UNIVERSITÀ
degli STUDI
di CATANIA

Funzioni in C

Corso di programmazione I (A-E / O-Z) AA 2022/23

Corso di Laurea Triennale in Informatica

Fabrizio Messina
fabrizio.messina@unict.it

Dipartimento di Matematica e Informatica

Indice

Introduzione alle funzioni in C

Definizione e struttura di una funzione in C

Le **funzioni** costituiscono la base della programmazione strutturata/procedurale.

Una funzione rappresenta un **blocco di codice** identificato da un **nome**.

Le funzioni andrebbero concepite per eseguire una o più attività **strettamente correlate tra loro**.

L'uso delle funzioni favorisce la **modularità** nei linguaggi come il C.

Definizione e struttura di una funzione in C

In C una funzione è costituita da:

- **nome** della funzione;
- **tipo di ritorno**;
- lista di argomenti o **parametri formali**;
- **corpo della funzione** (istruzioni) tra parentesi graffe;

```
1   int func(int dato){
2     return dato*2;
3   }
```

Definizione e struttura di una funzione in C

```
1   int func(int dato){
2     return dato*2;
3   }
4
5   int result = func(23);
6   printf("%d", result);
```

Per la semplice funzione denominata `func`

- `int` è il tipo di ritorno;
- `dato` è il parametro formale di tipo `int`;

Definizione e struttura di una funzione in C

```
1   int func(int dato){
2     return dato*2;
3   }
4
5   int result = func(23);
6   printf("%d", result);
```

L'istruzione alla linea 2 **costituisce il corpo della funzione**. La parola chiave `return` fa sì che il flusso di esecuzione prosegua con la istruzione successiva alla chiamata a funzione;

La linea 5 costituisce la **invocazione** (o chiamata) della funzione.

Invocazione di una funzione

```
1   int func(int dato){
2     return dato*2;
3   }
4
5   int result = func(23);
6   printf("%d", result);
```

Il valore di ritorno della funzione viene **copiato** nella variabile `ret`.

L'istruzione eseguita successivamente alla istruzione della linea 2 è quella della linea 6.

Definizione vs prototipo di funzione

Prototipo della funzione

```
1   //dichiarazione del prototipo
2   double sum(double , double );
```

Definizione della funzione.

```
1   //definizione
2   double sum(double p, double q){
3      double result = p + q;
4      return result;
5   }
```

Definizione vs prototipo di funzione

Prototipo:

```
double sum(double , double );
//oppure
double sum(double p, double q);
```

- **tipo di ritorno**;
- **segnatura**:
  - nome funzione;
  - lista parametri formali definiti da `tipo` e `nome` oppure semplicemente la lista dei tipi (i nomi dei parametri formali si possono omettere).

Definizione vs prototipo di funzione

È buona pratica:

- raccogliere le **dichiarazioni dei prototipi** delle funzioni in appositi file *header* (ES: `modulo1.h`). Un header contiene in genere:
  - direttive `#define` e altre direttive `#include`
  - dichiarazione di variabili (anche costanti) globali
  - prototipi di funzioni
- raccogliere la **definizione delle funzioni** (e metodi) in un modulo sorgente, ES: `modulo1.c`;
- usare la direttiva `#include "modulo1.h"` in ogni file sorgente in cui si fa uso di tali funzioni.

Definizione vs prototipo di funzione

I moduli contenenti una o più funzioni si possono compilare separatamente in uno o più file *oggetto* da assemblare successivamente.

```
$ gcc -c modulo1.c
$ gcc -c modulo2.c
...
$ gcc -c modulok.c
$ gcc -c main.c
```

Definizione vs prototipo di funzione

Il risultato sarà un set di file *oggetto*:

- `modulo1.o`
- `modulo2.o`
- `...`
- `modulok.o`
- `main.o`

Infine si possono assemblare (*fase di linking* – produce eseguibile):

```
$ gcc main.o modulo1.o modulo2.o ... modulok.o
```

Definizione vs prototipo di funzione

oppure

```
$ g++ main.c modulo1.c modulo2.c \
  [...] modulok.c
```

Definizione vs prototipo di funzione

15_00_main.c
15_00_func.c
15_00.h

Invocazione di funzioni

Parametri formali vs parametri attuali

La lista di argomenti presenti nella segnatura di una funzione o metodo è detta lista di **parametri formali**.

