Linguaggio C: Funzioni

Valeria Cardellini

Corso di Calcolatori Elettronici
A.A. 2019/20

Funzioni in C

- Definizione di funzioni
- Passaggio dei parametri
- Esecuzione di una funzione
- Variabili dichiarate in una funzione e visibilità
- Gestione della memoria su chiamate di funzioni

Funzioni in C

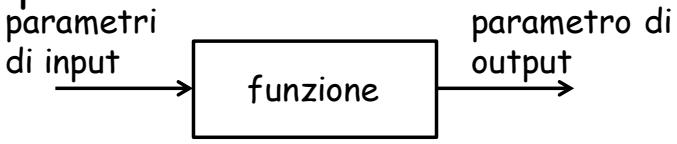
- Per progettare un programma complesso lo si suddivide in un insieme di componenti più piccoli ed indipendenti (*divide ed impera*)
  - Astrazione sui dati e astrazione funzionale
- In C l'astrazione funzionale si realizza attraverso la nozione di funzione
- **Funzione**: viene chiamata da un altro modulo di programma (`main` o altra funzione) per svolgere una specifica operazione
- Oltre a compiere una serie di azioni, la funzione comunica con l'ambiente chiamante:
  - ricevendo in ingresso una lista di parametri di input (o argomenti)
  - Restituendo, quando termina, un parametro di output

Funzioni in C

- Un programma in C è un insieme di funzioni
- Ogni funzione prende in ingresso una lista di argomenti e restituisce un valore
- In un programma deve esserci sempre la funzione principale `main()`
  - Finora negli esempi funzione main() e qualche funzione di libreria (ad es. input/ouput in #include <stdio.h>)
- Come si definiscono le funzioni?
- Come si usano le funzioni (chiamata o attivazione di funzione)?

Dichiarazione di funzioni in C

- Come le variabili che devono essere dichiarate prima di poter essere usate, anche le funzioni necessitano della dichiarazione
- La dichiarazione di una funzione deve specificare:
  - Il nome della funzione: identificatore che ne permette l'utilizzo nel programma
  - Gli argomenti (parametri di input) della funzione: quantità e tipo dei valori forniti in ingresso
  - Il tipo del valore in uscita (parametro di output)
- Sono ammesse funzioni senza parametri di input e/o di output



Esempi: dichiarazioni di funzioni in C

- Esempi di prototipi di funzioni

```
int fattoriale(int);    oppure
int fattoriale(int n);

int mcd(int, int);      oppure
int mcd(int a, int b);

int func(void);

void stampa_ris(double, int); oppure
void stampa_ris(double r, int a);
```

- Prototipo di funzione
  - Il nome dei parametri di input può essere incluso nel prototipo per documentare meglio il codice, comunque sarà ignorato dal compilatore

Definizione di funzione in C

```
tipoRisultato nomeFunzione(parametriFormali){
  istruzioni
}
```

- Intestazione e corpo della funzione
  - tipoRisultato: tipo del risultato restituito dalla funzione (es., `int`, `double`) oppure `void` se la funzione non restituisce risultato
  - nomeFunzione: nome della funzione
  - parametriFormali: lista (eventualmente vuota) di dichiarazioni di parametri (coppie tipo e nome) separate da ,
    - ✓ Se vuota può essere indicata con `void`
  - istruzioni: corpo della funzione contenente le istruzioni da eseguire alla chiamata della funzione stessa

Esempio: funzione in C

Codice sorgente: square.c

```
#include<stdio.h>

int square(int);  /* dichiarazione di funzione (o prototipo) */

int main(void) {
   int i;
   for (i=1; i<=10; i++)
     printf("Il quadrato di %d è %d\n", i, square(i));   /* chiamata
di funzione */
   return 0;
}

int square(int y) {      /* definizione di funzione */
  int r = y*y;
  return r;      /* alternativa più compatta: return y*y;
}
```

Altro esempio: lower2upper.c

Istruzione return

- Sintassi

  return espressione;

  - espressione: un'espressione il cui valore deve essere compatibile con il tipo del risultato dichiarato nell'intestazione della funzione

- Semantica
  - L'istruzione `return` eseguita all'interno di una funzione `func` termina l'esecuzione della stessa e restituisce il risultato nel punto del programma dove è stata invocata `func`

- Esempio

```
int main(void) {
    ...
    return 0;   /* 0 indica la terminazione normale */
}
```


Istruzione return

- L'esecuzione dell'istruzione `return` comporta la terminazione della funzione, anche se ci sono altre istruzioni che seguono
- Se il tipo del parametro di output di una funzione è `void`, allora `return` si può omettere, a meno che non si desideri interrompere l'esecuzione
  - Esempio: `return;`
- Se viene omesso `return` e la funzione restituisce un risultato diverso da `void`, si ha un warning del compilatore
  - warning: control reaches end of non-void function

Codice sorgente: err_return.c

Attivazione di funzioni

- Le funzioni vengono chiamate (attivate) all'interno di altre funzioni

```
int a, b, m;
...
a = 1; b = 2;
m = max(a, b);
```

