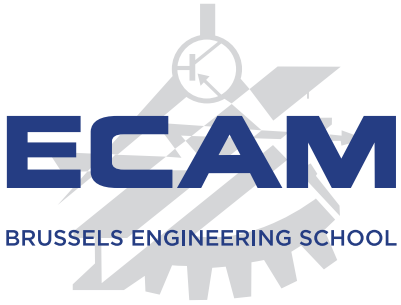

PL190μ Introduction to the
Go Programming Language

Session 1

Variable, Control Flow and Data Structure

Sébastien Combéfis

Winter 2020

Objectives

- Write, compile and run the first Go Hello World program

  Setup Go environment and discover basic command line tools

- Understand type, variable and execution flow control

  Write simple programs structured with functions

- Discover data structure built-in the Go programming language

  Declaring and using array, slice and map

Golang

Go Programming Language

- Go is a compiled C-like programming language

  Provides an easy low-level access to an operating system

- Open source programming language by a Google team

  Last stable 1.13 version from September 2019

- Provides excellent support for networking and concurrency

  Network, system, concurrent and distributed programming

Applications Written in Go

Hello World!

- Minimal Hello World! standalone program example in Go
  - The entry point if the `main` function
  - A standalone program must be in the `main` package
  - The imported `fmt` package used to print to stdout

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello World!")
7  }
```

```
Hello World!
```

Compile and Run

- Managing Go source codes with the go tool and commands
  - `build` to compile a package and its dependencies
  - `clean` to clean the workspace
  - `run` to compile and run a Go program

```
> go version
go version go1.13.6 darwin/amd64

> go run main.go
Hello World!

> go build -o program

> ./program
Hello World!
```

Format Code

- Go `fmt` command used to properly format source code

Format the code according to the Go formatting rules

```
> echo 'package main\nimport "fmt"\n          func main(   )
{fmt.Println(          "Hello World")}' > main.go

> cat main.go
package main
import "fmt"
          func main(   )     {fmt.Println(          "Hello World")
          }

> go fmt main.go
> cat main.go
package main

import "fmt"

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

Basic Type

- Go is a statically-typed programming language

  The type of all variables must be known at compile-time

- Four kinds of basic data types are available with Go
  - Integer number: `int`, `uint`, `int8`, `uint8`, ..., `int64`, `uint64`
  - Floating-point number: `float32`, `float64`
  - Boolean: `bool`
  - String: `string`

- Some additional data types are available with Go

  `byte` (alias `uint8`), `rune` (alias `int32`), `complex64` and `complex128`

Variable

- Variable declared by specifying its type and name

  Type can be inferred by the compiler for local variables

- Scope of a variable is limited to the enclosing block

  Possible to declare constants, whose value cannot be changed

```
1  package main
2
3  import "fmt"
4
5  const PI = 3.14                                  // Inferred type
6
7  func main() {
8     var radius float64 = 2                        // Explicit type
9     perimeter := 2 * PI * radius                  // Inferred type
10    fmt.Println("Perimeter: ", perimeter)
11 }
```

Shorthand Variable Declaration

- Variable initialised with a zero value by default

  Integer/floating-point number 0, *boolean* `false` *and string* `""`

- Several variables with the same type declared in one line

```
1  func main() {
2      var a, b string = "Hello", "World!"
3      fmt.Println(a, b)
4  }
```

- Variables with different types declared in a single block

```
1  func main() {
2      var (
3          a string = "Hello"
4          b int    = 42
5      )
6      fmt.Println(a, b)
7  }
```

Pointer

- Address of a variable obtained with reference operator `&`

  Reverse operation with the dereference operator `*`

- An address can be stored in a pointer variable

  Declared by prefixing the type with `*` like with C

```
1  func main() {
2      var a int = 42
3      var p *int = &a
4
5      fmt.Println(p, ":", *p)
6  }
```

```
0xc000092008 : 42
```