I valori passati nella invocazione della funzione vengono detti **parametri attuali**.

```
1 void foo(int x){ //x parametro formale
2   //...
3 }
4 int main(){
5   // ...
6   int a;
7   foo(a);  // a parametro attuale
8 }
```

Parametri formali vs parametri attuali

Nella programmazione strutturata/procedurale, il flusso è rappresentato da **una sequenza di invocazioni di funzioni**.

I parametri permettono alle funzioni di **scambiare dati**.

In C, per ottenere un programma eseguibile, è "obbligatorio" fornire al compilatore una funzione denominata `main`.

La prima istruzione della esecuzione della applicazione è rappresentata dalla invocazione della funzione `main()`.

Area stack e record di attivazione

La **porzione di memoria riservata** allo stoccaggio dei dati utili alla esecuzione delle **istruzioni contenute nel corpo delle funzioni** è denominata **stack** (pila).

Lo stack è una struttura in cui i dati vengono inseriti e prelevati in base al meccanismo `LIFO` (Last In - First Out).

Il singolo dato viene depositato (push) sempre sul top dello stack.

Si può prelevare un dato alla volta (pop), solo dal top dello stack.

Area stack e record di attivazione


```
1   void foo(int x){
2     double c, d;
3     //...
4   }
5   int main(){
6     int a,b;
7     //...
8     foo(a);
9     //..
10  }
```

Area stack e record di attivazione


```
1    void foo(int x){
2       double c, d;
3       //...
4    }
5    int main(){
6       int a,b;
7       //...
8       foo(a);
9       //..
10   }
```

1-Un record per la funzione **foo()** viene allocato sul segmento stack.

Area stack e record di attivazione


```
1   void foo(int x){
2     double c, d;
3     //...
4   }
5   int main(){
6     int a,b;
7     //...
8     foo(a);
9     //..
10  }
```

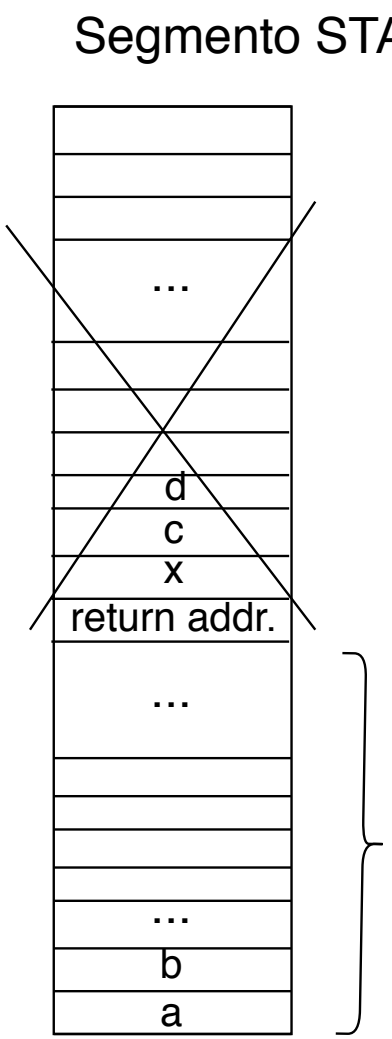
2-I parametri attuali vengono depositati nello stack: in questo caso il valore del parametro attuale b, denominato x nella lista di parametri formali;

Area stack e record di attivazione


```
1   void foo(int x){
2     double c,d;;
3     //...
4   }
5   int main(){
6     int a,b;
7     //...
8     foo(a);
9     //..
10  }
```

3-Vangono poi allocate le variabili locali definite all'interno della funzione (c e d);

Area stack e record di attivazione



Record di attivazione per la funzione main() (variabili locali, parametri attuali)

```
1   void foo(int x){
2     double c,d;
3     //...
4   }
5   int main(){
6     int a,b;
7     //...
8     foo(a);
9     //..
10  }
```

4-A seguito di una istruzione `return` o al arraggiungimento della fine della funzione `foo()`, il flusso prosegue con la istruzione successiva alla chiamata a `foo()`.

