Lecture 14: Pointers

Suppose we store information and employees and teams using the following types:

```go
type TeamInfo struct {
    teamName string
    meetingTime int
    members []Employee // note difference from class!
}

type Employee struct {
    id int
    name string
    salary float64
}
```

We can then store all the info about the company in a map from team names to `TeamInfo` structures:
Notice the lines marked with `// DUP!!!`: there are two entries for Carl because he is on two different teams. This is a waste of space, error prone because you must update each duplicate if any of the data changes, and slower because you always have to search for every duplicate.

The above structure of the data can be visualized as:
In class, we saw a fix to this that used the employee id to link employees into the teams. This situation is common enough, that most programming languages have a solution for it, which is pointers.

Pointers

We’d like to change the TeamInfo type so that each employee is represented once:
The way we can do this is to change the `TeamInfo` type to:

```go
type TeamInfo struct {
    teamName string
    meetingTime int
    members []*Employee // the "*" means "pointer"
}
```

The `[*Employee]` type is a list ([]) of pointers (*) to `Employees`.

**What is a pointer?**

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:

The location of the variable is called its **address**.

A pointer is a variable that contains the address of some other variable:
A pointer is a variable that contains an address of another variable. To get the address of a variable you use the `&` (address) operator:

```c
struct Employee {
...
}

var A int = 3

var E *Employee = 12

struct Employee {
...
}
```

**Setting what a pointer points to**

A pointer is a variable that contains an address of another variable. To get the address of a variable you use the `&` (address) operator:
Another example:

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

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

This creates the following situation:

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

Accessing the value of the variable that a pointer points to

Suppose you have the following statements:

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

Then `i`'s value is 10 and `p`'s value is the address of `i`. If you `fmt.Println(p)` you will get some large integer (probably in hexadecimal) that is the memory address where `i` is stored. If, however, you do

```
fmt.Println(*p)
```

you will print out "10". The `*` operator, when put before a pointer means "follow the pointer". The expression `*p` acts almost exactly like `i`. Consider the following statements:
Assigning to `*p` is exactly the same thing as assigning to `i`. And `*p` is really just another name for `i`. It's sort of like `i` is "The White House" and `*p` is "1600 Pennsylvania Avenue".

Here's an another example sequence of statements using pointers:

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

i = 11
fmt.Println(*p)
fmt.Println(p)

*p = 300
fmt.Println(*p)
fmt.Println(p)
fmt.Println(i)

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

Test Yourself! Write down what each `Println` statement above will print out and then check your answers by running the above statements in Go.

Other pointers

You can have pointers to any variable:
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.

```
var name *string          // ptr to string
var person *Employee     // ptr to Employee
var pj *int              // ptr to int
var m map[string]*Employee // map from strings to pointers to Employees
var pA *[]float64        // a ptr to a list of real numbers
var Apf [[]float64       // a list of pointers to real numbers
```

Pointers to structs

Just as with our Employee example, it is often the case that you have a pointer to a variable that holds a struct:

```
Leci n'est pas une pipe.
```

René Magritte
The statement `(*person).name` means: follow the `person` pointer; you arrive at a `struct` of type `Employee`; access the field `name` of that `struct`. This is so common, Go provides a shortcut:

```go
person.name = "Jerry"
```

You can access the fields of a struct from a pointer to that struct just as if you had the struct directly.

### Another example:

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

func main() {
    // c is a Contact structure
    var c Contact = Contact{name:"Dave", id:33}

    // p points to c
    var p *Contact = &c

    fmt.Println(c)  // will print out contents of c
    fmt.Println(*p) // will *also* print out context of c
    (*p).name = "Holly"  // will change c.name to "Holly"
    p.id = 33          // will change c.id to 33
    fmt.Println(*p)
}
```

### Passing a struct to a function:

What's wrong with this code?
When we pass \( c \) into \( \text{setContactInfo} \) we are passing in a copy, so any changes we make to \( d \) inside of \( \text{setContactInfo} \) don't effect \( c \). How can we fix it?

Have the function take a pointer to a \( \text{Contact} \) and then pass the address of \( c \) into the function:

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

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

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

**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

Glossary

- **address**: the location in memory of a variable.
- **pointer**: a variable that holds the address of another variable.
- **dereference**: means "following the pointer" and is denoted $*p$ when $p$ is a pointer.
- **address of operator**: $&x$ returns the address of variable $x$. 