Introduction to Go

Preparation for Lab 1

You can quickly run the code from this lecture by pasting the program into the Go Playground: play.golang.org

Why Go?

- Easy to use.
- Similar to C.
- Concurrent processing with channel communication built into the language.

Hello World

```
package main

import "fmt"

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

```
package main
```

Declare that this file belongs to the `main` package.

Only `main` packages can have executable `main()` functions.

```
import "fmt"
```

Import the package `fmt` (short for 'format') which implements formatted I/O with functions analogous to C and Java.

```
func main() {
    ...
}
```

Use the keyword `func` to declare a function.

Note that `main()` is a void function.

```
fmt.Println("Hello World")
```

Use the `Println` method from the `fmt` package to print the string `"Hello World"` followed by a newline character.

C-style `Printf` is also supported.

Running the program

```
$ go run hello.go

Hello World
```

Add

```
package main

import "fmt"

func add(a int, b int) int {
    sum := a + b
    return sum
}

func main() {
    firstNumber := 10
    secondNumber := 5
    firstNumber = add(firstNumber, secondNumber)
    fmt.Println(firstNumber) // = 15
}
```

```
firstNumber := 10
```

`:=` declares a **new** variable.

```
firstNumber = add()
```

`=` assigns a value to an **existing** variable

```
firstNumber := 10
```

is equivalent to

```
var firstNumber int = 10
```

However, you should always use the `:=` shortcut if the type can be inferred:

```
i := 42             // int
f := 3.142          // float64
g := 0.867 + 0.5i   // complex128
s := "bristol"      // string
```

String swap

```
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b) // = "world hello"
}
```

```
func swap(x string, y string) (string, string) {
    return y, x
}
```

Go allows multiple return values.

`swap()` takes two strings and **returns two strings**.

```
func swap(x string, y string) (string, string) {...
```

Both `x` and `y` are of type `string`.

Because of that the function signature could be written as:

```
func swap(x, y string) (string, string)
```

For loop

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
fmt.Println(sum) // = 45
```

Note the lack of parentheses around the 3 components of the for loop.

While loop

```
sum := 1
for sum < 1000 {
    sum += sum
}
fmt.Println(sum) // = 1024
```

While loops in Go work in the same way as in C or Java.

However, they are declared with the `for` keyword!

Infinite while loop

```
sum := 1
for {
    fmt.Println(sum)
    sum++
}
```

An infinite `while(true)` loop is declared with the `for` keyword and no boolean condition.

If

```
func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}
```

An if statement does not need parentheses.

It must use `{...}`.

A single line `if (cond) someFunc()` is not legal.

Vertex

```go
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    v := Vertex{1, 2}
    pointer := &v
    pointer.X = 1e9
    fmt.Println(v) // = {1000000000 2}
}
```

```
type Vertex struct {
    X int
    Y int
}
```

Declare a struct with two fields.

```
v := Vertex{1, 2}
```

Initialise a new variable of type `Vertex`.

```
pointer := &v
```

Get address of `v`.

`v` is of type `Vertex`.
`pointer` is of type `*Vertex`.

```
pointer.X = 1e9
```

Change the value of field `X`.

Note the use of `.` rather than `->` or `(*pointer).X`.

Arrays

```
var a [2]string
a[0] = "Hello"
a[1] = "World"
```

Declare a new array and assign some values.

```
a := [2]string{"Hello", "World"}
```

You can also use the shortcut notation.

Arrays are passed by VALUE

```
package main

import "fmt"

func goodbye(array [2]string) {
    array[0] = "Goodbye"
}

func main() {
    a := [2]string{"Hello", "World"}
    fmt.Println(a) // = ["Hello", "World"]

    goodbye(a)

    fmt.Println(a) // = still ["Hello", "World"]!
}
```

Calling `goodbye()` had no effect because the array was passed by value (i.e copied).

The bug could be fixed by returning the array.

```
func goodbye(array [2]string) [2]string {
    array[0] = "Goodbye"
    return array
}
```

Slices

In practice, 99% of the time, you need to use slices.

Slices are an abstraction of arrays. Arrays have fixed size, while slices are dynamically sized.

You can create a slice with `make` (~ `malloc`):

```
slice := make([]int, 5)
// slice = [0 0 0 0 0]
```

Or with the shortcut notation:

```
slice := []int{1, 2, 3, 4, 5}
```

Append

```
package main

func main() {
    slice := make([]int, 1)
    //slice  = [0]
    slice[0] = 5
    // slice = [5]
    slice = append(slice, 4)
    // slice = [5 4]
}
```

Append *may* need to allocate a new array if there is no space to fit in a new element.

All slices are just 192 bits*

A slice consists of:

- A pointer to the first element of the underlying array
- The length of the slice
- The capacity of the underlying array

* on a 64 bit system where the pointer and ints are 64 bits

```
s := []string{"Hello", "World"}
fmt.Println(s) // = ["Hello", "World"]

goodbye(s)

fmt.Println(s) // = ["Goodbye", "World"]
```

This previous example failed with arrays.

However, it works correctly with slices!

For-range loop

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Println("Value", v, "at position", i)
    }
}
```

gobyexample.com

tour.golang.org

Look at the above resources to learn more about Go's syntax and features.

Acknowledgements

Some of the presented code snippets have been adapted and modified from tour.golang.org under the BSD-3 license.

Some of the presented code snippets have been adapted and modified from gobyexample.com under the CC 3.0 license.