Passaggio mediante indirizzo e per valore

Passaggio “per valore” di un dato ad una funzione.

Il valore attuale del dato viene copiato sul record di attivazione dello stack (ES: a). .

```
1 void foo(int x){ //x parametro formale
2   //...
3 }
4 int main(){
5   // ...
6   int a = 10;
7   foo(a);
8   printf("%d", a);
9 }
```

Passaggio mediante indirizzo e per valore

Conseguenza del passaggio per valore: dato che la funzione opera su una copia di a, non può modificarne il valore.

```
1   void foo(int x){ //x parametro formale
2     //...
3     x = 90;
4   }
5   int main(){
6     // ...
7     int a = 10;
8     foo(a);  // parametro attuale e' VALORE in a
9     printf("%d", a); //stampa 10!
10  }
```

La istruzione alla linea 3 non ha alcun effetto sul valore di a!

Passaggio mediante indirizzo e per valore

Passaggio mediante indirizzo. La funzione riceve l'indirizzo del dato (il puntatore), quindi può operare modifiche al dato della "funzione chiamante" mediante **l'operatore di derefenziazione o indirezione.**

```
1   void spam(int *x){
2     //...
3     *x = 90;
4   }
5   int main(){
6     // ...
7     int a = 10;
8     spam(&a);  // parametro attuale e' INDIRIZZO di a
9     printf("%d", a); //stampa 90!
10  }
```

Classi di memorizzazione delle variabili, visibilità.

Allocazione automatica, dinamica, statica

Allocazione automatica:

- Dichiarazione di una **variabile locale** ad una funzione.
- Scope/visibilità **limitato al blocco di codice** in cui è stata dichiarata.
- Ciclo di vita del blocco allocato termina con la fine dell'esecuzione del blocco in cui viene
- Area di memoria usata è denominata `STACK`.

```
void foo(){
  int a = 0; // a visibile solo in foo()
}
```

Allocazione automatica, dinamica, statica

Allocazione dinamica

- Effettuata mediante funzione `malloc()` (`stdlib.h`) in qualunque punto del programma.
- Area di memoria usata è denominata `HEAP`.
- Ciclo di vita del blocco di memoria termina con invocazione di funzione `free()` sul puntatore.

```
//qualunque punto del programma
int *p = (int *) malloc(sizeof(int)); // cella int, valore
//...
free(p); // deallocazione della cella puntata da p
```

Allocazione automatica, dinamica, statica

Allocazione statica.

- Dichiarazione di variabili al di fuori da qualunque blocco.
- Segmento di memoria ospitante è detto **segmento DATA**.
- Ciclo di vita / scope: inizia e termina con il programma stesso.
- Variabile si dice globale.

```
//...
double data = 0.5;
//...
int main(){
  // ...
}
```

Esempi svolti

Esempi

`15_mem.c`

Passaggio di array a funzioni (allocazione automatica o statica)

Passaggio di array ad una dimensione

Il passaggio di un array come parametro di una funzione avviene sempre per indirizzo. (il nome di uno array è un puntatore costante al primo elemento dello array..).

```
1   void init(int *v, int n){
2     //...
3     for(int j=0; j<n; j++){
4       v[j] = 0;
5     }
6   }
7   int main(){
8     // ...
9     int x[10];
10    init(x, 10);
11  }
```

Passaggio di array ad una dimensione

Forma equivalente..

```
1   void init(int v[], int n){ //equivalente a int *v, ...
2     //...
3     for(int j=0; j<n; j++){
4       v[j] = 0;
5     }
6   }
7   int main(){
8     // ...
9     int x[10];
10    init(x, 10);
11  }
```

