Examining Go

Go Part 2, what makes it Go?

Admin

- Anyone new?
- Assignment:
  - Make a github account if you don’t have one and send me your githubID via MSTeams or email (some official BSU channel)
  - After today read chapters 4-5, maybe more if we get further.
-
- **BIG Caveat:**
  - Since you are advanced students, I’m showing the highlights here, you need to dig in further on your own.

Speaking of Constants

- A useful constant form in go :
  - Create a series of names constants with incremented values
    - Sort of like enums in C-like languages
  - `const(`
    - `val1 = iota`
    - `val2`
    - `...`
    - `valN`
    - `)`
  - val1 is zero, then each valX after gets next value

Arrays in Go

- Arrays in Go are interesting
  - Standard fixed size, homogeneous, contiguous data structure
    - Must declare type and size at compile time
      - Eg:
        - `var octoOfInts [8]int;`
        - `var tripleOfStrings [3]string = [3]string{“s”, “t”, “u”}`
      -
      - If not given initial values, then zero value for type is assigned.
        - fmt.Printf(octoOfInts) → [0,0,0,0,0,0,0,0]

Arrays II

- If you specify elements of array at creation time, Go can infer the size of the array
  - Use ellipsis as size
  - `BunchOfTemps := [...]int{74, 80, 54, 96, 97, 96}`
  - Finally you can specify locations of values when creating arrays: use those constants to help us
    - `type EmployeeId int`
    - `const(`
      - `BEN EmployeeId = iota`
      - `ANN`
      - `JJ`
      - `JED`
      - `)`

Arrays III

- Now we can use these to declare arrays
- `//jj was on vacation for the week and didn’t get any`
- `//hours`
- `PayrollForWeek := [...]float32{BEN: 300.67, JED: 500.99, ANN: 765.43}`
- A few things to note:
  - Multiline statements possible, just end line where a statement can’t end (implicit semicolons)
  - `fmt.Printf(“%v”, PayrollForWeek)` → `[300.67 765.43 0 500.99]`
  - Notice that we created the array ‘out of order’ but it prints in order

Arrays IV

- Arrays are passed by value in Go
  - like all params (except when passing pointers)
  - Not like C++/Java
- Array size is part of the type in go
  - And Go is a strongly typed language
  - So what does this mean for parameters in functions?

Arrays IV

- Arrays are passed by value in Go
  - like all params (except when passing pointers)
  - Not like C++/Java
- Array size is part of the type in go
  - And Go is a strongly typed language
  - So what does this mean for parameters in functions?
    - You need a different function for every size of array if you take an array. These differ:
      - `func reverse(ptr *[8]int){…`
      - `func reverse(ptr *[16]int){...`

Slices

- Arrays are great, but limited, no growth, typing is difficult
- So Go says: ‘use slices’
  - In Go slices are a “view” into a sequence data
    - Usually arrays, but also strings
  - Every slice has an array under it, but slices grow and have variable size.
  - Every slice has:
    - pointer to an array element (first item in slice)
    - len (how many elements in slice
    - cap (how many elements till end of underlying array)

Slices II

- Create an empty slice:
  - `var emptySlice []int`
  - len is 0; emptySlice == nil
- Create a slice with lots of zero values using make

```
names := make([]string, 5, 10)
```

  - Makes a sequence of type <first param> with len <second param> and capacity <third param>
    - If cap isn’t specified, len and cap are same
  - Going past len in a slice expands the slice
  - Going past cap, causes *panic*

Slices III

- Since slices are just these three values
  - The data pointer points at the data in the array
  - Slices are passed by value
    - Like all parameters
  - But like java, you can’t change what the slice points at in the caller, but you can change the value of the slice for the caller.

Maps

- Maps in Go
  - Are hash tables. Fast access given key to find value O(n) space.
  - Work (almost) just like dictionaries in python
  - Keys must be comparable
    - a type you can use == with
    - Values can be any type
  - Trying to retrieve a key/value not in the map returns the zero value
    - Check ok if you really need to know (see next slide)

Making Maps

- We can make a map with make
  - `wages:= make(map[string]float32)`
- Or with a literal map
  - `wages := map[string]:float32{`
    - `“Ed”: 450.17,`
    - `“Ann”: 375.99,`
    - `}`
  - Check ok:
  - `earnings, ok := wages[“John”]`
    - Earnings will be 0.0, ok will be false.

A brief Digression

- A quick aside
  - Here is how you might read from a text file in go
  - `Import “io/ioutil”`

```
byteArray, err := ioutil.ReadFile("file.txt") //in goland project directory is working directory

str := string(byteArray) // convert file contents to a string
```

In class exercise

- As an in class exercise,
  - Lets grab the silly ‘recommendation’ from the resources page
  - Write a program which opens the text file reads it in and prints out every other line to the screen.
  - Simple, but gives us a chance to actually write some go.

