

28/10/18

## Programmazione in "Go"

- ATOM (compilatore da installare) → hello.go (file sorgente che verrà compilato)
  - compilare → go build (comando) + (file sorg.) hello.go  
[go build hello.go]
  - esecuzione → ① Linux ./hello  
② windows hello.exe

P.S. Compila + esegui ms go run (comando)

- sito dove scaricare go: "golang.org"
- sito senza scaricare
- nulle e prove: "play.golang.org" (non si può automatizzare nulla)

### 1° Programma in Go

- ne hello.go (apro editor per scrivere)
  - package main
  - import "fmt" (utilizziamo funzionalità già comprese nel linguaggio: scrivere e leggere)
  - func main() {  
    fmt.Println("Hello, world!")  
}
- go build hello.go → compilo il programma (go run hello.go → eseguo il programma)
- go fmt hello.go (formattatore = ti sistema in maniera ordinata)

• Variabili <sup>"m"</sup>: locazioni di memoria sono catene, liste, stringhe.

↳ var i, n int (variabili: "i" e "n" intere)

• fmt. Scan (& n) → scrivo un numero (ex: 2) e questo mi permette di mettere quel numero ~~in~~ nella variabile "n".  
"2" → in "n"

```

package main
import "fmt"

func main() {
    var i, n int
    fmt.Println("Quante volte?")
    fmt.Scan(&n)
    for i, i < n, i++ {
        fmt.Println("ciao")
    }
}

```

7

- go doc `fmt.Scan` (NOME FUNZIONE) = descrizione di una funzione nel compilatore AFLX.  
 ↳ come un dizionario.

10/10/18

- TOKEN: sequenza di caratteri
- CLASSI:
  - parole chiave (token x costituti sintattici)
  - identificatore (possono essere definiti dall'utente)
    - Prima: Underscore - , oppure carattere alfabetico
    - Dopo: -, lettere, numeri
  - segni di interfusione o operatori
    - +, -, <, >, =, (), {}, "", "
  - costanti (inserite da noi nel programma)
- Numeri
  - FLOATING POINT (3.41723) → float32 o float64 (64bit)
  - NOTAZIONE SCIENTIFICA (3.417 · 10<sup>9</sup>)
  - INTERI ( $\pm 34$ ) → numero intero a 32 o 64 bit
- Testo
  - RUNE (carattere ISO) → alias di int a 32bit
  - STRING ("sequenza di caratteri") → immutabile
    - ↳ 1 SINGOLO TOKEN
- Cost. Booleane
  - BOOLEANS (true / false)

\* 4 Tipi di SPAZIO : ① Spazio - ② a capo - ③ tabulazione - ④ orelli a capo

\* IMPORTANTE!
 

- Si distinguono lettere maiuscole da minuscole
- Se la funzione si deve rendere accedibile per esterni, deve iniziare con lettera Maiuscola (ex: Println)

- COMMENTI: // commento → NON RILEVANTE per il compilatore - /\* Commento continua → \*/ Commento multilinea
- PACCHETTI: gruppi di file organizzati su disco in directory separate.

11/10/18

## Costrutto di FATTORIZZAZIONE

```
import f.  
f.fmt()  
f.main()  
:  
)
```

① .fmt → non ho più bisogno di scrivere sempre fnt. Print (ex.)  
 ② - main → "silenzia il pacchetto"

- VARIABILI: si possono dare nomi convenzionali

var {  
 • tipo  
 • Nome  
 • identificatore  
 • valore  
 • Lexical Scoping

| int  
| float64  
| String |

ex: var i int

→ Scope:  
 - definisco variabili nel blocco e posso usarle in tutto il blocco

- non posso dichiarare variabili uguali
- dichiaro variabili ovunque nel programma
- etc.

$v := 1$ $\{$ $v := 2$ $fnt. println(v)$ $\}$ $fnt. println(v)$	$\rightarrow$ OUTPUT : 2 1
--	----------------------------------

"var" viene sostituito da ":"

• Più precisamente: questo effetto di "nascondere" la variabile del blocco maggiore viene definita regola di visibilità SHADOWING

• Utilizzando ":" il compilatore indovina il tipo di variabile: posso avere definito come var x int ; x := 3  
 $x = 3$

15/10/18

- Funzioni Output:  
 • Print()  
 • Println()

Si possono mettere più espressioni separate da virgola.

- Println():  
 • va a capo  
 • mette sempre spazio tra membri separati da virgola

- Print():  
 • non va a capo  
 • non mette spazi tra membri separati da virgola

per mettere spazio: Print(3, " ", 4)

per andare a capo: Print("3 è minore di 4\n") → \n è carattere di MANDATA A CAPO

• Funzione Scan: legge/scannerizza valore inserito dall'utente.

- bisogna avere prima delle variabili
- non legge finché l'utente non preme

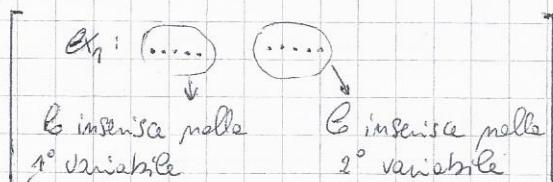
[Ex<sub>1</sub>: • var x int

- Scan (& x) → legge la variabile x inserita dall'utente.
- Println (x)

Ex<sub>2</sub>: • var anno int

- Print ("Inserisci il tuo anno di nascita: ")
- Scan (& anno)
- Println ("Hai ", anno, " anni")

→ divide in token saltando lo spazio:



P.S. ATTENZIONE!: differenza tra ① Scan (& h) e ② Scan (& h, & n)

① Più utile perché legge tutte quelle inserite dall'utente nella riga, quindi solo i token che ci sono: 3 VALORI → 3 Scan → inserisce 3 token? → Nel totale 2!

② Vengono fatte letture di file e devono essere inserite tutte per far funz. programma

• Funzioni MATEMATICHE:

$$\begin{array}{c} + \\ - \\ * \end{array} \left. \begin{array}{c} \\ \\ \end{array} \right\} \text{NESSUN PROBLEMA}$$

- / → divisione euclidea. (divisione tra intei!)
- dividendo e divisore negativi:  
resto più breve negativo.
  - divisione tra int e int
- ex:  $\frac{20}{100} = 0,2 \rightarrow \text{legge solo } 0!$

[Ex<sub>1</sub>: trovare solo parte frazionaria con go: (x inizializzata float64)]

- con int(x) trovo solo parte intera di x.
- Per fare operazioni le variabili devono essere dello STESO TIPO.
- x - float64 (int(x))

$\bar{X}_2$ : Annotare per difetto/ucciso:

- $\text{int}(x+0,5)$ 
    - Così da avere numero amotondato sempre verso zero, causa  $\text{int}(x)$ .
    - $0,5$  è una COSTANTE

Costanti in Go: Sono senza tipo → Ciò logge come tante per conto nostro, se mettiamo un valore float64, Go lo legge come tale

- +** aumentare      **• -** diminuire } si può fare solo con ~~restauaration~~ cancellare intere a causa  
del **SYNTACTIC SUGAR** (zucchero sintattico)

$\left[ \begin{array}{l} G_1: \quad X++ \rightarrow X+1 \\ \quad X-- \rightarrow X-1 \end{array} \right] \quad \text{In Go dopo aver fatto } ++ \circ -- \text{ non si può più rivedere} \\ \text{la } x.$

- Syntactic sugar = espressione con stesso valore di un'altra ( $\text{ex}_1^*$ )
  - Uguaglianze lessicali =
    - $+ = \rightarrow x := x + 3 \Rightarrow x += 3$  incremento di 3
    - $- = \rightarrow x := x - 3 \Rightarrow x -= 3$  decremento di 3
    - $* = \rightarrow x := x * 3 \Rightarrow x *= 3$
    - $/ = \rightarrow x := x / 2 \Rightarrow x /= 2$
    - $\% = \rightarrow x := x \% 2 \Rightarrow x \%= 2$

4 CASI DI CONTROLLO DEL FWSS:

17/10/18

- ① Sequenze
  - ② Costrutti di selezione : Gli anche dipende da quali di selezione (Non vediamo)
  - ③ Iterazione : [ciclo for]
  - ④

SELEZIONE BINARIA : if - if/else

```
func main () {
```

```

ex.1: func main () {
    var x int
    Scan (&x)
    if x == 0 {
        fmt.Println ("x e voglio a 0!")
    }
}

```

if/else

if COND {

} ELSE { ← importante posizione delle parentesi!  
..... → importante spaziatura degli argomenti.

if/else

A CASCATA

if COND.0 {

} else if COND.1 {

} else if COND.2 {

} else {

}

## Valori di Verità

Bool : TRUE / FALSE (True e false sono costanti)

(Bool = tipo di variabile.)

