**Module 1 – 1 Arrays**

1. Composite Data Types are data types that aggregate other data types together.
2. Arrays

* Fixed length series of elements of a chosen type
* Elements accessed using subscript notation, []
* Indices start at ()
* Elements initialized to zero value
* Example:

var x [5] int

x[0] = 2

fmt.Printf(x[1])

1. Array Literal

* An array pre-defined 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 number of initializers

x := […] int { 1, 2, 3, 4}

1. Iterating Through Arrays

* Use a for loop with the range keyword

x := [3] int { 1, 2, 3}

for i, v range x {

fmt.Printf(“ind %d, val %d”, i, v)

}

* Range return two values
* Index and element at index

**Module 1 – 2 Slices**

1. Slices

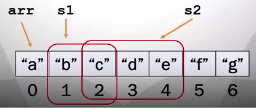
* A “window” on an underlying array
* Variable size, up to the whole array
* Pointer indicates the start of the slice
* Length is the number of elements in the slice
* Capacity is maximum number of elements, from start of slice to end of array

1. Slice Examples

arr := […] string {“a”, “b”, “c”, “d”, “e”, “f”, “g”}

s1 := arr[1:3]

s2 := arr[2:5]



1. Length and Capacity

* len() function returns the length
* cap() function returns the capacity
* Examples, results “1 3”:

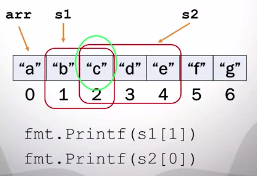
a1 := [3] string {“a”, “b”, “c”}

sli1 := a1[0:1]

fmt.Printf(len(sli1), cap(sli1))

1. Accessing Slices

* Writing to a slice changes underlying array
* Overlapping slices refer to the same array elements



1. Slice Literals

* Can be used to initialize a slice
* Creates the underlying array and creates a slice to reference it
* Slice points to the start of the array, length is capacity

**Module 1 – 1 Variables**

1. Make

* Create a slice (and array) using make()
* 2-arguments version: specify type and length/capacity
* Init to zero, length = capacity

sli = make([]int, 10)

* 3-arguments version: specify length and capacity separately

sli = make([]int, 10, 15)

1. Append

* Size of a slice can be increased by append()
* Adds elements to the end of a slice
* Inserts into underlying array
* Increases size of array if necessary

sli = make([]int, 0, 3)

* Length of sli is 0

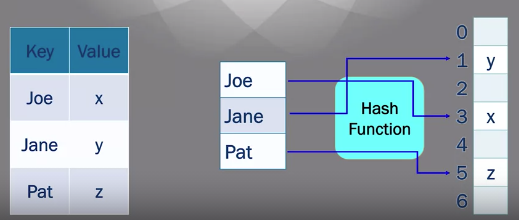
sli = append(sli, 100)

**Module 2 – 1 Hash Tables**

1. Hash Table

* Contains key/value pairs, such as SSN/email, GPS cors/address
* Each value is associated with a unique key
* Hash function is used to compute the slot for a key

1. Hash Table Example



1. Tradeoffs of Hash Tables

Advantages:

* Faster lookup than lists: constant time vs linear time
* Arbitrary keys: not ints, like slices or arrays

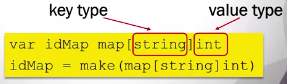
Disadvantages

* May have collisions: two keys hash to same slot

**Module 2 – 2 Maps**

1. Maps

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



* May define a map literal



1. Accessing Maps

* Referencing a value with [key]
* Returns zero if key is not present

fmt.Println(idMap[“joe”])

* Adding a key/value pair

idMap[“jane”] = 456

* Deleting a key/value pair

delete(idMap, “joe”)

1. Map Functions

* Two-value assignment tests for existence of the key

id, p := idMap[“joe”]

* id is value, p is presence of key
* len() return number values

fmt.Println(len(idMap))

1. Iterating Through A Map

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

for key, val := range idMap {

fmt.Println(key, val)

}

**Module 3 – 1 Structs**

1. Struct

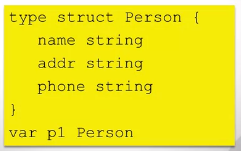
* Aggregate data type
* Groups together other objects of arbitrary type
* Example: Person Struct

Name, Address, Phone

Option 1: Have 3 separate variables, programmer remembers that they are related

Option 2: Make a single struct which contains all 3 vars

1. Struct Example



* Each property is a field
* p1 contains values for all fields

1. Accessing Struct Fields

* Use dot notation



1. Initializing Structs

* Can use new()
* Initializes field to zero



* Can initialize using a struct literal

