# Golang (autore: Vittorio Albertoni)

## Premessa

Il linguaggio di programmazione Go è stato inventato in casa Google per fare meglio e più facilmente tutto ciò che si può fare con i linguaggi C e C++.

Si tratta di un linguaggio particolarmente adatto alla realizzazione di software di sistema, per la gestione di server, per applicativi web ed altre cose altamente professionali.

Rispetto al C e al C++ è più facile da imparare, almeno se ci si limita alle prestazioni di base e se si lasciano le cose difficili ai professionisti.

Personalmente, pensando ai dilettanti cui sono solito dedicare i miei lavoretti, ritengo che non sia altrettanto facile di Python.

Rispetto a Python, che è un linguaggio interpretato, ha il grande vantaggio di consentirci di realizzare eseguibili stand alone, che, per funzionare, non hanno bisogno dell'installazione dell'interprete.

Probabilmente a causa dell'uso principale che ne viene fatto nessuno ha mai pensato di dotarlo della possibilità di creare programmi con interfaccia grafica per l'utente e non si presta alla programmazione per oggetti.

Qualcuno ha definito Go un linguaggio di scripting compilato. In effetti l'obiettivo dei suoi ideatori era quello di creare un linguaggio facile da imparare e da utilizzare come un linguaggio di scripting che avesse anche tutta la potenza dei linguaggi compilati.

In questo manualetto mi propongo di descrivere come Go possa essere utilizzato da un programmatore alle prime armi per fare cose non eccessivamente impegnative.

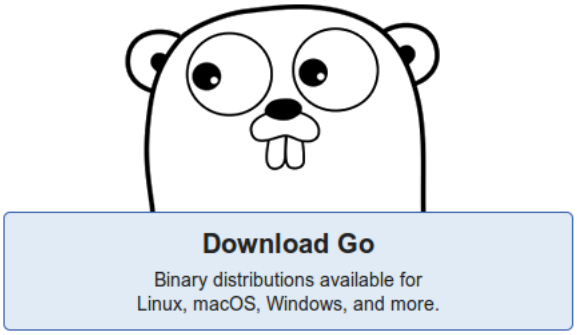
Per chi si appassioni e voglia fare di più, su *https://golang.org/*, nella sezione Documents, si trova tutta la documentazione necessaria a partire dai file The Go Programming Language Specification - The Go Programming Language.html e Packages - The Go Programming Language.html, che si possono considerare la bibbia del linguaggio Go.

Golang, immediatamente dopo il primo rilascio da parte di Google, è diventato software open source.

## Indice

# 1 Installazione

All'indirizzo *https://golang.org/*, cliccando sul pulsante




con la mascotte del Gopher, ci procuriamo l'installatore di Go adatto al nostro sistema operativo (Linux, Windows o Mac OS X). Chi usa Linux probabilmente si trova già installato il software (lo si può verificare digitando il comando `go` sul terminale) o lo può installare con il gestore dei programmi.

Troviamo anche tutta una serie di istruzioni per predisporre il workspace. Il rispetto di tutte queste istruzioni, di non banale esecuzione, servirebbe nel caso ci proponessimo un uso avanzato di Go, con la produzione di librerie riutilizzabili e importabili nei nostri programmi. Per quanto vedremo in questo manualetto è inutile che ci complichiamo la vita e possiamo benissimo lavorare nella solita directory Documenti della nostra area personale (magari potremmo creare, nella stessa directory Documenti o al suo fianco, una directory dedicata ai nostri lavori di programmazione, chiamandola, per esempio, Programmi e lavorare in questa).

# 2 Come funziona: il primo programma

Il linguaggio Go si basa su un insieme di file di testo, archiviati con estensione `.go`, chiamati Packages (si tratta praticamente di Librerie), che contengono elementi (variabili, costanti, tipi e funzioni) scritti secondo la sintassi del linguaggio stesso.

Il software di Go che abbiamo caricato sul computer contiene un sacco di packages con predisposte tantissime cose utili per fare i nostri programmi e, nel seguito, vedremo quelli che contengono le cose di utilizzo più ricorrente.

Anche il programma che facciamo noi deve essere scritto come package. Per essere compilato deve chiamarsi `main` e per essere eseguito deve contenere una funzione denominata `main`. Se il package avesse un nome diverso non verrebbe compilato e, se non contenesse una funzione `main`, non farebbe nulla.

Vediamo allora come dovremmo scrivere in linguaggio Go il famoso programmino Ciao mondo che qualsiasi manuale di programmazione porta come primo esempio.
Per scriverlo ci basta un qualsiasi editor di testo, tipo Gedit in Linux, TextEdit in Mac OS X e Blocco note in Windows (dobbiamo usare semplici editor di testo e non word processor).

```
package main
import "fmt"
func main() {
   fmt.Println("Ciao mondo")
}
```

Nella prima riga dichiariamo di voler scrivere un package chiamato main.
Nella seconda riga importiamo il package fmt, che contiene la funzione necessaria per scrivere qualche cosa nel terminale.

Infine scriviamo la funzione main all'interno della quale stabiliamo cosa debba fare il nostro programma: scrivere a terminale il saluto Ciao mondo.
Tra la parentesi graffa di apertura e quella di chiusura possono esserci quante istruzioni vogliamo: nel nostro caso ce n'è una sola, che richiama la funzione di scrittura dal package fmt che abbiamo importato e gli assegna la frase da scrivere.
La parentesi graffa di apertura del contenuto di una funzione deve essere scritta sulla stessa riga che contiene la definizione della funzione (se andassimo a capo il compilatore ci sgriderebbe).

Una volta scritto il programma lo salviamo nella nostra directory di lavoro attribuendo al file un nome sufficientemente descrittivo di ciò che fa il programma stesso e l'estensione `.go`: nel nostro caso chiamiamolo `ciao_mondo.go`.

Ora dobbiamo aprire il terminale del nostro sistema operativo: in Linux e Mac OS X si chiama Terminale e in Windows si chiama Prompt dei comandi e con il comando `cd` (sempre uguale in tutti i sistemi operativi) seguito dal nome della nostra directory di lavoro in cui abbiamo memorizzato il file del programma ci portiamo in questa directory.
Se vogliamo semplicemente vedere che cosa fa il nostro programma scriviamo il comando
`go run ciao_mondo.go`
e nella riga successiva vediamo comparire la scritta Ciao mondo.
Se vogliamo creare l'eseguibile dobbiamo scrivere il comando
`go build ciao_mondo.go`
e nella nostra directory di lavoro vedremo comparire il file eseguibile (in Linux e Mac OS X si chiamerà `ciao_mondo` e in Windows `ciao_mondo.exe`).

