# Chapter 8 – Maps

A Map is 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.* The keys must be unique.

They are ideal for looking up values fast: given the key, the corresponding value can be retrieved very quickly: in V O(1) or at least O(log n).

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.

This structure exists in other programming languages under other names such as Dictionary (dict in Python), hash, HashTable, and so on.

?? In V, they are currently implemented as a binary tree.

## 8.1 Making and using maps

(?? Nov 19 // Only maps with string keys are allowed for now )

A map is initialized as a literal map like this:

numbers := {

    'one': 1

    'two': 2

}

m1 := { 'foo': 'bar', 'baz': 'bar' }

m2 := {'ford' : 'mustang', 'chevrolet' : 'camaro', 'dodge' : 'challenger'}

In V, a empty map is defined as: mut m := map[string]int

Value of key: val := m[key]

Each (key, value) pair in the map is an item in an entries array, each pair has a key and a value

Put items in a map by using the array index-like [ ] notation.

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 in v the value corresponding to key1; if key1 is not present in the map, then v becomes the zero-value for the value type of map1.

The number of key-value pairs in a map m is its size property: m.size

Here is a first example:

Listing 8.1 – maps.v:

fn main() {

  mut m := map[string]int // Only maps with string keys are allowed for now

  m['one'] = 1

  m['two'] = 2

println(m.size) // => 2

  println(m['one']) // => "1"

  // println(m) // Nov 19: map\_int` needs to have method `str() string` to be printable

  for key, val in m {

    println('$key => $val')

  }

        /\*

        one => 1

        two => 2

        \*/

  mut m2 := map[string]bool

  m2['test'] = true

println(m['bad\_key']) // => "0"

  println('bad\_key' in m) //=> false

  println(m['two']) // => "2"

m.delete('two')

  println(m['two']) // => "0"

println('two' in m) // => false

  m.delete('five') // no error for non-existent key

}

To remove a (key, value)-pair from a map, use the delete method with the key to be deleted as argument.

If m is a map, then m.keys() is an array of its keys.

Maps now support array values.

Map struct fields are now initialized automatically, just like arrays.

Examples ??

**Constructing a map with an array:**

See *map\_array.v*

fn main(){

  mut entries := map[string]int

  arr := ["apple", "banana", "coconut"]

  for item in arr {

    entries[item] = entries[item] + 1

  }

  println(entries.keys().len) // => 3

  println(entries) // => {'apple': 1, 'banana': 1, 'coconut': 1}

}

## 8.2 Testing if a key-value item exists in a map and deleting an item

Testing the existence of key1 in map1:

We saw in § 8.1 that val1 = map1[key1] returns us the value val1 associated with key1. If key1 does not exist in the map, val1 becomes the zero-value for the value’s type; in our case it printed 0 for a non-existent key.

But this is ambiguous: now we can’t distinguish between this case, or the case where key1 does exist and its value is the zero-value!

In order to test this, use the **in** operator to check if an element is a member in a map, just like for arrays:

println(m['bad\_key']) // => "0"

println('bad\_key' in m) //=> false

Deleting an item with key1 from map1:

This is done with: map1.delete(key1)

When key1 does not exist, this statement doesn’t produce an error.

Here is a concrete example, using if and assigning the if result: *map\_in.v*

fn main() {

  m := {'ford' : 'mustang', 'chevrolet' : 'camaro', 'dodge' : 'challenger'}

  y := if 'chevrolet' in m {

            'The chevrolet in the list is a ' + m['chevrolet']

        }

        else {

            'There were no chevrolets in the list :('

        }

  println(y) // The chevrolet in the list is a camaro

}

## 8.3 The for in construct

This construct can also be applied to maps, for example for showing the map contents, or processing each value in turn (see map\_forin.v):

### Print the map with a for in loop

|  |
| --- |
|  |
|  | | for key, val in m { |
|  | | println('$key => $val') |
|  | | } |
|  | |  |

/\* Output:

  one => 1

        two => 2

\*/

We see that the for in loop is used (see ch 5), where key now takes the place of the index.

The first value (key) is the key of the map, the second is the value for that key; they are local variables only known in the body of the for-statement.