Function

- A function can be defined with a unique name

  And an optional list of parameters with types and names

- A function can be exited with the `return` statement

  Possibly with a return value with the right type

```
1  func sum(a int, b int) int {
2      return a + b
3  }
4
5  func main() {
6      fmt.Println(sum(17, 25))
7  }
```

Return Values

- Multiple values can be returned by a function

```
1  func sumdiff(a int, b int) (int, int) {
2      return a + b, a - b
3  }
```

- Possible to define named return value for a function

```
1  func sumdiff(a int, b int) (sum, diff int) {
2      sum = a + b
3      diff = a - b
4      return
5  }
```

Blank Identifier

- Multiple assignment to call multiple return values function

```
1  func main() {
2      a, b := sumdiff(17, 25)
3      fmt.Printf("Sum: %d\nDifference: %d\n", a, b)
4  }
```

- Some returned values can be ignored with the blank identifier

```
1  func main() {
2      a, _ := sumdiff(17, 25)
3      fmt.Printf("Sum:", a)
4  }
```

Control Flow

`if-else` Statement

- Decisions can be taken with the usual `if-else` statement

  Possible to chain several tests with `else if`

```
1  func main() {
2      var s string
3      temperature := 12
4
5      if temperature > 50 {
6          s = "Too hot!"
7      } else if temperature > 21 {
8          s = "Warm"
9      } else if temperature > 15 {
10         s = "Fine"
11     } else {
12         s = "Cold"
13     }
14     fmt.Println(s)
15 }
```

```
Cold
```

switch Statement

- More readable and performant else if with switch statement

  Possible to add an optional default case

```
1  func main() {
2      var s string
3      sex := "x"
4
5      switch strings.ToUpper(sex) {
6      case "M":
7          s = "male"
8      case "F":
9          s = "female"
10     default:
11         s = "undefined"
12     }
13     fmt.Println(s)
14 }
```

```
undefined
```

`for` Statement

- Repeating code can be done with the `for` statement

  Given a condition or with an `init` and `post` statements

```
1  func main() {
2      line := 1
3      for line <= 3 {
4          for i := 0; i < line; i++ {
5              fmt.Print("*")
6          }
7          fmt.Println()
8          line++
9      }
10 }
```

```
*
**
***
```

`defer` Statement

- Possible to `defer` the execution of code until the return

  The deferred code should be a function call

- Typically used to properly close or free allocated resources

  Similar to the `finally` or `ensure` from other languages

```
1  func main() {
2      defer fmt.Println("Bye bye!")
3      fmt.Println("Hello World!")
4  }
```

```
Hello World!
Bye bye!
```


Data Structure

Array

- Array of values initialised with a type and fixed size
  - Values directly defined or one by one with the access operator
  - Size of the array obtained with the `len` built-in function

```
1  func main() {
2      data := [5]int{1, 2, 3, 4, 5}
3      var squared [5]int                    // Size must be constant
4
5      for i := 0; i < len(data); i++ {
6          squared[i] = data[i] * data[i]
7      }
8
9      fmt.Println(data)
10     fmt.Println(squared)
11 }
```

```
[1 2 3 4 5]
[1 4 9 16 25]
```

range Statement

- The range form of for loop to iterate arrays

  The index and the value are returned for each iteration

```
1  func main() {
2      data := [5]int{1, 2, 3, 4, 5}
3      var squared [5]int                    // Size must be constant
4
5      for i, v := range data {
6          squared[i] = v * v
7      }
8
9      fmt.Println(data)
10     fmt.Println(squared)
11 }
```

```
[1 2 3 4 5]
[1 4 9 16 25]
```

Slice

- A slice is a dynamically-sized view into elements of an array

  Slices are wrapping arrays and manage size change

- Obtained from an array with low and high bounds

  High bound is excluded and the bounds can be omitted

```
1  func main() {
2      data := [5]int{1, 2, 3, 4, 5}
3
4      fmt.Println(data[2:4])
5      fmt.Println(data[4:])
6      fmt.Println(data[:])
7  }
```

```
[3 4]
[5]
[1 2 3 4 5]
```

