Introduction to Golang
Part 4

Calistus Igwilo
https://linkedin.com/in/calistus-igwilo
https://twitter.com/CalistusIgwilo

Overview

- Arrays
- Slices
- Variable Slices
- Maps
- Struct

Arrays in Go

- Fixed-length series of elements of a given type
- Stores multiple items of a given type under a single variable name
- Elements accessed using subscript notation [ ]
- Elements initialized to zero value

Arrays in Go

```go
package main

import "fmt"

func main(){
    var x [5]int

    x[2] = 6

    fmt.Printf("%d\n", x[1])   // 0
```

Array Literal

- An array predefined with values
  - var x [5]int = [5]{1, 2, 3, 4, 5}
  -
- Length of literal must be length of array

- ... for size in array literal infers size from the number of initializers
  - x := [...]{1, 2, 3, 4}

- Elements initialized to zero value

Iterating through Arrays

- Use a for loop with the range keywoard

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

for i, v := range y{
    fmt.Printf("Index: %d value: %d\n", i, v)
}
```

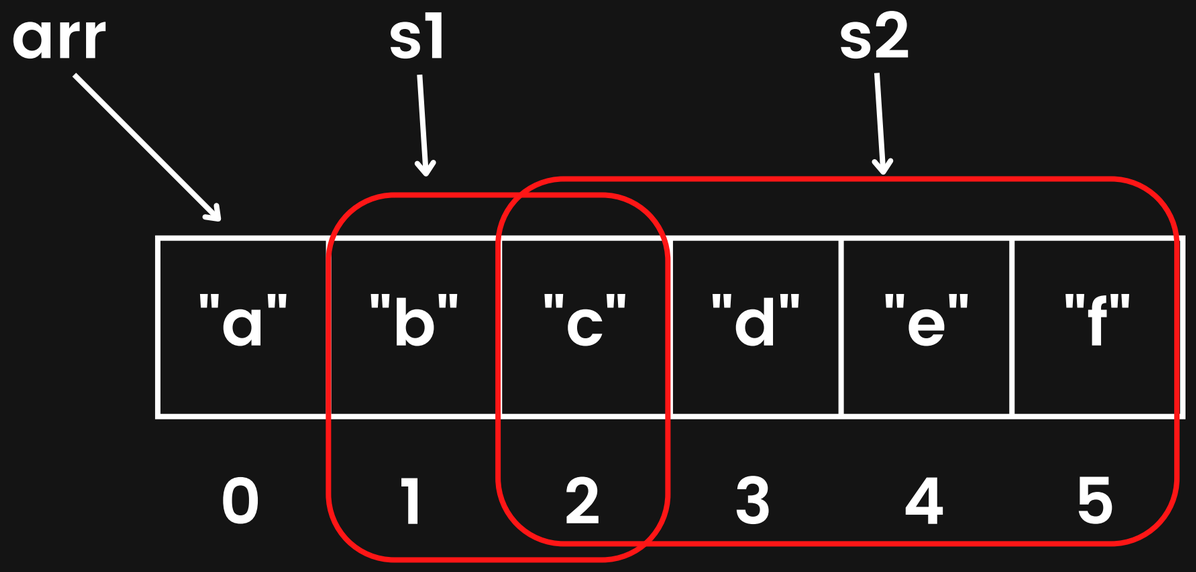
- range returns two values
  - Index and
  - Element at index

Slices

- Dynamically-sized, flexible view into the elements of an array.
- Variable size up to whole array.
- Pointer indicates the start of the slice
- Length is the number of elements in the slice
- Capacity is the maximum number of elements
  - from start of slice to end of array

Slice Examples