- Dopo l'if ci va qualcosa che restituisce un valore booleano.

= OPERATORI RELAZIONALI : servono per restituire 1.

•  $=$

•  $!=$

•  $<$

•  $>$

•  $\leq$

•  $\geq$

## Algebra booleana

• CONJUNZIONE LOGICA : quando due condizioni sono vere. COND 1 + COND 2 e - log. logica

• AND (in Go  $\Rightarrow$  "&&")

	T	F
T	T	F
F	F	F

• DISCIUNZIONE LOGICA : quando almeno una delle due è vera.

• OR (in Go  $\Rightarrow$  "||")

	T	F
T	T	T
F	F	T

• NEGAZIONE LOGICA : quando si invverte il valore

• NOT (in Go  $\Rightarrow$  " ! ")

	T	F
F	F	T

- Leggi di DeMorgan in Go =

$$\begin{aligned} \cdot ! (a \& b) &= !a \parallel !b && \left. \begin{array}{l} \text{scambi } \wedge \text{ con } \parallel \\ \text{con } \neg \end{array} \right. \\ \cdot ! (a \parallel b) &= !a \& !b \end{aligned}$$

### • VAWITAZIONE CONCERNITA.

Le espressioni vengono calcolate da sinistra a destra e si vede se la prima è vera oppure continua così finché deve. Non appena c'è un falsehood si stabilisce un risultato finale, l'operazione termina.

P.S. → Si può mettere istruzione parallela per VAWITAZIONE dell'IF.

$$\text{if } \Delta := b^*b - 4ac ; \Delta > 0 \{$$

!

22/10/18

### CICLO FOR

① ESEMPIO DI FOR IN GO: finché espressione è vera, il ciclo continua a funzionare e termina quando espressione diventa falsa.

- Primo esempio di divisione: Algoritmo di Euclides

```
[var x, y int
funt. Scan (&x, &y)
for x % y != 0 {
    x, y = y, x % y
}
```

← se inserisco un fuit. Println(x, y) nel ciclo, mi stampa ogni passo del ciclo.

- Radice quadrata

```
[var x, r int
funt. Scan (&x)
r = 1
for r * r < x {
    r++
}
```

mi calcola la radice quadrata. Se quadrato non perfetto arriva al numero dopo.

- Si può anche non mettere prime  $r=1$ , es: [for  $r=1$ ,  $r * r \dots$ ]

- ① FOR GN CONDIZIONE - SENZA INIZIALIZZAZIONE
- ② FOR GN CONDIZIONE + INIZIALIZZAZIONE
- ③ FOR GN CONDIZIONE + INIZIALIZZAZIONE + ISTRUZIONE DI INCREMENTO/DECRESIMENTO

→ Inizializzazione e iterazione può avvenire ponitattico: posso inizializzare variabili prima del ciclo for e iterare all'interno del ciclo.  
Meglio fare tutto nell'intestazione del ciclo.

[var n int  
funct. Scan (&n)]

for i := 1; i < n; i \*= 2 {  
    Println(i)  
}

• Stampa tutti i numeri che sono potenze di 2 fino a un numero n.

- Se abbiano ③ opzione + lsm. decremento  $\Rightarrow$  il ciclo avverrà n+1 volte!

### - Ciclo con Selezione (al suo interno)

[var n int  
funct. Scan (&n)  
primo := true]

for d := 2; d < n; d++ {  
    if n % d == 0 {  
        primo = false  
    }  
}

• Vedere se un numero è primo o no.

• nel caso: for d := 2; d < n && primo; d++ {  
il ciclo finisce appena il numero n non è più primo  
(primo è inizializzato vero!)

[if primo {  
    Println(n, "è primo")  
} else {  
    Println(n, "non è primo")  
}]

• Stampare se n è primo o no

- := crea e inizializza una variabile (cioè variabile di tipo bool, inizializzata a true)
- = assegnamento  $\rightarrow$  assegno a una variabile già creata! un valore.

[var n int  
funct. Scan (&n)  
var c int]

for c = 0; n > 0; n /= 10 {  
    c++  
}

Println(c, "cifre")

• Stampa numero di cifre di un numero inserito dall'utente.

- Dato un numero definito da noi = 100; l'utente inserisce un numero e noi stampiamo se numero utente è + grande, + piccolo o uguale.

```

x := 100
n := 0

for m != x {
    Print ("Indovina...")
    Scan (&n)
    if m < x {
        Println ("Troppo Piccolo")
    } else if m > x {
        Println ("Troppo Grande")
    } else {
        Println ("Indovinato!")
    }
}
  
```

- L'ultimo Println ("Indovinato!") sarebbe meglio inserito fuori da if, ma dentro for per ragioni stilistiche.

Facendo ciò non c'è else if, ma solo else.

- Tabella pitagorica fino a 10

```

for i := 1; i <= 10; i++ {
    for j := 1; j <= 10; j++ {
        Print (i * j, " ")
    }
    Println ()
}
  
```

Se metto i := 0; i <= 10 e j := 0; j <= 10;  
allora → Print ('i+1 \* j+1')

24/10/18

- Istruzione BREAK: si può trovare all'interno di un ciclo o all'interno di cicli annidati. Ma il campo di interrompe il ciclo più interno a cui si trova.
- In Go se la condizione del ciclo è FALSE, il ciclo non avviene (mentre, con do-while almeno una volta il ciclo avviene)
- Inserire Scan all'interno del ciclo fa in che ogni volta che repete il ciclo, mi riscriverebbe la variabile.
- SPAGHETTI CODE mm cercare significato.

```

s := 0
for {
    var x int
    Scan (&x)
    if x < 0 {
        break
    }
    s += x
}
  
```

- Il programma non si interrompe mai finché non inserisce un numero < 0.
- Somma di tutti i numeri scannerizzati ≥ 0.
- Il BREAK rompe il for non l'if !!!

- Se usciamo naturalmente dal ciclo (senza causa break), la condizione di controllo è falsa alla fine
  - Se usciamo cause BREAK, la condizione di controllo è ancora vera! ( $d < n \in$  Condizione di controllo)
- ↳ Per capire ieri dei due casi, bisogna dichiarare la variabile al di fuori del ciclo (prima) e non nell'intestazione del ciclo.

- CONTINUE:
  - interrompe solo il corpo del ciclo più interno.
  - Si usa davvero raramente...

```

S := 0
for S
    var x: int
    Scan(&x)
    if x < 0
        break
    if x % 2 == 15
        continue
    S += x
    }
  
```

- Continue ti manda alle fine delle parentesi più vicine
- In questo caso è come bleepo pag. prima; senza contare i numeri dispari (impegnoso, meglio usare if-else).
- Interrompe esecuzione corpo, ma non del ciclo totale.