Manipulating Slice

- Creating a slice with the `make` built-in function

  Possible to specify an initial size with default value elements

- The `append` built-in function is used to add elements to a slice

  It can also be used to delete elements from a slice

```go
func main() {
    data := make([]int, 3)

    data = append(data, 1, 2, 3, 4)
    fmt.Println(data)

    data = append(data[:1], data[3:]...)
    fmt.Println(data)
}
```

```
[0 0 0 1 2 3 4]
[0 1 2 3 4]
```

Copying Slice (1)

- A slice extracted from an array is referencing the array

  Changing the values in the slice changes them in the array

```
1  func main() {
2      data := [5]int{1, 2, 3, 4, 5}
3
4      slice := data[2:4]
5      slice[0] = 0
6      slice[1] = 0
7
8      fmt.Println(data)
9  }
```

```
[1 2 0 0 5]
```

Copying Slice (2)

- Copying a slice or part of a slice with the `copy` built-in function

Utility function to copy all the values into a fresh new slice

```
1  func main() {
2      data := [5]int{1, 2, 3, 4, 5}
3      slice := make([]int, 2)
4      copy(slice, data[2:4])
5
6      slice[0] = 0
7      slice[1] = 0
8      fmt.Println(data)
9  }
```

```
[1 2 3 4 5]
```

Array vs. Slice

- Array has a size fixed at creation and slice is dynamically-sized

```
1  func main() {
2      var a [3]int                    // Fixed-sized array
3      var b []int                     // Dynamically-sized slice
4
5      fmt.Println(a, b)
6  }
```

```
[0 0 0] []
```

- Default value for a slice is the special `nil` value

```
1  func main() {
2      var a []int
3      fmt.Println(a == nil)
4  }
```

```
true
```

Map

- A map is a set of pairs associating keys to values

  Sometimes referred to as an associative array

- A map is created with the `make` built-in function

  Need to specify the type of the keys and of the values

```
1  func main() {
2      inhabitants := make(map[string]int)
3
4      inhabitants["United States"] = 331002651
5      inhabitants["Belgium"] = 11589623
6
7      fmt.Println(inhabitants)
8  }
```

```
map[Belgium:11589623 United States:331002651]
```

Manipulating Map

- A map entry can be deleted with the `delete` built-in function

  Just specifying the key to remove the corresponding entry

```
1  func main() {
2      inhabitants := make(map[string]int)
3
4      inhabitants["United States"] = 331002651
5      inhabitants["Belgium"] = 11589623
6
7      delete(inhabitants, "United States")
8      fmt.Println(inhabitants)
9  }
```

```
map[Belgium:11589623]
```

References

- George Ornbo (2017). *Sams Teach Yourself Go in 24 Hours: Next Generation Systems Programming with Golang*, Pearson, ISBN: 978-0-672-33803-8.
- Caleb Doxsey (2012). *An Introduction to Programming in Go*, , ISBN: 978-1-478-35582-3.
- Himanshu Singh (2018). *Let's Go ~ A Complete Guide*, August 9, 2018.
  https://medium.com/mindorks/lets-go-a-complete-guide-147aec23fd5a
- Jerry J. Muzsik (2017). *A Space Themed Intro to Golang*, December 27, 2017.
  https://hackernoon.com/golang-the-highest-paying-technology-to-know-9c6089d7081d
- Kevin Goslar (2019). *Go is on a trajectory to become the next enterprise programming language*, April 10, 2019.
  https://hackernoon.com/go-is-on-a-trajectory-to-become-the-next-enterprise-programming-language-3b75d70544e

Credits

- Logos pictures from Wikipedia.
- Sam Thorogood, July 22, 2011, https://www.flickr.com/photos/samthor/5994939587.
- Grant Tarrant, June 3, 2007, https://www.flickr.com/photos/r_a_z_e/945477645.
- Maciej, December 15, 2012, https://www.flickr.com/photos/phaselockedloop/8522988876.