LE FUNZIONI

In questo Workshop parliamo nuovamente di programmazione, introducendo un nuovo utilissimo concetto: le funzioni, la loro dichiarazione e il loro utilizzo.

Nei programmi di esempio che hai potuto sperimentare ti è capitato più volte di utilizzare le istruzioni 'printf' e 'scanf' per gestire l'input e l'output testuale. Tali istruzioni sono, più precisamente, due **funzioni** messe a disposizione dalla libreria 'stdio.h'. Ma cos'è più precisamente una funzione in un linguaggio di programmazione? In modo molto semplice, **possiamo pensare una funzione come un costrutto che consente di raggruppare una porzione di codice e di richiamarla all'interno del programma in cui è dichiarata.**

FUNZIONI A PIACIMENTO

La possibilità di richiamare le funzioni è fondamentale in programmazione, ad esempio quando all'interno di un programma c'è necessità **di ripetere in più punti lo stesso blocco di codice**, oppure quando si vuole avere l'opportunità di **definire nuove istruzioni di alto livello**. In quest'ottica quindi la funzione è sia uno strumento che ci consente di semplificare la struttura del codice prodotto, ottenendo programmi più compatti e modulari, sia una tecnica di astrazione del linguaggio di programmazione stesso, che attraverso la creazione di nuove istruzioni di alto livello, può permetterci di ottenere un'ossatura del sorgente più vicina alla logica del programmatore. Vediamo come procedere per **definire una funzione**. Innanzitutto **ogni funzione è caratterizzata da quattro elementi fondamentali: il nome, il tipo ritornato, i parametri e la sua implementazione.** Il **nome** è semplicemente **l'identificativo mnemonico che consente di richiamarla all'interno dei propri programmi** e che, se scelto oculatamente, può aiutare il programmatore a comprendere il significato dell'istruzione stessa (le regole sintattiche per i nomi delle funzioni sono le stesse valide per i nomi delle variabili). Il **tipo caratterizza il dato che viene restituito come risultato dell'elaborazione.** Possono essere utilizzati tutti i tipi riferibili alle variabili, in base alla necessità (ad esempio, se la funzione che si vuole scrivere ha come obiettivo il calcolo della circonferenza di un cerchio, è plausibile che debba essere dichiarata come tipo 'float', proprio perchè il risultato sarà di tipo decimale). I **parametri** rappresentano **i dati di ingresso che possono essere impiegati dalla funzione durante la sua esecuzione.** Può essere utilizzato un qualsiasi numero di parametri, purché **ognuno di essi venga indicato attraverso un tipo e un nome.** L'ultimo elemento, è l'implementazione, che costituisce, invece, **il blocco di codice eseguito nel momento in cui viene chiamata la funzione.** La sintassi che consente di dichiarare una funzione è mostrata nel box sottostante. Come puoi notare dal contenuto di questo box, nel corpo della funzione compare la parola chiave

La dichiarazione delle funzioni

```
tipo_funzione nome_funzione ( tipo_par1 nome_par1,
tipo_par2 nome_par2, tipo_par3 nome_par3,...)
{
     ...............
      implementazione_della_funzione
     ...............
     return valore_ritornato;
}
```

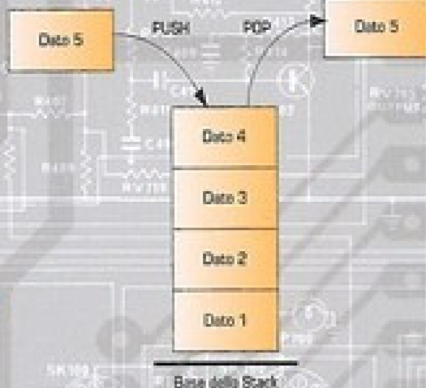
'return' (che hai già usato in passato negli StepbyStep). Essa **ha il compito di 'terminare' la funzione, restituendo il valore che la segue.** Se la funzione è 'tipizzata' (ossia è stata dichiarata con un tipo diverso da 'void') l'uso della parola chiave return è indispensabile e il tipo del valore ritornato deve essere compatibile con il tipo della funzione stessa. Inoltre, **poiché 'return' causa l'uscita dalla funzione, tutto il codice successivo all'esecuzione di tale comando viene ignorato.** Una seconda nota deve essere fatta necessariamente in merito a 'quando' e 'dove' si può dichiarare una funzione. Il discorso in realtà è molto articolato, ma per i nostri scopi è sufficiente tener presente che **ogni funzione deve essere sempre dichiarata prima del suo utilizzo** (di conseguenza tutte le funzioni ausiliarie devono essere definite prima della definizione della funzione principale 'main').

