Programming in Go

Matt Holiday
Christmas 2020

Composite Types

Composite types

`string`

a	r	e	a

`[4]int`

1	12	4	8

`[]int`

1	13	3	… …	17

`map[string]int`

"to"	1
"from"	12
"into"	3
"above"	0

Arrays

Arrays are typed by size, which is *fixed* at compile time

```
// all these are equivalent

var a [3]int
var b [3]int{0, 0, 0}
var c [...]{0, 0, 0}   // sized by initializer

var d [3]int
d = b                  // elements copied

var m [...]int{1, 2, 3, 4}

c = m                  // TYPE MISMATCH
```

Arrays are passed *by value*, thus elements are copied

Slices

Slices have variable length, backed by some array; they are copied when they outgrow that array

```
var a []int                 // nil, no storage
var b = []int{1, 2}         // initialized

a = append(a, 1)            // append to nil OK
b = append(b, 3)            // []int{1, 2, 3}

a = b                       // overwrites a

d := make([]int, 5)         // []int{0, 0, 0, 0, 0}
e := a                      // same storage (alias)

e[0] == b[0]                // true
```

Slices are passed *by reference;* no copying, updating OK

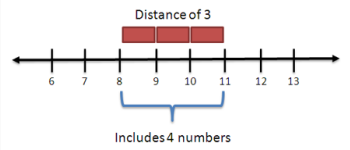
The off-by-one bug

Slices are indexed like `[8:11]`

(read as the starting element and *one past* the ending element, so this way we have $11 - 8 = 3$ elements in our slice)

For loops work the same way in most cases:

```
for i := 8; i < 11; i++ {  // in math written [8, 11)
    . . .
}
```



Read it on Wikipedia OB1

Slices

```go
package main
import "fmt"

func main() {
    t := []byte("string")   // 0:s 1:t 2:r 3:i 4:n 5:g
    fmt.Println(len(t), t)  // 6 bytes in t
    fmt.Println(t[2])       //   1 item
    fmt.Println(t[:2])      //   2 items
    fmt.Println(t[2:])      // 6-2 items
    fmt.Println(t[3:5])     // 5-3 items
}
```

```
6 [115 116 114 105 110 103]
114
[115 116]
[114 105 110 103]
[105 110]
```

Slices vs arrays

Most Go APIs take slices as inputs, not arrays

Slice	Array
Variable length	Length fixed at compile time
Passed by reference	Passed by value (copied)
Not comparable	Comparable (==)
Cannot be used as map key	Can be used as map key
Has `copy` & `append` helpers	—
Useful as function parameters	Useful as “pseudo” constants

Arrays as pseudo-constants

It can be useful to have fixed-size tables of values in some algorithms, treated as constant data

```
// from the file crypto/des/const.go in the DES package

// Used to perform an initial permutation of a 64-bit
// input block.
var initialPermutation = [64]byte{
    6, 14, 22, 30, 38, 46, 54, 62,
    4, 12, 20, 28, 36, 44, 52, 60,
    2, 10, 18, 26, 34, 42, 50, 58,
    0,  8, 16, 24, 32, 40, 48, 56,
    7, 15, 23, 31, 39, 47, 55, 63,
    5, 13, 21, 29, 37, 45, 53, 61,
    3, 11, 19, 27, 35, 43, 51, 59,
    1,  9, 17, 25, 33, 41, 49, 57,
}
```

Examples

```
var w = [...]int{1, 2, 3}   // array of len(3)
var x = []int{0, 0, 0}      // slice of len(3)

func do(a [3]int, b []int) []int {
    a = b                   // SYNTAX ERROR
    a[0] = 4                // w unchanged
    b[0] = 3                // x changed

    c := make([]int, 5)     // []int{0, 0, 0, 0, 0}
    c[4] = 42
    copy(c, b)              // copies only 3 elts

    return c
}

y := do(w, x)
fmt.Println(w, x, y)        // [1 2 3] [3 0 0] [3 0 0 0 42]
```

Maps

Maps are dictionaries: indexed by key, returning a value

You can read from a nil map, but inserting will panic

```
var m map[string]int          // nil, no storage
p := make(map[string]int)     // non-nil but empty

a := p["the"]                 // returns 0
b := m["the"]                 // same thing
m["and"] = 1                  // PANIC - nil map
m = p
m["and"]++                    // OK, same map as p now
c := p["and"]                 // returns 1
```

Maps are passed *by reference;* no copying, updating OK

The type used for the key must have == and != defined *(not slices, maps, or funcs)*

Maps

Maps can't be compared to one another; maps can be compared only to `nil` as a special case

```
var m  = map[string]int{
    "and": 1,
    "the": 1,
    "or":  2,
}

var n map[string]int

b := m == n                 // SYNTAX ERROR
c := n == nil               // true
d := len(m)                 // 3
e := cap(m)                 // TYPE MISMATCH
```

Maps

Maps have a special two-result lookup function

The second variable tells you if they key was there

```
p := map[string]int{}        // non-nil but empty

a := p["the"]                // returns 0
b, ok := p["and"]            // 0, false

p["the"]++

c, ok := p["the"]            // 1, true

if w, ok := p["the"]; ok {
    // we know w is not the default value
    . . .
}
```

Built-ins

Each type has certain built-in functions

```
len(s)              string     string length

len(a), cap(a)      array      array length, capacity (constant)

make(T, x)          slice      slice of type T with length x and capacity x
make(T, x, y)       slice      slice of type T with length x and capacity y

copy(c, d)          slice      copy from d to c; # = min of the two lengths
c=append(c, d)      slice      append d to c and return a new slice result

len(s), cap(s)      slice      slice length and capacity

make(T)             map        map of type T
make(T, x)          map        map of type T with space hint for x elements

delete(m, k)        map        delete key k (if present, else no change)

len(m)              map        map length
```

Make nil useful

Nil is a type of zero: it indicates the absence of something

Many built-ins are safe: `len`, `cap`, `range`

```go
var s []int
var m map[string]int

l := len(s)              // length of nil slice is 0

i, ok := m["int"]        // 0, false for any missing key

for _, v := range s {    // skip if s is nil or empty
    . . .
}
```

“Make the zero value useful.” — Rob Pike

“Understanding nil”

See Francesc Campoy’s video at
https://www.youtube.com/watch?v=ynoY2xz-F8s

GopherCon 2016: Francesc Campoy - Understanding nil

“Understanding nil”

Dimitri Fontaine liked

Programming Wisdom @CodeWisdom · 13h

“A language that doesn't affect the way you think about programming is not worth knowing.” - Alan J. Perlis

7 167 627