RUNE = carattere <sup>intero</sup> → 32 bit.

23/10/18

- Jan x rune  
 $x = '0'$  ← apici e non apostrofi perché apostrofi servono per le stringhe.
- introduce una lettera speciale → '\n' vai a capo
  - Se vogliano inserire ' → ''''
  - Se vogliano inserire \ → '||'
  - '\u02264' → less ≤ → numero in esadecimale.

↳ Con \ minuscolo possono mettere 4 caratteri che indicheranno un elemento.

- Printlu(x, string(x))

↓  
 stampa  
 il numero in  
 decimale

stampi il carattere  
 speciale

\u2264
Printlu(x) → 8864
Printlu(string(x)) → ≤

RUNE = è un tipo. Rappresenta un carattere in un intero a 32 bit.

→ più precisamente rappresenta un Code Point Unicode.

[ Val x Runne  
x = '0' ]

→ (ATTENZIONE!: spazi e non apostrofi perché apostrofi servono per le stringhe, spazi per le rune.)

\ introduce una lettera speciale → '\n' vai a capo

Se vogliamo inserire ' ' → ''  
Se vogliamo inserire \ → '\\'

\n → VAI A CAPO  
\b → BEEP DI AVVERTIMENTO  
\t → STAMPA UN TAB

Println (x, String(x))

• stampa il numero in decimale  
• stampa il carattere speciale

[ R-1 ]  
Val x Runne  
x = '\n 2264'

Println (x) → stampa : 2264  
Println (String(x)) → stampa ≤

Se voglia vedere la runa e non il codice, convertito in string.

ASCII = • American Standard Code for Information Interchange  
• Pubblicato dall'ANSI (American National Standard Institute) nel 1968.

• Codice per la codifica dei caratteri: è una lista di 128 numeri con il proprio carattere in base 8.

Man mano che si sviluppa si vengono a creare nuovi standard.

ISO / IEC 8859-1 = • Codice per la codifica dei caratteri orientato alla rappresentazione delle lingue dell'Europa Occidentale.  
• Viene standardizzato nel 1998  
• Costituisce la base degli insiemi di caratteri ad 8 bit.

• Codifica 191 caratteri originati dall'alfabeto latino.  
• Si riduce ad ASCII nella sua metà bassa con bit più significativo a zero.

UNICODE = • Sistema di codifica ovunque per tutto il mondo (viene continuamente aggiornato)  
• Ogni carattere è identificato da: NOME + CODE POINT (numero)  
• Nella prima parte incorpora la codifica ISO/IEC 8859-1

= Per enumerare tutti i caratteri previsti (come una parola di 21 bit), sono state previste CODIFICA A LUNGHEZZA VARIABILE, le più diffuse prende il nome di **UTF-8** =

- UTF-8 = • Unicode Transformation Format, 8 bit  
 • Codifica di caratteri in sequenze di lunghezza variabile di byte  
 • Usa gruppi di byte per rappresentare i caratteri Unicode
- UTF-8 usa da 1 a 4 byte per rappresentare caratteri Unicode:

- ~ I primi 127 Codepoints, corrispondono all'alfabeto ASCII, sono rappresentati da 1 byte

0xxxxxxx → 1 byte - 8 bit

~ I primi 1792 Codepoints (rappresentano caratteri usati dalle lingue occidentali), sono rappresentati - esclusi i punti - da 2 byte

110yyyyy | 10xxxxxx → 2 byte - 11 bit  
 1 byte      1 byte      {yyyyy xxxx xx}

~ I primi 65536 sono rappresentati - esclusi casi precedenti - da 3 byte

1110 yyyy | 10xxxxxx | 10zzzzzz → 3 byte - 16 bit  
 1 byte      1 byte      1 byte      {yyyy xxxx zzzzzz}

~ I restanti Codepoints sono rappresentati da 4 byte, con analogo ragionamento applicato in precedenza.

- ATTENZIONE!
  - papà ↪ 4 CARATTERI IN RUNE
  - 5 CARATTERI IN GO (CAUSA A CON ACCENTO - CARATERE SPECIALE)
  - 1 Codepoint Unicode = emoji = ☺

- STRING = è un tipo. ↪ CONCETTUALMENTE: insieme di rune  
 ↪ PRATICAMENTE: rune codificate in UTF-8 (quindi sequenza di byte rapp. in UTF-8)

[var s String]  
 $s = ""$ ] oppure [s := ""] → definisco variabile di tipo String  
 → inizializzo a stringa vuota

Scan (&s) → legge una parola per volta. Tokenizza INPUT!  
 Println (s, len(s)) → len(s) permette sapere lunghezza stringa

Concatenare più stringhe:

```
[ s, t := "", ""  
  Scan(&s)  
  Scan(&t)  
  Println(s, "e", t) ]
```

• STRINGHE LETTERALI COSTANTI:

Sequenza di caratteri delimitata da ""  
ex: "e"

• STRINGHE VARIABILI:

s è una stringa variabile

### STRINGHE MULTICHA:

- Si creano con il backtic (o apostrofo sinistro → sul pc è dritto ')
- ATTENZIONE! " " CARATTERE  
" " STRINGA

```
[ s := 'Ehi'  
    Ciao' ]
```

Stampa così come scriviamo. ex: "hv" così va a capo  
"hv" così lo scrive.

- s[1:3] range → legge parte della stringa. Parte da 0 e l'estremo sx incluso  
l'estremo dx escluso
- s += string (attacca il valore della stringa ad s)

For RANGE = prende una stringa e crea un ciclo.

È controllato da 2 variabili:  
① Posizione in byte  
② Ritme.

```
[ for i, r := range s  
  { Println(i, r, string(r)) } ]
```

Se dichiaro i e non le soffio stampare,  
non se il programma !!

Per non mettere la variabile i, insisto -

```
[ for _, r := range s ]
```

Il ciclo for range viene eseguito quante sono le righe (non in byte!)

- import ("unicode") → IMPORTANTE PER USUFRUIRE DEI COMANDI IsLetter,  
ToUpper, ToLower ecc...

## Per leggere stringhe da tastiera:

```

import ("bufio"; "os")
func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        s := scanner.Text()
        print(s)
    }
}

```

→ trova una riga

→ vero o falso se trova una riga o no

→ stampa ciò che scivi a tastiera

## RAPPRESENTAZIONE DELL'INFORMAZIONE

- ① NOTAZIONE POSIZIONALE:
- metodi di scrittura dei numeri, nel quale ogni posizione è collegata alle posizioni vicine da un moltiplicatore, chiamato *base*.
  - Valore di una cifra = cifra · base<sup>n</sup> posizione della cifra.

$$\text{ex: } 1125_{10} = 5 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 + 1 \cdot 10^3$$

$$864_8 = 4 \cdot 8^0 + 6 \cdot 8^1 + 8 \cdot 8^2$$

- ② • La precisione con cui rappresentiamo gli oggetti (numeri) è limitata.  
 • Le variabili hanno una gamma di valori definita  $\rightarrow$  Int usa almeno 32 bit.

- ③ Tipi:
- Ci è un linguaggio a TIPIZZAZIONE FORTE in quanto non permette di eseguire operazioni con valori di tipi diversi: somma intero 16bit e intero 32bit Non viene!
  - Per leggere valori con tipi differenti è necessario convertire le variabili.
  - Costanti e valori letterali, potendo essere usate senza tipo, sono più flessibili.

