employee struct {
 5     firstName, lastName string
 6     age, salary         int
 7 }
 8
 9 func main() {
10     var emp4 Employee //zero valued structure
11     fmt.Println("Employee 4", emp4)
12 }
13
Output:
Employee 4 {  0 0}
```

- Specify values for some fields and ignore the rest.
- The ignored field names are assigned zero values.

```
 1 package main  // Example 3-25
 2 import "fmt"
 3
 4 type Employee struct {
 5     firstName, lastName string
 6     age, salary         int
 7 }
 8
 9 func main() {
10     emp5 := Employee{
11         firstName: "John",
12         lastName:  "Paul",
13     }
14     fmt.Println("Employee 5", emp5)
15 }
16
Output:
Employee 5 {John Paul 0 0}
```

Accessing individual fields of a struct

- Use dot (.) operator to access the individual fields of a structure

```
 1 package main  // Example 3-26
 2 import "fmt"
 3
 4 type Employee struct {
 5     firstName, lastName string
 6     age, salary         int
 7 }
 8
 9 func main() {
10     emp6 := Employee{"Sam", "Anderson", 55, 6000}
11     fmt.Println("First Name:", emp6.firstName)
12     fmt.Println("Last Name:", emp6.lastName)
13     fmt.Println("Age:", emp6.age)
14     fmt.Printf("Salary: $%d\n", emp6.salary)
15 }
16
Output:
First Name: Sam
Last Name: Anderson
Age: 55
Salary: $6000
```

- Create a zero struct and then assign values to its fields later.

```
 1 package main  // Example 3-27
 2 import "fmt"
 3
 4 type Employee struct {
 5     firstName, lastName string
 6     age, salary         int
 7 }
 8
 9 func main() {
10     var emp7 Employee
11     emp7.firstName = "Jack"
12     emp7.lastName = "Adams"
13     fmt.Println("Employee 7:", emp7)
14 }
15
Output:
Employee 7: {Jack Adams 0 0}
```