Emiddio Ingenito - Programmazione I

Author	Emiddio Ingenito @emikodes

Hello World:

```
package main

import "fmt" //pacchetto che contiene le funzioni per la formattazione del test o.

func main() {
```

```
    fmt.Println("Ciao Mondo!")
}
```

Import

```
package main

import "fmt" //pacchetto che contiene le funzioni per la formattazione del test
o.
//import funzioniMatematiche "math" //rinominazione del pacchetto, così nel c
odice utilizzerò funzioniMatematiche.math()
//import . "pacchetto" permette di utilizzare le funzioni del pacchetto, senza s
pecificare il pacchetto in cui sono contenute, quindi al posto di fmt.Println, nel
mio codice utilizzerò Printnl e basta.

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

Dichiarazione di variabili:

```
package main

import "fmt"

func main() {
    var a, b int = 5, 6 //Dichiarazione Multipla.
    var student1 string = "John" //type is string
```

```
    var (
        i int = 10
        k float64) //definizione di più variabili in blocco.

    fmt.Println(student1)
}
```

Stampa formattata:

```
fmt.Printf("%v,%T", i, i)
//Utilizzo di "Printf" per effettuare stampa formattata
//con printing verbs (segnaposti) v per value e T per type.
```

Output: `10,int`

Inferenza di tipo:

```
package main

import "fmt"

func main() {
        var student2 = "Jane"        //type is automatically known by value
        x := 2
        //inferenza di tipo, dichiarazione e inizializzazione.

    fmt.Println(student2)

}
```

💡 **Attenzione**! L'Utilizzo dell'operatore ":=" prevede operazioni dichiarazione ed assegnamento, pertanto produrrà errore nel caso di utilizzo con variabili già dichiarate.

```
package main

import "fmt"

func main() {

	var x int
	x:=5 //NON COMPILA!
		//:= inizializza ed assegna una nuova variabile, ma x è gia dichiarat
a!
}
```

Tuttavia, quest'istruzione risulta corretta:

```
var x int

x,y := 5,6
```

La regola dice che a sinistra dell'operatore **":="**, dev'esserci almeno una nuova variabile.
Quindi assegna 5 a x, e crea y e assegna 6.

Definizioni di variabili "a livello di package":

```
package main
import "fmt"

var i int
```

```
func main(){
   i=10
   fmt.Println(i) //compila e gira, variabile definita a livello di package
                  //scope globale, utilizzabile e visibile da tutte le funzioni.
}
```

Operazioni tra variabili:

💡 **Attenzione**! Le operazioni tra variabili, sono compatibili solo se i tipi della variabili sono uguali.

```
var i int = 10
var j float64 = 3.9

i+=j //non compila! type mismatch, ricorro al casting.
i+=int(j)
```

Il casting ad int, effettua un "**troncamento**" della parte decimale, non un arrotondamento matematico, pertando il valore di 'i' sarà 13, non 14.

Input:

```
package main

import "fmt"

func main() {
   var i int
   var j float64
```

```
    num, err := fmt.Scan(&i, &j)
    /*Scan scans text read from standard input, storing successive space-sepa
rated values into successive arguments.
    Newlines count as space. It returns the number of items successfully scann
ed.
    If that is less than the number of arguments, err will report why*/
    fmt.Println(num, err)
}
```

La funzione "**Scan**", permette la lettura di valori da **stdin**, separati da spazio (o newline).

Restituisce due valori che possono opzionalmente essere memorizzati, il numero di elementi successivamente letti, e una variabile di tipo errore.

Esempio:

```
package main

import "fmt"

func main() {
    var i, j, a, b int
    num, err := fmt.Scan(&i, &j, &a, &b)
    /*Scan scans text read from standard input, storing successive space-sepa
rated values into successive arguments.
    Newlines count as space. It returns the number of items successfully scann
ed.
    If that is less than the number of arguments, err will report why*/
    fmt.Println(num, err)
    fmt.Println(i, j, a, b)
}
```

Input: `1 2 3 K` → Output: `3, expected integer. 1 2 3 0` (la lettura del vaore della variabile 'b' è fallita)

Variable Shadowing:

> ***“Dichiarazione di una variabile in un contesto lessicale più piccolo, con lo stesso nome di una variabile già dichiarata in un contesto lessicale più ampio.”***

In informatica e in particolare in programmazione, lo **shadowing** è la regola di visibilità secondo la quale una variabile locale “nasconde”, all'interno di un blocco, una variabile con lo stesso nome definita nel blocco superiore.

```
package main
import "fmt"

func main(){
   n:=3

   if n>0{ //si riferisce a n=3
      n:=2
      fmt.Println(n) //Output: 2
   }

   fmt.Println(n) //Output: 3
   }
```

```
package main
import "fmt"

func main(){
   n:=3

   if n:=2;n>0{ //dichiarazione nuovo variabile, IL NUOVO CONTESTO LESSIC
ALE,
                        //PARTE DA IF, NON DALLA GRAFFA
      fmt.Println(n) //Output: 2
```

```
    }

    fmt.Println(n) //Output: 3
    }
```

Ordinamento in GoLang:

> 💡 **Advanced**: https://hackthedeveloper.com/how-to-sort-in-go/

Il package “sort”, mette a disposizione una serie di metodi ed interfacce per effettuare l’ordinamento di alcune strutture dati in GoLang.

```
import "sort"
```

Sort Ints:

```
func main() {
    // Unsorted Array
    s := []int{5, 8, 3, 2, 7, 1}
    sort.Ints(s)
    fmt.Println(s)
}
```

Controllare se un array è ordinato (in modo crescente)

```
func main() {
    // Sorted in Ascending order
    s := []int{1, 2, 3, 4, 5}
    fmt.Println(sort.IntsAreSorted(s))
```

```
    // Sorted in Descending order
    s = []int{5, 4, 3, 2, 1}
    fmt.Println(sort.IntsAreSorted(s))

    // Unsorted order
    s = []int{4, 2, 3, 1, 5}
    fmt.Println(sort.IntsAreSorted(s))
}
```

Binary search:

La funzione "`Search`", restituisce un **indice**, che può rappresentare l'indice dell'elemento **trovato**, oppure l'indice dove l'elemento cercato **dovrebbe essere inserito** (parlando di un array ordinato).

> 💡 Attenzione, la funzione richiede che l'array **sia ordinato** per lavorare correttamente.

Il valore di ritorno potrebbe anche essere uguale alla lunghezza dell'array, a indicare che il valore cercato è più grande di tutti quelli all'interno dell'array, e andrebbe inserito in ultima posizione.

Siccome la funzione restituisce sempre un valore, è necessario controllare se esso sta ad indicare la posizione dell'elemento cercato (found), o dove esso andrebbe inserito (not found):

```
num := 1
n = sort.SearchInts(a, num)
if n < len(a) && a[n] == num {
    //se il numero in quella posizione è davvero quello cercato, o è dove andreb
be inserito.
```

```
  // found
}
```

💡 Le altre funzioni sort.**Float64s**, sort.**Strings**, sort.**SearchFloat64s**, sort.**SearchStrings**, sort.**Float64sAreSorted** e sort.**StringsAreSorted**, sono equivalenti a quanto scritto.

For-Range syntax

The `range` form of the `for` loop iterates over a slice or map.

When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

```
package main

import "fmt"

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

func main() {
    for index, value := range mioArray {
        fmt.Printf("All'indice %d, c'è l'elemento %d\n", index, value)
    }
}
```

Array:

Sequenza di dati numerata, di dimensione fissa, e omogenea (dati tutti uguali).

La sua lunghezza deve essere costante e non negativa, definita a compile time (per un massimo di 2GB)

La sua dimensione non può essere modificata durante l'esecuzione.