- Esistono diversi tipi:

- ① Logici
- ② Numeri Interi STANDARD
- ③ Numeri Interi SISTEMA DIPENDANTI  $\rightarrow$  3 differenti per Go
- ④ Numeri Flutuanti
- ⑤ Numeri COMPLESSI
- ⑥ TESTUALI
- ⑦ ARRAY
- ⑧ SEZIONE
- ⑨ MAP
- ⑩ PUNTATORE
- ⑪ STRUTTURA
- ⑫ INTERFAZIA
- ⑬ FUNZIONE
- ⑭ CANALE



1. Logici: • BOOL = rappresenta un valore logico booleano vero o falso espresso con le costanti TRUE e FALSE

2. Numeri interi:
- standard:
    - UINT8 (byte) = tipo uint8 e il suo alias byte, rappresentano un intero positivo a 8 bit (numero nel intervallo da 0 a 255)
    - UINT16 = intero positivo a 16 bit (intervallo da 0 a 65535)
    - UINT32 = intero positivo a 32 bit.
    - UINT64 = intero positivo a 64 bit.
  - INT8 = numero con segno a 8 bit (intervallo -128, 127)
  - INT16 = numero con segno a 16 bit
  - INT32 = numero con segno a 32 bit
  - INT64 = numero con segno a 64 bit

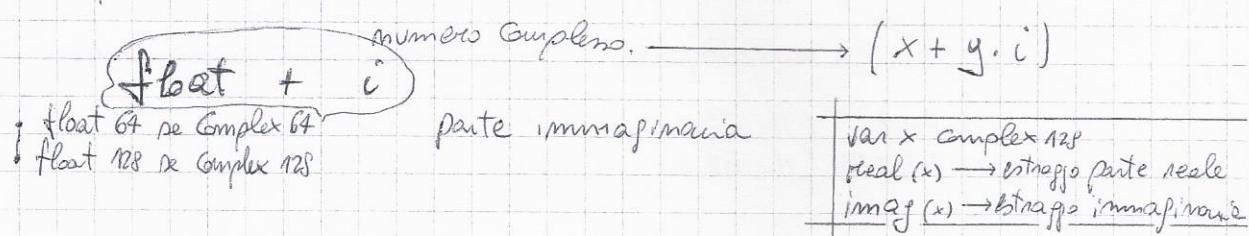
3. Numeri interi sistema dipendenti (dipendono dalla configurazione del sistema in cui il programma è eseguito)

- UINT = rappresenta un numero intero positivo a 32 bit o 64 bit
- INT = rappresenta un numero con segno a 32 bit o 64 bit
- UINTPTR = è numero intero positivo che rappresenta un puntatore.

4. Numeri decimali:

- FLOAT32 = numero decimale a 32 bit
- FLOAT64 = numero decimale a 64 bit

5. Numeri Complessi:



6. Extra:

- STRING
- RUNE = uint32
- BYTE = uint8
- FUNZIONI

## RAPPRESENTAZIONE DI INT

- Tutto ciò che è manipolabile dal calcolatore (numeri - caratteri - matrimoni - suoni - programmi)  $\rightarrow$  è rappresentato in forma binaria (al livello macchina).
- Quanti bit mi servono per manipolare  $N$  oggetti?  $K = \lceil \log_2 N \rceil$
- Per rappresentare gli int si usa la NOTAZIONE IN BASE 2, ovvero il codice binario.
- VARIABILI SENZA SEGNO:
  - se il numero non è rappresentabile, lo si butta (overflow o trabocco)
  - il risultato è il resto della divisione intera per 256
- VARIABILI CON SEGNO:
  - Si rappresentano in Complemento a 2

- Metodo più utile e diffuso
- Il bit più significativo ha peso negativo
  - o positivo  $\{ 0 = \text{positivo}$
  - $1 = \text{negativo}$
- Numero binario positivo si può rendere negativo: invertendo tutti i bit e somma 1.
- Valore assoluto numero binario negativo si ottiene con stessi procedimenti.
- insieme dominio:  $(-2^{K-1}; +2^{K-1}) - 1$

## RAPPRESENTAZIONE DI FLOAT

- Si utilizza il metodo floating-point (numero in virgola mobile)

- Numero formato da:
  - mantissa (M)
  - esponente (e) in una relativa base (b)
- Si utilizza 1 bit per il segno (s)
- Numero  $a = s * M * b^e$  ex:  $1,0110 * 2^6$
- Posso scrivere solo potenze inverse di:  $2(j/2^K)$
- Flx: ( $M=23$  bit,  $e=8$  bit)
- Flot64 ( $M=52$  bit,  $e=11$  bit)

## • Costanti (Const)

- In informatica una costante identifica una posizione di memoria il cui valore non varia nel corso dell'esecuzione di un programma.
- le costanti possono essere prevalentemente di tre tipi:
  - Costanti numeriche
  - Costanti di carattere
  - Costanti di stringhe
- Il COSTRUTTO CONST si utilizza come VAR, ma non dichiara una variabile locale, bensì una costante.

$\boxed{\text{const } x \text{ int} = 3}$  {

- Non posso più assegnare un'altra costante a x.
- x JAGGA per tutto il codice
- Se si modifica const x, verrà modificata in tutto il codice

}

- le costanti servono per risolvere il problema dei MAGIC NUMBER

→ Magic Number (numeri magici), sono numeri che si trovano in posizioni di codice e non si capisce dove vengono utilizzati.  
• Si fa una dichiarazione CONST e gli si dà un nome, risolto problema dei magic number.

ATTENZIONE! → Ci consente le costanti senza tipo, dove il tipo di costante si adatta alla situazione in cui si trova. Nessun problema tra int e float

$\boxed{\text{const } x = 3}$  {

- Senza tipo
- Non è variabile
- non è assegnabile

}

- Più semplici:
  - Le costanti sono valori letterali ai quali viene dato un nome, per esempio mi dà al valore 3,14 il nome pi greco.
  - Si dichiara una costante quando si crea una costante attribuibile un nome e un valore
  - Le costanti servono a rendere il codice più leggibile e più efficiente
  - Le costanti sono valori che non cambiano mai nel programma.
  - Ex: scrivo programma che gestisce un campionato di calcio e dichiara costante per numero di partite per girone; la quale può essere: SquadraGirone = 4. Cambiando il valore di questa costante nella sua dichiarazione, il programma userà il valore indicato ogni qualvolte la costante SquadraGirone verrà usata.

## Switch

- L'istruzione switch è un'istruzione decisionale a più vie che confronta un valore o un tipo per uscita inizialmente con altri valori o tipi.
- viene definita selezione multivoca perché ci sono più vie di uscite, a differenza di ELSE che è selezione binaria perché ha solo due uscite: true/false.

```
Switch r {  
    Case : ...  
    :  
    Case : ...  
    :  
    default: ...
```

- VENGONO ESEGUITI IN ORDINE ( $\downarrow$ ) I CASI E QUANDO UNO SI ATTIVA, LO SWITCH TERMINA.
- Switch senza espressione = Switch TRUE
- VAIUTA TUTTI I CASI FINCHÉ NON TROVA UN TRUE

- FALLTHROUGH = non fa terminare lo switch anche se il case è verificato e quindi attraversa anche i case successivi.

(Anche se Case è true, )  
grazie a fallthrough  
segue anche il caso succ.  
ovvero default.

```
for_, r: range s {  
    Switch r {  
        Case 'a', 'e', 'i', 'o', 'u':  
            r = 'u'  
            fallthrough  
        default:  
            Print(string(r))  
    }  
}
```

## • FUNzioni

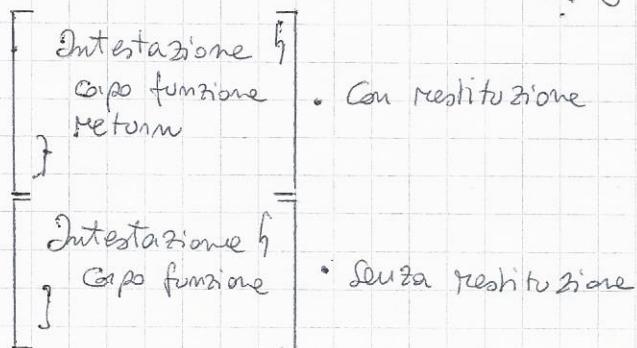
- Una funzione è un particolare costituto sintattico che permette di aggiungere, all'interno di un programma, una sequenza di istruzioni in un unico blocco. A partire da determinati input, svolgerà operazioni/azioni sui dati del programma stesso, così da restituire determinati output.
- Esistono due tipi di funzioni:
  - funzioni impostate.
  - funzioni create.
- In un programma si descrivono prima le funzioni create, poi per ultima la func main().