L'eseguibile potrà essere lanciato, sempre da terminale, scrivendo il suo nome preceduto da `./` in Linux e Mac OS X e scrivendo semplicemente il suo nome senza l'estensione `.exe` in Windows[1].

Il nostro lavoro può essere semplificato se utilizziamo editor di testo fatti apposta per la programmazione, che contengono anche pulsanti per compilare il programma e per lanciarlo.
Nel software libero abbiamo l'ottimo Geany, che consiglio per la leggerezza e la facilità di uso. Lo troviamo su *https://www.geany.org/download*, nelle versioni per Linux, Mac OS X e Windows. Al solito chi usa Linux, se non se lo trova già installato insieme al sistema operativo, lo può installare con il gestore dei programmi.

### Cross compilation

In genere per avere un eseguibile che funzioni in un sistema operativo occorre compilarlo in quel sistema operativo: ciò avviene per tutti i linguaggi compilati che conosco (dal C al Pascal, da ADA al C++).

Con il linguaggio Go è invece possibile generare eseguibili anche per sistemi operativi diversi da quello sul quale lavoriamo. Come dire che con un Macbook con sistema operativo OS X o con una macchina equipaggiata Linux possiamo creare un eseguibile per Windows e, ovviamente, viceversa.

Per ottenere questo meraviglioso risultato basta procedere alla compilazione con il comando
`GOOS=<sistema> GOARCH=<architettura> go build -o <nome_eseguibile> <nome_source>`
dove
. al posto di <sistema> scriviamo `darwin`, `linux`, `windows` a seconda, rispettivamente, se vogliamo ottenere un eseguibile per OS X, per Linux o per Windows;
. al posto di <architettura> scriviamo 386 o amd64 a seconda, rispettivamente, se vogliamo ottenere un eseguibile per l'architettura a 32 o a 64 bit;
. al posto di <nome_eseguibile> scriviamo il nome che vogliamo dare al file eseguibile;
. al posto di <nome_source> scriviamo il nome del file .go in cui abbiamo scritto il programma.

[1] In Windows gli eseguibili si possono lanciare anche facendo doppio click sul file. Se facciamo questo con il programma ciao_mondo.exe che abbiamo appena compilato vedremo comparire per un solo attimo la finestra del prompt dei comandi senza riuscire a leggerne il contenuto. Per evitare questo inconveniente esistono facili trucchetti nei linguaggi compilati C, C++ e Pascal ma non sono purtroppo riuscito a trovarne uno altrettanto facile in Go.

Sicché, se siamo su Linux e vogliamo creare un eseguibile per Windows del nostro programma ciao_mondo dobbiamo scrivere

```
GOOS=windows GOARCH=amd64 go build -o ciao_mondo.exe ciao_mondo.go
```

e nella nostra directory di lavoro troveremo il file `ciao_mondo.exe` pronto per un sistema Windows a 64 bit.

La cross compilation di Go può riguardare anche altri sistemi operativi (Solaris, FreeBSD, ecc.) e altre architetture: mi sono limitato ai casi più ricorrenti.

# 3 Tipi base

Tutti i valori su cui agisce un programma Go devono avere un tipo. Il tipo assegnato ad un valore identifica la natura di quel valore, le operazioni che si possono effettuare su di esso e le modalità con le quali esso viene inserito nella memoria del computer.

Il linguaggio Go, come tutti i linguaggi compilati, è a tipizzazione statica, cioè richiede che il tipo di ciascun valore venga stabilito nel codice sorgente[2].

In questo capitolo vediamo i tipi di uso più comune, i così detti tipi base.

## Tipi numerici

I tipi che possiamo assegnare ad un valore numerico sono i seguenti.

### Intero

Per i numeri interi, quelli che non hanno decimali, Go prevede una miriade di possibilità di definizione (con segno, senza segno, a 8, 16, 32, 64 bit), il tutto con l'evidente scopo di dosare l'utilizzo della memoria durante le elaborazioni a ciò che effettivamente serve. Salvo accorgersi che con queste limitazioni non si riesce a fare i calcoli richiesti dalla crittografia per la blockchain: allora, con un apposito package (chiamato math/big) si è creato un tipo particolare per i grandi numeri, il tipo big.Int (che, con la I maiuscola indica il tipo intero grande del pacchetto math/big).

Il tutto è di applicazione tutt'altro che banale e il neofita è bene si limiti ad utilizzare semplicemente il tipo `int` (con la i minuscola) sapendo che si tratta di un tipo con segno, di precisione legata all'architettura della macchina, precisione che normalmente arriva si e no alle 19 cifre[3].

### Decimale

I decimali sono i numeri reali, in informatica numeri a virgola mobile (floating point numbers) e sono quelli che contengono una parte decimale.

Possiamo scegliere tra `float32` (a precisione singola) e `float64` (a precisione doppia).

Per i programmini che può fare un neofita, dove non dovrebbe preoccupare l'impiego di memoria, tanto vale usare sempre il `float64`, rammentando che la precisione, anche doppia, può crollare quando, tra parte intera e parte decimale, superiamo le fatidiche 19 cifre.

### Complesso

Un numero complesso è composto da una parte reale e da una parte immaginaria.

Per quanto possa servire Go prevede la coppia di tipi `complex64` (precisione singola) e `complex128` (precisione doppia).

[2]Nei linguaggi interpretati, soprattutto nei linguaggi di scripting (come Python, Javascript, Perl, Ruby, ecc.), la tipizzazione è dinamica, cioè può avvenire anche durante l'esecuzione del programma.

[3]Sul sistema Linux a 64 bit su cui sto lavorando su un portatile di media tacca arrivo a calcolare esattamente il fattoriale di 20, che è un numero di 19 cifre. Per il fattoriale di numeri superiori di poco escono numeri negativi senza senso e, per numeri ancora superiori esce il risultato 0).

* * *

Con i tipi numerici possiamo compiere tutte le operazioni aritmetiche:
. somma attraverso l'operatore +
. sottrazione attraverso l'operatore -
. prodotto attraverso l'operatore *
. divisione attraverso l'operatore /
. resto intero attraverso l'operatore %
senza ricorrere ad alcun package.

I valori tipizzati sottoposti a queste operazioni devono avere lo stesso tipo e il risultato è un valore di questo stesso tipo. Pertanto, per quanto riguarda la divisione, occorre ricordare che se sia il dividendo sia il divisore sono interi il risultato è un intero e l'eventuale parte decimale si perde.

I tipi assegnati possono essere modificati con il casting, di cui parleremo quando ci occuperemo di variabili.

Tutto questo discorso vale, ovviamente, quando assegniamo un tipo ad un valore da memorizzare per essere riutilizzato nel corso del programma.