Passaggio di array multidimensionali

Per il passaggio di **array multidimensionali allocati sul segmento DATA o sullo STACK, nel parametro formale che rappresenta l'array, vanno specificate tutte le dimensioni, dalla seconda in poi**.

```
1   #define N 5
2   #define M 10
3   void init(int v[][M], int n);   // OK
4   void init(int v[N][M], int n);   // OK
5   void init(int v[][], int n); // Err. di compilazione!
6   void foo(){
7     int w[N][M];
8     init(w, N);
9   }
```

Passaggio di array multidimensionali

Attenzione!

```
1   void init(int *v[M], int n);
2   //...
3   int x[10][10]; //allocazione statica o automatica
4   init(x, 10); // NO: Errore di compilazione!
```

L'interpretazione del compilatore alla linea 1 è di **un vettore di M puntatori a int** (a causa della maggiore precedenza dello operatore [] rispetto a *).

La dichiarazione alla linea 1, **sebbene sintatticamente corretta**, provoca un **errore di compilazione in corrispondenza della invocazione alla linea 4**.

Passaggio di array multidimensionali

La seg. invece viene interpretata come **un puntatore a vettore di M interi**.

```
1   void init(int (*v)[M], int n); // OK
2   //...
3   int x[10][10]; //allocazione statica o automatica
4   init(x, 10); // OK
```

In questo caso la chiamata `init(x)` è lecita.

Passaggio di array multidimensionali

Per il passaggio di **array multidimensionali che siano VLA (Variable Length Array), le dimensioni che vanno specificate dalla seconda in poi**, si possono specificare anche mediante parametri formali. (Ricordiamo: non in C++!!)

```
1   void init(int m, int v[][m], int n);   // OK
2   void init(int m, int v[n][m], int n);   // OK, n viene ignor
3   void init(int v[][], int n); // Err. di compilazione!
4   void foo(){
5     int n = 10;
6     int m = 5;
7     int w[n][m];
8     init(m, w, n);
9   }
```

Passaggio di array multidimensionali

Array a tre dimensioni..

```
1  #define N 10
2  #define M 10
3  #define L 5
4  void init(int w[][M][L], int n);  //OK
5  void init(int w[N][M][L], int n);  //OK
6  void init(int w[][][L], int n); //Err. di compilazione!
```

Le linee 4 e 5 contengono dichiarazioni **valide**, ci sono

tutte le dimensioni che andavano specificate, dall'ultima alla seconda.

Alla linea 6 manca la seconda dimensione, quindi è una dichiarazione **non valida**.

Passaggio di array multidimensionali

Attenzione!

```
1   void init(int *v[M][L], int n);
2   int x[10][M][L];
3   init(x, 10); // Errore di compilazione!
```

Mentre la seg. è corretta...

```
1   void init(int (*v)[M][L], int n); // OK
2   int x[10][M][L];
3   init(x, 10); // OK
```

La dichiarazione alla linea 1 (il parametro formale v) viene interpretata come **un puntatore ad un array bidimensionale di interi dimensioni M x L**, quindi invocazione della linea 3 del secondo frame è valida.

Manipolazione di array con allocazione automatica o statica

Memorizzazione di array multidimensionali nello stack o nel segmento DATA

```
1   #define ROWS 3
2   #define COLS 4
3   int main(){
4       //...
5       int v[ROWS][COLS]; //segmento STACK
6       //...
7   }
```

La istruzione alla linea 5 implica la **allocazione** di un blocco di ROWS×COLS celle di memoria **contigue** di tipo `int` sullo *stack* (allocazione automatica).

Memorizzazione di array multidimensionali nello stack o nel segmento DATA

Allocazione statica (segmento DATA).

