Carl Kingford, 02-201, Fall 2015

Lecture 12: Grouping related varibles

Struct types

When working with random walks, we had to maintain 2 things: the x position, and the y position of the walker. This `(x,y)` pair is really one logical unit.

Another example: in Lindenmayer systems we had to manipulate 3 things: the x position, the y position, and the direction that the drawer was facing. These 3 things `(x,y,dir)` are logically one thing: the "state" of the drawer.

It would be nice if we could create a single variable that holds all the relevant related items. Luckily, we can! These are "structures":

```
type WalkerPosition struct {
    x float64
    y float64
    dir float64
}

type Pen struct { // details about the location of the drawing pen
    x float64
    y float64
    dir float64
}
```

Lists collect values of the same type, `struct`s can collect values of different types. Consider the Contacts application on your smartphone:

We can represent all the information for each contact using a struct:

```
type Contact struct {
    firstName string
    lastName string
    company string
    mobile []int
    homeEmail string
    homePage string
}
```

The above code creates a new type called `Contact`. Each one of the variables inside the struct is called a field. These variables are contained within a `Contact` structure and any time you create a Contact, these variables will be created automatically.

Creating structure variables

```
var person Contact   // creates a person variable of type Contact
```

Once you've created a Contact type, you can use it anyplace you used any of the builtin types. You can pass them into functions:

```
func printContact(c Contact) {
    // print the contact
}
```

You can return them from functions:

```
func createContact(n string) Contact {
    // create a contact from name n
}
```

Accessing fields in structures

You can get the fields of a struct using the "." (dot) syntax:

```
func printContact(c Contact) {
    fmt.Println("Name:", c.firstName + " " + c.lastName)
    fmt.Println("Company:", c.company)
    fmt.Println("Email:", c.homeEmail)
    fmt.Println("Web:", c.homePage)
}
```

You can assign to a field of a struct using the same "." syntax:

```
func createContact(n string) Contact {
    var c Contact
    c.firstName = n
    c.lastName = "Unknown"
    return c
}
```

These `c.firstName` variables act just like regular variables, and you can manipulate them in the same way. The only difference is that they are bundled together in a struct.

Example: A stack that contains `Pen` data

Suppose we want to have a stack that implements the `[` and `]` operators of L-systems:

```
// The pen state is now represented by a struct type
type Pen struct {
    x, y float64
    dir float64
}

// You can create a list of Pen structs just as you would any other list.
func createPenStack() []Pen {
   return make([]Pen, 0)
}


func pushPen(S []Pen, item Pen) []Pen {
   // You can manipulate the []Pen exactly as before.
   return append(S, item)
}

func popPen(S []Pen) ([]Pen, Pen) {
   if len(S) == 0 {
       panic(“Can’t pop empty stack!")
   }
   item := S[len(S)-1]
   S = S[0:len(S)-1]
   return S, item
}
```

Complex Data Structures

Maps, lists, structs, and regular variables can be combined in complicated ways in order to organize your data in a way that makes the most sense for what you want to do.

Maps of structs:

For example, you can create maps of structs:

```
var people map[string]Contact
```

Now `people["03ACX"].firstName` can contain the first name of employee with employee id "03ACX".

A Go quirk! Go disallows *assigning* values to fields of a struct contained in an array:

```
// BAD CODE BAD CODE BAD CODE
people["03ACX"].firstName = "Carl" // Go doesn't allow this
// BAD CODE BAD CODE BAD CODE
```

There is no good reason for this. The Go designers think there is a good reason, but they are wrong. The way around this is the following:

```
tmp := people["03ACX"]   // make a copy of the struct
tmp.firstName = "Carl"   // change it
people["03ACX"] = tmp    // put it back in the map
```

Lists of structs

Again, can create lists of struct types just as you would any other:

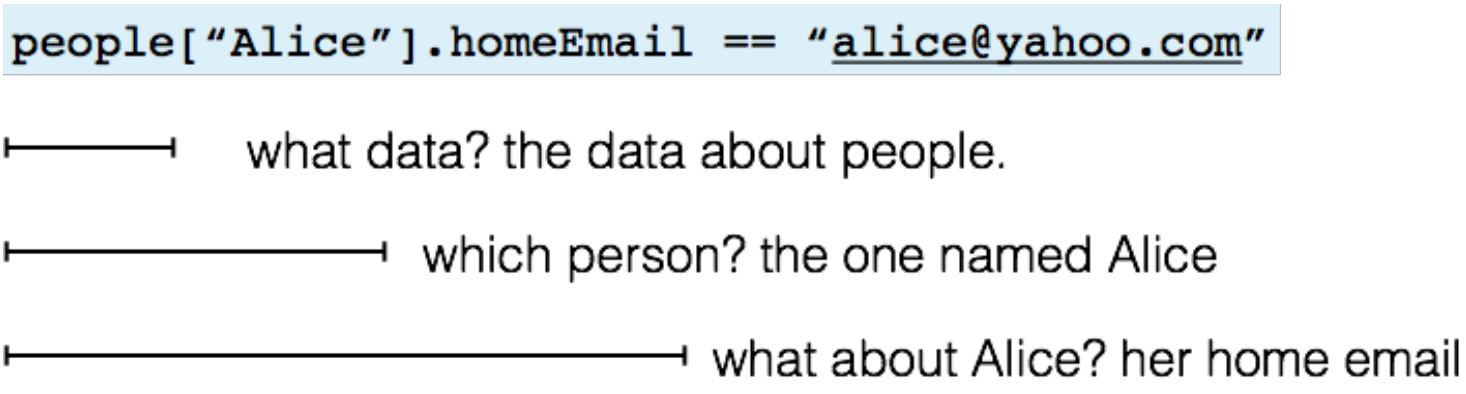
```
var employees = make([]Contact, 100)
```

You access the items as usual:

```
employees[10].mobile = make([]int, 10)
```

Note: when you create a Contact, it is initialized so that all its fields are their "0" value. This means any lists inside of the struct are `nil` and need to be `make` ed.

Combining data types let you organize data in complex ways:



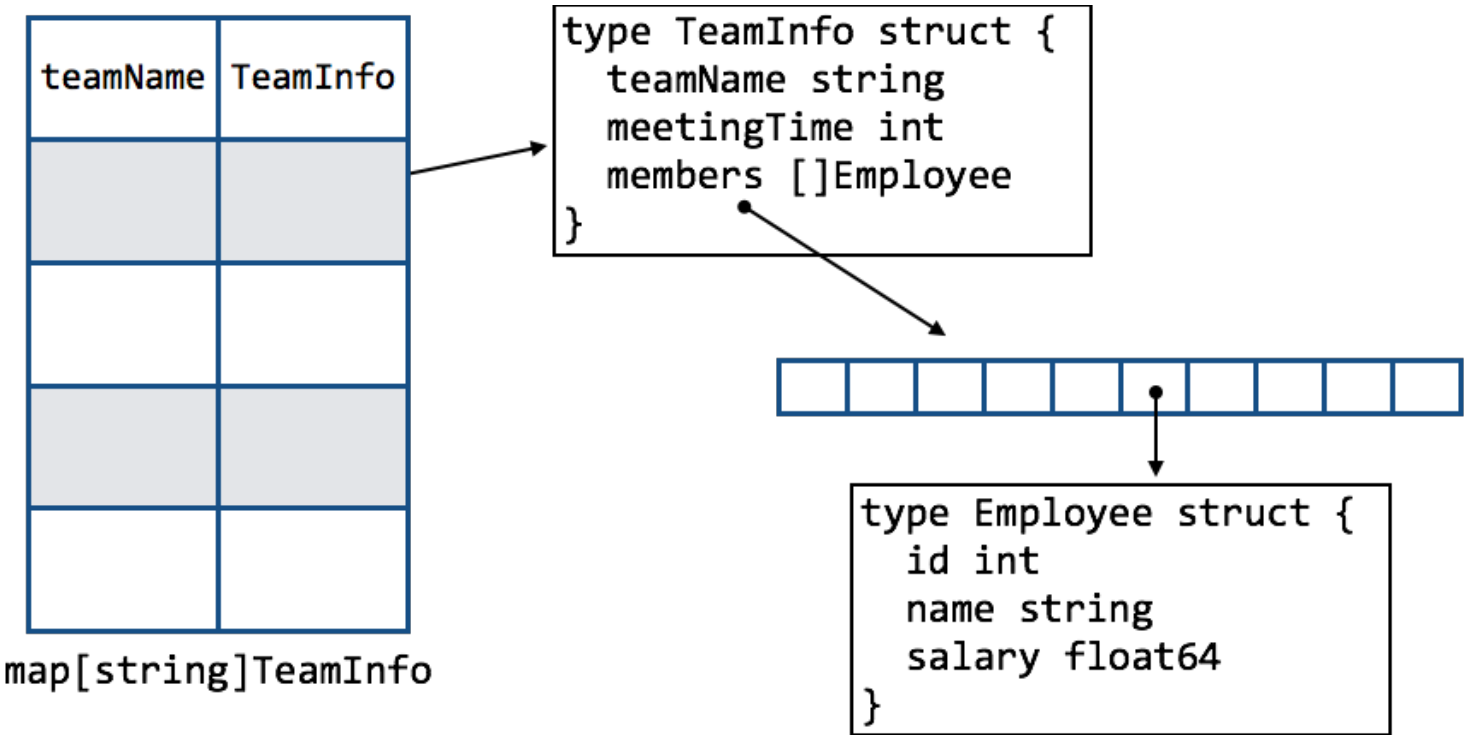
Example of a complex data structure

Suppose you run a small company that has several teams of employees. Each team has a name, a meeting time, a list of members. Each employee has an id, a name, and a salary.

You want to be able to:

- compute the total cost of a team, and
- see if any employee is on two different teams that meet at the same time

You might store this data in the following way:



Writing `teamCost()`:

The cost of a team is the total cost of the salaries of the members of the team.

Computing the total cost of a team:

```go
// returns the total cost of team t
func teamCost(teams map[string]TeamInfo, t string) float64 {
    var sum float64
    for i := range teams[t].members {
        sum = sum + teams[t].members[i].salary
    }
    return sum
}
```

Our organization of the data let's us find the members of a team with a simple `teams[t].members` statement, which has the type of a list that we can iterate over.

Writing `timeConflict()`:

We want to check if any employee is on two different teams that meet at the same time.

This is harder, since the way we organized the data doesn't let us directly find teams by meeting time or even the teams an employee is on.

Any ideas?

```go
// returns true if an employee has a time conflict
func timeConflict(teams map[string]TeamInfo) bool {

   // we create a data structure to reorganize our data to answer the
   // question. meetTimes[id][time] will be true if employee `id` has
   // a meeting at time `time`
    meetTimes := make(map[int]map[int]bool)

    // for every employee
    for _, info := range teams {
        for _, emp := range info.members {
            // if we haven't make the map for this employee yet
            _, exists := meetTimes[emp.id]
            if !exists {
                meetTimes[emp.id] = make(map[int]bool)
            }
            // if we added this meeting time to this emp in the past
            if meetTimes[emp.id][info.meetingTime] {
                fmt.Println("Employee", emp.name,
                    "has 2 meetings at", info.meetingTime)
                return true
            }
            meetTimes[emp.id][info.meetingTime] = true
        }
    }
    return false
}
```

> **Test yourself!** What data structuremight you use to represent a Minecraft world?

Summary

- Structs group a "small" number of related variables together to be manipulated as a unit.
- They are useful when your logical state has multiple parts to it.
- The `type` statement lets you define new types that work like the built-in types you've used many times already.
- Maps, slices, structs, variables let you create complex organization of your data to make answering the questions you want to answer easier.

Glossary

- struct: groups related variables together so they can be passed around together
- field: a struct contains fields of the same or different types