```
var array [5]int //al momento della dichiarazione, di default ogni cella di memo
ria
//è riempita con lo zero-value del tipo dell'array (0 per int).
array := [5]int{}
//equivalente

La dimensione di array è 5, i suoi indici vanno da 0 a 4.

array := [5]int{1,2,3,4,5}
array := [...]int{1,2,3,4,5} //il compilatore conterà gli elementi,
                                                    //e creerà comunque un array di dimensione
5.

array := [5]int{3: 1, 4:2}
//assegno valori solo alle posizioni 3 e 4, quindi il mio array sarà così riempito:
0 0 0 1 2



//accedo e modifico ad un singolo elemento
//di un array tramite indici (indicizzazione è zero based)
array[0] = 2
Gli array sono mutabili.

Gli array possono essere multidimensionali:
var array [2][5]int // 2 array da 5 elementi. indicizzazione come in C.
```

Quando passati ad una funzione, sono passati per copia, e ciò potrebbe occupare molta memoria.

Per risolvere ciò, posso procedere in due modi:

- Passare un puntatore all'array (Funzionale, ma poco ideomatico)

```
x := Sum(&array)

func Sum(arr *[3]int) (sum int){
   for _,v := range arr{
      sum+=v
   }

   return
}
```

- Utilizzare uno slice.

Slices:

E’ un riferimento ad una sezione di un array.

Questa sezione può essere l’intero array o un subset.

E’ come se fosse un’interfaccia dinamica per l’utilizzo di un array “sottostante”.

La sua dimensione può variare a runtime.

C’è una differenza tra lunghezza e capacità dello slice:

- Lunghezza - numero di elementi all’interno
- Capacità - dimensione dell’array sottostante allo slice

La capacità dell’array è sempre maggiore o uguale alla lunghezza dello slice.

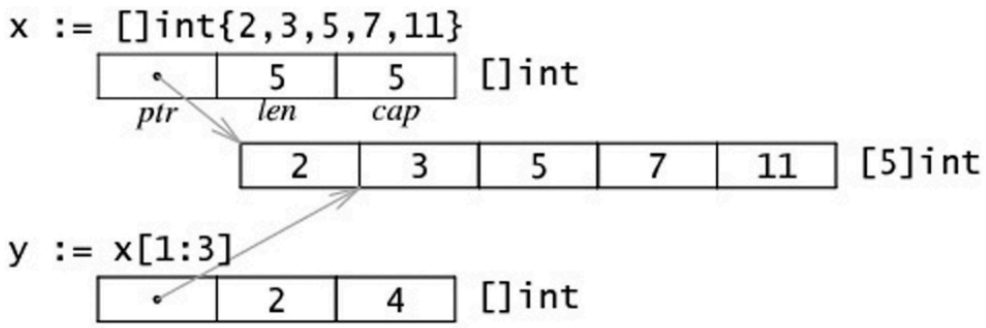
Due array diversi, rappresentano locazioni di memoria diverse.

Due slice invece, essendo riferimenti a locazioni di memoria, potrebbero riferirsi ad uno stesso array, e quindi avere degli elementi in comune.

```
slice := []int{1,2,3,4,5}
//crea un array SENZA NOME, e ne assegna il riferimento alla variabile "slice" (è un puntatore).
array := [5]int {1,2,3,4,5} //crea un array, alloca memoria e gli da un nome.

var slice2 []int = array[2:4]
//slice2 si riferisce ad una porzione di array, dalla posizione 2 alla posizione 4 (non inclusa)
slice2 = array[2:]//dalla posizione 2 fino alla fine.
slice2 = array[:3]//dalla posizione 0 alla posizione 3(non inclusa).

slice2 = slice2[0:5] //reslicing, ridimensiono il mio slice
                                //ad occupare tutta la capacità dell'array sottostante (5)

slice3 := &array //slice3 si riferisce alla totalità dell'array.
```

In memoria, uno slice è una struttura a 3 valori:

- Un Puntatore all’array sottostante
- La Lunghezza dello slice
- La Capacità dello slice



Sono passati per riferimento alle funzioni.

```
func Sum(slice []int) int{
    res:=0
    for _,v := range slice{
        sum+=v
    }
    return res
}

func main(){
    var arr = [5]int {1,2,3,4,5}
    Sum(arr[:])
    //creo uno slice temporaneo, equivalente all'array originale,
    //e lo passo alla funzione.
}
```

Append:

Siccome la lunghezza di uno slice è variabile, posso accodare degli elementi con l'utilizzo della funzione "append"

La funzione "**append**" è una **"variadic function"**, cioè una funzione che può essere chiamata con un qualsiasi numero di parametri.

Una variadic function, può essere chiamata specificando singoli parametri:

```
nums1 = append(nums1,2,3,4,6)
//aggiunge gli elementi '2','3','4','6' alla fine dell'array "nums1"
```

Oppure con la seguente notazione, se i più parametri da voler passare, sono contenuti all'interno di uno slice (array)

```
nums1 = append(nums1, nums2...)
//(notazione con tre puntini di sospensione)

s0 := []int{0, 0}
s1 := append(s0, 2)        // append a single element
                                        //s1 == []int{0, 0, 2}

s2 := append(s1, 3, 5, 7)  // append multiple elements
                                        //s2 == []int{0, 0, 2, 3, 5, 7}

s3 := append(s2, s0...)    // append a slice
                                        //s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}

