**Introduction**

One of the most useful data structures in computer science is the hash table. Many hash table implementations exist with varying properties, but in general they offer fast lookups, adds, and deletes. Go provides a built-in map type that implements a hash table.

**Declaration and initialization**

A Go map type looks like this:

map[KeyType]ValueType

where KeyType may be any type that is [comparable](https://golang.org/ref/spec#Comparison_operators) (more on this later), and ValueType may be any type at all, including another map!

This variable m is a map of string keys to int values:

var m map[string]int

Map types are reference types, like pointers or slices, and so the value of m above is nil; it doesn't point to an initialized map. A nil map behaves like an empty map when reading, but attempts to write to a nil map will cause a runtime panic; don't do that. To initialize a map, use the built in make function:

m = make(map[string]int)

The make function allocates and initializes a hash map data structure and returns a map value that points to it. The specifics of that data structure are an implementation detail of the runtime and are not specified by the language itself. In this article we will focus on the *use* of maps, not their implementation.

**Working with maps**

Go provides a familiar syntax for working with maps. This statement sets the key "route" to the value 66:

m["route"] = 66

This statement retrieves the value stored under the key "route" and assigns it to a new variable i:

i := m["route"]

If the requested key doesn't exist, we get the value type's *zero value*. In this case the value type is int, so the zero value is 0:

j := m["root"]

// j == 0

The built in len function returns on the number of items in a map:

n := len(m)

The built in delete function removes an entry from the map:

delete(m, "route")

The delete function doesn't return anything, and will do nothing if the specified key doesn't exist.

A two-value assignment tests for the existence of a key:

i, ok := m["route"]

In this statement, the first value (i) is assigned the value stored under the key "route". If that key doesn't exist, i is the value type's zero value (0). The second value (ok) is a bool that is true if the key exists in the map, and false if not.

To test for a key without retrieving the value, use an underscore in place of the first value:

\_, ok := m["route"]

To iterate over the contents of a map, use the range keyword:

for key, value := range m {

fmt.Println("Key:", key, "Value:", value)

}

To initialize a map with some data, use a map literal:

commits := map[string]int{

"rsc": 3711,

"r": 2138,

"gri": 1908,

"adg": 912,

}

The same syntax may be used to initialize an empty map, which is functionally identical to using the make function:

m = map[string]int{}

**Exploiting zero values**

It can be convenient that a map retrieval yields a zero value when the key is not present.

For instance, a map of boolean values can be used as a set-like data structure (recall that the zero value for the boolean type is false). This example traverses a linked list of Nodes and prints their values. It uses a map of Node pointers to detect cycles in the list.

type Node struct {

Next \*Node

Value interface{}

}

var first \*Node

**visited := make(map[\*Node]bool)**

for n := first; n != nil; n = n.Next {

**if visited[n] {**

fmt.Println("cycle detected")

break

}

**visited[n] = true**

fmt.Println(n.Value)

}

The expression visited[n] is true if n has been visited, or false if n is not present. There's no need to use the two-value form to test for the presence of n in the map; the zero value default does it for us.

Another instance of helpful zero values is a map of slices. Appending to a nil slice just allocates a new slice, so it's a one-liner to append a value to a map of slices; there's no need to check if the key exists. In the following example, the slice people is populated with Person values. Each Person has a Name and a slice of Likes. The example creates a map to associate each like with a slice of people that like it.

type Person struct {

Name string

Likes []string

}

var people []\*Person

**likes := make(map[string][]\*Person)**

for \_, p := range people {

for \_, l := range p.Likes {

**likes[l] = append(likes[l], p)**

}

}

To print a list of people who like cheese:

for \_, p := range likes["cheese"] {

fmt.Println(p.Name, "likes cheese.")

}

To print the number of people who like bacon:

fmt.Println(len(likes["bacon"]), "people like bacon.")

Note that since both range and len treat a nil slice as a zero-length slice, these last two examples will work even if nobody likes cheese or bacon (however unlikely that may be).

**Key types**

As mentioned earlier, map keys may be of any type that is comparable. The [language spec](https://golang.org/ref/spec#Comparison_operators) defines this precisely, but in short, comparable types are boolean, numeric, string, pointer, channel, and interface types, and structs or arrays that contain only those types. Notably absent from the list are slices, maps, and functions; these types cannot be compared using ==, and may not be used as map keys.

It's obvious that strings, ints, and other basic types should be available as map keys, but perhaps unexpected are struct keys. Struct can be used to key data by multiple dimensions. For example, this map of maps could be used to tally web page hits by country:

hits := make(map[string]map[string]int)

This is map of string to (map of string to int). Each key of the outer map is the path to a web page with its own inner map. Each inner map key is a two-letter country code. This expression retrieves the number of times an Australian has loaded the documentation page:

n := hits["/doc/"]["au"]

Unfortunately, this approach becomes unwieldy when adding data, as for any given outer key you must check if the inner map exists, and create it if needed:

func add(m map[string]map[string]int, path, country string) {

mm, ok := m[path]

if !ok {

mm = make(map[string]int)

m[path] = mm

}

mm[country]++

}

add(hits, "/doc/", "au")

On the other hand, a design that uses a single map with a struct key does away with all that complexity:

type Key struct {

Path, Country string

}

hits := make(map[Key]int)

When an Vietnamese person visits the home page, incrementing (and possibly creating) the appropriate counter is a one-liner:

hits[Key{"/", "vn"}]++

And it's similarly straightforward to see how many Swiss people have read the spec:

n := hits[Key{"/ref/spec", "ch"}]

**Concurrency**

[Maps are not safe for concurrent use](https://golang.org/doc/faq#atomic_maps): it's not defined what happens when you read and write to them simultaneously. If you need to read from and write to a map from concurrently executing goroutines, the accesses must be mediated by some kind of synchronization mechanism. One common way to protect maps is with [sync.RWMutex](https://golang.org/pkg/sync/" \l "RWMutex" \t "_blank).

This statement declares a counter variable that is an anonymous struct containing a map and an embedded sync.RWMutex.

var counter = struct{

sync.RWMutex

m map[string]int

}{m: make(map[string]int)}

To read from the counter, take the read lock:

counter.RLock()

n := counter.m["some\_key"]

counter.RUnlock()

fmt.Println("some\_key:", n)

To write to the counter, take the write lock:

counter.Lock()

counter.m["some\_key"]++

counter.Unlock()

**Iteration order**

When iterating over a map with a range loop, the iteration order is not specified and is not guaranteed to be the same from one iteration to the next. If you require a stable iteration order you must maintain a separate data structure that specifies that order. This example uses a separate sorted slice of keys to print a map[int]string in key order:

import "sort"

var m map[int]string

var keys []int

for k := range m {

keys = append(keys, k)

}

sort.Ints(keys)

for \_, k := range keys {

fmt.Println("Key:", k, "Value:", m[k])

}