```
arr := [...]string {"a", "b", "c", "d", "e", "f"}
s1 := arr[1:3]
s2 := arr[2:5]
```



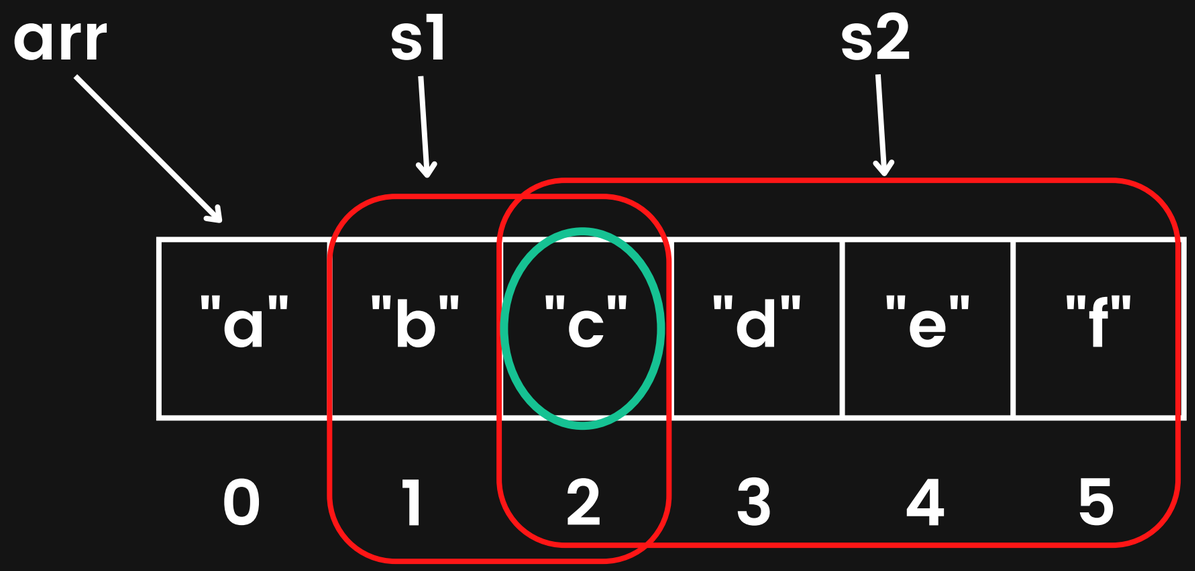
Length & Capacity

- len() function returns the length of the slice
- cap() function returns the capacity

```
a1 := [3]string("a", "b", "c")
sli1 := a1[0:1]
fmt.Printf(len(sli1), cap(sli1))
```

Result is " 1 3 "

Accessing Slices




```
fmt.Printf(s1[1])

