Pointers

02-201 / 02-601
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))
}
<table>
<thead>
<tr>
<th>teamName</th>
<th>TeamInfo</th>
</tr>
</thead>
</table>
| “appleWatch” | struct {
  
  
  teamName = “appleWatch”
  
  meetingTime = 10
  
  members = struct {
    
    id = 7
    
    name = “Carl”
    
    salary = 1.0
    
  } struct {
    
    id = 3
    
    name = “Dave”
    
    salary = 50.0
    
  } |
| “iPhone” | struct {
  
  
  teamName = “iPhone”
  
  meetingTime = 3
  
  members = struct {
    
    id = 4
    
    name = “Mike”
    
    salary = 101.0
    
  } struct {
    
    id = 8
    
    name = “Sally”
    
    salary = 151.0
    
  } |
| “iMac” | struct {
  
  
  teamName = “iMac”
  
  meetingTime = 10
  
  members = struct {
    
    id = 7
    
    name = “Carl”
    
    salary = 1.0
    
  } struct {
    
    id = 11
    
    name = “Teresa”
    
    salary = 92.0
    
  } struct {
    
    id = 10
    
    name = “George”
    
    salary = 75.0
    
  } |
Instead of storing Employee structs in the members slice directly, we can store a pointer to a struct.
Type a TeamInfo struct {
    teamName string
    meetingTime int
    members []*Employee
}

The "*" means "pointer to"

This is a slice of pointers to Employee structs

Can have pointers to most types:

```go
var name *string
var person *Employee
var pj *int
var m map[string]*Employee
var pA *[10]float64
var Apf [10]*float64
```
Your computer’s memory is a long chain of cells numbered 0 to some large number.

Each variable you declare take up some number of these cells.
A pointer is a variable that holds the address of some other variable.

```go
struct Employee {
    ...
}

var A int = 3

var E *Employee = 12
```
Setting What a Pointer Points To

var P Employee = createEmployee()
var person *Employee

// at this point, person == nil

person = &P

The “&” operator means “address of”

Another example:

var i int = 10
var p *int = &i

i == 10

p == &i
Accessing What a Pointer Points To

```go
var i int = 10
var j int = 10
var p *int = &i

i = 11
fmt.Println(*p) .......... 11
fmt.Println(p) .......... some big number

*p = 300
fmt.Println(*p) .......... 300
fmt.Println(p) .......... the same big number
fmt.Println(i) .......... 300

p = &j
fmt.Println(*p) .......... 10
*p = 12
fmt.Println(*p) .......... 12
```

You access what p points to by prefixing p with *
Pointers are “meta” things:
An Employee is a piece of data, an “object” of your program.
A *Employee is a reference to that object.
A variable of type *Employee is not an Employee.
Accessing the fields of a struct through a pointer

```go
var P Employee = createEmployee()
var person *Employee

// at this point, person == nil

person = &P

(*person).name = "Jerry"
```

This is so common, Go provides a shortcut: just use the pointer to a struct like a struct:

```go
person.name = "Jerry"
```
type Contact struct {
    name string
    id int
}

func main() {
    var c Contact = Contact{name:"Dave", id:33}
    var p *Contact = &c
    fmt.Println(c)
    fmt.Println(*p)
    (*p).name = "Holly"
    p.id = 33
    fmt.Println(*p)
}
Example: Passing A Struct to a Function

What’s wrong with this code?

```go
type Contact struct {
    name string
    id   int
}

func setContactInfo(c Contact) {
    c.name = "Holly Golightly"
    c.id = 101
}

func main() {
    var c Contact = Contact{name:"Dave", id:33}
    setContactInfo(c)
    fmt.Println(c)
}
```

How do we fix it?
Example: Passing A Struct to a Function

Pass the *address* of a Contact to `setContactInfo`:

```go
type Contact struct {
    name string
    id   int
}

func setContactInfo(c *Contact) {
    c.name = "Holly Golightly"
    c.id = 101
}

func main() {
    var c Contact = Contact{name:"Dave", id:33}
    setContactInfo(&c)
    fmt.Println(c)
}
```
Example: How is a Slice Implemented

- Conceptually, a slice is a struct containing 3 things:

```go
struct {
    startIndex int
    endIndex   int
    array      *[100]float64
}
```

- This is why:
  - Subslices point to the original data
  - Passing slices to functions doesn’t copy the data

- This is only a *conceptual* equivalence.
  Go treats slices differently than these structs.
**Pointer Summary**

- Pointers store addresses of other variables.

- Declare by prefixing type with `*`.

- Access the variable they point to by prefixing the pointer with `*`.

- Get the address of a variable (to assign to a pointer) via `&`.

- Most common use: pointers to structures.