Declaration and Initialization

This lesson describes the important concepts regarding maps, i.e., how to use, declare and initialize them.

WE'LL COVER THE FOLLOWING

Introduction

Maps are a special kind of data structure: an *unordered* collection of pairs of items, where one element of the pair is the *key*, and the other element, associated with the key, is the *data* or the *value*. Hence they are also called **associative arrays** or **dictionaries**. They are ideal for looking up values, and given the key, the corresponding value can be retrieved very quickly. This structure exists in many other programming languages under other names such as Dictionary (`dict` in Python), `hash`, `HashTable` and so on.

Concept

A map is a reference type and declared in general as:

```
var map1 map[keytype]valuetype
```

For example:

```
var map1 map[string]int
```

(A space is allowed between `[keytype]` and `valuetype`, but `gofmt` removes it.)

The *length* of the map doesn't have to be known at the declaration, which means a map can grow dynamically. The value of an *uninitialized* map is **nil**. The key type can be any type for which the operations == and != are defined, like *string*, *int*, and *float*. For arrays and structs, Go defines the equality operations, provided that they are composed of elements for which these operations are defined using element-by-element comparison. So arrays, structs, pointers, and interface types can be used as key type, but slices cannot because equality is not defined for them. The value type can be any type.

Maps are cheap to pass to a function because only a *reference* is passed (so 4 bytes on a 32-bit machine, 8 bytes on a 64-bit machine, no matter how much data they hold). Looking up a value in a map by key is fast, much faster than a linear search, but still around 100x slower than direct indexing in an array or slice. So, if performance is very important try to solve the problem with slices.

If `key1` is a key value of map `map1`, then `map1[key1]` is the value associated with `key1`, just like the array-index notation (an array could be considered as a simple form of a map, where the keys are integers starting from 0). The value associated with `key1` can be set to (or if already present changed to) `val1` through the assignment:

```
map1[key1] = val1
```

The assignment:

```
v:= map1[key1]
```

stores the value in `v` corresponding to `key1`. If `key1` is not present in the map, then `v` becomes the zero-value for the *value* type of `map1`.

As usual `len(map1)` gives the number of *pairs* in the map, which can grow or diminish because pairs may be added or removed during runtime.

Maps are *reference* types, as memory is allocated with the `make` function. Initialization of a map is done like:

```
var map1 map[keytype]valuetype = make(map[keytype]valuetype)
```

or shorter with:

```
map1 := make(map[keytype]valuetype)
```

Run the following program to see how maps are made in Go.

```
package main
import "fmt"

func main() {
  var mapLit map[string]int   // making map
  var mapAssigned map[string]int
  mapLit = map[string]int{"one": 1, "two": 2}   // adding key-value pair
  mapCreated := make(map[string]float32)        // making map with make()
  mapAssigned = mapLit
  mapCreated["key1"] = 4.5      // creating key-value pair for map
  mapCreated["key2"] = 3.14159
  mapAssigned["two"] = 3        // changing value of already existing key
  fmt.Printf("Map literal at \"one\" is: %d\n", mapLit["one"])
  fmt.Printf("Map created at \"key2\" is: %f\n", mapCreated["key2"])
  fmt.Printf("Map assigned at \"two\" is: %d\n", mapLit["two"])
  fmt.Printf("Map literal at \"ten\" is: %d\n", mapLit["ten"])
}
```

Make Maps

In the code above, in `main` at **line 5**, we make a map `mapLit`. The declaration of `mapLit` shows that its keys will be of *string* type and the values associated with its keys will be of *int* type. Similarly, at **line 6**, we make a map `mapAssigned`. The declaration of `mapAssigned` shows that its keys will also be of *string* type and the values associated with its keys will be of *int* type too. In the next line, we assign values to `mapLit`. You can see we make two keys named `one` and `two`. The values associated with `one` is **1** and with `two` is **2**.

At **line 8**, we create a map `mapCreated` using the `make` function, which is equivalent to `mapCreated := map[string]float{}`. The declaration of `mapCreated` shows that its keys will be of *string* type and the values associated with its keys will be of *float32* type. Then in the next line, we assign `mapAssigned` with the map of `mapLit`, which means that `mapLit` and `mapAssigned` now possess the same pairs.