// If you need only the keys:

  for key, \_ in m {

    println('$key')

  }

  /\*

  one

  two

  \*/

// or same output for:

**for key in m.keys()**{

    println('$key')

  }

// If you need only the values:

  for \_, val in m {

    println('$val')

  }

  /\*

  1

  2

  \*/

### Sorting a map on its keys

By default a map is not sorted, not even on the value of its keys.

If you want a sorted map, copy the keys (or values) to an array, sort the array, and then print out the keys and/or values using a for-in on the slice.

(see next section)

The order in which elements are visited when iterating over a map using a for range statement is unpredictable, even if the same loop is run multiple times with the same map: the first element in a map iteration is chosen at random.

This behavior allows the map implementation to ensure better map balancing. Your code should not assume that the elements are visited in any particular order.

The following code snippet prints out the contents of a map, sorted on its keys:

mut keys := m.keys()

  keys.sort()

  // Print the map

  for key in keys {

    val := m[key]

    println('$key => $val')

  }

**Showing all pairs of a map:**

see *pairs.v*

fn main() {

params := {

'k1': 'v1',

'k2': 'v2',

'k3': 'v3'

}

keys := params.keys()

pairs := keys.map('$it=${ params[it] }')

println(pairs.join(' - ')) // => k1=v1 - k2=v2 - k3=v3

}

## 8.4 An example of using a map: word\_counter.v

This is one of the examples packaged with v, you can find it at /path to v/examples/word\_counter.

It takes a file with text (for example: cinderella.txt) as argument: word\_counter [text\_file]

And produces a listing of all of its words with their frequency:

a => 25

able => 2

after => 1

afterwards => 1

again => 10

against => 2

all => 12

allow => 1

…

would => 9

yet => 2

you => 34

young => 1

younger => 2

your => 6

This is clearly a job for a map structure see word\_counter.v:

**import os**

fn main() {

  mut path := 'cinderella.txt'

  if os.args.len != 2 { (1)

    println('usage: word\_counter [text\_file]')

    println('using $path')

  }

  else {

    path = **os.args[1]** (2)

  }

  contents := **os.read\_file(path.trim\_space()) or {** (3)

    eprintln('failed to open $path')

    return

  }

  mut m := map[string]int (4)

  for word in extract\_words(contents) { (5)

    m[word] = m[word] + 1 // TODO m[key]++

  }

  // Sort the keys

mut keys := m.keys()  (6)

  keys.sort()

  // Print the map

  for key in keys { (7)

    val := m[key]

    println('$key => $val')

  }

}

// Creates an array of words from a given string

fn **extract\_words**(contents string) []string {

  mut splitted := []string

  for space\_splitted in contents.to\_lower().split(' ') {

    if space\_splitted.contains('\n') {

      splitted << space\_splitted.split('\n')

    }

    else {

      splitted << space\_splitted

    }

  }

  mut results := []string

  for s in splitted {

    result := filter\_word(s)

    if result == '' {

      continue

    }

    results << result

  }

  return results

}

// Removes punctuation

fn **filter\_word**(word string) string {

  if word == '' || word == ' ' {

    return ''

  }

  mut i := 0

  for i < word.len && !word[i].**is\_letter()** {

    i++

  }

  start := i

  for i < word.len && word[i].is\_letter() {

    i++

  }

  end := i

  return word[start..end]

}

Here a lot of V idioms we discussed earlier are applied.

Because we need to give up the name of the text file as a command-line argument, we need os.args, for that we need to import the module os. If the command doesn’t have 2 arguments, a help message is shown (see line (1)). By default the accompanying cinderella.txt file is used.

The file is read in its entirety into the contents variable with the os function read\_file

It’s result is an option type, so an or block is needed in the case no file is found. Then a message is printed to the error output with eprintln: failed to open cinderella2.txt

Then the map is created in line (4), and filled up in line (5). Then the map is sorted on its keys, and printed out (lines 6 and 7)

The helper functions extract\_words and filter\_word mainly use the split function to split the text on spaces and newlines. The is\_letter() function from module builtin is used to check whether a word character is a letter.

Exercise 8.1: map\_drinks.v

Construct a collection which maps English names of drinks to the French (or your native language) translations; print first only the drinks available, and then print both (the name and the translation). Then produce the same output, but this time the English names of the drinks must be sorted.

Exercise 8.2: map\_days.go

Make a map to hold together the number of the day in the week (1 -> 7) with its name.

Print them out and test for the presence of Tuesday and holiday.