sum(1,2,3)
sum(nums1...) //somma tutti gli elementi di uno slice (array)
```

Il Valore di ritorno di “append” è un nuovo slice, e potrebbe riferirsi allo stesso array (se la capacità è abbastanza per contenere anche i nuovi elementi) oppure ad un nuovo array (append crea un nuovo array, ci sposta gli elementi dell’array precedente, aggiunge gli altri elementi, e restituisce uno slice che si riferisce al nuovo array), pertanto può essere stampato o assegnato a una nuova variabile.

Utilizzi di “append”:

1) Append a slice b to an existing slice a:
```
a = append(a, b...)
```
2) Copy a slice a to a new slice b:
```
b = make([]T, len(a))
copy(b, a)
```
3) Delete item at index i:
```
a = append(a[:i], a[i+1:]...)
```
4) Cut from index i till j out of slice a:
```
a = append(a[:i], a[j:]...)
```
5) Extend slice a with a new slice of length j:
```
a = append(a, make([]T, j)...)
```
6) Insert item x at index i:
```
a = append(a[:i], append([]T{x},
       a[i:]...)...)
```
7) Insert a new slice of length j at index i:
```
a = append(a[:i], append(make([]T,
       j), a[i:]...)...)
```
8) Insert an existing slice b at index i:
```
a = append(a[:i], append(b,
       a[i:]...)...)
```
9) Pop highest element from stack:
```
x, a = a[len(a)-1], a[:len(a)-1]
```
10) Push an element x on a stack:
```
a = append(a, x)
```

Cos'è una stringa?

Una stringa in GoLang, è uno slice di bytes, che può contenere valori arbitrari.

In memoria, è una struttura a due valori:

- Un puntatore all'array di bytes sottostante
- La sua lunghezza.

Generalmente, è utilizzato per memorizzare sequenze di caratteri codificati secondo UTF-8, per cui ogni carattere, può assumere lunghezza variabile da 1 a 4 bytes.

Per questo motivo, le stringhe in GoLang sono dette "variable-width".

Le stringhe in GoLang non prevedono un carattere di terminazione, come lo '\0' in C per esempio.

Posso applicare gli operatori ==, !=, >=, <=, per effettuare una comparazione "lessicografica".

> More at: https://go.dev/blog/strings

```
var s string
s = "aaaabbbbccc\n" //stringa interpretata, le sequenze di escape sono interp
retate
                                    //(/n /r /t /u).
s = `stringaaa\n` //"Raw string", le sequenze di escape non sono interpretate.
                                  //backtick: AltGr + ?

Concatenazione di stringhe:
a:="Ciao"
b:="Mondo"

c = a+b

fmt.Println(c) //stampa "Ciao Mondo".
```

```
fmt.Println(s[0]) //stampa il carattere in posizione 0, 'a'.
```

Attenzione! L'Indicizzazione in una stringa,si riferisce ai byte che la compongono
(Per questo abbiamo detto che una stringa è uno slice di bytes),e non ai caratteri.
Quindi l'indicizzazione si traduce in un carattere solo in caso della rappresentazione ASCII.

```
fmt.Println([s[0:2]) //Anche alle stringhe posso applicare il subslicing.
                                    //estraggo una sottostringa che va
                                    //dal primo al secondo byte (escluso)
```

Attenzione! Anche l'estrazione di una sottostringa, lavora per **byte**, non per caratteri, quindi anche qui in caso la stringa contenesse caratteri unicode codificati su più caratteri, potrebbe capitare di estrarre porzioni errate di caratteri.

Immutable strings e bytes arrays:

Le stringhe in GoLang, sono immutabili, cioè è illegale fare ciò:

```
str = "ciao"
str[1]='b' //ILLEGALE.
```

Se avessi bisogno di cambiare il valore di una runa in una stringa, dovrei passare per un array di bytes:

```
str = "ciao"
arrBytes:=[]byte(str)

arrBytes[1]='b'
str = string(arrBytes)

fmt.Println(str) //Output: "cbao".
```

Loop through UTF-8 codepoints (Runes):

```
const nihongo = "日本語"
   for index, runeValue := range nihongo {
      fmt.Printf("%#U starts at byte position %d\n", runeValue, index)
   }
```

Loop through bytes:

```
for i := 0; i < len(sample); i++ {
}
```

Pacchetto “strings” e “strconv”:

```
package main

import (
    "fmt"
    "strconv"
    "strings"
)

func main() {
    str := "Questa è una stringa, termina con AA"
    fmt.Println(strings.HasPrefix(str, "AA")) //se i primi caratteri sono "AA"
    fmt.Println(strings.HasSuffix(str, "AA")) //se gli ultimi caratteri sono "AA"
    //Output: false,true.

    fmt.Println("La stringa, contiene la sottostringa \"termina\"?", strings.Contain
s(str, "termina")) //true

    fmt.Println("Indice della prima occorrenza del carattere 'è' : ", strings.Index
Rune(str, 'è')) //7
    fmt.Println(str[strings.IndexRune(str, 'è')])
    // NON STAMPA 'è', perchè la e accentata, NON E' ASCII.

    fmt.Println("Indice della prima occorrenza della stringa \"con\" : ", strings.In
dex(str, "con"))                //31
    fmt.Println("Indice della prima occorrenza della stringa o carattere \"con\" :
", strings.Index(str, "cavallo")) //-1, sottostringa non trovata.
```

```
    fmt.Println("Indice della prima e dell'ultima occorrenza del carattere \"a\" : ", strings.Index(str, "a"), strings.LastIndex(str, "a")) //5 , 29

    stringaNuova := strings.Replace(str, "termina", "finisce", 1)
    fmt.Println(stringaNuova) //"Questa è una stringa, finisce con AA."

    stringaNuova = "ciao cane,come va cane?"
    fmt.Println("numero di \"cane\" nella stringa: ", strings.Count(stringaNuova, "cane")) //2

    fmt.Println(strings.ToUpper(str), strings.ToLower(str)) //QUESTA È UNA STR INGA, TERMINA CON AA questa è una stringa, termina con aa

    //Trimming:
    fmt.Println(strings.TrimSpace(str)) //toglie tutti gli spazi PRIMA E DOPO IL P RIMO E L'ULTIMO CARATTERE "SIGNIFICATIVO".

    //Splitting:

    arrayStringhe := strings.Split(stringaNuova, ",") //divide la stringa in sottostri nghe, usa ',' come divisore.

    for i, v := range arrayStringhe {
        fmt.Println(i, v)
    }
    /*Output:
    0 ciao cane
    1 come va cane?
    */

    joinedString := strings.Join(arrayStringhe, "|") //unisce i vari elementi dell'ar ray, divise da un '|'.
    fmt.Println(joinedString)                    //ciao cane|come va cane?

