Structs

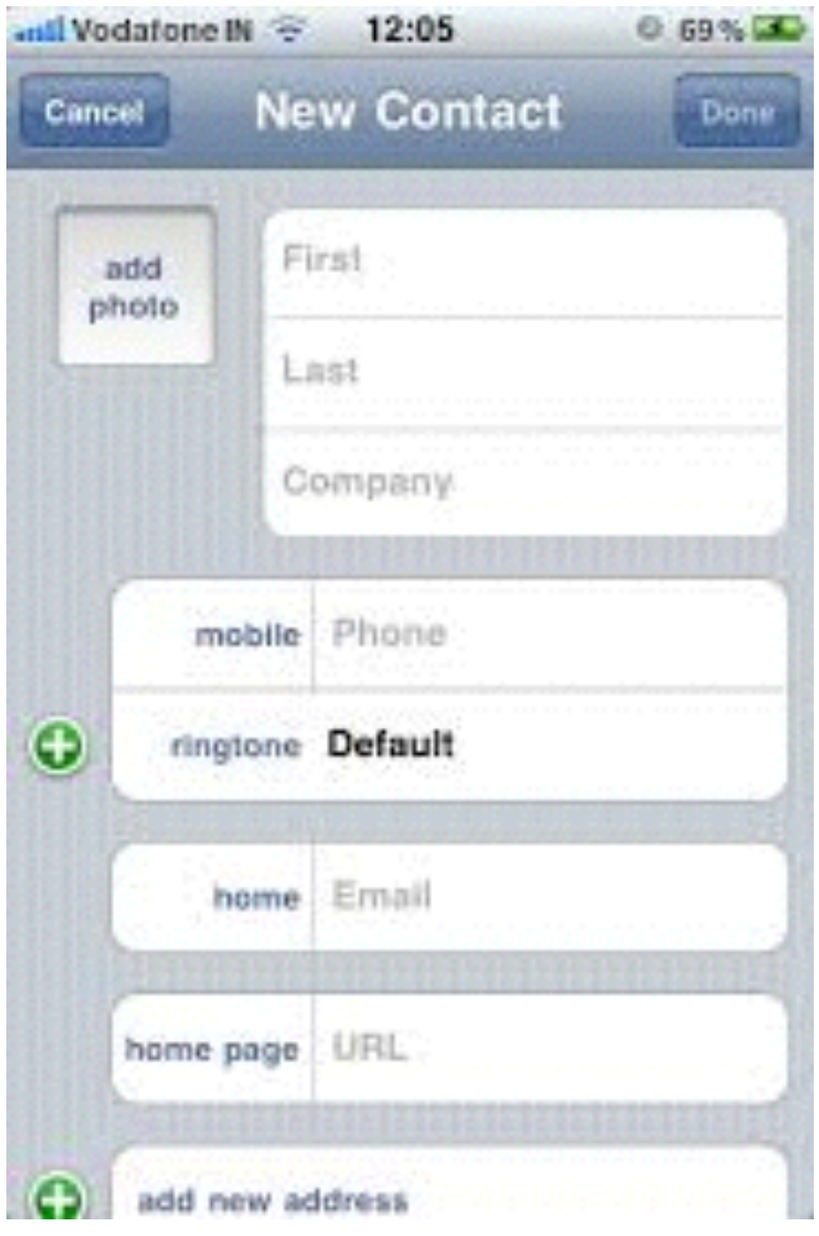
02-201 / 02-601

Lindenmayer Stack Example

```
Xstack := createStack()
Ystack := createStack()
DirStack := createStack()

// …

Xstack = push(Xstack, x)
Ystack = push(Ystack, y)
DirStack = push(DirStack, dir)

// …

Xstack, x = pop(Xstack)
Ystack, y = pop(Ystack)
DirStack, dir = pop(DirStack)
```

- Needed to create 3 stacks to hold 3 different values.
- *Logically* (x,y,dir) is one thing: the current state of our drawing pen
- This is a common situation:
  - an experiment might measure temperature, humidity, salinity
  - a person might have have name, an id#, an address, and a phone number.
- You will want to manipulate these *logical entities* as a single *program entity.*

structs




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

- Creates a new type called “Contact”
- This type contains within it *fields* corresponding to other variables.
- These variables are contained within a Contact struct and any time you create a Contact, these variables will be created automatically.