Possiamo utilizzare numeri interi e numeri decimali con gli operatori visti sopra in espressioni finalizzate alla sola scrittura di un altro numero, come elemento letterale.
Se scriviamo `fmt.Println(3.5)` otteniamo la stampa di 3.5.
Se scriviamo `fmt.Println(7.0/2)` otteniamo sempre la stampa di 3.5.
Se scriviamo `fmt.Println(1+2.5)` otteniamo ancora la stampa di 3.5.
Negli ultimi due casi, infatti, 7.0/2 e 1+2.5 erano soltanto modi diversi di scrivere il numero 3.5.
Notare come, per la divisione, almeno uno degli operandi sia stato scritto come numero decimale (7.0 e non 7) in modo da evitare la divisione tra interi (7/2 avrebbe dato il risultato 3).

### Tipo stringa

La stringa è una successione di caratteri di lunghezza fissa usata per rappresentare testo e corrisponde all'identificativo `string`.

Il suo contenuto si crea scrivendo i caratteri all'interno di una coppia di doppi apici " o di una coppia di apici rovesciati ‘. Mentre la stringa tra apici rovesciati è sensibile al new line generato con il tasto INVIO, in quella tra doppi apici è possibile inserire il new line solo usando la sequenza di escape `\n`.

* * *

Più stringhe possono essere sommate attraverso l'operatore + e il risultato è una nuova stringa contenente i caratteri delle stringhe di partenza in successione senza spazi intermedi (questi, se si vogliono nel risultato, vanno precedentemente inseriti come stringhe vuote o come caratteri vuoti alla fine delle stringhe interessate alla somma).

Possiamo contare i caratteri contenuti in una stringa utilizzando la funzione `len()`, che ritorna un numero intero corrispondente al numero di caratteri contenuti nella stringa, contando anche gli eventuali spazi bianchi.
Se scriviamo `fmt.Println(len("Vittorio"))` otteniamo la stampa del numero 8.

La posizione di ogni carattere è indicizzata partendo con l'indice 0 per il primo carattere a sinistra. Se alla stringa facciamo seguire l'indice tra parentesi quadre viene ritornato un numero corrispondente al codice ASCII del carattere corrispondente all'indice. Trasformando questo numero in stringa, con il casting, otteniamo la lettera.
Se scriviamo `fmt.Println("Vittorio"[3])` otteniamo la stampa del numero 111 (carattere ASCII corrispondente alla lettera t minuscola, il quarto carattere partendo da indice 0 della stringa Vittorio).

Se scriviamo `fmt.Println(string("Vittorio"[3]))` otteniamo la stampa della corrispondente lettera t minuscola.

### Tipo booleano

Il tipo booleano è uno speciale intero di un bit che rappresenta i valori `true` (1) e `false` (0).

* * *

Con i valori booleani possiamo utilizzare i tre operatori logici
`&&` che sta per `and`,
`||` che sta per `or`,
`!` che sta per `not`.

Se scriviamo `fmt.Println(true && true)` otteniamo la stampa di `true`,
Se scriviamo `fmt.Println(true && false)` otteniamo la stampa di `false`,
Se scriviamo `fmt.Println(true || true)` otteniamo la stampa di `true`,
Se scriviamo `fmt.Println(true || false)` otteniamo la stampa di `true`,
Se scriviamo `fmt.Println(!true)` otteniamo la stampa di `false`.

## 4 Variabili e costanti

Le variabili e le costanti sono delle locazioni di memoria, delle scatole, alle quali diamo un nome, destinate a contenere valori di un certo tipo.

Negli esempi che abbiamo visto finora (il primo programma ciao_mondo visto nel Capitolo 2 e l'uso della funzione Println per stampare altre cose viste nel Capitolo 3) davamo in pasto alla funzione Println dei così detti elementi letterali, anche in forma di espressioni numeriche, volanti, il tutto finalizzato, appunto, semplicemente alla loro stampa.

Se ci limitassimo all'uso di elementi letterali non andremmo lontano con i nostri programmi.

In un programma che debba fare qualche cosa in più di quattro conticini, che debba sviluppare algoritmi complessi nel corso dei quali siano da prendere decisioni su cosa fare in presenza di un valore piuttosto che di un altro è necessario tenere a disposizione i valori, poterli modificare e richiamare quando serve. A questo servono le variabili e le costanti.

### Variabili

La variabile è una locazione di memoria destinata a contenere un valore modificabile nel corso del programma.

Essa va creata dichiarandone il nome e dichiarando il tipo di valore a lei destinato, eventualmente indicando anche un valore iniziale.

La sintassi per la dichiarazione di una variabile è
`var` <nome> <tipo>
nel caso la si voglia anche inizializzare
`var` <nome> <tipo> `=` <valore>.
Se scriviamo `var x int` creiamo la variabile denominata x destinata a contenere un intero.
Se scriviamo `var raggio float64 = 7.56` creiamo la variabile raggio (per esempio di un cerchio) destinata a contenere un numero decimale e le assegniamo il valore iniziale 7, 56 (attenzione alla virgola, che in Go è un punto).
Se scriviamo `var saluto string = "Ciao"` creiamo la variabile saluto e le assegniamo il valore iniziale Ciao.

Per i nomi possiamo utilizzare qualsiasi parola che non sia riservata al linguaggio[4].

[4]Le parole riservate al linguaggio sono `break default func interface select case defer go map struct chan else goto package switch const fallthrough if range type continue for import return var`.

Per il tipo possiamo indicare uno di quelli visti nel Capitolo 3.

Il valore può essere un numero, una espressione numerica, più o meno includente altre variabili, o una stringa (in tal caso sarà una successione di caratteri racchiusa tra apici).

Se lavoriamo all'interno di una funzione (`func`) abbiamo una possibilità stenografica di dichiarare e inizializzare una variabile utilizzando l'operatore := in luogo dell'operatore di assegnazione =.
In questo caso la sintassi per la dichiarazione con inizializzazione diventa
<nome> := <valore>
e il tipo viene automaticamente assegnato a seconda della natura del letterale indicativo del valore (se si tratta di un numero senza decimali viene assegnato il tipo int, se con decimali il tipo float, se tra apici il tipo string).

Trattandosi di variabili, non solo possiamo modificarne il valore ma anche il tipo, attraverso un accorgimento chiamato casting.
Per fare questo dobbiamo scrivere l'identificativo del tipo che vogliamo riassegnare alla variabile seguito, tra parentesi tonde, dal nome della variabile stessa.
Rammentiamo che la riassegnazione del tipo int a una variabile float comporta la perdita della parte decimale del numero.
Sicché, avendo una variabile x, di tipo float64, con valore 7.65, se scriviamo `var y int = int(x)` o semplicemente `y := int(x)` creiamo una variabile y di tipo int contenente il valore numerico 7.
Così come se scriviamo `fmt.Println(int(x))` otteniamo la stampa del numero 7.

