CS 677: Big Data

Slightly More Advanced Go

Lecture 4

Going, Going, Gone

So, we started writing some Go last class. Let’s look at some more features of the language.

Structs

- Go does not have classes. A `struct` is the closest relative:

```
type LogEntry struct {
    accessTime time.Time
    userIP     net.IP
    pageURL    url.URL
}
```

- You can still write object-oriented programs, but the data structures and functions are defined separately

Receiver Functions

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

// This is like a Java .toString() method:
func (p Person) String() string {
        return fmt.Sprintf("Hello, my name is %s %s! I am %d years old. Bye!",
                p.firstName, p.lastName, p.age)
}

func main() {
        bob := Person{"Bob", "Bobberton", 38}
        fmt.Println(bob)
}
```

Visibility

- Like Java, you can control function/method visibility
- Functions are either *exported* or *unexported*
  - Or in other words, visible outside their **package**
  - Not every function has to belong to a class like in Java, so we deal with package scoping instead
- **C**apitalized() = exported
  - As our `String()` function was in the previous example
- **u**nCapitalized() = not exported

Creating an Array

Arrays behave similarly to other languages. As usual, the declaration looks a bit different:

```
var nums [100]int
nums[0] = 24
nums[1] = 22
nums[99] = 1000
//nums[100] = 10 (will not compile -- out of bounds!)
```

- Indexes are 0-based and work as you'd expect.
- Each element is set to their data type's default initial value (0 for `int`)

Iterating: len()

We can iterate through an array similarly to how we would in C/Java. Use `len()` to determine its length:

```
for i := 0; i < len(nums); i++ {
    fmt.Printf("%d %d\n", i, nums[i])
}
```

- On a related note: go only supports ‘postfix increment’
  - `i++`
  - Cannot be used as an expression (i.e., `a := i++` is not allowed!)

Ranges (“for each” loop)

```
var nums [100]int
nums[0] = 24
nums[1] = 22
nums[99] = 1000
//nums[100] = 10 (will not compile -- out of bounds!)

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

- Note: `range` is a keyword. It does not take parameters.

Unused Variables

Say we want a `for each` loop but don't need the index:

```
// This won't compile...
for i, value := range nums {
    fmt.Printf("%d\n", value)
}
```

Use _ to throw away (ignore) a variable:

```
for _, value := range nums {
    fmt.Printf("%d\n", value)
}
```

Pointers

- When variables are passed to a function, a **copy** is made
- If we want to be able to change a variable from inside a function, we can use **pointers**

```
// Takes in a pointer to an array of 100 ints:
func printArrayPointer(arr *[100]int) {
    for _, value := range arr {
        fmt.Printf("%d\n", value)
    }
}
...
printArrayPointer(&nums)
// Any changes to nums made in printArrayPointer WILL be visible here
```

Array Sizing

- Thus far, you've seen us setting an explicit size for the arrays being passed to a function.
- Umm, is that required?
  - Yes. 🙅
- But don't worry. It's not a big deal…

Slices

- Arrays are always a fixed size
- To resize an array, we can use a **slice**:

```
// [] indicates a slice (note no size is given)
func printArray(arr []int) {
    for _, value := range arr {
        fmt.Printf("%d\n", value)
    }
}
```

Slices vs Arrays [2/2]

- Go's arrays are a lot like Java's
  - You use them, but not *that* often
  - `ArrayList` (or other implementations of `List<>`) are what we use more in Java
- In go, you'll see slices being used very frequently
- So what **is** a Slice?
  - A pointer to an array
  - A size
  - A capacity

Slicing and Dicing

Use them to create “views” of your arrays:

```
// Create a slice with the first 10 elements of 'nums':
chopChop := nums[1:10]
for _, value := range chopChop {
    fmt.Printf("%d\n", value)
}
```

Or we can create a new, empty slice:

```
slicey := make([]int, 0, 100)
// len(slicey) = 0
// cap(slicey) = 100
```

In Memory

- Slices are “views” of arrays: when you *re-slice* a slice, you’re just changing where the pointer points in the array!
  - If you change the underlying array, the slice contents change too!
- A slice’s capacity is fixed since it is based on its backing array
  - **BUT** we can resize a slice easily!
  - `someSlice = append(someSlice, someNewThing)`