Creating a struct variable

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

Once you've created a Contact type, you can use it anyplace you used any of the builtin types:

You can create Contact variables:

```
var me Contact
```

Pass Contact variables into functions:

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

Return Contact variables from functions:

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

Accessing the fields of a struct

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

- 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)
}
```

Setting the Values of a Struct

- 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.

A Better Lindenmayer Stack

```
type Pen struct {
    x, y float64
    dir float64
}

func createPenStack() []Pen {
   return make([]Pen, 0)
}

func pushPen(S []Pen, item Pen) []Pen {
   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
}
```

The pen state is now represented by a struct type

You can create a slice of Pen structs just as you would any other slice.

You can manipulate the []Pen exactly as before.

Using the Pen Stack

```go
func drawPlant(s string) {
    const w, h = 10000, 10000
    pic := CreateNewCanvas(w, h)

    var myPen Pen
    myPen.x, myPen.y = 0.5*w, 0.5*h
    myPen.dir = 0.0
    step := 2.0

    penStack := createPenStack()

    pic.MoveTo(myPen.x,myPen.y)
    for _, c := range s {
        switch c {
        case 'F':
            myPen.x = myPen.x + step * math.Cos(myPen.dir)
            myPen.y = myPen.y - step * math.Sin(myPen.dir)
            pic.LineTo(myPen.x, myPen.y)

        case '+':
            // turn left
            myPen.dir = myPen.dir + math.Pi * (25.0 / 180.0)

        case '-':
            // trun right
            myPen.dir = myPen.dir - math.Pi * (25.0 / 180.0)

        case '[':
            // save
            penStack = pushPen(penStack, myPen)

        case ']':
            // restore
            penStack, myPen = popPen(penStack)
            pic.MoveTo(myPen.x,myPen.y)

        case 'X':

        default:
            panic("Wow, somethings really wrong.")
        }
    }
    pic.Stroke()
    pic.SaveToPNG("Plant.png")
}
```

Instead of creating x,y,dir individually, we create a single Pen variable

We can now push and pop Pens directly onto our Pen stack.

Suppose we wanted to add new rules like:

- ^: increases pen width
- v: decreased pen width
- R: changes pen color to red

We could add fields to our Pen struct and only need to add code to handle these new Lindenmayer commands.

Struct Literals

- This code initializes the value of the myPen struct.
- It's a little clunky (repeat "myPen" a lot, e.g.)

```
var myPen Pen
myPen.x, myPen.y = 0.5*w, 0.5*h
myPen.dir = 0.0
```

- Setting the initial values of a struct is a very common thing to do.
- Can use "struct literals" to do it:

```
var myPen = Pen{x: 0.5*w, y: 0.5*h, dir:0.0}
```

The name of the struct type

A field name

The value for the field

Using struct literals

```go
func drawPlant(s string) {
    const w, h = 10000, 10000
    pic := CreateNewCanvas(w, h)

    var myPen = Pen{x:0.5*w, y:0.5*h, dir:0.0}
    step := 2.0

    penStack := createPenStack()

    pic.MoveTo(myPen.x,myPen.y)
    for _, c := range s {
        switch c {
        case 'F':
            myPen.x = myPen.x + step * math.Cos(myPen.dir)
            myPen.y = myPen.y - step * math.Sin(myPen.dir)
            pic.LineTo(myPen.x, myPen.y)

        case '+':
            // turn left
            myPen.dir = myPen.dir + math.Pi * (25.0 / 180.0)

        case '-':
            // trun right
            myPen.dir = myPen.dir - math.Pi * (25.0 / 180.0)

        case '[':
            // save
            penStack = pushPen(penStack, myPen)

        case ']':
            // restore
            penStack, myPen = popPen(penStack)
            pic.MoveTo(myPen.x,myPen.y)

        case 'X':

        default:
            panic("Wow, somethings really wrong.")
        }
    }
    pic.Stroke()
    pic.SaveToPNG("Plant.png")
}
```

Can create an initialize the pen at the same time

Notice code is getting:

(a) shorter

(b) clearer since the program entities now better correspond to the logical things we're modeling

Another Common Case: maps of structs

- You can create maps where the values are structs:

```
var people map[string]Contact
people[“Carl”].company = “Carnegie Mellon”
people[“Dave”].firstName = “Mike”
```

- These data structures let you organize data in complex ways.

```
people[“Alice”].homeEmail = “alice@yahoo.com”
```

what data? the data about people.

which person? the one named Alice

what about Alice? her home email

Side Note:

- You don't *need* to define a struct as a new type to use structs:

```
var people map[string]struct{
    company string
    firstName string
}
people[“Carl”].company = “Carnegie Mellon”
people[“Dave”].firstName = “Mike”
```

- But this quickly becomes tiring to type and it makes it harder to pass structs around to functions, etc.

- Tip: always make a new type for your structs.

Another Common Case: Slices of Structs

- Again, can create slices 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 slices inside of the struct are **nil** and need to be “**make**”ed.

Data Structure Example

Example: 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

Data Structure Example

Example: 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

teamName	TeamInfo

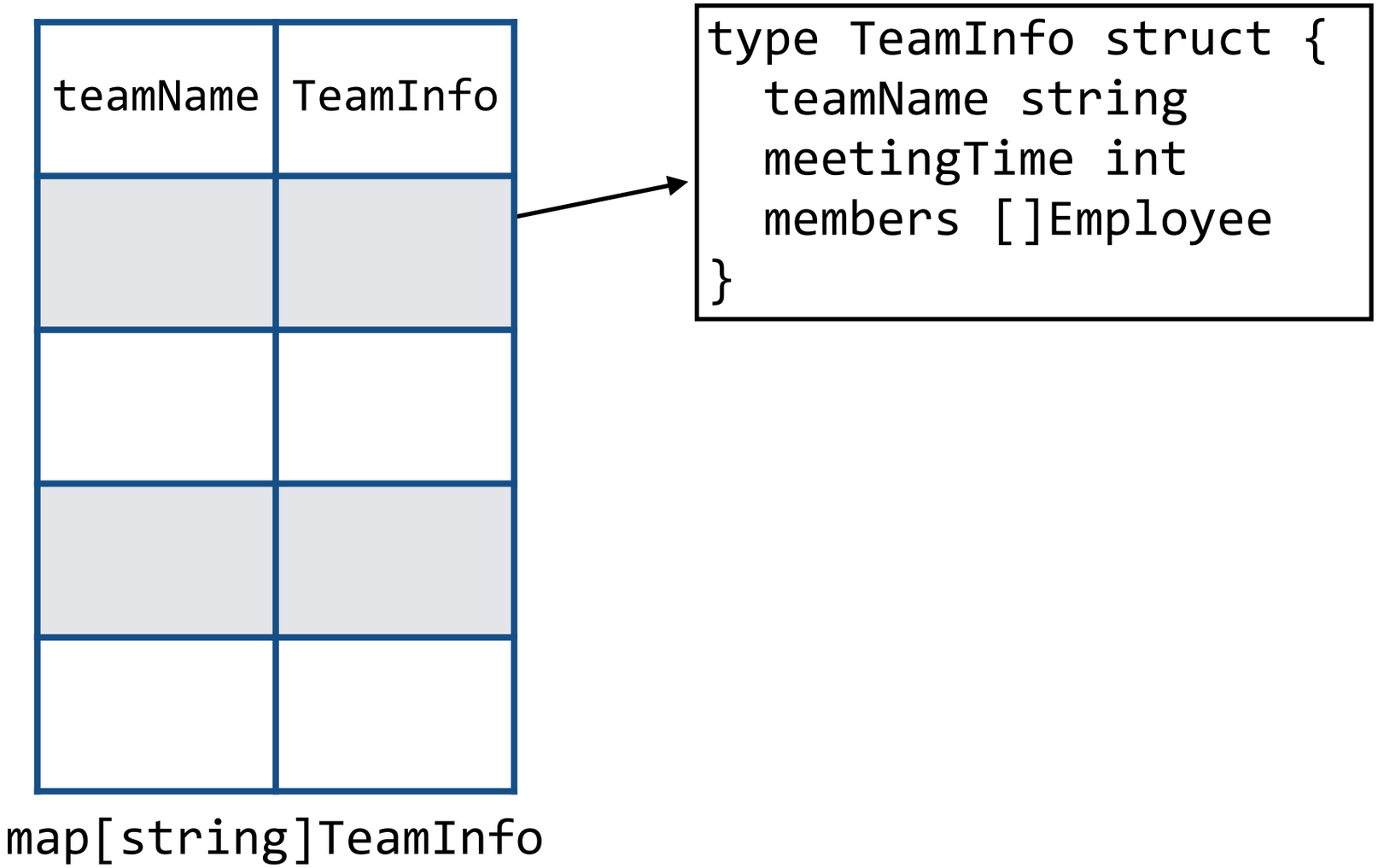
map[string]TeamInfo

Data Structure Example

Example: 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

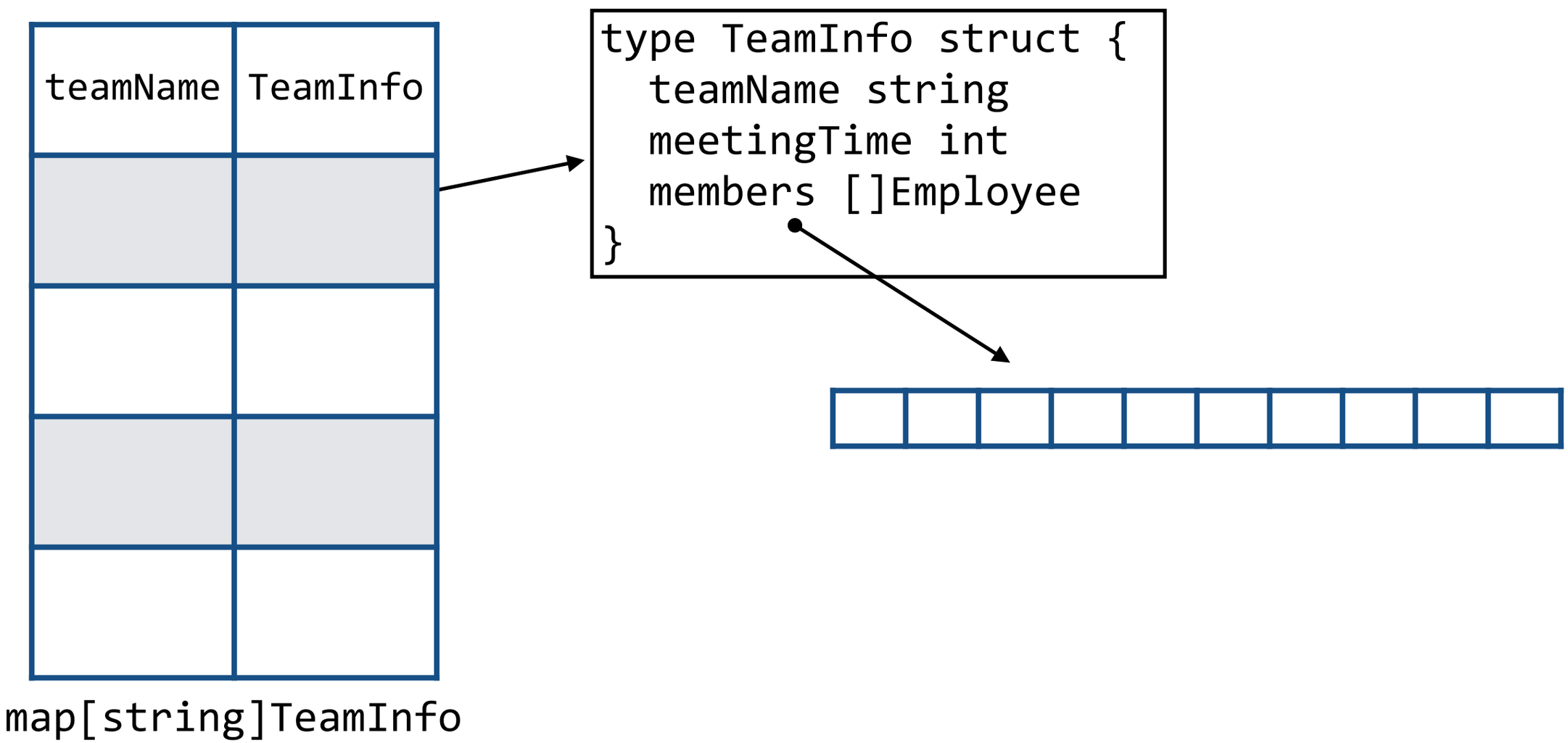


Data Structure Example

Example: 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

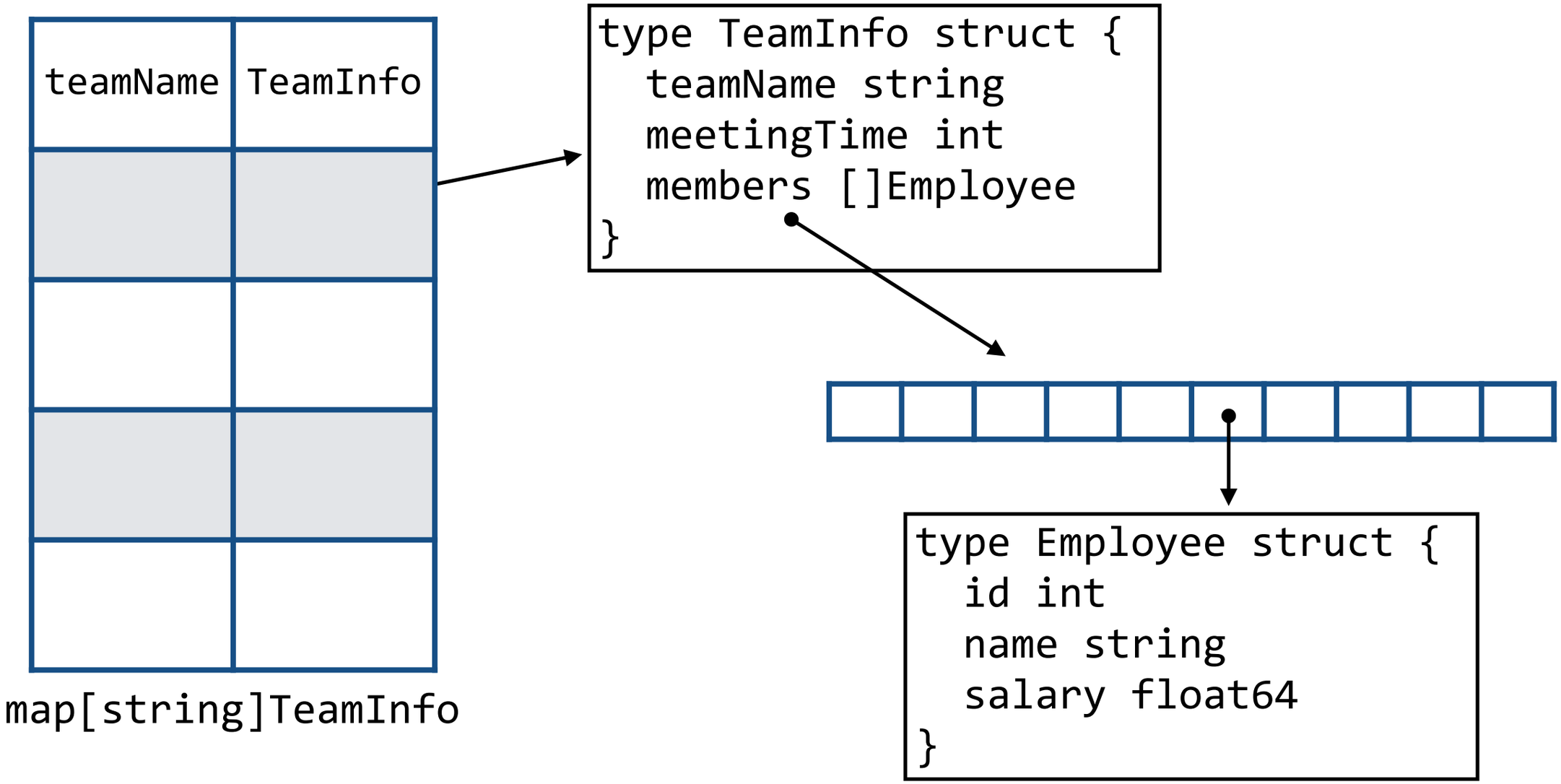


Data Structure Example

Example: 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



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:

```
// returns the total cost of team t
func teamCost(teams map[string]TeamInfo, t string) float64 {
```

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

```
// returns the total cost of team t
func teamCost(teams map[string]TeamInfo, t string) float64 {
    var sum float64
```

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

```
// returns the total cost of team t
func teamCost(teams map[string]TeamInfo, t string) float64 {
    var sum float64
    for _, emp := range teams[t].members {
```

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

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

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

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

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

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

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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:

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

- Our organization of the data let's us find the members of a team with a simple "teams[t].members" statement.

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?

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

    // for every employee
```

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

    // for every employee
    for _, info := range teams {
```

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

    // for every employee
    for _, info := range teams {
        for _, emp := range info.members {
```

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

```go
// returns true if an employee has a time conflict
func timeConflict(teams map[string]TeamInfo) bool {
    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[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

```go
// returns true if an employee has a time conflict
func timeConflict(teams map[string]TeamInfo) bool {
    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[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

```go
// returns true if an employee has a time conflict
func timeConflict(teams map[string]TeamInfo) bool {
    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[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Writing timeConflict()

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

`meetTimes[id][time]` will be true if employee with `id` has a meeting at `time`.

Complex Literal Data Example

```go
func main() {
    company := make(map[string]TeamInfo)

    company["appleWatch"] = TeamInfo{
        teamName: "appleWatch",
        meetingTime: 10,
        members: []Employee{
            Employee{id: 7, name: "Carl", salary: 1.0},
            Employee{id: 3, name: "Dave", salary: 50.0},
        },
    }

    company["iPhone"] = TeamInfo{
        teamName: "iPhone",
        meetingTime: 3,
        members: []Employee{
            Employee{id: 4, name: "Mike", salary: 101.0},
            Employee{id: 8, name: "Sally", salary: 151.0},
        },
    }

    company["iMac"] = TeamInfo{
        teamName: "iMac",
        meetingTime: 10,
        members: []Employee{
            Employee{id: 7, name: "Carl", salary: 1.0},
            Employee{id: 10, name: "George", salary: 75.0},
            Employee{id: 11, name: "Teresa", salary: 92.0},
        },
    }

    fmt.Println(teamCost(company, "appleWatch"))
    fmt.Println(timeConflict(company))
}
```

- Typically you would read in your data from a file or user input (we'll see how soon)
- But sometime (especially for testing) it's useful to be able to specify your data right in the program.
- Example at left.

Summary

- Structs group a “small” number of related variables together to be manipulated as a unit.
- Good 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.