

# Maps

02-201 / 02-601












# Arrays Store Lists of Variables

3	12	3	3	7	8	10	-2	30	6	11	11	11	32	64	80	99	-1	0	12
---	----	---	---	---	---	----	----	----	---	----	----	----	----	----	----	----	----	---	----

- A list of filenames
- A list of prime numbers
- A column of data from a spreadsheet
- A collection of DNA sequences
- Factors of a number
- etc.

Arrays are fundamental *data structures*  
Useful whenever you have a collection of  
things you want to work with together.

# What if you want to store populations of US states?

State or territory	Population estimate for July 1, 2013
California 	38,332,521
Texas 	26,448,193
New York 	19,651,127
Florida 	19,552,860
Illinois 	12,882,135
Pennsylvania 	12,773,801
Ohio 	11,570,808
Georgia 	9,992,167
Michigan 	9,895,622
North Carolina 	9,848,060
New Jersey 	8,899,339

Arrays: `var statePop []int`

Maps: `var statePop map[string]int`

```
statePop["PA"] = 12773801
statePop["CA"] = 38332521
```

Access and use like an array, but:

- you can associate data with an arbitrary *key*
- maps grow and shrink as needed as you add items

# Declaring a map variable

Basic syntax: `map[KEYTYPE]DATATYPE`

```
var grades map[string]int           // strings to ints
var rules map[string]string        // strings to strings
var multi map[string][]string      // strings to string slices
var pop map[string]float64         // strings to floats
var ssn map[int]string             // ints to strings
var families map[string]map[string]int
```

As with slices, you have to “make” a map:

```
grades = make(map[string]int)
rules = make(map[string]string)
multi = make(map[string][]string)
pop = make(map[string]float64)
ssn = make(map[int]string)
families = make(map[string]map[string]int)
```

# Mental Image of a Map

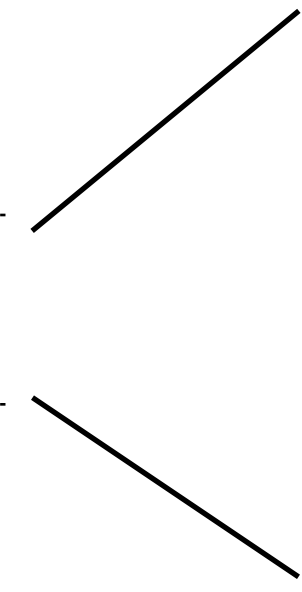
<b>Key</b>	<b>Value</b>
Albert	50.5
Bob	30.2
Ethan	65.45
Vivian	83
Dave	76.7
Rebecca	90.5
Susan	100
Charlie	82
Mike	33
Kelly	76
Sarah	95
Margaret	25
Lauren	21
Betty	91

# Mental Image of a Map of Maps

Key	Value
CMU	
Bob	•
Ethan	•
Vivian	•
Dave	•
Rebecca	•
Susan	•

Key	Value
Albert	50.5
Bob	30.2
Ethan	65.45
Vivian	83
Dave	76.7
Rebecca	90.5
Susan	100



# Using Maps

- Maps look like slices, but now you index the elements using the key:

```
grades["Carl"] = "A+++"  
fmt.Println("Rule for", x, "is", rules[x])  
ssn[627729183] = "Dave"  
paPop = pop["PA"]
```

- After you “make”, items start at their 0 value:

```
fmt.Println(grades["Chuck"]) // will print ""
```

# Map Example

- Recall this function we wrote for the Lindenmayer system:

```
// gets the Rhs for a given Lhs for a rule
func getRhsFor(char string, lhs, rhs []string) (string, bool) {
    for i, l := range lhs {
        if l == char {
            return rhs[i], true
        }
    }
    return "", false
}
```

- This assumed we had rules encoded like this:

```
lhs := []string{"A", "B"}
rhs := []string{"B-A-B", "A+B+A"}
```

- But the rules are more logically encoded as a map from a string (lhs) to another string (rhs)



# Map Example, continued

- But the rules are more logically encoded as a map from a string (lhs) to another string (rhs)

```
rules := make(map[string]string)
rules["A"] = "B-A-B"
rules["B"] = "A+B+A"
```

- Now we can write getRhsFor() much easier:

```
// gets the Rhs for a given Lhs for a rule
func getRhsFor(char string, rules map[string]string) (string, bool) {
    rhs, exists := rules[char]
    return rhs, exists
}
```

- This is (a) clearer, and (b) more efficient (no loop)

# Checking if a map contains a key

- You can check to see if a map value has ever been set explicitly:

```
paPop, exists := pop["PA"]
if !exists {
    fmt.Println("Never set PA pop!")
}
```

```
paPop := pop["PA"]
```

paPop will be whatever is stored in pop["PA"], or "" if nothing was stored there

```
paPop, exists := pop["PA"]
```

paPop will be set as above, but exists will be false if nothing was stored there


You can use any variable name here (exists is a bool variable)

# Deleting an element

- You can remove an item from a map (so it looks like you never set it to a non-zero value):

```
delete(pop, "PA")  
delete(rules, "A")  
delete(ssn, x)
```


map name, key value



# Map Literals

- Just as with arrays and slices, we can explicitly list what we want to be in a map:

```
rules := map[string]string{
    "A": "B-A-B",
    "B": "A+B+A",
}
```



(if you put this on multiple lines, the last one must have a “,” just like the others)

# Getting the Number of Elements in a Map

- Use the **len()** function to get the number of things that have been added to a map:

```
len(pop)
```

- Example:

```
m := make(map[int]int)
m[1] = 0
m[7] = 10
m[8] = 0
fmt.Println(len(m))
```

Will print 3

# Looping Through the Items in a Map

- Just as with arrays and slices, we can loop using the **for...range** loop:

```
for k, v := range pop {  
    fmt.Println("The population of", k, "is", v)  
}
```

- Note: there is **no guarantee** about which order the elements of the map will be accessed in a **for...range** statement.

# Summary

- Maps store associations between a key and a value.
- Keys must be unique within a map.
- You can use them like slices, but with more general keys.
- Maps are extremely useful.