Maps
02-201 / 02-601
Arrays Store Lists of Variables

- 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?

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

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`

```go
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:

```go
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albert</td>
<td>50.5</td>
</tr>
<tr>
<td>Bob</td>
<td>30.2</td>
</tr>
<tr>
<td>Ethan</td>
<td>65.45</td>
</tr>
<tr>
<td>Vivian</td>
<td>83</td>
</tr>
<tr>
<td>Dave</td>
<td>76.7</td>
</tr>
<tr>
<td>Rebecca</td>
<td>90.5</td>
</tr>
<tr>
<td>Susan</td>
<td>100</td>
</tr>
<tr>
<td>Charlie</td>
<td>82</td>
</tr>
<tr>
<td>Mike</td>
<td>33</td>
</tr>
<tr>
<td>Kelly</td>
<td>76</td>
</tr>
<tr>
<td>Sarah</td>
<td>95</td>
</tr>
<tr>
<td>Margaret</td>
<td>25</td>
</tr>
<tr>
<td>Lauren</td>
<td>21</td>
</tr>
<tr>
<td>Betty</td>
<td>91</td>
</tr>
</tbody>
</table>
### Mental Image of a Map of Maps

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMU</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>•</td>
</tr>
<tr>
<td>Ethan</td>
<td>•</td>
</tr>
<tr>
<td>Vivian</td>
<td>•</td>
</tr>
<tr>
<td>Dave</td>
<td>•</td>
</tr>
<tr>
<td>Rebecca</td>
<td>•</td>
</tr>
<tr>
<td>Susan</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albert</td>
<td>50.5</td>
</tr>
<tr>
<td>Bob</td>
<td>30.2</td>
</tr>
<tr>
<td>Ethan</td>
<td>65.45</td>
</tr>
<tr>
<td>Vivian</td>
<td>83</td>
</tr>
<tr>
<td>Dave</td>
<td>76.7</td>
</tr>
<tr>
<td>Rebecca</td>
<td>90.5</td>
</tr>
<tr>
<td>Susan</td>
<td>100</td>
</tr>
</tbody>
</table>
Using Maps

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

```go
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:

```go
fmt.Println(grades[“Chuck”]) // will print “”
```
Map Example

- Recall this function we wrote for the Lindenmayer system:

```go
// 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:

```go
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)

```go
rules := make(map[string]string)
rules[“A”] = “B-A-B”
rules[“B”] = “A+B+A”
```

• Now we can write getRhsFor() much easier:

```go
// 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:

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

- `paPop` will be whatever is stored in `pop["PA"]`, or "" if nothing was stored there.
- `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):

```java
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:

```go
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:

```go
m := make(map[int]int)
m[1] = 0
m[7] = 10
m[8] = 0
fmt.Println(len(m))  // Will print 3
```

• Example:
Looping Through the Items in a Map

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

```go
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.