    //strconv
```

```
    interoStringa := "143"
    fmt.Println(interoStringa) //"143"

    intero, _ := strconv.Atoi(interoStringa) //restituisce due valori, un intero, e un codice di eventuale errore di conversione.
    intero += 50

    interoStringa = strconv.Itoa(intero) //restituisce solo una stringa, la conversione va sempre a buon fine.
    fmt.Println(interoStringa)          //"193"

}
```

Funzioni

```
func stampa(s string,n int){
    for i:=0;i<n;i++{
        fmt.Println(s);
    }
}

func perdiv2(n int) (int,float64){
    return n*2, float64(n)/2 //funzione che prende un intero 'n', e restituisce due valori
}

func perdiv3 (n int) (c int){ //definisco una variabile di ritorno, che chiamo c
    c = n/3 //assegnamento, perchè la definizione l'ho già fatta a riga precedente
    return //posso scrivere solo "return", perchè la mia funzione già sa
                //di dover restituire la variabile 'c'
//Attenzione! La definizione di c come variabile di output, non mi impedisce di
//restituire un valore diverso da c.
    return 20 //legal.
```

```
}

func main(){
    stampa("ciao",4) //stampa "ciao", 4 volte.
    x,y := perdiv2(5) //assegnamento di due variabili
}
```

Switch case

```
package main

import (
    "fmt"
    "time"
)

func main() {

    x := 4
    y := 2

    switch x { //case valutati dall'alto verso il basso
    case 1:
        fmt.Println("x vale uno")
    case 2, 3, y * y:
        fmt.Println("x vale 2,3 o y*y")

        fallthrough
        //di default,il GO si comporta come se in C
        //avessi un break dopo ogni case, per far si che continui al case dopo,
        //scrivo "fallthrough".
        //ATTENZIONE FALLTHROUGH INDICA DI IGNORARE IL CONTROLLO DEL
CASE SUCCESSIVO,
        //E DI ESEGUIRLO CECAMENTE!
```

```
        //QUINDI IN QUESTO CASO, STAMPEREBBE ANCHE "x è uguale al valor
e.."
        //ANCHE SE NON E' VERO!!!
case funzione():
        fmt.Println("x è uguale al valore di ritorno della funzione")
    default:
        fmt.Println("default")
    }


    // uno switch senza condizione, corrisponde ad uno switch true, cioè ad un
a
    // forma compatta di un if-else.
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }

}
```

Cos’è un puntatore?

Metodo con cui il linguaggio di programmazione permette di manipolare la memoria.

Per ogni tipo di variabile, esiste un tipo di puntatore associato (*type).

```
var puntatore *int
//variabile che rappresenta un indirizzo di memoria di una variabile di tipo inte
```

```
ro.
```

Da un punto di vista astratto, sono riferimenti ad una variabile dello stesso tipo del puntatore.

Alla dichiarazione, il puntatore ha valore NIL cioè non punta a nessuna variabile.

Operatore “ampersand” &:

Passo dalle variabili ai loro indirizzi di memoria

La scrittura &nomeVariabile, restituisce un valore di tipo “puntatore a posizione di memoria di “nomeVariabile”.

Questo valore, può essere assegnato ad un puntatore.

```
var x int
var puntatore *int

puntatore = &x
```

Oppure con assegnamento breve:

```
var x int
pointer := &x //inferenza di tipo.
```

Operatore di “deferenziazione” *:

Passo dagli indirizzi di memoria ai valori.

Permette di ricavare il valore associato ad una specifica posizione di memoria.

Dato un puntatore, *puntatore restituisce il valore della variabile puntata dal puntatore.

```
var x int = 5

puntatore := &x
```

```
fmt.Print(*puntatore) //stampa 5.
```

A Livello sintattico, *puntatore corrisponde a ‘x’, pertanto posso anche assegnare valori alla variabile ‘x’ tramite i puntatori.

```
var x int

puntatore:=&x

*puntatore = 5

fmt.Print(x) //Stampa 5
```

Essendo “*puntatore” corrispondente a x, l’espressione “&*puntatore” corrisponde all’indirizzo di memoria di x.

Allo stesso modo, *&x, stampa a schermo il valore di x. (con & passo dalla variabile x al suo indirizzo di memoria, con * dall’indirizzo di memoria di nuovo al valore, praticamente vado avanti e indietro).

💡 Attenzione! La deferenziazione di un puntatore a valore “nil”, restituisce un errore di tipo “segmentation violation”.

```
var puntatore *int
fmt.Print(*puntatore)
```

Output:

panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x47ae76]

---

new()

La funzione new, permette di allocare memoria in modo dinamico, restituendo un puntatore ad essa.

```
p := new(int) //x è di tipo puntatore ad int.
*p = 5
fmt.Println(p) //indirizzo di memoria dell'area di memoria
fmt.Println(*p) //stampa 5
```

In caso perdessi il riferimento all'area di memoria appena creata, essa rimarrebbe "orfana" (dangling pointers), cioè no avrei più modo di accedervi, e sarà liberata, in un secondo momento non definito, in parallelo all'esecuzione del programma, dal garbage collector.

```
j:= new(*int) //**int

//essendo che new restituisce un puntatore, il valore restituito
//sarà **int
```

Esempio codice:

```
package main

import "fmt"

func incrementa(pointer *int) {
   *pointer++
}

func main() {
```

```
    var x int
    var puntatore *int

    puntatore = &x

    x = 5

    fmt.Println(*puntatore) //stampa 5

    *puntatore = 10

    fmt.Println(x) //stampa 10

    fmt.Println(&puntatore, puntatore, &*puntatore) //indirizzo di memoria dov'è
memorizzato il puntatore, indirizzo di memoria di x, indirizzo di memoria di x.

    incrementa(&x)
    fmt.Println(x) //Stampa 11

    var puntatoreDiPuntatore **int //riceve un valore di tipo "indirizzo di memori
a di un puntatore"

    puntatoreDiPuntatore = &puntatore
    **puntatoreDiPuntatore++ //il valore del valore del puntatore viene increme
ntato, quindi x++

    fmt.Println(**puntatoreDiPuntatore) //Stampa 12.

    var k, y int = 5, 70 //scambia il valore di due variabili.
    k, y = y, k
    fmt.Println(k, y)

    p := new(int) //x è di tipo puntatore ad int.
    *p = 5
    fmt.Println(p) //indirizzo di memoria dell'area di memoria
```

```
    fmt.Println(*p)
}
```

Mappe:

Array associativo, Dizionario, insieme di coppie chiave-valore.

Dichiarazione: `var nomeMappa map[tipoChiave]tipoValore`

Inizializzazione mappa vuota: `variabile := make(nomeMappa )`

Dichiarazione e inizializzazione: `var mappa map[string]int{"one":1, "two":2, "three":3}`

Aggiunta valori alla mappa : `mappa[chiave] = valore`

Passate a funzioni per riferimento.

Il tipoValore, potrebbe essere un array, per avere un dizionario chiave - valori.

Test presenza elemento:

Per testare la presenza di un elemento, utilizzo la notazione comma ok:

```
valore, ok = mappa[chiave]

ok sarà true or false, in base a se l'elemento mappa[chiave] esiste.

