Module Six

Maps

C++ is about objects. Go is about algorithms.

Unknown

Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint-Exupéry

6.1 Introduction

Maps in Go are are implemented as references to underlying hash tables. Hash tables tend to perform in close to constant time, depending on how collisions are handled, which makes the one of the most efficient data structures available to us. We are not going to explore how maps work at the level of the hash table since going off on that tangent gets us away from talking about maps in Go and the focus of this module which is using Go maps in our code.

Because maps are references, the cost of passing them around as arguments to functions is minimal because we we are just moving pointers around and not the underlying hash table. However direct array access is generally faster in Go than map access, some authors have cited 100x faster, so this suggests that if performance is an issue then slices may be a better choice for managing data. I cannot verify or disprove these claims about performance.

6.2 Declaring, Creating and Initializing Maps

The map data structure can be thought of as a set of key value pairs (key, value) where the keys are unique and are used to reference or find their corresponding values. The underlying low-level details of how this is done is beyond the scope of this course.

6.2.1 Declaring Maps

Maps are a reference type, just like slices, and are declared using the general syntax

```
var mapname[keytype]valuetype
```

This declaration does not create the underlying hash table any more than [ ]int creates the underlying array. What has been created is a variable that can be used to point to or reference some underlying hash table. Since that table does not yet exist, we call this the nil map since it doesn't reference anything... yet.

Maps, like arrays, have types. For example the following code snippet declares severity to be a reference to a map that has keys of type string and values of type string.

```
var severity[string]string
```

If we tried to use severity to reference a map that had int as the key type and string as the value type, then we would get a compile time error because map[string]string and map[int]string are two different map types.

6.2.2 Choice of Key Data Types

The actual choice of what data type the key ought to be should be dictated by what type makes sense naturally for the real world data we are working with. As far as Go is concerned, the only requirement for a type to be usable as a key is that the comparison operators `==` and `!=` be defined for that type. Since `==` is not defined for slices, for example, they could not be used as a key type.

However, just because a data type can be used as a key does not mean it should. Several discussion questions in the lab explore why some types are not good choices for keys.

6.2.3 Declaring versus Creating Maps

Declaring a map does not create the hash table. There are two ways that we can create the underlying hash table, which, as far as we are concerned, are the same conceptually as creating the map object.

Our map reference severity declared above is the nil map because it doesn't point to anything. If we try to add data to severity, then Go will generate a panic because there is no place yet to put anything. The nil map which severity references can be thought of as being like a zero pointer in C or a null pointer in Java.

There are two ways we can create a map structure to go along with our declaration.

6.2.4 Map Literals

By supplying a map literal as an initializer, the Go compiler automatically creates the underlying hash table to hold the literal in the same way it created an underlying array when we declared a slice initialized with an array literal.

```
// Example 06-01 Creating maps with literals

package main

import "fmt"

var errs map[int]string

func main() {
  severity := map[string]string{
       "Blue":   "normal",
       "Orange": "moderate",
       "Red":    "severe"}

  severity["Black"] = "apocalyptic"

  fmt.Println(severity, " size =", len(severity))
  fmt.Println(errs, " size =", len(errs))
}
```

```
[Module06]$ go run ex06-01.go
map[Blue:normal Orange:moderate Red:severe Black:apocalyptic]
size = 4
map[]  size = 0
```

In the example above, a literal, which is a list of key value pairs, has been provided to initialize the map severity. We know the map exists because we can add an element to it.

6.2.5 Using make()

The other way to create the underlying hash table is to use make() which operates analogously to new in Java and C++ and malloc() in C. The make() function creates a hash table and returns a pointer or reference to the newly created map. Once the map has been created, then it can be accessed dynamically.

```go
// Example 06-02 Creating maps with make()

package main

import "fmt"

var errs map[int]string

func main() {
  errs = make(map[int]string)
  errs[0] = "Hardware"
  errs[1] = "Segmentation fault"
  fmt.Println(errs, " size =", len(errs))
  errs[0] = "Firmware fault"
  fmt.Println(errs, " size =", len(errs))
  fmt.Println("The errorcode '0' is a ", errs[0])
}
```

```
[Module06]$ go run ex06-02.go
map[1:Segmentation fault 0:Hardware]  size = 2
map[0:Firmware fault 1:Segmentation fault]  size = 2
The errorcode '0' is a  Firmware fault
```

Just as a point of clarification: there are two kinds of maps that are hard to tell apart because both have length 0. The first is the nil map which, as we saw earlier, is when we have a map variable declared but which does not reference an underlying hash table. The other type is the empty map where the map variable references an existing hash table with no elements. Both have length 0 and both print out the same way – with the empty string.

The difference is illustrated in example 06-03 on the next page.

```
// Example 06-03 Empty versus nil maps

package main

import "fmt"

var errs map[string]string

func main() {
  fmt.Println("Check for nil before make() ", errs == nil)
  fmt.Println("Length of errs ", len(errs))

  errs = make(map[string]string)

  fmt.Println("Check for nil after make() ", errs == nil)
  fmt.Println("Length of errs ", len(errs))
}
```

```
[Module06]$ go run ex06-03.go
Check for nil before make()  true
Length of errs  0
Check for nil after make()  false
Length of errs  0
```

6.2.6 Map Capacity

Maps grow dynamically. Every time an element is added to a map, the size of the map increases by one. In terms of efficiency, this may mean a lot of behind the scenes memory management as the underlying data is potentially reorganized every time the map grows.

