Pointers and Methods

02-201 / 02-601
Pointers
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
  }
<p>|</p>
<table>
<thead>
<tr>
<th>teamName</th>
<th>TeamInfo</th>
</tr>
</thead>
</table>
| appleWatch | struct {
|          |   
|          |   
|          |   
|          |   }               |
| iPhone   | struct {
|          |   
|          |   
|          |   }               |
| iMac     | struct {
|          |   
|          |   
|          |   }               |

Instead of storing Employee structs in the members slice directly, we can store a pointer to a struct.
**Pointer Types**

```go
type 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 takes up some number of these cells.
A pointer is a variable that holds the address of some other variable.
Setting What a Pointer Points To

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

// at this point, person == nil
person = &P

The “&” operator means “address of”

Another example:
```
```go
var i int = 10
var p *int = &i
```
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"
```
```go
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?

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 *

• Most common use: pointers to structures
“Objects” & Methods
“Objects”

• Top-down design: start from the big problem and break it into smaller problems, writing a function for each of the smaller problems.

• Another useful way of thinking: describe the organization of your data and have that reflected in your program.
  
  • A contact management program will manipulate **Contacts**
  
  • A drawing program will manipulate a **Canvas**, and perhaps **Lines**, **Colors**, and **Shapes**
  
  • Facebook will manipulate **Users**, **Posts**, and **Advertisements**
  
  • Twitter will manipulate **Tweets**, **Users**, **Advertisements**

• These are the “nouns” of these programs.
These two ways of thinking complement each other
Objects + Operations

- Once you've decided on the “nouns”, you choose the “verbs” that apply to those nouns.

Example:

Your “noun” is a Tweet:

```go
type Tweet struct {
    text string
    time uint64
    who *User
}
```

- Get Hashtags in Tweet
- Get Direct Mentions in Tweet
- Shorten URL in Tweet
- Get URLs in Tweet
- Get Short Version of Tweet

Your “noun” is a User:

```go
type User struct {
    name string
    followers []*User
    following [][]*User
}
```

- Direct Message User
- Add Follower
- Remove Follower
- Add Following User
- Remove Following User
- Get All Tweets from Followed Users
Example 2: Contacts

Operations you will need to perform on a Contact:

- Get First Name
- Get Last Name
- Set First Name
- Set Last Name
- Get Formatted Phone Number
- Call
- Count Friends
- Add Friend
- Give Raise

Your “noun” is a Contact:

type Contact struct {
    name string
    id int
    salary float64
    friends []*Contact
    phone []int
}

Example 3: Spatial Games

```go
type Field struct {
    cells [][]Cell
}
```

- Count Cell Kinds in Neighborhood
- Read Field From File
- Evolve Single Step
- Save Field To File
- Draw Field
- Check Field is Valid
- Zero All Scores

```go
type Cell struct {
    kind string
    score float64
    prevKind string
}
```

- Zero Score
- Set Kind
- Get Kind
- Get Previous Kind
- Get Cell Color
Example 4: Real World Example

- Too complex to be a good example for class, but I wanted to show that this kind of thinking is actually used:
Example 5: Canvas

```go
type Canvas struct {
    gc    *draw2d.ImageGraphicContext
    img   image.Image
    width int
    height int
}
```

Operations you can perform on a Canvas:

- `MoveTo(c *Canvas, x, y float64)`
- `LineTo(c *Canvas, x, y float64)`
- `SetStrokeColor(c *Canvas, col color.Color)`
- `SetFillColor(c *Canvas, col color.Color)`
- `SetLineWidth(c *Canvas, w float64)`
- `Stroke(c *Canvas)`
- `FillStroke(c *Canvas)`
- `Fill(c *Canvas)`
- `ClearRect(c *Canvas, x1, y1, x2, y2 int)`
- `SaveToPNG(c *Canvas, filename string)`
- `Width(c *Canvas)`
- `Height(c *Canvas)`

These operations are logically related: they are the things you can do to a Canvas.

They are functions called “methods”.

They all take a *Canvas as their first parameter.

Go provides a special syntax for this situation (next slide).
Go’s Method Syntax

• Same as a function definition, with one addition:

```go
func (c *Canvas) SetStrokeColor(col color.Color) {
    c.gc.SetStrokeColor(col)
}
```

Move the logical “first” parameter to before the name of the function.

Can use “c” just like any other parameter.

• Now use “dot” syntax to call the method:

```go
var pic *Canvas = MakeCanvas()
pic.SetStrokeColor(blue)
```
What’s the Point?

• Logically groups operations with the data they operate on

• Supports the “noun” / “verb” way of designing programs directly

• Let’s you use the same function name for different object types:
  • (c *Canvas) Draw()
  • (b *Button) Draw()

are different functions.
Method Summary

• Methods are functions that are associated with a type.

• If you have a variable X, you can call any of its methods using:

   X.methodName(param1, param2)

   This works like a normal function call.

• This is “object-oriented programming”