fmt.Printf(s2[0])
```

Slice Literals

- Can be used to initialize a slice
- Creates an underlying array and references it
- Slice points to the start of the array, length is the capacity

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

Make

- Create a slice (and an array) using make()

- 2 argument version
  - Specify type and length/capacity
  - initialize to zero, length = capacity
  - sli = make ([]int, 10)
  -
- 3 argument version
  - specifiy length and capacity separately
  - sli = make ([]int, 10, 15)

Append

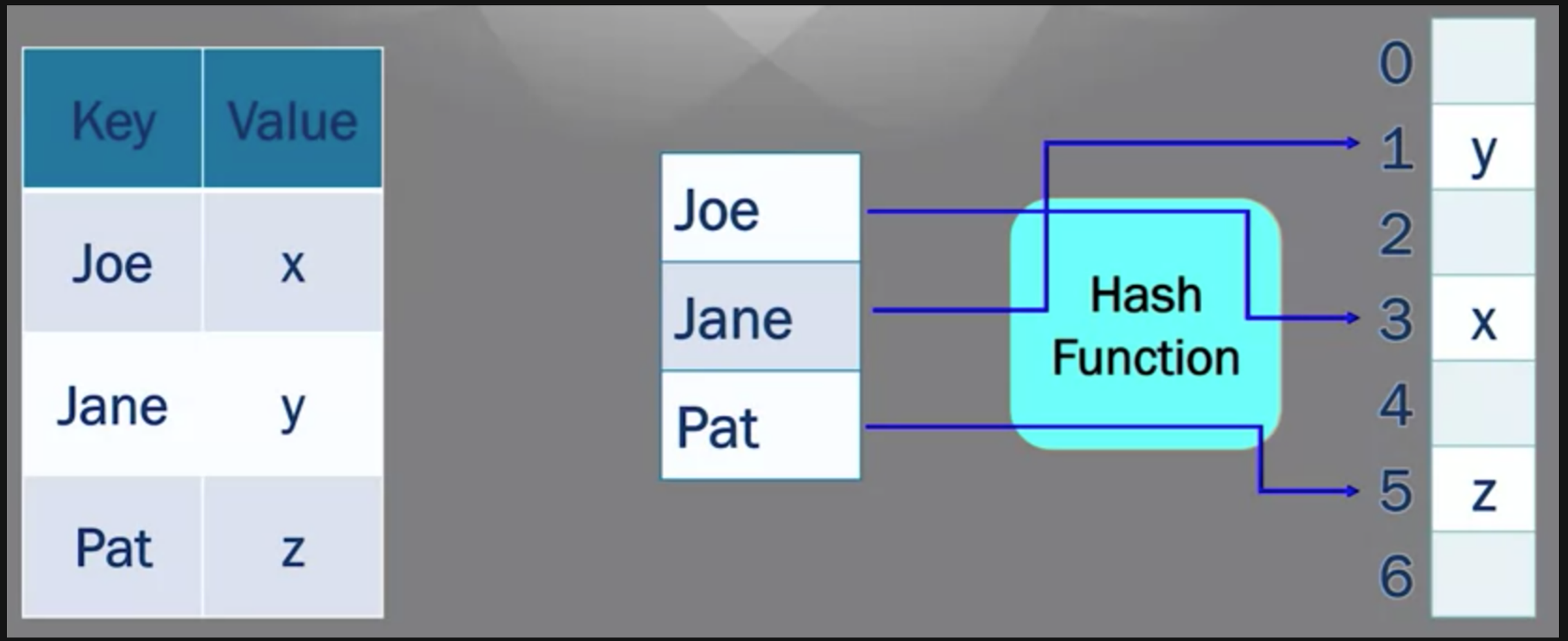
- Size of a slice can be increased by append()
- Adds element to the end of a slice
- Increases size of array if necessary
  - sli = make ([]int, 0, 3)
  - length of sli is zero
  - sli = append (sli, 100)

Hash Table

- Contains key/value pairs
  - email/name
- Each value is associated with a unique key
- Hash function is used to compute the slot for a key

Hash Table Example

Key	Value
Joe	x
Jane	y
Pat	z



Tradeoffs of Hash Table

Advantages

- **Faster lookup thank lists**
  - **constant time vs linear time**

- **Arbitrary keys**
  - not integers like slices and arrays

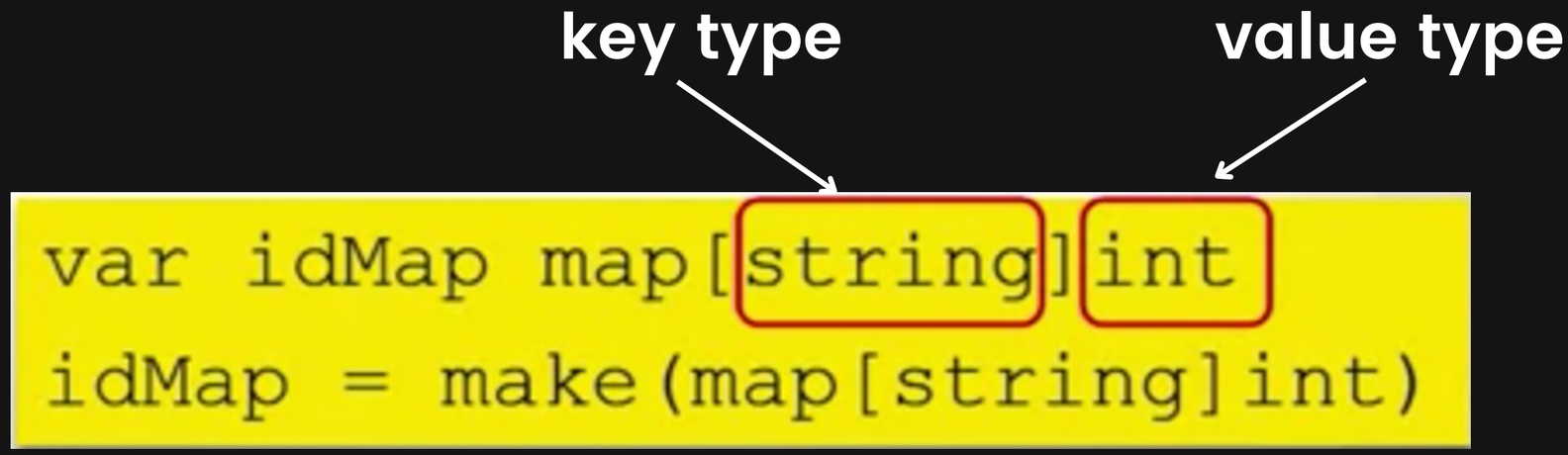
Tradeoffs of Hash Table

Disadvantages

- Collision may occur
  - Two keys hash to the same slot
    - can be sloved using:
      - linked list
      - bit manipulation
      - rehashing all keys (growing)
  -
    - But it slows things down a bit

Maps

- Implementation of a hash table
- Use make() to create a map

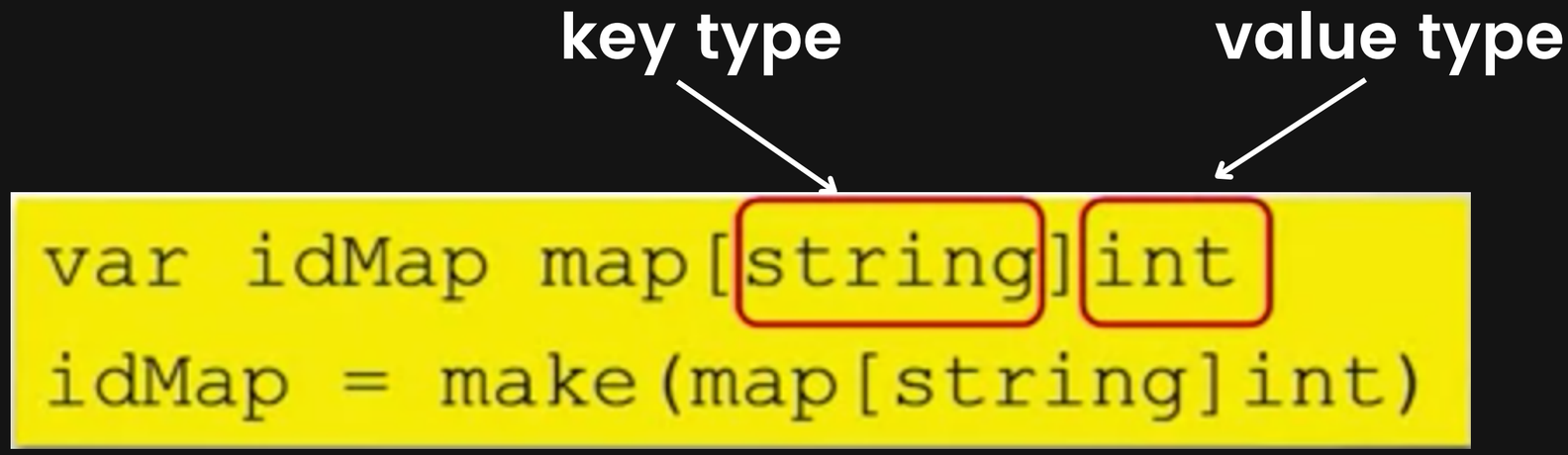


- May define a map literal

