Arrays

CR1

// Sample program to show how to declare and iterate over

// arrays of different types.

package main

import "fmt"

func main() {

// Declare an array of five strings that is initialized

// to its zero value.

var fruits [5]string

fruits[0] = "Apple"

fruits[1] = "Orange"

fruits[2] = "Banana"

fruits[3] = "Grape"

fruits[4] = "Plum"

// Iterate over the array of strings.

for i, fruit := range fruits {

fmt.Println(i, fruit)

}

// Declare an array of 4 integers that is initialized

// with some values.

numbers := [4]int{10, 20, 30, 40}

// Iterate over the array of numbers.

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

fmt.Println(i, numbers[i])

}

}

CR2

// Sample program to show how arrays of different sizes are

// not of the same type.

package main

import "fmt"

func main() {

// Declare an array of 5 integers that is initialized

// to its zero value.

var five [5]int

// Declare an array of 4 integers that is initialized

// with some values.

four := [4]int{10, 20, 30, 40}

// Assign one array to the other

five = four

// ./example2.go:21: cannot use four (type [4]int) as type [5]int in assignment

fmt.Println(four)

fmt.Println(five)

}

CR3

// Sample program to show how the behavior of the for range and

// how memory for an array is contiguous.

package main

import "fmt"

func main() {

// Declare an array of 5 strings initialized with values.

friends := [5]string{"Annie", "Betty", "Charley", "Doug", "Edward"}

// Iterate over the array displaying the value and

// address of each element.

for i, v := range friends {

fmt.Printf("Value[%s]\tAddress[%p] IndexAddr[%p]\n", v, &v, &friends[i])

}

}

CR4

// Sample program to show how the for range has both value and pointer semantics.

package main

import "fmt"

func main() {

// Using the pointer semantic form of the for range.

friends := [5]string{"Annie", "Betty", "Charley", "Doug", "Edward"}

fmt.Printf("Bfr[%s] : ", friends[1])

for i := range friends {

friends[1] = "Jack"

if i == 1 {

fmt.Printf("Aft[%s]\n", friends[1])

}

}

// Using the value semantic form of the for range.

friends = [5]string{"Annie", "Betty", "Charley", "Doug", "Edward"}

fmt.Printf("Bfr[%s] : ", friends[1])

for i, v := range friends {

friends[1] = "Jack"

if i == 1 {

fmt.Printf("v[%s]\n", v)

}

}

// Using the value semantic form of the for range but with pointer

// semantic access. DON'T DO THIS.

friends = [5]string{"Annie", "Betty", "Charley", "Doug", "Edward"}

fmt.Printf("Bfr[%s] : ", friends[1])

for i, v := range &friends {

friends[1] = "Jack"

if i == 1 {

fmt.Printf("v[%s]\n", v)

}

}

}

Slices

CR1

// Sample program to show how the capacity of the slice

// is not available for use.

package main

import "fmt"

func main() {

// Create a slice with a length of 5 elements.

fruits := make([]string, 5)

fruits[0] = "Apple"

fruits[1] = "Orange"

fruits[2] = "Banana"

fruits[3] = "Grape"

fruits[4] = "Plum"

// You can't access an index of a slice beyond its length.

fruits[5] = "Runtime error"

// Error: panic: runtime error: index out of range

fmt.Println(fruits)

}

CR2

// Sample program to show the components of a slice. It has a

// length, capacity and the underlying array.

package main

import "fmt"

func main() {

// Create a slice with a length of 5 elements and a capacity of 8.

fruits := make([]string, 5, 8)

fruits[0] = "Apple"

fruits[1] = "Orange"

fruits[2] = "Banana"

fruits[3] = "Grape"

fruits[4] = "Plum"

inspectSlice(fruits)

}

// inspectSlice exposes the slice header for review.

func inspectSlice(slice []string) {

fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))

for i, s := range slice {

fmt.Printf("[%d] %p %s\n",

i,

&slice[i],

s)

}

}

CR3

// Sample program to show how to grow a slice using the built-in function append

// and how append grows the capacity of the underlying array.

package main

import "fmt"

func main() {

// Declare a nil slice of strings.