Il programmino ciao_mondo che abbiamo scritto nel Capitolo 2, se vi introduciamo l'uso di una variabile, diventa

```
package main
import "fmt"
var saluto string = "Ciao mondo"
func main() {
   fmt.Println(saluto)
}
```

Come si vede, abbiamo innanzi tutto creato una variabile stringa che contiene la frase Ciao mondo e poi, nella funzione main, abbiamo detto al metodo Println di stampare quella variabile richiamandone il nome.
Il risultato è lo stesso che si ottiene con la scrittura del programma senza l'uso della variabile, ma ora abbiamo realizzato il vantaggio di avere nel programma una variabile utilizzabile in altre parti del programma e in altre funzioni semplicemente richiamandone il nome.
Per quanto detto prima, dal momento che la variabile è stata creata non all'interno di una funzione, in questa posizione non avremmo potuto crearla con la scrittura abbreviata `saluto := "Ciao mondo"`, come invece potremmo fare se creassimo la variabile all'interno della funzione main.

A proposito della posizione in cui creiamo la variabile è necessaria una precisazione circa la sua visibilità (nell'inglese informatico *scope*).
Nell'esempio sopra abbiamo creato la variabile fuori dalla funzione main.
In questo caso la variabile è visibile anche per essere utilizzata in altre funzioni dovessimo inserire nel nostro programma: come si suol dire, abbiamo creato una variabile globale.
Se avessimo creato la stessa variabile, questa volta potendo usare la scrittura abbreviata, all'interno della funzione main, la variabile sarebbe visibile ed utilizzabile solo all'interno della funzione: come si suol dire, avremmo creato una variabile locale.

### Costanti

La costante è praticamente una variabile il cui contenuto non può cambiare.

Essa va creata dichiarandone nome, tipo di valore a lei destinato e valore.

La sintassi per la dichiarazione di una costante è
`const` <nome> <tipo> = <valore>

ove, per nome, tipo e valore vale quanto detto per le variabili nel precedente paragrafo.

Dal momento che parliamo di una cosa che non può cambiare, per le costanti non è ammessa la possibilità di modificarne il tipo con il casting.

Per la visibilità vale quanto detto per le variabili.

# 5 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

### Operatori aritmetici

Li abbiamo già visti parlando dei tipi numerici.

Mi limito a citare quelli di uso più comune per i principianti.

`+` somma, si applica a valori interi, float, complessi e stringhe,
`-` differenza, si applica a valori interi, float e complessi,
`*` prodotto, si applica a valori interi, float e complessi,
`/` divisione, si applica a valori interi, float e complessi,
`%` resto intero, si applica a valori interi.

Già abbiamo detto, trattando dei tipi e delle variabili, dei problemi circa l'utilizzo di questi operatori in relazione al tipo degli operandi.

Come si vede non c'è un semplice operatore per l'elevamento a potenza, operazione per la quale dobbiamo trovare altrove lo strumento.

### Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano.

Sono i seguenti.

`==` uguale,
`!=` non uguale,
`<` minore,
`<=` minore o uguale,
`>` maggiore,
`>=` maggiore o uguale.

Gli operandi assoggettati al confronto devono avere lo stesso tipo, che deve essere int, float o string.

### Operatori logici

Li abbiamo già visti parlando del tipo booleano, il risultato che forniscono è un valore booleano e sono i seguenti.

`&&` che sta per `and`,
`||` che sta per `or`,
`!` che sta per `not`.

# 6 Funzioni

La funzione, in altri contesti detta anche procedura o subroutine, è una sezione del programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Nel Capitolo 2 abbiamo visto che in un programma Go deve esistere almeno una funzione, la funzione main: è la funzione che viene automaticamente chiamata al lancio del programma.

Né basta questa sola funzione per far fare qualche cosa al programma: semplicemente per ottenere che il nostro programma scriva un salutino a terminale abbiamo dovuto chiamare un'altra funzione, la funzione Println del package fmt.

Le istruzioni contenute nella funzione, salvo che noi stessi la programmiamo, ci sono ignote: sappiamo solo che, chiamando la funzione e inserendo gli eventuali parametri richiesti, otteniamo un certo risultato. Come di fronte a una scatola nera.
Per scrivere Ciao mondo abbiamo chiamato la funzione Println, le abbiamo passato il parametro (la stringa "Ciao mondo") e ci siamo trovata la scritta a terminale, senza sapere minimamente quali istruzioni siano contenute nella funzione Println per ottenere questo risultato.

Il linguaggio Go ci mette a disposizione tantissime funzioni, raggruppate nei packages e, nel prossimo Capitolo, passeremo in rassegna i packages di utilizzo più ricorrente per conoscere le funzioni che contengono.

Potremmo noi stessi scrivere funzioni e inserirle in packages in modo da averle a disposizione quando serve, ma nell'economia di questo manualetto per principianti direi che non è il caso di complicarci la vita per fare cose da professionisti.

Tuttavia, per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per dichiarare e scrivere una funzione è
`func` <nome> `(`<nome_e_tipo_parametri`)` <tipo_ritorno> `{`
<istruzioni>
`}`
dove <nome> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla, <nome_e_tipo_parametri> è il nome e il tipo del o dei parametri (dati) da inserire quando la richiamiamo, <tipo_ritorno> è il tipo del risultato che intendiamo ottenere. Le <istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.

La sintassi per chiamare una funzione ed eseguirla è
<nome_funzione> `(`<parametri>`)`
dove, se la funzione appartiene ad un package importato, <nome_funzione> si indica scrivendo il nome del pacchetto e il nome della funzione divisi da un punto.

Come esercizio esemplificativo supponiamo di voler scrivere un programma che calcoli Combinazioni e Disposizioni, semplici senza ripetizione, di n numeri presi k a k.
Le formule che dobbiamo utilizzare sono
$C(n,k) = \frac{n!}{(n-k)!k!}$ per le combinazioni,
$D(n,k) = \frac{n!}{(n-k)!}$ per le disposizioni,
nelle quali ricorre spesso il calcolo del fattoriale.
Il fatto che la necessità di questo calcolo ricorra spesso e comporti una non semplice scrittura di istruzioni per effettuarlo suggerisce di isolare queste istruzioni in una funzione dedicata al calcolo del fattoriale in modo da poter richiamare queste istruzioni in blocco quando serve.
La funzione per calcolare il fattoriale può essere scritta così:

```
func fattoriale (n int) int {
    if n == 0 {
       return 1
    }
    return n * fattoriale (n - 1)
}
```