```
idMap := map[string]int {
    "joe": 123}
```

Maps

- Implementation of a hash table
- Use make() to create a map



- May define a map literal

```
idMap := map[string]int {
    "joe": 123}
```

Accessing Maps

- Referencing a value with key
- Returns zero if key is not present

```
fmt.Println(idMap["Hassan"])
```

- Adding a key/value pair

```
idMap["Kunle"] = 465
```

- Deleting a key/value pair

```
delete(idMap, "Joe")
```

Map Functions

- Two value assignment tests for the existencse of a key

```
id, p := idMap["Hassan"]
```

  - id is value, p is presence of key

- len() returns number of values

```
fmt.Println(len(idMap))
```

Iterating through a Map

- Use a for loop with range keyword
- Two value assignment with range

```
for key, val := range idMap {
        fmt.printlin(key, val)
}
```

Struct

- Aggregate data type
- store multiple values of different data types into a single variable

Struct

Example: Person struct

Name, Address, Phone

- Option 1: have 3 different variables say name1, address1, phone1. Programmer has to figure out that they are related

- Option 2: make a single struct which has all the variables

Struct Example

```
type struct Person {
    name string
    addr string
    phone string
}
var p1 Person
```

- Each property is a field (name, addr, phone)
- p1 contains values for all fields

Accessing Struct Fields

- Use dot notation

```
p1.name = "Zainab"
x  =  p1.address
```

Initializing Structs

- Can use new()
- Initializes fields to zero

```
p1 := new Person
```

- Can initializes using a struct literal

```
p1 := new Person(name: "Esther",
address: "7 Agudama", phone: "1234"
```