Funzioni che restituiscono (return): func collatz (x int) int {

Funzioni che non restituiscono: func collatz (x int) {

- INTESTAZIONE FUNZIONE:

func + nome (dato da noi) + (argomenti + tipo) + tipo di restituzione.



- Le variabili delle funzioni sono locali e non condivise tra di loro, quindi possono essere chiamate con nomi differenti.
- La variabile utilizzata per creare la funzione viene utilizzata come variabile locale e viene sostituita al valore inserito dall'utente → Si passa alla funzione il valore, NON LA VARIABILE!
- ATTENZIONE!: le variabili possono avere nomi differenti, ma i tipi delle variabili devono essere uguali per svolgere operazioni.

- Le funzioni possono avere più di un parametro, ma ci sono delle regole:

- ① Posso avere quanti argomenti voglio, ma separati tra di loro da una virgola
- ② Posso restituire più valori
- ③ Per estendere parte di argomenti di una func - mettiamo - underscore sugli altri argomenti.

ex:

func somalif (a, b float64) (float64, float64) {  
    return a+b, a-b

- Per convenzione, le funzioni vanno dichiarate con le prime lettere maiuscola ? "unicole. IsLetter" ↗

- Per eseguire funzioni salvate su un file a parte e non implementate all'interno del file principale da eseguire:

go run b1.go file1.go file2.go

- Il compilatore guarda, compila i segmenti e infine chiama all'interno del file principale.

SCANDIRE STRINGA DAL FONDO ALL'INIZIO:  
ciclo for → for  $i := \text{len}(s) - 1$ . ecc....

- Prendere una stringa dalla func main e restituirne intero :  $\rightarrow$  func binario (s string) int

- Nell'intestazione posso non mettere parentesi quando abbiano un tipo senza nome:

↓ ex: func binario (s String), int → meglio mettere sempre le parentesi

PACCHETTO <sup>b1</sup> stronj <sup>"u"</sup>

- stconv. Atoi ( $^{16} - 42$ )
  - stconv. Itoa ( $-42$ )

## func Atoi (Ascii to Integer)

- Converte stringa a intero
  - da anche un errore come risultato
  - errore = mil Se e andato tutto bene

\* func Itoa (int to string)

• Stesso procedimento di Atof.

## PACCHETTO "strings"

- strings. Contains
  - strings. HasPrefix
  - strings. Index

## func Contains

- guarda se la seconda stringa è contenuta nella prima

## func HashPrefix

- Guido se le stringe comunica con una certa  
peregrina di cosa tien

## func Index

- guarda il punto di potenza della Sottostringa  
di 5, all'interno di 5

## PUNTATORI

- I puntatori sono tipi di dati che rappresentano la posizione (usando indirizzi di memoria) di elementi del programma come variabili, oggetti, strutture dati, sottoprogrammi.
- Diversi tipi:
  - 1 - Tipi BASE: complex, string, int ...
  - 2 - Tipi COMPOSTI: formati da strutture (contengono più variabili correlate)
  - 3 - Tipi INTERFAZIA: si specifica solo che tipi di funzioni si possono applicare al tipo e non come è fatto il tipo.
- I tipi composti (2-) rappresentano la posizione in memoria delle variabili
  - stanno davanti al nome del tipo originale ex  $*int$        $*string$
  - Si può inizializzare una variabile con puntatore: var p \*int  
(variabile p con puntatore di tipo int)
  - Esistono puntatori con valore indefinito, quindi con il puntatore stesso che non punta a nulla:  $nil$  (JAVA  $\Rightarrow$  null)  
 $C \Rightarrow NULL$   
 $Go \Rightarrow nil$
  - $\&$  = ampersand  $\rightarrow$  simbolo che consente di ottenere un puntatore a una data variabile, ovvero calcolare l'indirizzo.

## OPERAZIONI SUI PUNTATORI:

- ① OPERAZIONE DI DEREFERENZIAZIONE:
- è rappresentata da un operatore speciale
  - produce come risultato l'oggetto puntato.

Ex:  $\begin{bmatrix} \text{var } n \text{ int} \\ \text{var } p \text{ *int} \\ *p = 3 \end{bmatrix} \rightarrow$  assegna 3 alla variabile int puntata da p, ovvero n

- ② OPERAZIONE PER CALCOLARE INDIRIZZO:
- non è fornita da tutti i linguaggi
  - consente di ottenere un puntatore a una data variabile, ovvero calcolare l'indirizzo.

Ex:  $\begin{bmatrix} \text{var } n \text{ int} \\ \text{var } p \text{ *int} \\ p = \& n \\ *p = 4 \end{bmatrix}$

assegna al puntatore p l'indirizzo della variabile n e finché p mantiene tale valore, qualsiasi uso di \*, avrà effetto sulla variabile n puntata da p.

Ex: fint. Scan (&x).

$\boxed{\begin{array}{l} \text{var } x \text{ int} \\ p = \&x \\ *p = 0 \end{array}}$  → metto 0 nella variabile puntata da p, ovvero  $x$  non è 0  
 → ATTENZIONE! nil è inindirizzabile (non si può fare indirizzamento)  
 Se NIL

- & e \* sono come operazioni inverse.

CASO<sub>1</sub>

$x \equiv * \& x \rightarrow * \&$  si cancellano, ottieniamo solo  
oggetto semantico  $[x \equiv x]$

CASO<sub>2</sub>

$p \equiv \& * p \rightarrow$  con p puntatore valido! non ottieniamo solo  
oggetto semantico. ♪

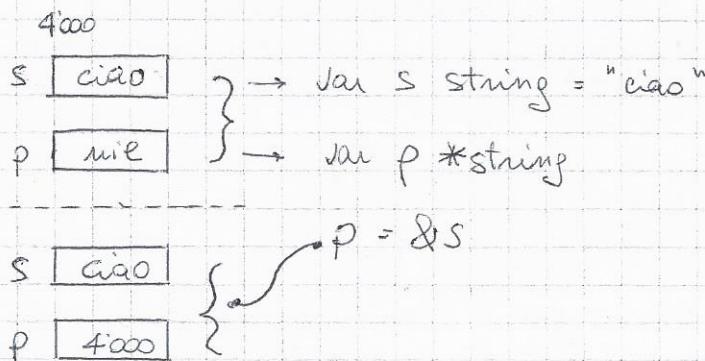
posizione in  
memoria della var  
puntata da p

$\& * p \rightarrow$  operazione  
 $p \rightarrow$  variabile

ex: var s string = "ciao"  
var p \*string

$p = \&s$

$\text{frmt. Println}(p) \Rightarrow 4000$   
 $\text{frmt. Println}(*p) \Rightarrow \text{ciao}$   
 $\text{frmt. Println}(*p[0]) \Rightarrow 98$  { sarebbe il numero del carattere in posizione zero delle  
parole ciao ⇒ c = 98



- ALIUSING =  $*p$  è esempio di Aliasing. Nell'esempio sopra,  $*p$  è come se fosse un altro nome di s.