E' un bell'esempio di formula ricorsiva (funzione che richiama sé stessa) che ci dimostra come il linguaggio Go supporti la ricorsività.
Per scrivere questa funzione sono ricorso all'istruzione `if` che il lettore ancora non conosce. E' stato necessario per inserire nella funzione la convenzione per cui il fattoriale di 0 è 1.
Il programma per calcolare combinazioni e disposizioni di 10 numeri presi 3 a 3 diventa il seguente:

```
package main
import "fmt"
func fattoriale (n int) int {
    if n == 0 {
        return 1
    }
    return n * fattoriale (n-1)
}
func main() {
    var n int = 10
    var k int = 3
    var c int
    var d int
    c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
    d = fattoriale(n)/fattoriale(n-k)
    fmt.Println("combinazioni:  ", c)
    fmt.Println("disposizioni:  ", d)
}
```

Salviamo il nostro programma nel file calcolo_combinatorio.go e lo compiliamo.
All'esecuzione del programma verrà scritto sul terminale il risultato in questi termini:

```
combinazioni:  120
disposizioni:  720
```

# 7 Packages

Il package è una raccolta di funzioni, una libreria.

Il linguaggio Go ce ne mette a disposizione parecchi, anche per fare cose molto professionali.

Qui mi limito a richiamare quelli di uso più ricorrente in programmi semplici alla portata di un dilettante, limitandomi altresì alle funzioni più ricorrenti.

Il panorama completo possiamo averlo consultando il file Packages - The Go Programming Language.html che troviamo all'indirizzo *https://golang.org/*, nella sezione Documents.

Per utilizzare nei nostri programmi funzioni contenute in un package dobbiamo importarlo con l'istruzione

`import "<nome_package>"`

Se dobbiamo usare più package possiamo importarli con l'istruzione

```
import (
    "<nome_package>"
    "<nome_package>"
    ....
)
```

Le funzioni si chiamano con la sintassi

<nome_package>.<nome_funzione>(<parametri_richiesti>)

Tutto ciò l'abbiamo già sperimentato negli esempi fin qui visti usando il package `fmt` e la sua funzione `Println`.

### Package fmt

Contiene funzioni per l'input e l'output formattati.

Per le nostre esigenze da novizi ci basta conoscere tre delle oltre venti funzioni che contiene.

**Println**

Questa funzione scrive l'output usando una formattazione di default, accostando gli elementi da scrivere separati da uno spazio e nell'ordine in cui sono inseriti come parametri separati da una virgola.

Gli elementi da scrivere, che passiamo come parametri, possono essere stringhe (parole o frasi racchiuse tra doppi apici), letterali numerici o espressioni numeriche (e ne verrà scritto il risultato), nomi di variabili (e ne verrà scritto il valore).

Println inserisce automaticamente un new line e, dopo avere scritto, va a capo.

Esempio:

Supponendo di avere una variabile a contenente il valore 16.5, l'istruzione

```
fmt.Println("3 moltiplicato 4 fa", 3*4, "e la variabile a vale", a)
```

scrive a terminale quanto segue

```
3 moltiplicato 4 fa 12 e la variabile a vale 16.5
```

**Printf**

Scrive l'output formattandolo secondo le indicazioni fornite attraverso le così dette direttive di formattazione, che funzionano come dei segnaposto nella stringa che contiene quanto va scritto.

Le direttive di formattazione si aprono con il simbolo % cui fanno seguito, per citare quelli di più ricorrente uso, i seguenti altri simboli:

`s` ad indicare l'inserimento di una stringa,

`d` ad indicare l'inserimento di un numero intero non particolarmente formattato,

`f` ad indicare l'inserimento di un numero decimale non particolarmente formattato,

`<n>d` con n numero intero, ad indicare l'inserimento di un intero allineato a destra in un campo di n caratteri,

`.<n>f` con n numero intero, ad indicare l'inserimento di un decimale con una parte decimale di n caratteri (l'ultimo dei quali arrotondato).

Printf non inserisce il new line e, per andare a capo, occorre terminare la stringa con il carattere di escape `\n`.

Alla fine della stringa, dopo una virgola, si elencano i letterali numerici o le espressioni numeriche (e ne verrà scritto il risultato) o i nomi di variabili (e ne verrà scritto il valore) che dovranno occupare i segnaposto, ponendo il tutto nell'ordine dei segnaposto stessi.

Esempio:

Supponiamo di avere tre variabili: a intera 78, b intera 3456 e c decimale 786,89567.

Se scriviamo le seguenti istruzioni

```
fmt.Printf("2 moltiplicato 3.5 fa %.2f e la variabile c vale %.3f\n", 2*3.5, c)
fmt.Printf("la variabile a vale%6d\nla variabile b vale%6d\n", a, b)
```

otteniamo a terminale quanto segue

```
2 moltiplicato 3.5 fa 7.00 e la variabile c vale 786.896
la variabile a vale    78
la variabile b vale  3456
```

Ovviamente, se non abbiamo necessità di formattazione, tanto vale usare la più semplice funzione Println.

**Scan**

Legge un input da tastiera e lo assegna a una variabile precedentemente dichiarata.

La sintassi è

```
fmt.Scan(&<variabile>)
```

E' molto importante far precedere al nome della variabile ove memorizzare l'input l'operatore di indirizzamento `&`. Se omettiamo questo operatore chissà dove finisce il nostro input.

Altra cosa importante da tener presente è che la funzione Scan, come tutte le sue varianti contenute nel pacchetto fmt, non scandisce lo spazio vuoto: ciò significa che l'input deve essere costituito da una sola parola o da un solo numero. L'inserimento di uno spazio nell'input provoca l'acquisizione della sola parte che precede lo spazio e una segnalazione di errore.

### Package bufio

Contiene funzioni che agevolano l'input/output di testi attraverso il buffering. Delle tante utilità, per le nostre esigenze dilettantesche viene comoda quella che ci consente di inserire in una variabile stringa più parole, con un input che contiene spazi. In tal modo possiamo superare la limitazione che abbiamo appena visto per la funzione Scan del package fmt.

Per fare tutto questo dobbiamo importare, oltre al package bufio, anche il package os, che contiene funzioni di interfacciamento con il sistema operativo.

La sintassi per leggere il contenuto di una riga di testo anche formato da più parole ed allocarla in una variabile stringa precedentemente dichiarata è la seguente

```
scanner := bufio.NewScanner(os.Stdin)
scanner.Scan()
<variabile> = scanner.Text()
```

Nella prima riga costruiamo uno scanner avvalendoci del costruttore NewScanner del package bufio e lo abilitiamo per lo standard input (la tastiera) con la funzione Stdin del package os.
Nella seconda riga utilizziamo la funzione Scan di questo scanner per leggere la riga di testo che scriviamo sulla tastiera.
Nella terza riga, attraverso la funzione Text di questo scanner che ritorna il testo letto, lo assegniamo alla variabile stringa precedentemente dichiarata per contenerlo.