var data []string

// Capture the capacity of the slice.

lastCap := cap(data)

// Append ~100k strings to the slice.

for record := 1; record <= 1e5; record++ {

// Use the built-in function append to add to the slice.

value := fmt.Sprintf("Rec: %d", record)

data = append(data, value)

// When the capacity of the slice changes, display the changes.

if lastCap != cap(data) {

// Calculate the percent of change.

capChg := float64(cap(data)-lastCap) / float64(lastCap) \* 100

// Save the new values for capacity.

lastCap = cap(data)

// Display the results.

fmt.Printf("Addr[%p]\tIndex[%d]\t\tCap[%d - %2.f%%]\n",

&data[0],

record,

cap(data),

capChg)

}

}

}

CR4

// Sample program to show how to takes slices of slices to create different

// views of and make changes to the underlying array.

package main

import "fmt"

func main() {

// Create a slice with a length of 5 elements and a capacity of 8.

slice1 := make([]string, 5, 8)

slice1[0] = "Apple"

slice1[1] = "Orange"

slice1[2] = "Banana"

slice1[3] = "Grape"

slice1[4] = "Plum"

inspectSlice(slice1)

// Take a slice of slice1. We want just indexes 2 and 3.

// Parameters are [starting\_index : (starting\_index + length)]

slice2 := slice1[2:4]

inspectSlice(slice2)

fmt.Println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

// Change the value of the index 0 of slice2.

slice2[0] = "CHANGED"

// Display the change across all existing slices.

inspectSlice(slice1)

inspectSlice(slice2)

fmt.Println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

// Make a new slice big enough to hold elements of slice 1 and copy the

// values over using the builtin copy function.

slice3 := make([]string, len(slice1))

copy(slice3, slice1)

inspectSlice(slice3)

}

// inspectSlice exposes the slice header for review.

func inspectSlice(slice []string) {

fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))

for i, s := range slice {

fmt.Printf("[%d] %p %s\n",

i,

&slice[i],

s)

}

}

CR5

// Sample program to show how one needs to be careful when appending

// to a slice when you have a reference to an element.

package main

import "fmt"

type user struct {

likes int

}

func main() {

// Declare a slice of 3 users.

users := make([]user, 3)

// Share the user at index 1.

shareUser := &users[1]

// Add a like for the user that was shared.

shareUser.likes++

// Display the number of likes for all users.

for i := range users {

fmt.Printf("User: %d Likes: %d\n", i, users[i].likes)

}

// Add a new user.

users = append(users, user{})

// Add another like for the user that was shared.

shareUser.likes++

// Display the number of likes for all users.

fmt.Println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for i := range users {

fmt.Printf("User: %d Likes: %d\n", i, users[i].likes)

}

// Notice the last like has not been recorded.

}

CR6

/\*

https://blog.golang.org/strings

Go source code is always UTF-8.

A string holds arbitrary bytes.

A string literal, absent byte-level escapes, always holds valid UTF-8 sequences.

Those sequences represent Unicode code points, called runes.

No guarantee is made in Go that characters in strings are normalized.

----------------------------------------------------------------------------

Multiple runes can represent different characters:

The lower case grave-accented letter à is a character, and it's also a code

point (U+00E0), but it has other representations.

We can use the "combining" grave accent code point, U+0300, and attach it to

the lower case letter a, U+0061, to create the same character à.

In general, a character may be represented by a number of different sequences

of code points (runes), and therefore different sequences of UTF-8 bytes.

\*/

// Sample program to show how strings have a UTF-8 encoded byte array.

package main

import (

"fmt"

"unicode/utf8"

)

func main() {

// Declare a string with both chinese and english characters.

s := "世界 means world"

// UTFMax is 4 -- up to 4 bytes per encoded rune.

var buf [utf8.UTFMax]byte

// Iterate over the string.

for i, r := range s {

// Capture the number of bytes for this rune.

rl := utf8.RuneLen(r)

// Calculate the slice offset for the bytes associated

// with this rune.

si := i + rl

// Copy of rune from the string to our buffer.

copy(buf[:], s[i:si])

// Display the details.