ALCUNE NOTE SULLE FUNZIONI

Prima di dedicarci alla pratica, concludiamo con due semplici osservazioni: la prima concerne **l'impiego della memoria.** Abbiamo detto che il ricorso alle funzioni permette in un certo senso di 'compattare' il codice. Diminuisce quindi la dimensione dell'eseguibile, ma aumenta, di contro, la quantità di memoria richiesta durante l'esecuzione del programma stesso. Ciò è legato ai processi di funzionamento interni del computer stesso. L'esecuzione di una funzione, infatti, 'congela' il programma principale per il tempo necessario alla sua esecuzione. In questa fase di congelamento il microprocessore esegue un '**salto**' da un punto all'altro del programma in esecuzione con la necessità, però, di poter riprendere l'esecuzione del programma chiamante al termine della funzione. Ciò è possibile memorizzando i parametri di rientro in una particolare porzione di memoria chiamata '**stack**' (vedi box sotto), che aumenta di dimensioni ogni volta che viene richiamata una funzione e si riduce alla sua conclusione. Il problema sorge se il software contiene una grande quantità di funzioni che si richiamano reciprocamente senza mai concludersi (come nel caso delle **procedure ricorsive**, un'alternativa all'iterazione che non affronteremo). In questa situazione critica, lo stack continua ad aumentare la sua dimensione sino a saturare l'area di memoria, con la generazione di un errore di sistema noto come '**stack overflow**' (straripamento dello stack). Oltre alle questioni legate alla memoria vi sono una serie di problematiche inerenti al cosiddetto '**passaggio di parametri**', ossia al modo in cui la funzione riceve i parametri. In C, infatti, i parametri possono essere trasmessi alle funzioni in diversi modi. Negli esempi dello StepbyStep sfrutteremo il **passaggio 'per copia'**, che prevede appunto la copia dei valori usati come parametro della funzione chiamante in apposite aree di memoria riservate alla funzione chiamata. I parametri vengono quindi 'duplicati', con la conseguenza che se la funzione applica modifiche su di essi, queste non si ripercuotono sulla funzione chiamante.

LO STACK

Lo stack è una speciale area di memoria presente nei computer (e che ritroveremo anche nei microcontrollori) caratterizzata, da una struttura di accesso 'a pila' (tecnicamente detta LIFO, ossia *Last In, First Out*, che può essere approssimativamente tradotto come 'l'ultimo inserito, è il primo estratto'). Il comportamento di questa struttura è simile a una pila di piatti, sulla quale sono possibili solo due operazioni: l'aggiunta di un piatto sulla cima (operazione di *push*) e la rimozione di un piatto, sempre dalla cima (operazione di *pop*, vedi schema a destra). Nel campo della programmazione lo stack riveste una importanza enorme in quanto viene utilizzato, tra l'altro, per immagazzinare i parametri di rientro ogniqualvolta si richiami una funzione.



STEPbySTEP

PERIMETRO E AREA DI UN RETTANGOLO

In questo primo esercizio puoi sperimentare concretamente la definizione di due semplici funzioni: la prima ti permette di calcolare il perimetro di un rettangolo a partire dai suoi lati (passati come parametri); la seconda calcola, invece, la sua area utilizzando base e altezza. Le due funzioni lavorano su dati di tipo intero.

Perimetro e area di un rettangolo

```
#include <stdio.h>
/* funzione che calcola e restituisce il perimetro di
un rettangolo partendo dalla sua base e dalla sua altezza */
int calcola_perimetro_rettangolo( int lato1, int lato2)
{
      int perimetro;
      perimetro = 2*lato1 + 2*lato2;
      return perimetro;
}

/* funzione che calcola e restituisce l'area di un rettangolo
partendo dalla sua base e dalla sua altezza */
int calcola_area_rettangolo(int base, int altezza)
{
      int area;
      area = base*altezza;
      return area;
}

/*programma principale*/
int main()
{
  int base_letta, altezza_letta; /*  base e altezza inserite
                                      dall'utente */

  /* acquisisce la base */
  printf("\nInserisci la base del rettangolo: ");
  scanf("%d", &base_letta);

  /*acquisisce l'altezza */
  printf("\nInserisci l'altezza del rettangolo: ");
  scanf("%d", altezza_letta);

 /* calcola e stampa a video il perimetro del rettangolo */
  printf("\n\nIl perimetro del rettangolo è: %d",
         calcola_perimetro_rettangolo(base_letta,altezza_letta));


/* calcola e stampa a video l'area del rettangolo */
   printf("\nL'area del rettangolo è: %d",
         calcola_area_rettangolo(base_letta,altezza_letta));
/* attende la pressione di un tasto e termina il programma */
getch();
return 1;
}
```

LA MODIFICA DEI PARAMETRI PASSATI

Il secondo esercizio che proponiamo ti sarà utile per verificare che la modifica, all'interno di una funzione, di un parametro passato 'per copia' non altera le variabili della funzione chiamante. La funzione definita è di tipo 'void' e, come tale, non ritorna valori.

La tavola pitagorica

```
#include <stdio.h>

/* funzione che applicherà alcune semplici modifiche matematiche sul
parametro ricevuto */
void modifica_parametri(int p1)
{
    printf("\nParametro originale ricevuto dalla funzione: %d",
            p1);
    p1++;
    printf("\nParametro incrementato della funzione: %d", p1);
    p1*=2;
    printf("\nParametro incrementato e raddoppiato dalla funzione:
            %d", p1);
}


/* funzione main del programma */

int main()
{
    /* parametro che verrà utilizzato */
    int parametro;

    /* acquisizione del parametro dall'utente */
    printf("\nInserisci il parametro: ");
    scanf("&d", &parametro);

    /* stampa del parametro memorizzato*/
    printf("\nIl parametro nella funzione main vale: %d ",
            parametro);

    /* chiamata della funzione */
    modifica_parametri(parametro)

    /*stampa del parametro dopo la chiamata della funzione */
    printf("\nDopo la chiamata della funzione il parametro vale:
            %d ", parametro);

    /* attende la pressione di un tasto e termina il programma */
    getch();
    return 1;
}
```