### Package math

Contiene costanti e funzioni matematiche che ci possono servire per qualsiasi calcolo possiamo immaginare. Esaminiamole per raggruppamenti.

#### Costanti

Tra le tante cito le tre più famose costanti matematiche `E` (numero e, base dei logaritmi naturali), `Pi` (pi greco) e `Phi` (fi greco, il così detto rapporto aureo) e le loro radici quadrate `SqrtE`, `SqrtPi`, `SqrtPhi`, oltre alla radice di 2 `Sqrt2`.

#### Radici e potenze

`Pow(`<base>`,` <esponente>`)` ritorna in float64 la base elevata all'esponente indicato, espressi in float64,
`Pow10(n)` ritorna in float64 10 elevato a n, espresso come intero,
`Exp(x)` ritorna in float64 il numero e elevato a x, espresso come float64,
`Sqrt(x)` ritorna in float64 la radice quadrata di x, espresso in float64,
`Cbrt(x)` ritorna in float64 la radice cubica di x, espresso in float64.
Per radici ennesime dobbiamo ricorrere alla funzione Pow indicando come esponente il reciproco dell'indice di radice, dato che $\sqrt[n]{x} = x^{\frac{1}{n}}$.

#### Funzioni trigonometriche

Tutte le funzioni trigonometriche hanno l'argomento espresso in float64 radianti e ritornano un valore in float64. Le inverse hanno l'argomento in float64 e ritornano un valore in float64 radianti.

Abbiamo le funzioni `Sin(x)`, `Cos(x)`, `Tan(x)` e le funzioni inverse `Asin(x)`, `Acos(x)`, `Atan(x)`.

**Funzioni iperboliche**

Tutte le funzioni iperboliche hanno l'argomento espresso in float64 e ritornano valori in float64.

Abbiamo le funzioni `Sinh(x)`, `Cosh(x)`, `Tanh(x)` e le funzioni inverse `Asinh(x)`, `Acosh(x)`, `Atanh(x)`.

**Funzioni varie**

Tutte queste funzioni operano su tipi float64.
Abs(x) ritorna il valore assoluto di x,
Ceil(x) ritorna, in float64, il più piccolo intero maggiore o uguale a x,
Floor(x) ritorna, in float64, il più grande intero minore o uguale a x,
Log(x) ritorna il logaritmo naturale di x,
Log10(x) ritorna il logaritmo decimale di x,
Trunc(x) ritorna la parte intera di x.

**Sottopacchetto per la generazione di numeri casuali**

Il package math contiene un sottopacchetto rand che contiene funzioni sulla casualità.

Per utilizzarlo va importato come `math/rand`.

Per noi può essere interessante usarlo in due semplici casi: generare un numero casuale intero compreso tra 0 e n o generare un numero casuale decimale compreso tra 0 e 1.

In ogni caso dobbiamo preventivamente ancorarci alla sorgente di generazione, che, per garantire una perfetta casualità, deve variare ogni volta. Per ottenere questa grandezza mutabile si usa leggere l'orologio del computer che fornisce un valore che cambia ad ogni battere di ciglio, pertanto, insieme al package `math/rand` dobbiamo importare anche il package `time`. Per generare la sorgente dobbiamo scrivere queste due righe di codice

```
s := rand.NewSource(time.Now().UnixNano())
r := rand.New(s)
```

Così ottenuto il generatore `r`, ancorato alla lettura dell'orario segnato dall'orologio del computer ne richiamiamo le funzioni generatrici dei numeri casuali che sono
`r.Intn(n)` per generare un casuale intero compreso tra 0 e n,
`r.Float64()` per generare un casuale decimale compreso tra 0 e 1.
Esempio:
Questo codice

```
package main
import (
"fmt"
"time"
"math/rand"
)
func main() {
    s := rand.NewSource(time.Now().UnixNano())
    r := rand.New(s)
    fmt.Println(r.Intn(20))
    fmt.Println(r.Float64())
    numero_casuale := r.Intn(20)
    fmt.Println(numero_casuale)
}
```

una volta compilato, produce un programma che scrive un numero casuale tra 0 e 20, poi scrive un decimale casuale compreso tra 0 e 1, poi inserisce nella variabile numero_casuale un altro numero casuale tra 0 e 20 e stampa il contenuto di questa variabile: come si vede se si esegue

il programma, anche a distanza di millesimi di secondo, il secondo numero casuale tra 0 e 20 generato è diverso dal primo.

## 8 Interattività con l'utente

Negli esempi che abbiamo visto finora abbiamo scritto programmi che contenevano già i dati da elaborare e, una volta lanciati, fornivano il risultato. Se ci accontentassimo di questo, per calcolare le combinazioni e le disposizioni di 18 numeri presi 4 a 4, dovremmo prendere il programma che abbiamo scritto nel Capitolo 6 per calcolare combinazioni e disposizioni di 10 numeri presi 3 a 3, ricopiarlo sostituendo al valore della variabile n il numero 18 e al valore della variabile k il numero 4, ricompilarlo e rilanciarlo.

Ma con quanto abbiamo visto nel precedente Capitolo 7 possiamo ora fare meglio e modificare il programma in modo che, una volta lanciato, chieda all'utente per quali valori vuole eseguire i calcoli in modo che il programma diventi finalizzato a calcolare combinazioni e disposizioni su numeri qualsiasi.

Forti di quanto abbiamo appreso esaminando tutte le funzioni del package fmt basta che modifichiamo il programma scritto nel Capitolo 6 in questo modo

```
package main
import "fmt"
func fattoriale (n int) int {
    if n == 0 {
        return 1
    }
    return n * fattoriale (n-1)
}
func main() {
    var n int
    var k int
    var c int
    var d int
    fmt.Println("Quanti elementi utilizzi?")
    fmt.Scan(&n)
    fmt.Println("Con quanti elementi formi i raggruppamenti?")
    fmt.Scan(&k)
    c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
    d = fattoriale(n)/fattoriale(n-k)
    fmt.Println("combinazioni:  ", c)
    fmt.Println("disposizioni:  ", d)
}
```

Lanciando questo programma saremo richiesti di indicare il numero di elementi utilizzati (n), il numero di elementi per i raggruppamenti (k) e in un attimo vedremo il risultato. Purtroppo, a causa del pesante calcolo dei fattoriali implicati nelle formule, per quanto già detto nel Capitolo 3 trattando dei tipi numerici interi, avremo risultati sballati se il numero di elementi (n) supera la ventina[5].

