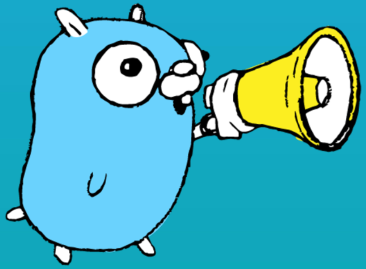
Fundamentals of Programming – Term 1/2020

Program Structure

Asst.Prof.Orathai Sangpetch, Dr.Akkarit Sangpetch

KMITL / CMKL University

Computer Innovation Engineering



Today's (glorious) blather.

Names & Declarations	01
Variables	02
Assignments	03
Type Declarations	04
Packages and Files	05
Scope	06

Building Blocks of Computer Program

- What is a **computer program**?

  A set of instructions that tells the computer how to manipulate data / information

- **Statements**: Tell the computer to do something
- **Data Types**: Data is divided into different types
- **Variables**: Allow you to store data and access stored data
- **Operators**: Allow you to manipulate data
- **Conditional Statements**: Execute if a condition is satisfied
- **Functions**: Mini self-contained programs

Names & Declarations

Names used to refer to functions, variables, constants, types, labels and packages

```
25 Reserved keywords (cannot use for declarations)

break     default     func    interface select
case      defer       go      map       struct
chan      else        goto    package   switch
const     fallthrough if      range     type
continue  for         import  return    var

Other predeclared names (can be redeclared)
Constants: true    false    iota    nil
Types:     int int8 int16 int32 int64
           uint uint8 uint16 uint32 uint64 uintptr
           float32 float64 complex128 complex64
           bool byte rune string error
Functions: make len cap new append copy close delete
           complex real imag
           panic recover
```

Names begin with a letter or _
Case matters (*hello* is different from *Hello*)
Go uses *CamelCase* or *camelCase*

Go Declarations

```
package main

import “fmt”

const boilingF = 212.0

func main() {
  var f = boilingF
  var c = (f - 32) * 5 / 9
  fmt.Printf(“boiling point = %g F or %g C\n, f, c)
}
```

A declaration names a program entity and specifies some or all of its properties.

Four kinds of declarations in Go

var	variables
const	constants
type	data types
func	functions

Function for encapsulation & reuse

Uses function fToC to encapsulate Fahrenheit to Celsius conversion logic

```
package main
import “fmt”

func main() {
  const freezingF, boilingF = 32.0, 212.0
  fmt.Printf(“%g F = %g C”, freezingF,
                            fToC(freezingF))
  fmt.Printf(“%g F = %g C”, boilingF,
                            fToC(boilingF)
}

func fToC(f float64) float64 {
  return (f – 32) * 5 / 9
}
```

Variables

Var declaration creates a variable of a type, attaches a name to it, and sets its initial value

```
var name type = expression

var s string  // initialized to “”
var b,f,s = true, 2.3, “four” // bool, float64, string

// Short variable declarations infer types from expressions
freq := rand.Float64() * 3.0
t := 0.0
i, j := 0, 1
```

A ***variable*** is a piece of storage containing a value.

If a type is not specified, Go will infer the variable type from the expression.

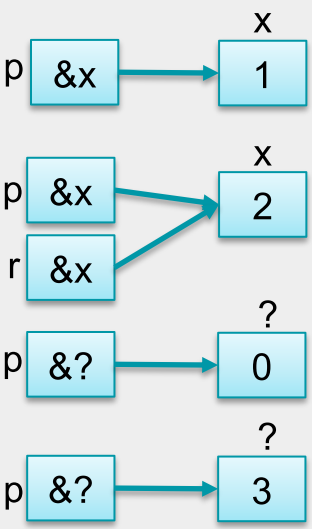
If an expression is not specified, the variable will be initialized to a zero value (*0, false, “”, nil*)

The zero value of an aggregate type (array / struct) has zero value of all its elements or fields

Pointers

Pointers allow indirect read/write of a variable without using or knowing the name of the variable.

```go
x := 1
p := &x // p as *int
fmt.Println(*p) // 1
*p = 2
fmt.Println(x)  // 2
r := p
p = new(int)
fmt.Println(*p) // 0
*p = 3
fmt.Println(*p) // 3
fmt.Println(*r) // 2 (value of x)
```



A ***pointer*** value is the address of a variable.