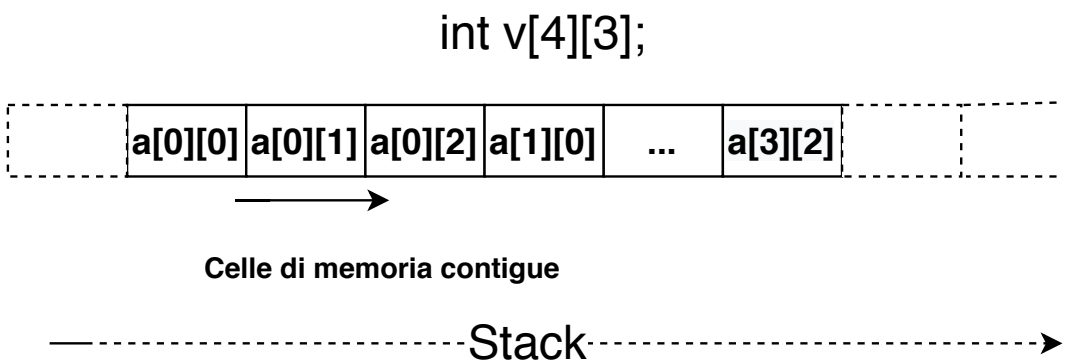
```
#define ROWS 3
#define COLS 4

int v[ROWS][COLS]; //segmento DATA
int main(){
   //...
}
```

Allocazione di un blocco ROWS×COLS celle **contigue** di tipo `int` nel segmento DATA.

Memorizzazione di array multidimensionali nello stack o nel segmento DATA

```c
#define ROWS 3
#define COLS 4
int main(){
  //...
  int v[ROWS][COLS]; //sullo stack
  //...
}
```



Accesso ad indirizzi e valori

```
int v[ROWS][COLS];
```

Per v dichiarato come sopra, la seg. expressione (1)

```
(1)  &v[i][j]; // indici
```

è equivalente alle seg. espressioni (2) e (3):

```
(2)  (*(v+i) + j); //aritmetica dei punt.
(3)  (v[i] + j); //aritmetica dei punt.  e indici
```

Entrambe le espressioni rappresentano **l'indirizzo della cella agli indici (i,j)**.

Accesso ad indirizzi e valori

```
int v[ROWS][COLS];
```

La seg. expressione (4)

```
(4)  v[i][j]
```

è equivalente alle segg. espressioni (5) e (6) :

```
(5)  *(*(v+i) + j); //arimetica dei punt.
(6)  *(v[i] + j); //arit.  punt.  + operatore [] .
```

Entrambe le espressioni rappresentano il **valore contenuto nella cella agli indici (i,j)**.

Accesso ad indirizzi e valori

Le segg. espressioni rappresentano indirizzi di memoria e danno identico risultato (indirizzo del primo elemento della riga di indice i)

`(v+i)` oppure `v[i]`
equivale a ..
$\&v[0][0] + i \times COLS$.

Di conseguenza, applicando l'operatore di indirezione `*` è possibile ottenere l'indirizzo dello elemento v[i][j], ovvero `&v[i][j]`.

`(*(v+i) + j)` oppure `*v[i] + j`
equivale a ...
$\&v[0][0] + i \times COLS + j$.

Esempi svolti

15_01.c
15_02.c
PTR_01.c
...
PTR_13.c

Homework H15.1

Codificare una funzione in C che prenda in input una coppia di interi short con segno e restituisca il valore assoluto del prodotto di tali interi.

Invocare la funzione all'interno di una funzione main() con alcuni valori di test.

Homework H15.2

Modificare il precedente esercizio affinchè la funzione prenda in input, oltre alla coppia di interi short, anche un puntatore ad numero short unsigned.

La funzione dovrà scrivere il risultato della computazione nella locazione di memoria rappresentata dal puntatore in input. La funzione non avrà alcun tipo di ritorno.

Invocare la funzione all'interno di una funzione `main()` con alcuni valori di test. Stampare opportunamente i risultati.

Homework H15.3

Codificare una funzione in C che prenda in input un array di caratteri. Tale funzione dovrà restituire la lunghezza della stringa.