Un altro facile esempio:

[5]Quello dello stack overflow, cioè del non riuscire a trattare numeri al di là di certe dimensioni, è un grande limite di tutti i linguaggi compilati (Pascal, C, C++, ADA). Come ho accennato nel Capitolo 3 parlando dei tipi numerici, i creatori di Go hanno superato questo limite inventandosi i grandi interi da trattarsi con le funzioni incluse nel package math/big. Ma il tutto è di uso parecchio laborioso e complicato. Se vogliamo essere tranquilli e poter usare grandi numeri senza alcuna complicazione è meglio che ricorriamo al linguaggio Python, dove è tutto più semplice.

```
package main
import (
"fmt"
"bufio"
"os"
)
func main() {
    var nome string
    fmt.Println("Come ti chiami?")
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Scan()
    nome = scanner.Text()
    fmt.Println("Ciao,", nome)
}
```

In questo programmino ci viene chiesto il nome e ci viene rivolto un piccolo saluto personalizzato. Dal momento che il nome può essere composto da più parole oppure vogliamo inserire anche il cognome, per la lettura dell'input sono state usate le funzioni del package bufio.

# 9 Strutture di controllo

Come tutti i linguaggi di programmazione, Go ci offre il modo di controllare l'esecuzione del programma attraverso istruzioni per la ripetizione di blocchi oppure per condizionare l'esecuzione di un blocco al verificarsi di certe condizioni. Con un lodevole intento semplificante, i creatori di Go sono stati parchi nel predisporre queste istruzioni, che sono praticamente solo tre contro le quattro o più che troviamo in altri linguaggi e che, comunque e pur con diverse sfumature, fanno le stesse cose.

### for

L'istruzione `for` si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per l'uso di questa istruzione è

```
contatore := 1
for contatore <= n {
   <istruzioni>
   contatore = contatore + 1
}
```

La variabile che ho battezzato `contatore` per richiamarne la natura, in genere viene nominata semplicemente `i` e serve per contare le volte che viene eseguito il blocco di istruzioni contenuto tra le parentesi graffe: vediamo infatti che, dopo le istruzioni, se ne prevede l'incremento di una unità.

L'istruzione `for` lancia l'esecuzione del blocco di istruzioni fino a che il contatore è minore o uguale a un certo numero `n`, deciso dal programmatore.

Questo piccolo programma

```
package main
import "fmt"
func main() {
    i := 1
    for i <= 3 {
        fmt.Println("Ciao, Vittorio")
        i = i + 1
    }
}
```

scrive la frase Ciao, Vittorio per tre volte.

Quest'altro

```
package main
import "fmt"
func main() {
    i := 1
    for i <= 6 {
        fmt.Println(i)
        i = i + 1
    }
}
```

elenca i numeri da 1 a 6.

In entrambi gli esempi il blocco da ripetere contiene una sola istruzione oltre a quella di aumentare di 1 il contatore ma, ovviamente, può contenerne quante vogliamo.

### if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if <condizione> {
   <istruzioni>
}
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite le istruzioni contenute tra le parentesi graffe, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if <condizione> {
   <istruzioni>
} else {
   <istruzioni>
}
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco altrimenti vengono eseguite quelle contenute nel secondo blocco dopo else (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `else if` con questa sintassi

```
if <condizione> {
   <istruzioni>
} else if <condizione> {
   <istruzioni>
} else if <condizione> {
   <istruzioni>
} else if <condizione> {
   <istruzioni>
}
.....
```

e proseguire quanto vogliamo.

Per esemplificare l'uso dell'istruzione if propongo un perfezionamento del programmino che, nel precedente paragrafo, elencava i numeri da 1 a 6 in modo che a fianco di ciascun numero venga indicato se si tratta di un numero pari o dispari.

```
package main
import "fmt"
func main() {
    i := 1
    for i <= 6 {
        if i%2 == 0 {
            fmt.Println(i,"pari")
        } else {
            fmt.Println(i,"dispari")
        }
        i = i + 1
    }
}
```

Infine, per un ripasso che si estende utilmente all'uso degli operatori aritmetici, propongo quest'altro programma che, con una semplificazione che ne limita l'uso agli anni del calendario gregoriano, affida al computer l'algoritmo del grande matematico Gauss (che peraltro ha fatto cose ben più importanti di quella di calcolare la data delle pasque) per la determinazione del giorno della Pasqua cristiana (prima domenica successiva alla luna piena di primavera).

```
package main
import "fmt"
func main() {
    var anno,a,b,c,d,e,g,h,j,k,m,r,n,p int
    var mese string
    fmt.Println("Inserisci l'anno")
    fmt.Scan(&anno)
    if anno < 1583 {
        fmt.Println("Fornirei una indicazione senza senso.")
        fmt.Println("Mi baso sul calendario gregoriano,")
        fmt.Println("adottato nel 1583.")
    }
    a = anno%19
    b = int(anno/100)
    c = anno%100
    d = int(b/4)
    e = b%4
    g = int((8*b+13)/25)
    h = (19*a+b-d-g+15)%30
    j = int(c/4)
    k = c%4
    m = int((a+11+h)/319)
    r = (2*e+2*j-k-h+m+32)%7
    n = int((h-m+r+90)/25)
    p = (h-m+r+n+19)%32
    if n == 3 {
        mese = "marzo"
    } else {
        mese = "aprile"
    }
    fmt.Println("Pasqua",anno,":",p,mese)
}
```

Dal momento che ho semplificato l'algoritmo riducendolo al calcolo del giorno della Pasqua per gli anni del calendario gregoriano attualmente utilizzato, il primo `if` verifica che l'anno introdotto dall'utente non sia precedente a quello di adozione del calendario gregoriano: se fosse precedente verrebbe scritto il messaggio esplicativo e si interromperebbe l'esecuzione del programma.
Il secondo `if` serve semplicemente per tradurre in lettere il mese che i calcoli determinano indicando, nella variabile n, il numero 3 o il numero 4.

### switch

Abbiamo visto che l'istruzione `if` può essere applicata anche a una ramificazione multipla attraverso le istruzioni aggiuntive `else if`.

L'istruzione switch può convenientemente essere usata per semplificare la realizzazione della ramificazione multipla nel caso essa dipenda dal confronto tra diverse alternative che una variabile può assumere.

La sintassi è la seguente

```
switch <variabile> {
    case <valore>: <istruzione>
    case <valore>: <istruzione>
    case <valore>: <istruzione>
    .......
}
```

dove <variabile> può essere di tipo intero o di tipo stringa e <valore> è uno dei valori che la variabile può assumere in corrispondenza al quale eseguire una certa istruzione.