Assignment

- At this point go over the *simple* first go project
- Some of you might have seen this already

Structs

- Structs in Go are aggregate data types
  - If you squint hard enough they look like c-structs
  - A collection of named typed fields
  - Eg:

  -
    ```
    type Player struct {
        name string
        health int
        jumpDistance int
    }
    ```
  - Creates a struct type with 3 fields,
    - `var player1 Player //creates a variable player1 of type Player with zero value for the fields`
  - Access fields in a C-like manner
    - `player1.jumpDistance = 3`

Structs II

- Two structs have the same type if:
  - They have the same number of fields
  - Of the same type
  - In the same order
- Can create struct with literal
  - `var player2 = Player{“Mario”, 1, 2}`
  - Or
  - `Var player3 = Player{name: “Luigi”, health:1}`
  - `//note jump distance not supplied so zero.`
- Looking at this code, what can you tell me about the packages for this code and the Player struct?

Structs II

- Can create struct with literal
    - `var player2 = Player{“Mario”, 1, 2}`
  - Or
    - `Var player3 = Player{name: “Luigi”, health:1}`
    - `//note jump distance not supplied so zero.`
- Looking at this code, what can you tell me about the packages for this code and the Player struct?
  - They have to be in the same package
    - The struct name is Capitalized and exported
    - But the field names are not – so can’t access fields from another package.
    - Of course don’t do this in ‘real life’

Structs III

- Structs can have another struct as a member
  - But no recursive definitions
  - Must use pointer for recursive.
- Embedded structs have a “lazy programmer” hack
- type saveGame struct{
  - p Player
  - Size int
  - }

Structs IV

- Suppose someone hands me a saveGame from another method
  - `MySave := <some function call here>`
- I want to find and display the name of the player from the save
  - Access with MySave.p.name
  - This could be a pain if there are lots of embedded structs – so see next slide
  -

Structs V

- Use struct with anonymous fields
  - Eg
    - `type SaveGame struct{`
    - `Player`
    - `Size int`
    - `}`
  - Now
    - `MySave :=` *<some function call here>*
    - `MySave.name`
  - Ahh “programmers are lazy” works as long as there are no fields with same name in anonymous fields

And now

- Now a deeper look at functions.
- Remember go functions
  - Func <name> (<parameter list) (return list){
    - Function body
  - }
- No default param values in go
- Parameters and return variables are local variables with widest scope in function

Return variables

- Return variables – lets look at this function

```
func factorial(n int) (answer int, err error){
    if n<0{
        answer = -1; err = errors.New("can use negative number for factorial"); return
    }
    answer = 1
    for ;n>0; n--{
        answer *= n
    }
    return
}
```

  - answer and err are return variables
    - Initialized to zero values when function starts
    - Need to give useful values before function returns
    - Functions that have return types must end in return

Anonymous Return

- Same function without return variables

```go
func factorial(n int) (int, error){
    if n<0{
        return -1, errors.New("can use negative number for factorial")
    }
    answer := 1
    for ;n>0; n--{
        answer *= n
    }
    return answer, nil
}
```

  - Now return values explicitly

Recursion

- Recursion works in go – as always make sure to have your base case first.
- Book is only place I’ve seen that walks through recursion with multivalue returns, have a look.
- Page 126-127

Errors

- As discussed
  - No exceptions in go
  - Errors as (traditionally/conventionally) the last return value in multi value return
  - Since you can’t have an unused variable, must handle the error
  - What about _ ?
    -

Errors

- As discussed
  - No exceptions in go
  - Errors as (traditionally/conventionally) the last return value in multi value return
  - Since you can't have an unused variable, must handle the error
  - What about _ ?
    - don't, just don't, it would be a terrible, horrible, no good very bad idea.
    - And most of the time will not compile anyway

Errors

- Error is an interface
  - More on that later
- Functions that always succeed: no need for errors
- Functions that throw exceptions in other languages
  - Likely need error return values
- Can create your own error types
  - And check the type of the error in the caller to see what sort of error it was
    - React appropriately (eg page 132)
  - Sorta like exceptions

Errors

- As we’ve seen,
  - Idiomatic in Go to handle errors before success (and then forget that the error occurred)
    - If the error is not recoverable
      - log.fatal will record in log file and then exit program
      - For lesser errors, warn and continue with reduced functionality (network down for example)

Functions are first class

- Functions are first class values in Go
  - Like python and rust (but not java and C/C++)
  - Can assign a function (not the result but the function itself) to a variable
  - Functions are not comparable (no ==)
    - So not as keys to map.
    - But can be compared to nil (zero value for function)

Functions in functions

- You can declare functions in other functions
  - Example from golang-book.com

  -
    ```
    func main() {
      add := func(x, y int) int {
        return x + y
      }
      fmt.Println(add(1,1))
    }
    ```
  - Parameter types and return type signature defines go function types

Function Types

- Given the assignment to add in previous slide, one of these will compile and one will error. Which is which?

```
add= func(x, y int64) int64{
        return x+y
}

add = func(first, second int) int{
        return first + second
}
```