Se vogliamo verificare la sola presenza dell'elemento, senza memorizzarlo:

if _,ok = mappa[chiave]{

}
```

Eliminazione elemento:

```
delete(mappa, chiave) //elimina l'elemento in base alla chiave.
```

Ordinare una mappa:

Per effettuare il sort di una mappa, bisogna passare ad uno slice.

Copio tutte le chiavi della mappa in uno slice, ordino lo slice, e ricavo i valori corrispondenti alle chiavi ora ordinate:

```
var (
	barVal = map[string]int{“alpha”: 34, “bravo”: 56, “charlie”: 23,
	“delta”: 87, “echo”: 56, “foxtrot”: 12, “golf”: 34, “hotel”: 16,
	“indio”: 87, “juliet”: 65, “kilo”: 43, “lima”: 98}
)
func main() {
	fmt.Println(“unsorted:”)
	for k, v := range barVal { //stampa non ordinata

	   fmt.Printf(“Key: %v, Value: %v / “, k, v)
	}

	keys := make([]string, len(barVal)) //crea slice
	i := 0
	for k, _ := range barVal { //copia tutte le chiavi nello slice
	   keys[i] = k
	   i++
	}
	sort.Strings(keys)

	fmt.Println(“sorted:”)
	for _, k := range keys {
	   fmt.Printf(“Key: %v, Value: %v / “, k, barVal[k])
	   //stampa chiavi, e valori associati alle chiavi.
```

```
        }
    }
```

Output:

```
unsorted:
Key: indio, Value: 87 / Key: echo, Value: 56 / Key: juliet, Value: 65 / Key: charlie,
Value: 23 / Key: hotel, Value: 16 / Key: lima, Value: 98 / Key: bravo, Value: 56 / Key:
alpha, Value: 34 / Key: kilo, Value: 43 / Key: delta, Value: 87 / Key: golf, Value: 34 /
Key: foxtrot, Value: 12 /
sorted:
Key: alpha, Value: 34 / Key: bravo, Value: 56 / Key: charlie, Value: 23 / Key: delta,
Value: 87 / Key: echo, Value: 56 / Key: foxtrot, Value: 12 / Key: golf, Value: 34 / Key:
hotel, Value: 16 / Key: indio, Value: 87 / Key: juliet, Value: 65 / Key: kilo, Value: 43
/ Key: lima, Value: 98 /
```

> Se avessi bisogno di una mappa ordinata, sarebbe più efficiente utilizzare uno slice di structs, che posso ordinare col package sort:

```
type struct {
    key string
    value int
}
```

Keyword “type”:

In Go (Golang), ci sono due modi principali per dichiarare un nuovo tipo utilizzando la keyword `type`. La differenza tra `type nuovotipo int` e `type nuovotipo = int` è sottile, ma ha un impatto sulla facilità di manutenzione del codice e sull'interoperabilità con altri tipi.

1. **Dichiarazione di tipo senza assegnazione di valore (`type nuovotipo int`):**

   ```
   goCopy code
   type nuovotipo int
   ```

Con questa sintassi, stai dichiarando un nuovo tipo denominato `nuovotipo` che è equivalente al tipo sottostante `int`. Questo significa che `nuovotipo` e `int` sono due tipi distinti, anche se possono essere usati in modo intercambiabile. Ad esempio, se hai una funzione che accetta un `nuovotipo`, non accetterà un valore di tipo `int` anche se sono correlati.

2. **Dichiarazione di tipo con assegnazione di valore (`type nuovotipo = int`):**

```
goCopy code
type nuovotipo = int
```

Con questa sintassi, stai dichiarando un alias per il tipo `int` chiamato `nuovotipo`. In questo caso, `nuovotipo` e `int` sono considerati identici e possono essere usati in modo intercambiabile senza alcuna conversione. Le operazioni su `nuovotipo` sono le stesse di quelle su `int`.

Quindi, in breve:

- Con `type nuovotipo int`, stai creando un nuovo tipo distinto che richiede conversioni esplicite tra `nuovotipo` e `int`.
- Con `type nuovotipo = int`, stai creando un alias per `int`, rendendo i due tipi indistinguibili.

La scelta tra i due dipende dalle esigenze specifiche del tuo codice. Se vuoi un tipo distintivo, scegli la prima opzione. Se vuoi un alias che può essere utilizzato in modo intercambiabile con il tipo sottostante, scegli la seconda opzione.

Scanner

```
package main
```

```
import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "time"
)

func main() {
    scanner := bufio.NewScanner(os.Stdin) //nuovo scanner, che legge da uno stream di byte (un reader, in questo caso stdin)
    //scanner è una struttura che fornisce dei metodi per la lettura formattata.

    for scanner.Scan() { //legge un token (di default fino a newline), restituisce true finchè ha letto qualcosa, false quando incontra EOF (ctrl-d) o un errore.
        fmt.Println(scanner.Text())    //.Text restituisce il testo letto durante l'ultima chiamata scanner.Scan()
        if scanner.Text() == "basta" { //finisce di leggere
            break
        }
    }

    fmt.Println("ho finito")

    numeroACaso := rand.New(rand.NewSource(time.Now().UnixNano()))
    //creo un generatore di numeri pseudocasuali, a cui passo una sorgente di byte casuali, a cui passo un seme
    fmt.Println(numeroACaso.Intn(1000)) //numero a caso da 0 a 1000
}
```

Differenza tra var nomeslice []string e nomeslice:=make([]string,0)

Le tre dichiarazioni che hai fornito in Golang si differenziano nel modo in cui dichiarano e inizializzano uno slice:

1. `var nomeslice []string`

   Questa dichiarazione crea una variabile `nomeslice` di tipo slice di stringhe senza inizializzarla. Lo slice sarà inizializzato con il valore zero per uno slice, che è `nil`. Puoi successivamente assegnare uno slice già esistente o inizializzarlo con `make` o assegnare uno slice ottenuto da un altro metodo.

   ```
   goCopy code
   var nomeslice []string
   // nomeslice è ora nil
   ```

2. `nomeslice := make([]string, 0)`

   Questa dichiarazione crea uno slice di stringhe inizializzato con una capacità di 0 utilizzando la funzione `make`. Questo è utile se si prevede di aggiungere elementi allo slice successivamente. Inoltre, assicura che lo slice non sia `nil`, ma sia un'istanza valida di uno slice.

   ```
   nomeslice := make([]string, 0)
   // nomeslice è ora uno slice vuoto con capacità 0
   ```

In breve, la prima dichiarazione dichiara solo uno slice senza inizializzarlo. La seconda dichiarazione è scorretta senza l'inizializzazione appropriata, mentre la terza dichiarazione utilizza `make` per creare uno slice inizializzato con una capacità specifica.

Cosa si intende per capacità nella funzione make?

Sì, la "capacità" di uno slice in Golang si riferisce alla lunghezza del sottostante array che supporta lo slice. Quando utilizzi `make` per creare uno slice con capacità specifica, stai effettivamente creando uno slice associato a un array di quella capacità.

Esempio:

```
goCopy code
nomeslice := make([]string, 0, 5)
```

In questo caso, `nomeslice` è uno slice di stringhe inizializzato con una lunghezza di 0 e una capacità di 5. Significa che puoi accedere agli indici da 0 a 4, ma la lunghezza effettiva dell'array è 0 inizialmente.

Puoi assolutamente fare un `append` a uno slice con capacità specifica. Se la lunghezza dello slice raggiunge la capacità massima, lo slice verrà automaticamente riallocato con una capacità maggiore (di solito il doppio di quella precedente) e tutti gli elementi esistenti verranno copiati nel nuovo array.

Ad esempio:

```
nomeslice := make([]string, 0, 5)
nomeslice = append(nomeslice, "elemento1")
// La capacità rimane 5, la lunghezza diventa 1

nomeslice = append(nomeslice, "elemento2")
// La capacità rimane 5, la lunghezza diventa 2

nomeslice = append(nomeslice, "elemento3")
// La capacità rimane 5, la lunghezza diventa 3

nomeslice = append(nomeslice, "elemento4")
// La capacità rimane 5, la lunghezza diventa 4

nomeslice = append(nomeslice, "elemento5")
// La capacità rimane 5, la lunghezza diventa 5
```

```
nomeslice = append(nomeslice, "elemento6")
// La capacità viene automaticamente aumentata, la lunghezza diventa 6
```

Quindi, anche se hai specificato una capacità iniziale di 5, puoi aggiungere più di 5 elementi a causa del meccanismo di riallocazione automatica di Golang quando la capacità massima viene raggiunta.

Cosa cambia tra nomeslice := make([]string,5,5) e make([]string,0,5)?

Se scrivi `nomeslice := make([]string, 5, 5)`, stai creando uno slice con una lunghezza iniziale di 5 e una capacità di 5. Questo significa che all'inizio avrai uno slice con 5 elementi vuoti inizializzati al valore zero del tipo di elemento (nel caso di stringhe, sarebbero stringhe vuote `""`).

```
goCopy code
nomeslice := make([]string, 5, 5)
// nomeslice contiene ["", "", "", "", ""]
```

In questo caso, la lunghezza dello slice è 5 (poiché hai specificato 5 elementi iniziali), e la capacità è anche 5. Puoi accedere agli indici da 0 a 4 e anche aggiungere elementi fino alla capacità massima prima che venga eseguita una riallocazione.

In breve, la differenza tra `make([]string, 5, 5)` e `make([]string, 0, 5)` è che nel primo caso hai uno slice con lunghezza 5 e capacità 5, mentre nel secondo caso hai uno slice vuoto con capacità 5. La capacità determina quanto può crescere lo slice prima che venga automaticamente riallocato.

Se faccio var nomeslice []string, e poi cerco di fare nomeslice[0]="Ciao", ottengo errore? perche? e se facessi var nomeslice []string e poi nomeslice = append(nomeslice,"ciao")? funziona? e cosa succede in memoria?

Se dichiari uno slice senza inizializzarlo, come in `var nomeslice []string`, otterrai uno slice vuoto con un puntatore a un array di stringhe nullo. Se provi ad assegnare un valore a un indice specifico di questo slice, otterrai un errore di runtime noto come "panic" perché stai cercando di accedere a un'area di memoria non inizializzata.

Esempio con errore:

```
var nomeslice []string
nomeslice[0] = "Ciao"  // Questa riga causerà un panic
```

Invece, se utilizzi `append` su uno slice non inizializzato, funzionerà correttamente perché `append` si occupa automaticamente di allocare memoria e gestire il riallocamento se necessario.

Esempio con `append`:

```
var nomeslice []string
nomeslice = append(nomeslice, "Ciao")  // Questo funziona correttamente
```

In questo caso, `append` capirà che lo slice è vuoto e creerà un nuovo array con un elemento "Ciao" e aggiornerà lo slice in modo appropriato.

In memoria, la rappresentazione potrebbe apparire come segue:

Dopo `var nomeslice []string`:

```
plaintextCopy code
+----------------------+
| Puntatore (nil)      |
| Lunghezza (0)        |
| Capacità (0)         |
+----------------------+
```

Dopo `nomeslice = append(nomeslice, "Ciao")`:

```
plaintextCopy code
+----------------------+
| Puntatore (nuovo array)|
| Lunghezza (1)        |
| Capacità (nuova capacità)|
+----------------------+
```

In questa rappresentazione semplificata, il "nuovo array" rappresenta l'array appena allocato con l'elemento "Ciao", e "nuova capacità" rappresenta la capacità dell'array appena allocato.

E per quanto riguarda l'inizializzazione di una mappa? cosa cambia tra var mappa map[string]int e make(map[string]int)

La dichiarazione e l'inizializzazione di una mappa in Go possono essere fatte in due modi principali:

1. **Dichiarazione con `var`:**

```
var mappa map[string]int
```

In questo caso, `mappa` è una variabile di tipo mappa (`map[string]int`) dichiarata ma non inizializzata. La mappa è `nil` e non può essere utilizzata fino a quando non viene effettivamente inizializzata. Tentare di aggiungere elementi o accedere a elementi in una mappa non inizializzata causerà un errore di runtime.

2. **Inizializzazione con `make`:**

```
mappa := make(map[string]int)
```

Con `make`, stai inizializzando la mappa. La mappa sarà pronta per l'utilizzo e può essere immediatamente utilizzata per inserire, recuperare e manipolare dati.

Ecco un esempio pratico:

```
goCopy code
package main

import "fmt"

func main() {
    // Dichiarazione senza inizializzazione
    var mappa1 map[string]int
    // mappa1 è nil e non può essere utilizzata

    // Inizializzazione con make
    mappa2 := make(map[string]int)
    // mappa2 è pronta per essere utilizzata

    // Aggiungiamo alcuni valori a mappa2
```

```
    mappa2["uno"] = 1
    mappa2["due"] = 2
    mappa2["tre"] = 3

    // Stampa i valori di mappa2
    fmt.Println(mappa2) // Output: map[uno:1 due:2 tre:3]
}
```

In sintesi, con `var mappa map[string]int`, la mappa è dichiarata ma `nil` (non allocata), mentre con `make(map[string]int)`, la mappa è allocata e pronta per l'utilizzo. Utilizza `make` quando vuoi una mappa pronta per l'uso fin dall'inizio.

Interfaces

Un'interfaccia, è una collezione di prototipi metodi.
Una variabile di tipo interfaccia, deve rispettare (avere implementato) tali metodi.

```
type list interface{
    addNewNode(val int)
    printList()
    searchList(val int)(bool,*listNode)
    deleteIten(val int) bool
    len() int
}
```

Tutti questi metodi, ricevono implicitamente (come receiver), una variabile del tipo dell'interfaccia.