Se riprendiamo il programma della Pasqua mostrato nel paragrafo precedente, dove abbiamo scritto

```
if n == 3 {
        mese = "marzo"
    } else {
        mese = "aprile"
    }
```

per far corrispondere al valore 3 della variabile n il nome marzo e far corrispondere al valore alternativo il nome aprile, potremmo scrivere, in modo più semplice

```
switch n {
    case 3:  mese = "marzo"
    case 4:  mese = "aprile"
}
```

Propongo un altro esempio nel quale il confronto è basato su valori stringa.

```
package main
import "fmt"
func main() {
    var moneta string
    fmt.Println("Scrivi il nome di una moneta")
    fmt.Scan(&moneta)
    switch moneta {
        case "euro":  fmt.Println("Europa")
        case "dollaro":  fmt.Println("USA")
        case "sterlina":  fmt.Println("Regno Unito")
        case "yen":  fmt.Println("Giappone")
        case "yuan":  fmt.Println("Cina")
        default:  fmt.Println("non so")
    }
}
```

Tra i due esempi si noterà una differenza: nel secondo caso ho introdotto un'istruzione di default che nell'esempio precedente non c'era.

Questa differenza è dovuta al fatto che nel primo caso il programmatore sa che la variabile n, per come sono configurati i calcoli che la determinano, può assumere o il valore 3 o il valore 4 e non c'è bisogno di prevedere cosa debba succedere in presenza di altri valori che non si possono verificare.

Nel secondo caso, invece, il programma risponde indicando la zona geografica dove si usa una certa valuta indicata dall'utente e, anche se il programmatore si sforza di prevederle tutte, può sempre succedere che l'utente ne indichi una non prevista: al che il programma risponde «non so».

# 10 Puntatori

Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile.

La variabile puntatore si crea con la sintassi
`<puntatore> := &<variabile>`
dove <puntatore> è il nome che gli attribuiamo
e <variabile> è la variabile puntata.
L'operatore `&` è l'operatore di indirizzamento (quello stesso che abbiamo conosciuto trattando della funzione `Scan` del package `fmt` nel Capitolo 7).
Attraverso questo operatore attribuiamo alla variabile puntatore l'indirizzo di memoria della variabile puntata e la variabile puntatore acquisisce un tipo, detto tipo puntatore, simile a quello della variabile puntata. Se questa contiene un numero intero abbiamo un puntatore a un intero.

Con l'istruzione
`*<puntatore>` otteniamo il valore della variabile puntata.
L'operatore `*` che utilizziamo, facendolo precedere al nome della variabile puntatore, si chiama operatore di indirezione e significa «il contenuto di».
Con questo stesso operatore, fatto precedere al normale identificatore di tipo, indichiamo anche il tipo puntatore: `*int` indica il tipo puntatore a intero, `*float64` indica il tipo puntatore a float64, ecc.

Poniamo di avere la variabile x contenente il numero intero 100.
Se creiamo un puntatore alla variabile x con l'istruzione px := &x,
questa variabile puntatore px ha come valore l'indirizzo di memoria di x
ma con l'istruzione *px otteniamo il valore 100 della variabile puntata.

Per comprendere a cosa servono i puntatori vediamo questo piccolo programma.

```
package main
import "fmt"
func azzera(x int) {
    x = 0
}
func main() {
    x := 5
    azzera(x)
    fmt.Println("x vale:",x)
    x = 0
    fmt.Println("ora x vale:",x)
}
```

Se compiliamo e lanciamo questo programma otteniamo questi risultati

```
x vale: 5
ora x vale: 0
```

Come mai, avendo tentato di azzerare la variabile x richiamando una funzione che abbiamo scritto apposta per farlo, la variabile x non si è affatto azzerata e per azzerarla abbiamo dovuto farlo attribuendole il valore 0 con apposita istruzione?
Il fatto è che quando passiamo una variabile come parametro a una funzione lo passiamo per valore e la funzione, in realtà, non lavora sulla variabile ma lavora su una sua copia. Pertanto tutto va bene se la funzione si limita ad utilizzare il valore della variabile, copiato, per fare elaborazioni e calcoli. Ma se, con la funzione, pretendiamo di modificare il valore della variabile, in realtà modifichiamo la copia di quel valore ma non il valore originario della variabile.

Lavorando con i puntatori diventa possibile, invece, raggiungere la variabile anche lavorando con una funzione, come avviene con quest'altro programmino.

```
package main
import "fmt"
func azzera(px *int) {
*px = 0
}
func main() {
x := 5
azzera(&x)
fmt.Println("x vale:",x)
}
```

Qui la funzione azzera prevede un parametro che non è la variabile ma il valore del puntatore alla variabile di tipo puntatore a intero: in poche parole alla funzione azzera dobbiamo passare non una variabile ma il suo indirizzo. La funzione, così, azzera il valore della variabile raggiungendola per indirezione con il puntatore, cioè andando al suo indirizzo.
Nella funzione main, dopo aver creato la variabile con il valore 5, la azzeriamo passando alla variabile azzera l'indirizzo della variabile x e così riusciamo ad azzerarla.
Con questo sistema non abbiamo passato alla funzione il parametro per valore ma, come si dice in informatica, abbiamo passato il parametro per riferimento.

Non so se e quando un dilettante programmatore alle prime armi possa trovare convenienza ad utilizzare i puntatori. Convenienza che emerge quando, per accelerare l'esecuzione di un programma ed evitare appesantimenti di uso di memoria, diventa pressante evitare le continue copiature di dati che avvengono quando alle funzioni si passano i dati per valore.

Ho comunque ritenuto utile parlarne per creare un tantino di consapevolezza su come lavora un computer.

Inoltre, visto che i padri di Go sono riusciti a semplificare parecchio la sintassi per l'uso dei puntatori rispetto a quanto avviene nei linguaggi C e C++, direi che in Go l'uso dei puntatori, una volta capito a cosa servono, può essere affrontato anche da un dilettante.

## 11 Conclusione

Verso la fine del secolo scorso, all'apparire del linguaggio di programmazione Java, ricordo di aver letto il seguente apprezzamento da parte di un programmatore: «Mi piace Java perché, rispetto a C e C++, mi ha liberato dall'incubo della garbage collection e dei puntatori».

Come abbiamo visto nel precedente Capitolo, invece, Go continua a tenerci liberi dalla garbage collection ma ci ripropone, pur in modo semplificato, i puntatori.

Abbiamo inoltre visto come persistano altri ancoraggi al non facile linguaggio C e come non sia poi così semplice, per un neofita della programmazione, affrontare questo C semplificato che sarebbe il linguaggio Go.

Probabilmente le semplificazioni sono più sensibili per affrontare problemi di programmazione che esulano da ciò che può fare un dilettante.

Al nostro livello, pertanto, teniamoci questo manualetto come un bell'esercizio ma, se vogliamo faticare meno ed essere tranquilli sui risultati anche lavorando su grandi numeri, continuiamo a preferire Python.