Resizing

- When we `append` to a slice, internally we are:
  1. Checking if we've exceeded the array capacity. If so, allocate a new slice with a backing array that's double the size
     - `make`
  2. Copy the elements over to the new slice
     - `copy`
  3. Return the new slice!
- Slices have a "slice header" that's basically a struct with this information included

Pointers after Resize

- What happens if your backing array gets resized? Are the old “slice pointers” updated?
  - **No.** They still point at old data. The Go garbage collector won’t delete it until it is unreferenced
- This sounds horrible, but in practice you probably won’t pass pointers to slice elements around
  - Generally you pass the slices around your code, so they always contain up to date pointers

Go Maps

Go has a built in Map, my favorite data structure of all time:

Let’s create one:

```
myMap := make(map[string]int)
/*                    |     |
                      |     \--> value's type
                      \--> key's type
*/
```

And put something in it: `myMap["test"] = 42`

Pre-Populating a Map

- We can create a map and add entries to it at the same time:

```
myMap := map[string]int{
    "thing1":          1,
    "thing2":          2,
    "thing3":          45,
    "thing4":          99,
    "something else": 10000,
}
```

- Don't forget the comma on the last line! (`,`)
- By the way: you can auto-format your code like above by running `go fmt`

Adding to a Map

```
ages := make(map[string]int)
ages["matthew"] = 45
ages["alice"] = 22
ages["joe"] = 99

...

ages["joe"] = 95 // The entry for 'joe' is updated w/ new value
```

Print functions can handle maps automatically:

```
fmt.Println("Here's everyone's ages:", ages)
Here's everyone's ages: map[alice:22 joe:128 matthew:45]
```

Deleting from a Map

The built-in `delete` removes items from the map. `len` reports it size, same as arrays or slices:

```
ages["matthew"] = 99
fmt.Printf(">>> %d\n", len(ages))
>>> 3

delete(ages, "matthew")

fmt.Printf(">>> %d\n", len(ages))
>>> 2
```

Checking for Items

We can use a 2nd optional return value when retrieving from a map to determine whether the element is present or not:

```
lookup := "bill"
_, present := ages[lookup]
if present {
    fmt.Println("We have " + lookup + " !")
} else {
    fmt.Println("There is no " + lookup + " here :-(")
}
```

A Common Pattern for “Contains”

- You can use the “comma ok” idiom to test for keys in a set:

```
// Assume we have a map of things:
if thing, ok := things[foo]; ok {
    // 'foo' was in the map
}

// or, if we don't care about the value:
if _, ok := things[foo]; ok {
    // 'foo' was in the map
}
```

Default Values

Let’s try accessing an item that doesn’t exist in the map:

```
fmt.Println(ages["bobby"])
```

What happens? An error? Runtime exception? Panic?

```
0
```

The default value for the datatype (`int` in this case) is returned.

Implementing a Set

- In Go, a set is a map with any type of key and a `bool` for its value:

```
professors := map[string]bool {
    "Alark"  : true,
    "Dave"   : true,
    "Sophie" : true,
}

if professors[name] {
    // The professor exists
}
```

- Since the default value of the set will be `false`, checking for a non-existing entry will return `false`

What Can be a Key?

- We can use anything that's comparable as a key. This includes:
  - boolean, numeric, string, pointer, and structs or arrays that contain only those types
  - With structs, all the members are used to evaluate equality
- We cannot use slices, maps, and functions as keys

Strings and Runes

- All go source files are UTF-8 and the language provides great support for Unicode
- Strings are represented as arrays of bytes
  - But that is problematic if we have characters outside the usual ASCII range
  - Most of the time, we interpret strings arrays of `rune` instead (32-bit integers)

Runes

```
package main
import "fmt"
// Let's check out the difference between these two loops...
func main() {
    const str = "হ্যালো 🐿️ 🤓"
    fmt.Println("--- " + str + " ---")
    for i := 0; i < len(str); i++ {
        fmt.Printf("%02d %c\n", i, str[i])
    }
    fmt.Println()
    fmt.Println("--- " + str + " ---")
    for i, runeValue := range str {
        fmt.Printf("%02d %c\n", i, runeValue)
    }
}
```