fmt.Printf("%2d: %q; codepoint: %#6x; encoded bytes: %#v\n", i, r, r, buf[:rl])

}

}

CR7

// Sample program to show how to declare and use variadic functions.

package main

import "fmt"

// user is a struct type that declares user information.

type user struct {

id int

name string

}

func main() {

// Declare and initialize a value of type user.

u1 := user{

id: 1432,

name: "Betty",

}

// Declare and initialize a value of type user.

u2 := user{

id: 4367,

name: "Janet",

}

// Display both user values.

display(u1, u2)

// Create a slice of user values.

u3 := []user{

{24, "Bill"},

{32, "Joan"},

}

// Display all the user values from the slice.

display(u3...)

change(u3...)

fmt.Println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for \_, u := range u3 {

fmt.Printf("%+v\n", u)

}

}

// display can accept and display multiple values of user types.

func display(users ...user) {

fmt.Println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for \_, u := range users {

fmt.Printf("%+v\n", u)

}

}

// change shows how the backing array is shared.

func change(users ...user) {

users[1] = user{99, "Same Backing Array"}

}

CR8

// Sample program to show how the for range has both value and pointer semantics.

package main

import "fmt"

func main() {

// Using the value semantic form of the for range.

friends := []string{"Annie", "Betty", "Charley", "Doug", "Edward"}

for \_, v := range friends {

friends = friends[:2]

fmt.Printf("v[%s]\n", v)

}

// Using the pointer semantic form of the for range.

friends = []string{"Annie", "Betty", "Charley", "Doug", "Edward"}

for i := range friends {

friends = friends[:2]

fmt.Printf("v[%s]\n", friends[i])

}

}

CR9

// Sample program to show how slices allow for efficient linear traversals.

package main

import (

"encoding/binary"

"fmt"

)

func main() {

// Given a stream of bytes to be processed.

x := []byte{0x0A, 0x15, 0x0e, 0x28, 0x05, 0x96, 0x0b, 0xd0, 0x0}

// Perform a linear traversal across the bytes, never making

// copies of the actual data but still passing those bytes

// to the binary function for processing.

a := x[0]

b := binary.LittleEndian.Uint16(x[1:3])

c := binary.LittleEndian.Uint16(x[3:5])

d := binary.LittleEndian.Uint32(x[5:9])

// The result is zero allocation data access that is mechanically

// sympathetic with the hardware.

fmt.Println(a, b, c, d)

}

CR advanced

// Sample program to show how to use a third index slice.

package main

import "fmt"

func main() {

// Create a slice of strings with different types of fruit.

slice := []string{"Apple", "Orange", "Banana", "Grape", "Plum"}

inspectSlice(slice)

// Take a slice of slice. We want just index 2

takeOne := slice[2:3]

inspectSlice(takeOne)

// Take a slice of just index 2 with a length and capacity of 1

takeOneCapOne := slice[2:3:3] // Use the third index position to

inspectSlice(takeOneCapOne) // set the capacity to 1.

// Append a new element which will create a new

// underlying array to increase capacity.

takeOneCapOne = append(takeOneCapOne, "Kiwi")

inspectSlice(takeOneCapOne)

}

// inspectSlice exposes the slice header for review.

func inspectSlice(slice []string) {

fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))

for i, s := range slice {

fmt.Printf("[%d] %p %s\n",

i,

&slice[i],

s)

}

}

Maps

CR1

// Sample program to show how to initialize a map, write to

// it, then read and delete from it.

package main

import "fmt"

// user represents someone using the program.

type user struct {

name string

surname string

}

func main() {

// Declare and make a map that stores values

// of type user with a key of type string.

users := make(map[string]user)

// Add key/value pairs to the map.

users["Roy"] = user{"Rob", "Roy"}

users["Ford"] = user{"Henry", "Ford"}

users["Mouse"] = user{"Mickey", "Mouse"}

users["Jackson"] = user{"Michael", "Jackson"}

// Read the value at a specific key.

mouse := users["Mouse"]

fmt.Printf("%+v\n", mouse)

// Replace the value at the Mouse key.

users["Mouse"] = user{"Jerry", "Mouse"}

// Read the Mouse key again.

fmt.Printf("%+v\n", users["Mouse"])

// Delete the value at a specific key.

delete(users, "Roy")

// Check the length of the map. There are only 3 elements.

fmt.Println(len(users))

// It is safe to delete an absent key.

delete(users, "Roy")

fmt.Println("Goodbye.")

