CS 521: Systems Programming

Go

Lecture 17

Time to get go-ing

- Okay, but all accounts we are now all C geniuses.
  - Well, kinda?
- C isn't going away any time soon (you'll be using it in Systems Foundations)
  - Some of the limitations of C can be really annoying
  - BUT C gives us a great idea of how things work at a lower level
    - Hey, even the Python interpreter was written in C!
- However, needing to do things like implementing your own arraylist is kinda a drag on productivity…

Let's go

- The rest of the semester, we'll be using go (aka `golang`)
  - If you need help and want to search Google etc, use `golang` in your query
- Let's install it on our VMs:
  - `sudo pacman -Sy go`
- Unlike C, go is more *portable*: if I write a go program on my Mac, there's a pretty good chance it'll work fine on Linux
  - We will still use the VMs for testing

Greeting the World (again)

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world!")
}
```

- Neat, there's a `Println` so you don't always have to remember the `\n`!
- Packages?! What is this, Java?! 😉 (for now we'll use `main`)
- Hmm, no return value or arguments?
- NO SEMICOLONS?!

Compiling and Running

- `go build whatever.go` will produce an executable called `whatever`
  - No need for the `-o whatever` flag, or running executables named `a.out`!
- After compiling, you can just run `./whatever`
  - Even puts Java to shame!
- Or, you can do both in a single step:

  `go run whatever.go`
  - Note: an executable will not be produced in this case

What About Variables?

```
package main

// Use this format for multiple imports. Convention is to always do this,
// even if you only have a single import.
import (
    "fmt"
    "strconv"
)
func main() {
    var str = "Hello " + "world" + "!"     // Concatenating with +... Whoohoo!
    var magicNumber int = 42               // Notice how we gave this one a type
    f := 3.0                               // := is shorthand for 'var f = ...'
    var a, b, c = 1, 2, 3                  // Creating and assigning many ints

    fmt.Println(str + "  The magic number is: " + strconv.Itoa(magicNumber))
    fmt.Printf("We can still printf! f = %f\n", f)
    fmt.Printf("a = %d, b = %d, c = %d\n", a, b, c)
}
```

Comments

- `//` and `/* ... */`, just like C and Java. 'Nuff said.
- What about Doxygen / Javadoc style comments?
  - Write a comment above whatever it is you're documenting. That's it! (No special `/**` or similar to identify document comments)
    - Generate documentation with `godoc`
  - There are no special identifiers, e.g., `@param`.
    - The idea here is your variables should be named clearly enough that these are not necessary

Types

- As you've seen, go uses *type inference* to figure out what the type is for each variable
  - We can be explicit about the type if we want
- Basic types:
  - `bool`
  - `string`, `rune`
  - `int`, `uint`
  - `byte`
  - `float32`, `float64`

Conditionals

```
a = 35

if a > 64 {
    fmt.Println("Bigger than 64!")
} else if a > 32 {
    fmt.Println("Bigger than 32!")
} else {
    fmt.Println("Not bigger than 32 or 64!")
}
```

- Support for `if`, `else if`, and `else`
- No parentheses
- No **ternary if** in the language (`something ? true : false`)

Loops

There is only one type of loop: `for`

```
// The syntax we're used to, just without parentheses:
for i := 0; i < 100; i++ {
    fmt.Println(i)
}

// This is more like a 'while' loop:
i := 0
for i < 100 {
    fmt.Println(i)
    i = i + 1
}

// "Forever" infinite loop (like while(true) { ... })
for {
    fmt.Println("loop")
    break // 'continue' is also supported in for loops!
}
```

Functions

Getting `func` -y

```
func doSomething() { // Takes no params, doesn't return anything

}

func areWeWritingC() bool { // Returns a boolean
    return false
}

func addThree(numberOne int, numberTwo int, numberThree int) int {
    return numberOne + numberTwo + numberThree
}

// We can omit all but the last type if they are the same:
func addThree(numberOne, numberTwo, numberThree int) int {
    return numberOne + numberTwo + numberThree
}
```

Creating an Array

Arrays behave similarly to other languages. As usual, the declaration looks a bit different:

```
var nums [100]int
nums[0] = 24
nums[1] = 22
nums[99] = 1000
//nums[100] = 10 (will not compile -- out of bounds!)
```

- Indexes are 0-based and work as you'd expect.
- **Each element is set to their data type's default initial value (0 for `int`)**

Iterating: len()

We can iterate through an array similarly to how we would in C. Use `len()` to determine its length:

```
for i := 0; i < len(nums); i++ {
    fmt.Printf("%d %d\n", i, nums[i])
}
```

- On a related note: go only supports 'postfix increment'
  - `i++`
  - Cannot be used as an expression (i.e., `a := i++` is not allowed!)

Ranges

One common pattern in many languages is the `for each` loop. Go has something similar: `range`