- NEW (tipo che soglio creare) = Come creare una var senza nome, dalla quale abbiamo solo il puntatore.

ex:  $\boxed{\begin{array}{l} \text{var } p \text{ *int} \\ p = \text{new(int)} \end{array}}$

EX:  $x := 5$   
 $p := \&x$   
 $\text{val } q \text{ } **\text{int} = \&p$

→ è di tipo \*int

{

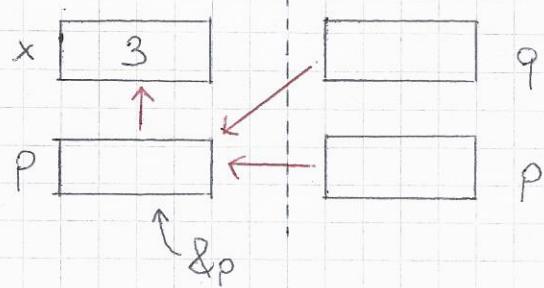
x	5
p	4000
q	3000

}

- ex:
- func piúumo (&x) → x è di tipo int
  - func piúumo (p) → p è già puntatore, non serve &
- 

- & → indica l'indirizzo della locazione di memoria
- \* → indica il contenuto della locazione di memoria
- && → NON si PUÒ FARE perché un indirizzo è un numero!
- \*\* → Seguo due volte e leggo il contenuto.

EX: func main      func piúumo



\*\* (q)++ :

- Seguo due volte e ando a x, poi leggo contenuto

```

func piúumo (p **int) {
    q := p
    (**q)++
}

func main () {
    x := 3
    p := &x
    piúumo (&p)
    fmt.Println(x)
}

```

## TYPE / Struct

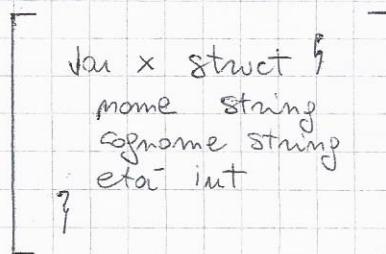
TYPE = funzione per definire nuovi tipi ~ Alias di "tipo"

- Ex: type anno int (uguale a scrivere type anno = int)
- Non si può fare l'assegnazione diretta
- In "type anno int" anno è comodo come nome, ma in questo caso non c'è nessuna differenza tra tipo int e tipo anno.

Struct = struttura - permette l'aggregazione di più variabili di tipoeterogeneo (diverse tra loro) in un unico "sacchettino".

- L'aggregazione avviene in modo simile a quello degli array, ma a differenza di questi non è in maniera ordinata e omogenea proprio perché una struttura può contenere variabili di tipo diverso.
- per denotare una struttura si usa la parola chiave struct.

1) STRUTTURE COMPOSTE =



• Variabile x è una struttura composta

• Per richiamarla usiamo la "DOT NOTATION"

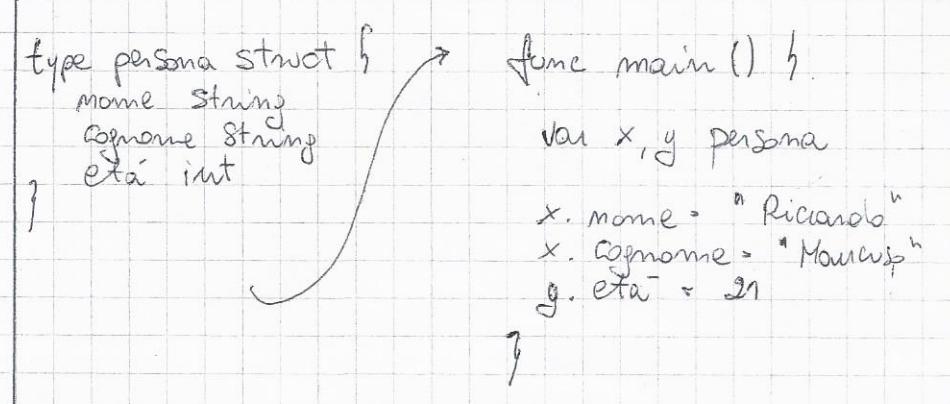
Ex: `[x. nome = "Sebastiano"]`

{ "notazione col punto", schema  
 sintattico per riferirsi a proprietà  
 degli oggetti come attributi.  
 - NOTAZIONE -  
 oggetto. proprietà

•  $x = y \Rightarrow$  tutti i dati nella struttura di y vengono assegnati a x

2) TYPE + Struct = L'uso di type, combinato con struct, ci permette di creare tipi di dato molto complessi

Ex:



- le strutture possono essere definite letteralmente : (1.1)

```
[ func main() {  
    x := persona { nome: "Riccardo", cognome: "Monaco", eta: "21" }  
    :  
}
```

ATTENZIONE : • usare ":" e dividere con ","

- Struttura che contiene struttura : (2)

```
[ func main() {  
    var x matricola  
    x.persona.nome = "Luca"  
}
```

ATTENZIONE : se non utilizzo un tipo initializz.  
in una struct , se stampo variabile  
di tipo struct senza usare "nome"  
per esempio , mi stampa la stringa  
vuota . Con tipo int ("eta" per esempio)  
mi stampa 0

## ARRAY e SLICE

ARRAY = Un array / vettore / matrice è una sequenza numerata di elementi di un unico tipo.

{  
    ARRAY = usa [ ] ; tipo omogeneo  
    STRUCT = usa { } ; tipieterogenesi}

- Serve per memorizzare collezioni di oggetti omogenei
- Si può costruire un array a partire da un qualsiasi tipo base
- Il numero di elementi da cui è costituito si chiama lunghezza
- DICHIARARE UN ARRAY:

- var a [5] int  $\rightarrow$  array "a" formato da 5 elementi di tipo int

- tra le [ ] ci deve essere un valore costante
- I suoi elementi vanno da 0 a K-1
- len(a) dice quanto è lungo l'array a

- INIZIAZZARE UN ARRAY:

- var a [5] int = [5] int {1, 2, 3, 4, 5}

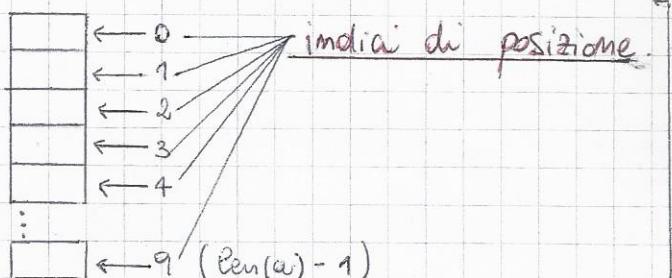
- a := [5] int {1, 2, 3, 4, 5}

- a := [...] int {1, 2, 3}  $\rightarrow$  indica lui il numero di elementi

- POSIZIONARE UN VALORE IN UNA POSIZIONE SPECIFICA DELL'ARRAY

a := [10] int {4:5}  $\rightarrow$  mette in posizione 4 (5° elemento) il valore 5

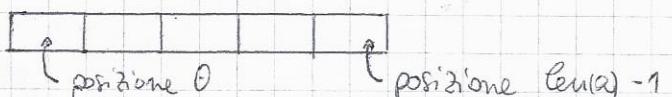
var a [10] int



I vettori / array sono di dimensione fissa, quindi valutabili al tempo di compilazione.

ALTRI MODO  
PER L'APPRESENTAZIONE  
UN ARRAY

var a [5] int



ATTENZIONE!

- Meglio passare un puntatore al vettore anziché il vettore stesso

p := \* [10'000'000] int

## Ex 1. MEDIA VALORI DELL'ARRAY

- Inizializzo un array di 10 indici, senza nessun valore.