}

CR2

// Sample program to show how maps behave when you read an

// absent key.

package main

import "fmt"

func main() {

// Create a map to track scores for players in a game.

scores := make(map[string]int)

// Read the element at key "anna". It is absent so we get

// the zero-value for this map's value type.

score := scores["anna"]

fmt.Println("Score:", score)

// If we need to check for the presence of a key we use

// a 2 variable assignment. The 2nd variable is a bool.

score, ok := scores["anna"]

fmt.Println("Score:", score, "Present:", ok)

// We can leverage the zero-value behavior to write

// convenient code like this:

scores["anna"]++

// Without this behavior we would have to code in a

// defensive way like this:

if n, ok := scores["anna"]; ok {

scores["anna"] = n + 1

} else {

scores["anna"] = 1

}

score, ok = scores["anna"]

fmt.Println("Score:", score, "Present:", ok)

}

CR3

// Sample program to show how only types that can have

// equality defined on them can be a map key.

package main

import "fmt"

// user represents someone using the program.

type user struct {

name string

surname string

}

// users defines a set of users.

type users []user

func main() {

// Declare and make a map that uses a slice as the key.

u := make(map[users]int)

// ./example3.go:22: invalid map key type users

// Iterate over the map.

for key, value := range u {

fmt.Println(key, value)

}

}

CR4

// Sample program to show how to declare, initialize and iterate

// over a map. Shows how iterating over a map is random.

package main

import "fmt"

// user represents someone using the program.

type user struct {

name string

surname string

}

func main() {

// Declare and initialize the map with values.

users := map[string]user{

"Roy": {"Rob", "Roy"},

"Ford": {"Henry", "Ford"},

"Mouse": {"Mickey", "Mouse"},

"Jackson": {"Michael", "Jackson"},

}

// Iterate over the map printing each key and value.

for key, value := range users {

fmt.Println(key, value)

}

fmt.Println()

// Iterate over the map printing just the keys.

// Notice the results are different.

for key := range users {

fmt.Println(key)

}

}

CR5

// Sample program to show how to walk through a map by

// alphabetical key order.

package main

import (

"fmt"

"sort"

)

// user represents someone using the program.

type user struct {

name string

surname string

}

func main() {

// Declare and initialize the map with values.

users := map[string]user{

"Roy": {"Rob", "Roy"},

"Ford": {"Henry", "Ford"},

"Mouse": {"Mickey", "Mouse"},

"Jackson": {"Michael", "Jackson"},

}

// Pull the keys from the map.

var keys []string

for key := range users {

keys = append(keys, key)

}

// Sort the keys alphabetically.

sort.Strings(keys)

// Walk through the keys and pull each value from the map.

for \_, key := range keys {

fmt.Println(key, users[key])

}

}

CR6

// Sample program to show that you cannot take the address

// of an element in a map.

package main

// player represents someone playing our game.

type player struct {

name string

score int

}

func main() {

// Declare a map with initial values using a map literal.

players := map[string]player{

"anna": {"Anna", 42},

"jacob": {"Jacob", 21},

}

// Trying to take the address of a map element fails.

anna := &players["anna"]

anna.score++

// ./example4.go:23:10: cannot take the address of players["anna"]

// Instead take the element, modify it, and put it back.

player := players["anna"]

player.score++

players["anna"] = player

}

CR7

// Sample program to show how maps are reference types.

package main

import "fmt"

func main() {

// Initialize a map with values.

scores := map[string]int{

"anna": 21,

"jacob": 12,

}

// Pass the map to a function to perform some mutation.

double(scores, "anna")

// See the change is visible in our map.

fmt.Println("Score:", scores["anna"])

}

// double finds the score for a specific player and

// multiplies it by 2.

func double(scores map[string]int, player string) {

scores[player] = scores[player] \* 2

}