```
var nums [100]int
nums[0] = 24
nums[1] = 22
nums[99] = 1000
//nums[100] = 10 (will not compile -- out of bounds!)

for i, value := range nums {
    fmt.Printf("%d %d\n", i, value)
}
```

- Note: `range` is a keyword. It does not take parameters.

Unused Variables

Say we want a `for each` loop but don't need the index:

```
for i, value := range nums {
    fmt.Printf("%d\n", value)
}
```

```
$ go run array.go
./array.go:5:9: i declared but not used
```

Use _ to throw away (ignore) a variable:

```
for _, value := range nums {
    fmt.Printf("%d\n", value)
}
```

Passing Arrays to Functions

Unlike in C, when arrays are passed to a function, a **copy** is made!

- Whoo, no more forgetting that `arr` is just a pointer to the first element in the array 😄

```
func printArray(arr [100]int) {
    for _, value := range arr {
        fmt.Printf("%d\n", value)
    }

    // This change is not visible outside the function:
    arr[6] = 24
}
```

Pointers

- But what if we **want** to be able to change an array (or any variable for that matter) from inside a function?
- Guess what, go supports **pointers**!

```
// Takes in a pointer to an array of 100 ints:
func printArrayPointer(arr *[100]int) {
    for _, value := range arr {
        fmt.Printf("%d\n", value)
    }
}

...

printArrayPointer(&nums)
// Any changes to nums made in printArrayPointer WILL be visible here
```

Array Sizing

- Thus far, you've seen us setting an explicit size for the arrays being passed to a function.
- Umm, is that required?
  - Yes. 🤦
- But don't worry. It's not a big deal…

Variable Array Size

The following code will compile:

```
func printArray(arr []int) {
    for _, value := range arr {
        fmt.Printf("%d\n", value)
    }
}
```

…but `arr` here is **NOT** an array! It's something else called a *slice*

- To convert to a slice, we'd pass an array like:

  `printArray(nums[:])`

Slices vs Arrays [1/2]

- In C, we use arrays quite often
- However, they are quite difficult to work with
  - Must be a fixed size…
  - …or we can `malloc` and `realloc` them
- In go, arrays are very similar but have a few of the "gotchas" removed
  - Out of bounds indexing, pointer to first element, etc…

Slices vs Arrays [2/2]

- Go's arrays are a lot like Java's
  - You use them, but not *that* often
  - `ArrayList` (or other implementations of `List<>`) are what we use more in Java
- In go, you'll see slices being used very frequently
- So what **is** a Slice?
  - A pointer to an array
  - A size
  - A capacity
- Wait, didn't we… *build* that already, several weeks ago in C?

Slicing and Dicing

The cool thing about slices is you can use them to create "views" of your arrays:

```
// Create a slice with the first 10 elements of 'nums':
chopChop := nums[1:10]
for _, value := range chopChop {
    fmt.Printf("%d\n", value)
}
```

Or we can create a new, empty slice:

```
slicey := make([]int, 0, 100)
// len(slicey) = 0
// cap(slicey) = 100
```

In Memory

- Remember, slices are “views” of arrays: when you *re-slice* a slice, you’re just changing where the pointer points in the array!
  - If you change the underlying array, the slice contents change too!
- A slice’s capacity is fixed since it is based on its backing array
  - **BUT** we can resize a slice easily!
  - `someSlice = append(someSlice, someNewThing)`

Resizing

- When we `append` to a slice, internally we are:
  1. Checking if we've exceeded the array capacity. If so, allocate a new slice with a backing array that's double the size
     - `make`
  2. Copy the elements over to the new slice
     - `copy`
  3. Return the new slice!
- Slices have a "slice header" that's basically a struct with this information included

Strings and Runes

- We mentioned that go has a `rune` type
- All go source files are UTF-8 and the language provides great support for Unicode
- Strings are still represented as arrays of bytes (just like in C)
  - But that is problematic if we have characters outside the usual ASCII range
  - Most of the time, we interpret strings arrays of `rune` instead (32-bit integers)

Runes

```
package main
import "fmt"

// Let's check out the difference between these two loops...
func main() {
    const str = "হ্যালো 🐿️ 🤓"

    fmt.Println("--- " + str + " ---")
    for i := 0; i < len(str); i++ {
        fmt.Printf("%02d %c\n", i, str[i])
    }

    fmt.Println()
    fmt.Println("--- " + str + " ---")
    for i, runeValue := range str {
        fmt.Printf("%02d %c\n", i, runeValue)
    }
}
```