$a := [10] \text{ float64 } \{ \}$

- Scannizzatore da tastiera valori da mettere nell'array:

```
for i := 0; i < len(a); i++ {  
    Print.Scan(&a[i])  
}
```

- Sommo elementi dell'array (VALORE)

$s := 0.0$

```
for i := 0; i < len(a); i++ {  
    s += a[i]  
}
```

## SLICE =

Una slice / fetta è un riferimento a un segmento contiguo di un array e contiene una sequenza numerata di elementi di tale array.

- Uguale agli array, ma le slice non possono allungare nel tempo
- Una slice, una volta inizializzata, è sempre associata ad un array di base; ogni volta che aggiunge un valore essa inizializza un pezzo di array.
- Slice hanno:
  - Lunghezza = numero di elementi presenti nella slice
  - Capacità = numero di elementi possibili da mettere.
    - ↳ slice's capacity = quante info posso aggiungere prima di generare un errore.

- All'inizio la slice contiene nil, come una slice vuota.

- CREARE UNA NUOVA SLICE: - var a []int {} man ciascuno la lunghezza

{ - a = make ([]int, 10) -> voglio CREARE UNA SLICE DI INTERI, FORMATA DA 10 ELEMENTI E METTERE IN a  
Conoscendo la lunghezza  
devo solo dare - a = make ([]int, 10, 16) -> CREO UNA SLICE DI 10 ELEMENTI, Poi NE AGGIUNGO ANCORA 6.

- AGGIUNGERE ELEMENTO ALLA SLICE:

- Append -> a = append (a, s)

• `a := make([]int, 5)`  $\rightsquigarrow$  5 indica anche la lunghezza

• Slice e array possono essere momeisti con il for range:

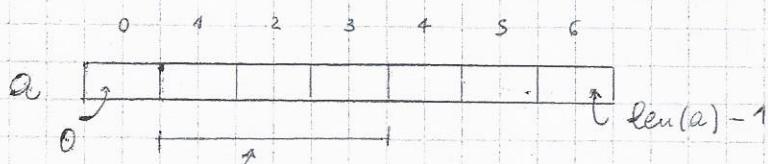
for  $i, x := \text{range } a$  {

    fmt.Println( $i, x$ )

$i \rightarrow$  INDICA L'INDICE  
 $x \rightarrow$  INDICA IL VALORE

}

• SUBSlicing = andiamo ad "affettare" un array e lo facciamo diventare una slice di array, oppure affettiamo slice per farla diventare slice di slice.



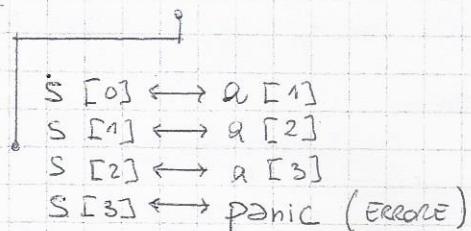
$a[1:4]$   $\rightsquigarrow$  indice iniziale: INCLUSO  
indice finale: NON INCLUSO

• OBTENIAMO COSÌ UNA FETTA DELL'ARRAY, OVVERO UNA SLICE CHE CONTIENE:

Inizio
fine
$\&a$

1      4      } Slice è un puntatore, ma  
        } contiene anche inizio e fine

• Gli accessi alla slice avvengono così:  $s := a[1:4]$



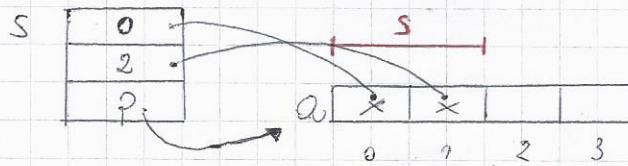
• Una slice è un modo differente di vedere l'array sovraccorrente (in questo caso  $a$ )

• Si può costruire una slice a partire da un'altra slice. (POSSIAMO FARLE FETTE DI SLICE QUANTE VOLTE VOGLIAMO)

**ATTENZIONE!**: le stringhe sono slice immutabili di array di byte.

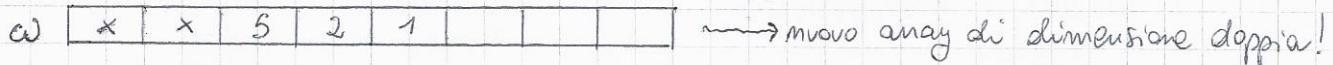
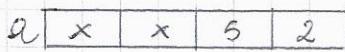
## COME FUNZIONA UNA SLICE :

- Ricordiamo : ARRAY → fisso  
SLICE → espandibile
- Usando Append, aggiungiamo elementi alle Slice. La Slice fa sempre riferimento a un array (senza elementi in questo caso) e ogni append aggiunge ad utilizzare la parte dell'array che non stiamo usando.



(Ricordiamo : 2 NON COMPRESO ! )

- Quando usiamo tutti gli spazi disponibili dell'array a cui facciamo riferimento, viene creato un altro array 2 volte più grande.  
In questo caso : a len = 4 → w len = 2 · 4 = 8



- Cap slice s "I CASO" = 4 / len(s) "I CASO" = 2
- Cap slice s "II CASO" = 8 / len(s) "II CASO" = 5

ATTENZIONE ! : Se dalla slice "s", ne creo un'altra di nome "f" e dopo modifico gli estremi di "f" → NON MODIFICO ANCHE "S" !

ATTENZIONE ! : Se creo una nuova funzione dove modifica una slice, devo restituire la slice e non una sua sottofetta !

- APPEND si può fare anche tra due slice (non più di due).

```

S := [] int {1, 2}
t := [] int {9, 10}
front. Println (append (S, t))
    
```

- Pacchetto "os" → Args (argomenti scritti su riga di comando del terminale)
  - quando chiamo argomenti → os.Args / os.Args [0] / os.Args [1]

Ex: ① func main() {  
 front. Print (os.Args)  
}

② if len (os.Args) != 2 {

## STAMPA FORMATTATA

Dobbiamo prima spiegare come vogliano venire rappresentati i dati, poi scriviamo il dato che vogliano stampare.

: `fmt.Println`:

	<code>%v</code>	stampa dato come faresti normalmente con <code>Print</code>
Per int	<code>%d</code>	stampa dato in <u>DECIMALE</u>
	<code>%x</code>	stampa dato in <u>ESADECAZIALE</u>
	<code>%b</code>	stampa dato in <u>BINARIO</u>
	<code>%o</code>	stampa dato in <u>BASE8</u>
	<code>%T</code>	stampa il <u>TIPO</u> del dato (int, float64, ecc...)
	<code>%g</code>	si usa per i <u>Floating-point</u>
Per string	<code>%s</code>	stampa dato <u>STRINGA</u>

~~~~ CON `GO VET` + nome programma -go , esaminiamo PROGRAMMA e cerchiamo di trovare DEGLI ERRORE

|                   |                                              |
|-------------------|----------------------------------------------|
| <code>%20d</code> | Capitula da 20 cifre                         |
| <code>.5g</code>  | 5 cifre decimali dopo il punto               |
| <code>08d</code>  | voglio numero allineato con 0, non con spazi |
| <code>%%</code>   | stampa simbolo percentuale.                  |

(EX:  $x :=$  )  
 $\%02 \rightarrow 01$   
 $\%04 \rightarrow 0001$ )

# MAPPE

Mappe = è un gruppo di elementi non ordinati di un tipo (chiamato "tipo dell'elemento") indiziato da un insieme di chiavi ovunque di un altro tipo (chiamato "Tipo chiave")

- Mappe permettono di associare un output ad un input
- Bisogna necessariamente dare una chiave e un valore (natura arbitraria delle chiavi)
- Valore di una mappa non inizializzata è nullo

- Dichiarazione:

- var m map[int] int  $\rightsquigarrow$  mappa che prende un intero e sostituisce un intero

- Se map vale null, provando ad accedervi: panic

- Inizializzazione / Creazione:

- m := make(map[int] int)

- m[1] = 5  $\rightsquigarrow$  voglio associare alla chiave 1 il valore 5

- Dichiarazione letterale:

- m := map[string]int{ "Marco": 3, "Sebastiano": 5 }

- Tipo CHIAVE / CHIAVE:

- Gli operatori di confronto (`==` e `!=`) devono essere completamente definiti per i operandi del tipo chiave, quindi il tipo chiave non può essere `ARRAY` o `SLICE`.

- Se il tipo chiave è un tipo interfaccia, operatori di confronto devono essere definiti per i valori della chiave disponibili (in caso di errore, panic nel tempo di esecuzione)

- c, ok := m[0]  $\rightsquigarrow$  { "ok" ci dice se ha trovato la chiave } { "c" ci dice il valore della chiave. } { "ok" oppure no (mappa con chiave 0) }

- delete(m, 0)  $\rightsquigarrow$  come togliere chiave dalla mappa

- Lunghezza mappa: - `len(m)` - il numero di elem. di una mappa rappresenta la sua lunghezza.

- For range: `for nome, punti := range m {  
 fnt. Println(nome, punti)  
}`

- Capacità iniziale mappa non vincola le sue dimensioni: le mappe crescono per accogliere il numero di elementi memorizzati al suo interno.
- Cerchando di prelevare un valore di mappa con una CHIAVE NON PRESENTE NELLA MAPPA, ritornerà il valore zero rispetto al tipo di elemento della mappa.
  - ↳ Ex: ① LA MAPPA CONTIENE INTERI } ③ mi ritorna il VALORE 0
  - ② CERCO CHIAVE NON ESISTENTE }

## CREAZIONE e DISTRUZIONE DI VARIABILI

- Quando un programma va in esecuzione, le variabili vengono memorizzate in 3 possibili posti:

### ① MEMORIA STATICÀ:

- Variabili definite all'esterno di funzioni
- Rimangono per tutta l'esecuzione del programma

### ② STACK (struttura dati):

- Variabili definite all'interno di funzioni (variabili locali)
- Nello stack possiamo aggiungere elementi e tirare fuori solo l'ultimo elemento aggiunto.
- Stack è molto veloce - Si rappresenta per mappe storiche così:



### ③ HEAP (mucchio):

- Variabili create con malloc o new
- Heap è zona di memoria gestita da Go in maniera automatica
- Quando non utilizzano più l'oggetto, il programma elimina l'oggetto dallo heap e crea spazio.
- Heap è posto rispetto allo stack.

## RECORD di ATTIVAZIONE:

• è la zona di memoria che viene creata per ogni singola funzione

- Per ogni chiamata funzionale, si attiva Record di attivazione che assimila e ricorda ① DA DOVE VIENE CHIAMATO (riga) ② VARIABILI / PARAMETRI LOCALI
- Quando termina la funzione termina anche il record di attivazione correlato, quindi si cancella e svanisce (con riga e variabili)



# Funzioni Ricorsive

• RICORSIONE: tecnica di programmazione fondamentale, permette di far richiamare a una funzione se stessa.

- FUNZIONI RICORSIVE:
- funzioni che chiamano se stesse
  - STACK OVERFLOW (tubizzo delle pile): errore che ti informa che è stato creato un loop infinito di chiamate.
  - per ogni funzione ricorsiva bisogna trovare:
    - ① CASO/i BASE
    - ② CASO GENERALE

- La tecnica ricorsiva permette di scrivere algoritmi eleganti e sintetici e rende più efficienti gli algoritmi iterativi.
- ESEMPI:

## ① CALCOLO FATTORIALE DI $n$ .

• Caso base:  $0! = 1$  per  $n = 0$

• Caso generale:  $n! = n * (n-1)!$  per  $n > 0$

```
[func fact (n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}]
```

## ② CALCOLO SEQUENZA FIBONACCI

• Caso base:  $F(0) = 0$   
 $F(1) = 1$

• Caso generale:  $F(n) = F(n-1) + F(n-2)$

```
[func fib (n int) int {
    if n < 2 {
        return n
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}]
```

### 3) STABILIRE SE STRINGA È PALINDROMA:

- Caso base : 1) Stringa vuota è palindroma } if  $\text{len}(a) \leq 2$  }  
2) Stringa da un carattere è palindroma } RETURN TRUE }
- Caso generale : ciclo dove 1° e ultimo carattere devono essere uguali
  - if  $a[0] \neq a[\text{len}(a)-1]$  } RETURN FALSE ?
  - RETURN funzione palin len stringa ( $a[1:\text{len}(a)-1]$ )

# PACCHETTI - ATTACCAMENTO DI METODI

- WORKSPACE
  - spazio di lavoro di Go (trovabile scrivendo "go env")
  - importante per usare librerie scritte da noi
  - formato da:
    - 1) PKG (Package): contiene i pacchetti che abbiano installato
    - 2) SRC (Source):
      - librerie contengono struttura, funzioni, ma non contiene main
      - scarica i seguenti dei pacchetti installati (contenuti nel pkg)
      - scarica i seguenti dalle librerie, compila e poi mai più ricompilato.

## CREARE MIEI PACCHETTI:

- ① DECIDO NOME DEL MIO PACCHETTO:  
per: palin
  - organizzo funzioni come voglio, ma in un sottopacchetto (ex: math/rand) non posso mettere funzioni con stesso nome. Posso farlo solo in sottopacchetti differenti.
- ② CD PALIN: palin è nome deciso da me. Cd Palin faccio nel terminale.
- ③ GO BUILD PALIN: Compilo il pacchetto da me creato.
- ④ GO INSTALL PALIN: installo il mio pacchetto, memo' non sarebbe eseguibile da altri programmi.
- ⑤ IMPORT ("PALIN"): per usarlo in un altro programma devo importarlo, come faccio con "fmt"; "strconv"; "os" ecc...

- ⑥ RICORDARE !:
  - nel mio pacchetto/libreria non c'è la func main, ma posso la func IsPalin (come esempio)
  - La func nel pacchetto è una func normale:

```
func IsPalin (s string) bool {  
    :  
    return true  
}
```

← Solo questa!

P.S. Molto utile avere variabili globali nel pacchetto.

P.S. GO PATH (variabile d'ambiente): definisce quale è lo spazio di lavoro di Go.

## VARIABILI E PACCHETTI

① os.Stdout → è una variabile che si trova in un pacchetto ("os") e serve per scrivere il flusso standard di input (TASTIERA o ALTRA SORGENTE, PER ESEMPIO: <miotest.txt>)

② os.Stderr → è una variabile contenuta in "os", serve per scrivere in output (non usata da noi)

③ os.Stdin →

④ Scanner → serve per leggere riga per riga e vuole sapere quale è l'input, avendo (os.Stdout)

## METODI

• METODO: indiciamo con il termine metodo un SOTOPROGRAMMA associato ad una STRUTTURA e che rappresenta un'operazione eseguibile sugli oggetti/argomenti di quella struttura.

• STRUTTURA, NOME DEL METODO (ARGOMENTI) : sintassi molto simile a quelle di funzione,

- ↳ Come chiamare un metodo.
  - + FUNZIONE:  $f(a, x)$
  - + METODO:  $a.f(x)$

• func (a \*struttura) nome di f (x tipo):

ex: METODO func(p \*persona) aumenta (x int) {

FUNZIONE func aumenta(p \*persona, x int) {

• METODI DICHIARATI PER PUNTATORE : possiamo dichiarare metodi sia per valore che per puntatore, ma nel 90% dei casi per puntatore  
Casi Cambiano valori nel main

P.S. Metodi possono avere stessi nomi e stessi argomenti (a differenza delle funzioni), ma NECESSARIAMENTE bisogna avere strutture diverse.