The zero value for a pointer of any type is ***nil***

Two pointers are equal if and only if they point to the same variable or both are nil.

new(T) can be used to create an unnamed variable of type T and returns its pointer (*T)

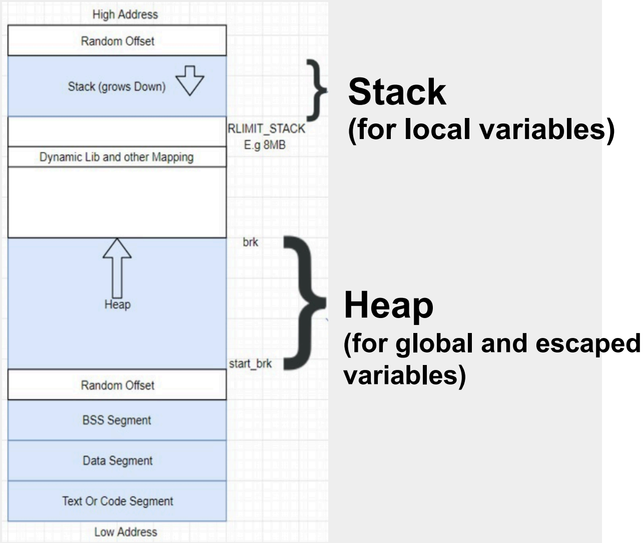
Lifetime

Lifetime of a variable is the interval of time during which it exists as the program executes.

```go
var global *int

func f() {
  var x int = 1
  global = &x
}

func g() {
  y := new(int)
  *y = 1
}
```



A package-level variable exists for the entire execution of the program.

A local variable lives on until it becomes *unreachable*, at which point its storage may be recycled by the garbage collector.

Escaped variables: A variable which remains reachable after its declared function should be heap-allocated.

A variable which becomes unreachable after its function return could be stack-allocated.

Assignments

Variables can be updated by an assignment statement (lhs = expression)

```
x = 1               // named variable
*p = true           // indirect variable
person.name = “bob”      // struct field

// array, slice or map element
count[x] = count[x] * scale

// assignment operator
count[x] *= scale
```

Tuple Assignment

Assigning several variables with a single statement

```
x, y = y, x
a[i], a[j] = a[j], a[i]

i, j, k = 2, 3, 5

v, ok = m[key]   // map lookup
v, ok = x.(T)    // type assertion
v, ok = <-ch     // channel receive

_, err = io.Copy(dst, src) // discard byte count
_, ok = x.(T)    // check type but discard result
```

All right-hand side expressions are evaluated before variable updates.

Unwanted values can be assigned to the blank identifier

_

Assignability

```
// slice literal
medals := []string{“gold”, “silver”, “bronze”}

// equivalent to
medals := make([]string, 3)
medals[0] = “gold”
medals[1] = “silver”
medals[2] = “bronze”
```

Assignment can be done implicitly (function calls, literal expression of composite types)

Assignment is always legal if lhs and rhs have the same type.

Constants (untyped) have flexible assignment rules.

Quick exercise

- **Fibonacci & GCD**

```go
func fib(n int) int {
  x, y := 0, 1
  for i := 0; i < n; i++ {
    x, y = ___, ___
  }
  return x
}

func gcd(x, y int) int {
  for y != 0 {
    x, y = ___ , ___
  }
  return x
}
```

Fill in the blank to (iteratively) calculate the n^{th} Fibonacci number & greatest common divisor

Type Declarations

The type of a variable define the characteristics of the values it may take on

```
type name underlying-type

type Celsius float64
type Fahrenheit float64

const (
  AbsoluteZeroC Celsius = -273.15
  FreezingC     Celsius = 0
  BoilingC      Celsius = 100
)

func CToF(c Celsius) Fahrenheit {
  return Fahrenheit(c*9/5+32)
}

func FToC(f Fahrenheit) Celsius {
  return Celsius((f-32)*5/9)
}

// methods = named type's associated functions
func (c Celsius) String() string {
  return fmt.Sprintf("%g C", c)
}
```

A type declaration defines a new ***named type*** that has the same underlying type as an existing type.