```
package main

import (
    "fmt"
    "strconv"
)
```

```
type Pippo interface { //definito sintatticamente come struct.
    pippo(int) int
    pluto() string
}

type TipoACazzo int //TipoACazzo soddisfa l'interfaccia Pippo,
//perchè ha un metodo pippo che riceve un int e restituisce un int,
//e ha un pluto che non riceve nulla e restituisce una stringa.

func (t TipoACazzo) pippo(x int) int {
    return x * 2
}

func (t TipoACazzo) pluto() string {
    return "ciao"
}

type TipoACazzoDue int

func (t TipoACazzoDue) pippo(x int) int {
    return x * 100
}

func (t TipoACazzoDue) pluto() string {
    return "cazzo"
}

func prova(i Pippo) { //riceve un qualsiasi tipo che rispetti l'interfaccia Pippo,
quindi so già che potrò usare i metodi pippo e pluto
    fmt.Println(i.pippo(6))
}

func (t TipoACazzo) String() string { //metodo "Stringer" del tipo "TipoACazz
o", cioè un metodo di stampa che "fmt" va a cercare e utilizza (se implementa
```

```
to).
    return "il Valore della variabile di tipo \"tipoacazzo\" è: " + strconv.Itoa(int(t))
}

func main() {
    variabile := TipoACazzo(2)
    variabileDue := TipoACazzoDue(5)

    fmt.Println(variabile)              //"fmt" la prima cosa che fa è cercare se il tipo implementa il metodo stringer, in questo caso viene trovato, viene chiamato, e fmt.Println ne stampa la stringa restituita.
    var str string = variabile.String() //utilizzo metodo stringer

    fmt.Println(str) //stampa la stessa cosa di linea 49

    prova(variabile)
    prova(variabileDue)

    var i Pippo
    //i è un puntatore che può contenere valori di tipo puntatore a "TipoACazz o" o "TipoACazzoDue", perchè può contenere puntatori a un qualsiasi tipo che rispetti l'interfaccia Pippo.
    i = TipoACazzo(4)
    fmt.Println(i.pippo(5)) //qui il GO deve fare "duck typing", cioè capire se in i c'è TipoACazzo o TipoACazzoDue, e capire quale "pippo" deve andare a chia mare.

    //a livello di memoria, un tipo Interfaccia, è una coppia di valori: un puntator e al valore associato (TipoACazzo(4), puntatore ad una variabile di quel tipo),
    //e un puntatore al tipo di quella variabile, un puntatore ad una itable, che as socia il nome dell'interfaccia, al tipo di dato che c'è dentro.

}
```

```
package main

import (
    "fmt"
    "math"
)

// Questa è una semplice interfaccia per le forme geometriche.
// sintatticamente definito come fosse una struct.
type geometry interface {
    area() float64
    perim() float64
}

// Per il nostro esempio implementeremo questa inferfaccia
// per le struct `rect` e `circle`.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}

// Per implementare un interfaccia in Go è sufficiente implementare
// tutti i metodi dell'interfaccia.
// Qui stiamo implementando l'interfaccia `geometry` per il tipo `rect`.
func (r rect) area() float64 { //è un metodo associato al tipo rect, cioè potrò far
e variabile.area(),si differenziano per avere il paramtro ricevuto prima del nom
e della funzione.
    return r.width * r.height
}
func (r rect) perim() float64 { //è un metodo associato al tipo rect, cioè potrò f
are variabile.perim(),si differenziano per avere il parametro ricevuto prima del
nome della funzione.
    return 2*r.width + 2*r.height
```

```
}

// ora rect implementa correttamente l'interfaccia `geometry`

// Qui invece l'implementazione per `circle`.
func (c circle) area() float64 { //è un metodo associato al tipo circle, cioè potr
ò fare variabile.area(),si differenziano per avere il paramtro ricevuto prima del
nome della funzione.
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 { //è un metodo associato al tipo circle, cioè pot
rò fare variabile.perim(),si differenziano per avere il paramtro ricevuto prima d
el nome della funzione.
    return 2 * math.Pi * c.radius
}

// ora circle implementa correttamente l'interfaccia `geometry`

// Se una variabile ha il tipo di un'interfaccia possiamo invocare i metodi
// dell'interfaccia stessa. Qui è presente una funzione `measure` generica
// che funzionerà su ogni variabile di tipo `geometry`.
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // Sia `circle` che `rect` implementano l'interfaccia
    // `geometry`, possiamo quindi passare alla funzione
    // `measure` istance di queste due struct.
    r.area()
    measure(r)
```

```
    measure(c)
}
```

Come fa GoLang a sapere però, data una variabile di tipo interfaccia, quale metodo richiamare?

Una variabile di tipo interfaccia, è memorizzata come una tupla di valori (value,type).

Controllando il tipo della variabile, GoLang sa quale metodo richiamare.

Type assertion

Permette di accedere al valore concreto della variabile di tipo interfaccia.

Esempio: avendo un'interfaccia che può avere valore di tipo stringa o float:

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string) //s contiene il valore stringa di i
    fmt.Println(s) //stampa "hello"

    s, ok := i.(string)
    fmt.Println(s, ok) //stampa "hello, true"

    f, ok := i.(float64) //comma ok, il valore di tipo float della variabile i non esist
e
    fmt.Println(f, ok) //Stampa "0, false"

    f = i.(float64) // panic! non ho controllato con sintassi comma ok
```

```
    fmt.Println(f)
}
```

Interface{} o any

Per generalizzare tipi, possiamo utilizzare la scrittura `interface{}` per indicare “un qualsiasi tipo” (ogni tipo primitivo infatti, implicitamente implementa un’interfaccia senza metodi)

```
type list interface{
    addNewNode(val interface{})
    printList()
    searchList(val interface{})(bool,*listNode)
    deleteItem(val interface{}) bool
    len() int
}
```

Alternativamente, per aumentare la leggibilità, posso usare la keyword `any`, alias di `interface{}`

```
type list interface{
    addNewNode(val any)
    printList()
    searchList(val any)(bool,*listNode)
    deleteItem(val any) bool
    len() int
}
```

Files:

Utilizziamo le interfacce Read e Writer, per la lettura e la scrittura su e da stream di byte, e l’interfaccia close, per la deallocazione del “file” sorgente di byte.

Qualsiasi cosa in grado di fornire dati è un file, un processo, una connessione http, in UNIX “everything is a file”. (anche stdin e stderr sono files)

Ogni file implementa un Reader, un Writer e un’interfaccia close.

Apertura file:

La funzione “os.Open”, restituisce un reader, uno stream di bytes, con sorgente file.

```
file, err := os.Open("nomeFile")
if err != nil {
    log.Fatal(err) //stampa errore
}
```

Lettura:

```
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    filePointer, err := os.Open(os.Args[1]) //acquisisco la risorsa di tipo file (APE
RTA IN SOLA LETTURA)
    if err != nil {
        log.Fatal(err) //stampa messaggio di logging, con formato standard che i
nclude anche data e ora.
        //log.Fatal o log.Panic, in base alla gravità dell'errore (Panic è più grave)
    }
    sliceSuCuiLeggere := make([]byte, 100000)
```

```
    num, err := filePointer.Read(sliceSuCuiLeggere) //legge su tutta la dimensio
ne dell'array di byte. (in questo caso legge 100 byte, per leggerne meno devo
fare subslicing)
    //restituisce numero di bytes letti, e un errore.

    fmt.Println(string(sliceSuCuiLeggere))
    fmt.Println("Ho letto ", num, " bytes, ho terminato la lettura con errore ", err)
    //se ho una slice gigante e leggo meno bytes della sua lunghezza, non è un
errore, semplicemente non riempirò tutta la mia slice.

    defer filePointer.Close() //dealloco l'apertura di un nuovo file al S.O



}
```