At **line 10** and **line 11**, we are making *key-value* pairs (each pair line by line) for `mapCreated`. At **line 10** we create key `key1` and give the value **4.5** to it. At **line 11** we create the key `key2` and give the value **3.14159** to it. Now at **line 12**, we are changing the value associated with the key of `mapAssigned`, already present before. We are giving the key `two` a new value of **3**.

Now, from **line 13** to **line 16**, we are printing certain values to verify the maps' behaviors. At **line 13**, we are printing the value associated with key `one` of `mapLit` which is **1**. In the next line, we are printing the value associated with key `key2` of `mapCreated`, which is **3.14159**. At **line 15**, we are printing the value associated with key `two` of `mapLit`, which is **3** (because of reference, as the value changes at **line 12**). In the last line, we are trying to print the value associated with the key `ten` of `mapLit`, which doesn't even exist. So **0** will be printed because `mapLit` contains *int* value types, and the default value for an integer is **0**.

> **Note**: Don't use `new`, always use `make` with *maps*.

If you by mistake allocate a reference object with `new()`, you receive a pointer to a **nil** reference, equivalent to declaring an uninitialized variable and taking its address:

```
mapCreated := new(map[string]float)
```

Then we get in the statement:

```
mapCreated["key1"] = 4.5
```

the compiler error: `invalid operation: mapCreated["key1"] (index of type *map[string] float)`.

To demonstrate that the value can be any type, here is an example of a map that has a **func() int** as its value:

```
package main
import "fmt"

func main() {
  mf := map[int]func() int{ // key type int, and value type func()int
    1: func() int { return 10 },
    2: func() int { return 20 },
    5: func() int { return 50 },
  }
  fmt.Println(mf)
}
```

Value Type of Maps

In the code above, in `main` at **line 5**, we make a map `mf`. The declaration of `mf` shows that its keys will be of *int* type and the values associated with its keys will be of **func() int** type. From **line 6** to **line 8**, we are making *key-value* pairs for `mf`. At **line 6** we create a key `1` (of type *int*) and give the value: `func() int { return 10 }` (a function returning **10**) to it. At **line** 7, we create the key `2` (of type *int*) and give the value: `func() int { return 20 }` (a function returning **20**) to it. At **line 8**, we create key `5` (of type *int*) and give the value: `func() int { return 50 }` (a function returning **50**) to it. On printing the map at **line 10**, you'll notice that *int* type keys are mapped to the *address* of the functions which are assigned to them.

Map capacity

Unlike arrays, maps grow dynamically to accommodate new key-values that are added; they have no fixed or maximum size. However, you can optionally indicate an initial capacity `cap` for the map, as in:

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

For example:

```
map2 := make(map[string]float, 100)
```

When the map has grown to its capacity, and a new key-value is added, then the size of the map will *automatically* increase by **1**. So for large maps or maps that grow a lot, it is better for performance to specify an initial capacity; even if this is only known approximately. If you don't specify the initial capacity, a default capacity of 8 bytes is taken (on a 64-bit machine). Here is a more concrete example of a map. Map the *name of a musical note* to its *frequency* in Hz:

```
noteFrequency := map[string]float32{
"C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83, "G0": 24.50, "A0": 27.
50, "B0": 30.87, "A4": 440}
```

In the above snippet, `noteFrequency` is a map. The declaration of `noteFrequency` shows that its keys will be of *string* type and values associated with its keys will be of *float32* type. The initialization is done along with the

declaration. For example, the `C0` key is given value **16.35**, the `D0` key is given value **18.35** and so on.

Slices as map values

When a key has only one associated value, the value can be a *primitive type*. What if a key is associated with many values? For example, when we have to work with all the processes on a Unix-machine where a parent process as key (process-id pid is an int value) can have many child processes (represented as a slice of ints with their pid's as items). This can be elegantly solved by defining the value type as a *[ ]int* or a *slice* of another type. Here are some examples defining such maps:

```
mp1 := make(map[int][]int)
mp2 := make(map[int]*[]int)
```

Now, you are familiar with the use of maps. In the next lesson, you'll study how to examine values present in maps independently.