This process can be made more efficient if we know in advance roughly how many entries the map will have. For example, suppose I want to automate looking up the error codes for a legacy application. Looking at my administration guide I can see that there are 2000 error codes that will eventually have to be entered into the map, however only 200 of the error codes account for 99% of all the errors that actually occur but I don't know which ones those 200 are.

I've decided I have better things to do with my time than write out a 2000 item map literal. The alternative approach I decide on is to create an initial map of 200 items and as each error occurs, it will get added to the map. The initial capacity of the map ensures that for most of the 200 additions I make to the map, there will no need for the map to re-size. If there are more than 200, then I can live with a couple of re-sizings or, if I can't, I'll just set the initial capacity to 250.

In that contrived example, the problem is small enough that there probably will not be any real savings in setting up an initial capacity, but when we scale up the possible numbers of entries or the size of the entries or frequencies of additions by several orders of magnitude – a situation we might find at a Google or Facebook – then these savings start to become significant.

The syntax for specifying an initial capacity is:

```
make(map[keytype]valuetype, capacity)
```

```
// Example 06-04 Map capacity

package main

import "fmt"

var errs map[string]string

func main() {
  errs = make(map[string]string, 200)
  fmt.Println("Length of errs ", len(errs))
}
```

```
[Module06]$ go run ex06-4.go
Length of errs  0
```

Example 06-04 shows that the capacity of a map is not the same as the length. Because there are no elements in the map, its length is 0 but it has been allocated with enough memory to store up to 200 elements without needing to re-size.

6.3 Key Existence

In the examples so far in this module, we have seen the basic syntax for adding, accessing and updating elements in a map. We have not spent a lot of time on this since presumably you are familiar with this sort of associative array syntax already from other programming languages.

However these basic operations alone are not enough for us to do all the things we would like to do in terms of processing the elements in a map. There are a couple of scenarios when we want to ensure that a specific key value either exists or doesn't exist before we perform some operation on the map. We could just test if we can fetch the item from the map first to see if it exists, but aside from being a rather cumbersome way to code, that still doesn't solve all of our problems

Fetching the element is not a solution because when we try to fetch an element that does not exist in the map, ie. we cannot find an element with the requested key, the [ ] operator returns the default zero value for the map's value type. Which means we don't really know if the entry exists and has a zero value or doesn't exist and Go is just returning a zero value.

Consider the following cases:

6.3.1 Avoiding Overwrites

Using the syntax map[key] = value, the operation actually performs two different actions.

1. If the key does not exist, then a new element is added to the map.
2. If the key does exist, the existing value associated with the key is overwritten

However I may only want the first effect to take place – if the key is already in the map, I don't want its existing value to be overwritten. How do I ensure this doesn't happen? Getting a default zero back does not give me a definitive answer.

6.3.2 Deleting Elements

Elements are deleted from maps with the following function

```
delete(mapname, keyvalue)
```

The problem is that if the key value does not exist, then a panic occurs. We would like to know if the value actually exists before we try to delete it. Even trying to fetch it doesn't help because of the default zero value again.

6.3.3 Selective Processing

Suppose that we are given a list of keys for elements that might be in our map. We want to go through our map and only process the elements whose keys that are in the list we have been given. The problem we encounter is that when we execute the statement

```
v := mapname[keyval]
```

we always get a value back. If keyval is in the map, we get the associated value back, but if the keyval is not in the map, I get back the zero value for the valuetype. Now the problem is trying to figure out if the element {keyval, value} actually exists and value just happens to be zero value, or is it that the element {keyval, value} is not in the map.

6.3.4 Comma ok form

We can test for existence by using this form:

```
value, ok = mapname[keyval]
```

ok is a Boolean value that is true if the element exists in the map and false if the element does not exist.

In example 06-05, the comma ok form is used to ensure that we only update existing entries in a map and do not add new entries.

```
// Example 06-05  comma ok update
package main

import "fmt"

func update(m map[int]int, key int, val int) {
  _, ok := m[key]
  if ok {
       m[key] = val
  }
}
func main() {
  data := map[int]int{1: 100, 3: 300}
  update(data, 1, -1)
  update(data, 2, 200)
  fmt.Println(data)
}
```

```
[Module06]$ go run ex06-05.go
map[1:-1 3:300]
```

6.4 Iteration

Just like with arrays and slices, we can iterate through a map. Unlike an array, the order of the keys is arbitrary from our perspective because the map organizes them internally in order to provide the most efficient access path.

The range operation iterates through the map in the internal order of the keys and each time through provides two items, the key and the corresponding value.

Example 6-6 demonstrates this. Notice that the elements were not printed out in the order in which they were defined nor in any recognizable lexicographic order.

```
// Example 06-06 Iteration

package main

import "fmt"

func main() {

  dotw := map[string]int{
       "Sun": 1, "Mon": 2, "Tue": 3, "Wed": 4,
       "Thu": 5, "Fri": 6, "Sat": 7}
  for day, num := range dotw {
       fmt.Printf("(%s : %d) ", day, num)
  }
}
```

```
[Module06]$ go run ex06-06.go
(Thu : 5) (Fri : 6) (Sat : 7) (Sun : 1) (Mon : 2) (Tue : 3)
(Wed : 4)
```