Per la lettura di file grandi, conviene operare per una lettura di tipo “buffered”, piuttosto che leggere N bytes tutto in RAM:

```
sliceSuCuiLeggere := make([]byte, 100000)

for{ //lettura bufferizzata ad N byte alla volta (10000)
    num,err:= filePointer.Read(sliceSuCuiLeggere)
    if num == 0 || err != nil{ //quando ho finito di leggere
                                        //(stringa vuota o errore di lettura)
        break
    }
}
```

```
file, _ := os.Open("largefile.txt")
scanner := bufio.NewScanner(file)
for scanner.Scan() { //lettura fino a newline
processLine(scanner.Text())
}
file.Close()
```

Scrittura:

```
package main

import (
    "log"
    "os"
)

func main() {
    filePointer, err := os.Create(os.Args[1]) //acquisisco la risorsa di tipo file (AP
ERTA IN SCRITTURA, CANCELLO CONTENUTO PRECEDENTE)
    if err != nil {
        log.Fatal(err) //stampa messaggio di logging, con formato standard che i
nclude anche data e ora.
        //log.Fatal o log.Panic, in base alla gravità dell'errore (Panic è più grave)
    }
    sliceDaScrivere := make([]byte, 100000)
    sliceDaScrivere = []byte("ciao a tutti, scrivo su fileee")

    _, err = filePointer.Write(sliceDaScrivere)
    if err != nil {
        log.Fatal(err)
    }
    //Oppure filePointer.WriteString("\nafammocc")

    defer filePointer.Close() //dealloco l'apertura di un nuovo file al S.O

}
```

Logging:

Attività di descrivere il funzionamento di un programma con interfaccia a riga di comando. (esempio descrivere l'esecuzione di un server web)

Il logging viene scritto all'interno di un'apposito sistema di log (di default, il sistema di log è stderr), al posto che su stdin.

Defer:

defer è una funzione che permette di postporre l'esecuzione di una funzione, a quando la funzione in la defer è eseguita termina.

defer significa "esegui questa funzione,quando esco dalla funzione in cui essa è eseguita".

Negli esempi sopra, il file viene chiuso quando il main finisce per un qualsiasi motivo, per fine "naturale", per panic, errore o altro.

Se ho più funzioni in "esecuzione differita", le funzioni vengono eseguite in ordine inverso all'ordine di comparsa nel codice, dall'ultima istruzione fino alla prima.

Ciò perchè potrei avere un file,e più risorse che dipendono dal file, e se chiudessi prima il file, le risorse che dipendono andrebbero in errore, quindi chiudo prima le risorse, poi il file.

Valutazione parametri:

```
package main

import "fmt"

func main() {
    x := 1

    defer func(y int) {
```

```
        //I PARAMETRI ALLA CHIAMATA FUNZIONALE,
        //SONO VALUTATI AL MOMENTO DELLA CHIAMATA A DEFER, anche se l
a funzione è
        //eseguita alla fine del main in questo caso.
        fmt.Println("La variabile x alla valutazione del defer vale ", y, "ma alla fine
vale ", x)
    }(x)

    //funzione anonima, che chiamo col parametro x
    //(tra parentesi sarebbe la chiamata)

    x = 2

    //OUTPUT: La variabile x alla valutazione del defer vale 1,ma alla fine vale 2
}
```

Recover from panic:

```
defer func() {
    r := recover();
    if r != nil { //sono in panico
        log.Println("Recovered from error:", r)
    }else{ //non sono in panico, posso scrivere altro codice.

    }
}()

panic("This is an error!")
```

La funzione differita viene invocata quando il mio codice genera un panico, passando un puntatore a un panico, che acquisisco con la funzione recover().

Funzioni anonime e variabili funzionali

```
package main

import . "fmt"

func pippo(x int, parametroFunzionale func(int) int) int {
    return parametroFunzionale(x)
}

func doppio(x int) int {
    return 2 * x
}

func main() {
    funzione := func(x int) int { //funzione anonima, viene dichiarato sul momen
to il contenuto della funzione
        return 2 * x
    }
    Println(funzione(10), doppio(10))                  //same output
    Println(pippo(20, func(x int) int { return x * x })) //funzione anonima dichiara
ta direttamente nel print
    Println(pippo(20, doppio))                         //funzione anonima dichiarata dir
ettamente nel print

    //Output: 20,20,400,40
}
```

Metodi associati a tipo:

```
package main

import (
```

```
    "fmt"
)

type Celsius float64
type Fahrenheit float64
type Reamur float64

func (c Celsius) toFahrenheit() Fahrenheit { //è un metodo associato al tipo Ce
lsius, cioè potrò fare variabile.ToFahrenheit(),si differenziano per avere il para
mtro ricevuto prima del nome della funzione.
    return Fahrenheit((float64(c) * 9 / 5) + 32) //conversione
}

func (f Fahrenheit) toCelsius() Celsius {
    return Celsius((float64(f) - 32) * 5 / 9)
}

func (r Reamur) toCelsius() Celsius { //due metodi con stesso nome, ma non
c'è ambiguità perchè ogni metodo è associato a un tipo diverso.
    return Celsius((float64(r) / 0.8))
}

func main() {
    var a Celsius = 35
    var b Fahrenheit = 100
    var c Reamur = 200

    converitoFahrenheit := a.toFahrenheit()
    converitoCelsius := b.toCelsius()
    convertitoReamurCelsius := c.toCelsius()

    fmt.Println("celsius: ", a, "fehrenheit: ", b, "reamur: ", c, "valore convertito in
fahrenheit: ", converitoFahrenheit, "valore convertito in celsius: ", converitoCel
sius, "valore in reamur convertito in celsius: ", convertitoReamurCelsius)
}
```

Metodi associati a tipo struct:

```
package main

import(
    "fmt"
)

type Persona struct{
    nome,cognome string
    età int
}

func (p *Persona) nomeCognome(nomeOCognome bool) string{ //passaggio per riferimento, ottimizzo memoria
    if nomeOCognome{
        return p.nome
    }else{
        return p.cognome
    }
}

func main(){
    strutturaPersona := Persona{"er","patata",18}
    puntatore := &strutturaPersona

    fmt.Println(strutturaPersona.nomeCognome(true)) //estrae l'indirizzo di memoria di strutturaPersona, e lo passo alla funzione.
    fmt.Println(puntatore.nomeCognome(false)) //ho già l'indirizzo di memoria, sono già pronto per passarlo alla funzione.
    //LE RIGHE SOPRA SONO EQUIVALENTI.

    //fmt.Println(Persona{"sebastiano","vigna",56}.nomeCognome())
    //NON FUNZIONA!! perchè voglio passare quella struct per riferimento, ma
```

```
quella struct in memoria non c'è, è un letterale, non ha un indirizzo di memori
a.
    //se la funzione ricevesse p Persona e non p *Persona, funzionerebbe anch
e coi letterali.


}
```