Named type provides a way to separate different / incompatible uses of the underlying type

Conversion operation ***T(x)*** converts the value x to type T (if both have the same underlying type)

Quick Checks

```
fmt.Printf("%g\n", BoilingC-FreezingC)     // 100
boilingF := CToF(BoilingC)                 // 212
fmt.Printf("%g\n", boilingF-CToF(FreezingC) // 180
Fmt.Printf("%g\n", boilingF-FreezingC) // compile error

var c Celsius
var f Fahrenheit
fmt.Println(c == 0)  // true
fmt.Println(f >= 0)  // true
fmt.Println(c == f)  // compile error: type mismatch
Fmt.Println(c == Celsius(f)) // true (both are zeros)
```

Packages and Files

Packages support modularity, encapsulation, separate compilation & reuses

gopl.io/ch2/tempconv.go

```go
package tempconv

type Celsius float64
type Fahrenheit float64

func CToF(c Celsius) Fahrenheit {
  return Fahrenheit(c*9/5+32)
}

func FToC(f Fahrenheit) Celsius {
  return Celsius((f-32)*5/9)
}

func (c Celsius) String() string {
  __________

}
func (f Fahrenheit) String() string {
  __________
}
```

gopl.io/ch2/cf.go

```go
package main

import (
  "fmt"
  "os"
  "strconv"
  "gopl.io/ch2/tempconv"
)

func main() {
  arg := os.Args[1]
  t, err := strconv.ParseFloat(arg, 64)
  if err != nil {
    os.Exit(1)
  }
  f := tempconv.Fahrenheit(t)
  c := tempconv.Celsius(t)

  fmt.Printf("%s = %s, %s = %s\n",
    f, tempconv.FToC(f),
    c, tempconv.CToF(c))
}
```

Each package serves as a separate namespace for its declarations.

Exported identifiers start with an upper-case letter

By default, each package's short name matches the last segment of its import path

Package Initialization

```go
package popcount

// pc[i] is the population count of i.
var pc [256]byte

func init() {
  for i := range pc {
    pc[i] = pc[i/2] + byte(i&1)
  }
}

// PopCount returns the population count (number of set bits) of x.
func PopCount(x uint64) int {
  return int(pc[byte(x>>(0*8))] + pc[byte(x>>(1*8))] +
          pc[byte(x>>(2*8))] + pc[byte(x>>(3*8))] +
          pc[byte(x>>(4*8))] + pc[byte(x>>(5*8))] +
          pc[byte(x>>(6*8))] + pc[byte(x>>(7*8))])
}
```

Package-level variables are initialized in the order in which they are declared.

Complex data can be initialized using ***init()*** function which will be executed when the program starts, in the order in which they are declared.

Can be used to precompute values.

Scope

Scope is a region of the program where a use of the declared name refers to the declaration

```
func f() {}
var g = "g"
func main() {
  f := "f"
  fmt.Println(f) // local f shadows package-level func f
  fmt.Println(g) // package-level var
  fmt.Println(h) // undefined
}


// for loop creates two lexical blocks; one explicit block
// for its body & another implicit block encloses variables
// declared by the initialization clause
// How many variables are named x in the following code?
func main() {
  x := "hello"
  for _, x := range x {
    x := x + 'A' - 'a'
    fmt.Printf("%c", x)  // "HELLO"
  }
}
```

A syntactic *block* is a sequence of statements enclosed in braces.

A *lexical block* includes grouping of statements in braces and other declarations. A lexical block for the entire source code is called the *universe block*.

A declaration's lexical block determines its scope
- *Package-level* outside functions
- *File-level* imported packages

Inner declaration can *shadow* or *hide* the outer one.

Nested Scope & Implicit Blocks

for, if, switch statements create implicit blocks for initialization clauses

```
if x := f(); x == 0 {
   fmt.Println(x)
} else if y := g(x); x == y {
   fmt.Println(x, y)
} else {
   fmt.Println(x, y)
}
fmt.Println(x, y)      // error: x and y is not visible here
```

Variables declared in first statement's initializer are also visible within the second block.

Nested Scope & Implicit Blocks

Watch out for scope of implicit blocks

```
if f,err := os.Open(fname); err != nil {
  return err           // compile error: unused f
}
f.ReadByte()           // compile error: undefined f
f.Close()              // compile error: undefined f


--------------------------

f, err := os.Open(fname)
if err != nil {
  return err
}
f.ReadByte()
f.Close()
```

The scope of implicit declaration (f, err) is within the enclosed statement (if)