Homework H15.4

Codificare una funzione in C che prenda in input due array di caratteri. La funzione restituisca un risultato intero che rappresenta il confronto lessicografico tra le due stringhe (siano `s1` ed `s2` le due stringhe):

- -1 se $s1 < s2$;
- 1 se $s1 > s2$;
- 0 se $s1 = s2$.

Homework

NB: Specificare opportunamente tipo ed eventuali qualificatori dei parametri formali: i parametri formali dovranno/potranno essere `(char *)` oppure `(const char *)`? Perchè?

NB2: Per il calcolo della lunghezza della stringhe, usare la funzione del precedente esercizio.

Homework H15.5

Codificare una funzione C che prenda in input un array di caratteri ed un parametro che indica la sua lunghezza.

La funzione dovrà riempire l'array di caratteri con caratteri pseudo-casuali nell'intervallo [a-z].

La funzione dovrà restituire il numero di vocali all'interno della stringa.

Per l'accesso in scrittura agli elementi dell'array di caratteri, usare sia la notazione con '*' e aritmetica dei puntatori, sia la notazione con parentesi quadre.

NB: porre attenzione al carattere di terminazione stringa.

Homework H15.6

Codificare una funzione che prenda in input due array di interi A e B di eguale lunghezza, ed un parametro formale che indichi la lunghezza stessa degli array.

Tale funzione dovrà operare nel seguente modo (solo se $b_i \neq 0$):

- se $\frac{a_i}{b_i} < 1$, allora i) pone $b_i = a_i$; ii) scrive nella cella a_i un numero pseudo-casuale nel range [100,200];
- se $\frac{a_i}{b_i} > 1$, allora pone ii) $a_i = b_i$; ii) scrive nella cella b_i un numero pseudo-casuale nel range [0,100];
- se $\frac{a_i}{b_i} = 1$, allora i) pone $b_i = a_i$; ii) scrive nella cella a_i il numero 0.

La funzione restituisca il numero di volte in cui $\frac{a_i}{b_i} = 1$.

Homework H15.7

Codificare una funzione in C che prenda in input un array bidimensionale di dimensioni $N \times M$ di interi, ovvero una matrice, ed un numero double y.

La funzione restituisca il numero di colonne della matrice che soddisfano il seguente requisito: la media aritmetica degli elementi della colonna stessa sia minore o uguale ad y.

Testare la funzione mediante la codifica di una funzione main() nella quale si dichiari (allocazione automatica) un un array bidimensionale di dimensioni a piacimento. Riempire tale matrice con elementi pseudo-casuali in un range a piacere.

Homework H15.8

Codificare una funzione in C che prenda in input un array bidimensionale A di dimensioni $n \times n$ di double, ovvero una matrice quadrata, ed un parametro formale in virgola mobile δ.

la funzione restituisca il numero di elementi della diagonale secondaria tali che

$$|d_k - \frac{1}{n}\sum_{i=0}^{n-1} a_{ik}| < \delta$$

dove per d_k si intende l'elemento della diagonale secondaria appartanente alla colonna di indice k, e a_{ik} rappresenta il generico elemento della matrice stessa.

Homework H15.9

Codificare una funzione in C che prenda in input un array bidimensionale di dimensioni $n \times n$ di interi, ovvero una matrice quadrata.

La funzione restituisca la differenza, in valore assoluto, tra la media aritmetica degli elementi della diagonale principale, e la media aritmetica degli elementi della diagonale secondaria.

Homework H15.10

Codificare una funzione in C che prenda in input un array bidimensionale di dimensioni $n \times m$ di interi senza segno, due short unsigned mobile a e b (si assuma $0 \leq a < b \leq n$), ed un terzo parametro in virgola mobile α.

La funzione restituisca il numero di righe della matrice per cui vale la seguente proprietà: la media aritmetica degli elementi a_{ik} (i rappresenta l'indice della generica riga) tali che $a \leq k \leq b$ è un numero minore di α.

FINE