- Sintassi per la chiamata di funzioni

  identificatore(parametriAttuali)
  - identificatore: nome della funzione
  - parametriAttuali: lista di espressioni separate da ,
- I parametri attuali devono corrispondere in numero, posizione e tipo ai paramenti formali
  - Errore: #parametri attuali < #parametri formali

    error: too few arguments to function call

Coercizione degli argomenti

- Una chiamata di funzione non corrispondente al prototipo provoca un errore di sintassi
- In alcuni casi sono tollerate chiamate con tipi diversi: coercizione degli argomenti
- Es.: sqrt ha come argomento un `double`, però:

```
int x = 4;
printf("%lf", sqrt(x));
```

  - stampa 2.00…0
  - Il compilatore promuove 4 a 4.0 e quindi la funzione viene chiamata correttamente
- Le conversioni avvengono secondo le regole di promozione del C già esaminate

Attivazione di funzioni

- Semantica dell'attivazione di una funzione g nella funzione f
  - L'attivazione di una funzione è un'espressione
  - Quando viene attivata g in f:
    - ✓ viene sospesa l'esecuzione di f e si passa ad eseguire le istruzioni di g (a partire dalla prima)
    - ✓ quando termina l'esecuzione di g, si torna ad eseguire le istruzioni di f
- Prima di poter essere chiamata, una funzione deve essere stata dichiarata oppure definita sopra
- Vedi esempi

Codice sorgente: massimo.c e massimo_prot.c

Visibilità di funzioni

- In C non è possibile definire funzioni all'interno di funzioni
- Prima di essere chiamata una funzione deve essere definita o dichiarata
  - Il compilatore deve controllare che i parametri formali corrispondano (per numero e tipo) ai parametri attuali
- Prima dell'attivazione deve essere conosciuto il prototipo
  - Se la funzione non è definita, almeno deve essere già dichiarata
- E' pratica comune usare quest'ordine:
  1. Dichiarazione di tutte le funzioni tranne main
  2. Definizione di main
  3. Definizione di tutte le altre funzioni

File header (o intestazioni)

- Ogni libreria standard ha un corrispondente file header contenente:
  - definizioni di costanti
  - definizioni di tipo
  - dichiarazioni di tutte le funzioni della libreria
- Esempi:
  - `<stdio.h>`: input/output
  - `<stdlib.h>`: memoria, numeri casuali, utilità generali
  - `<string.h>`: manipolazione di stringhe
  - `<limits.h>`: limiti del sistema per valori interi
  - `<float.h>`: limiti del sistema per valori reali
  - `<math.h>`: funzioni matematiche
- Il programmatore può scrivere una libreria e definire il corrispondente file header

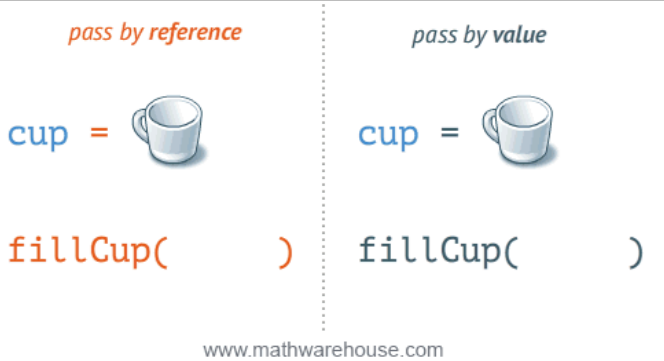
Passaggio dei parametri

- La definizione di una funzione presenta nell'intestazione una lista di parametri formali
- Questi parametri sono utilizzati come variabili all'interno del corpo della funzione
  - La chiamata di una funzione contiene i parametri da utilizzare come argomenti della funzione stessa
  - Questi parametri sono detti parametri attuali, per distinguerli dai parametri formali presenti nell'intestazione della definizione della funzione
- Quando, attraverso la chiamata, attiviamo una funzione, i parametri attuali vengono collegati (binding) ai parametri formali
- Come effettuare questo legame?

Passaggio dei parametri: per valore e per riferimento

- Esaminiamo due tecniche per il passaggio dei parametri
- Passaggio **per valore** (pass-by-value)
  - All'atto della chiamata viene preparata una **copia** dei valori dei parametri attuali: la funzione opera sulla copia, mentre il valore dei parametri attuali non viene modificato
- Passaggio **per riferimento** (pass-by-reference)
  - Alla funzione viene passato il **riferimento** alle variabili che costituiscono i parametri attuali: la funzione opera direttamente su di esse e quindi può modificarne il valore
- Il C prevede *esplicitamente* solo il passaggio dei parametri per valore
- Esistono anche altre tecniche (tra cui passaggio per copia/ripristino), non esaminate

Passaggio dei parametri: una spiegazione informale



https://www.mathwarehouse.com/programming/passing-by-value-vs-by-reference-visual-explanation.php

Passaggio dei parametri per valore

- **pa**: parametro attuale presente nella chiamata della funzione
- **pf**: corrispondente parametro formale nella definizione della funzione
- Al momento del passaggio dei parametri:
  1. Viene valutato pa (in generale, un'espressione)
  2. A pf viene associata una locazione di memoria
  3. Tale locazione viene inizializzata al valore di pa
- Durante l'esecuzione la funzione opera sui parametri formali, ovvero sulle locazioni a loro associate durante la chiamata, non modificando quindi il valore dei parametri attuali

Passaggio dei parametri per valore

- pf si comporta come una variabile dichiarata all'interno della funzione e creata al momento di ciascuna chiamata della funzione
  - La memoria è allocata per pf solo durante l'esecuzione della funzione
- Al termine dell'esecuzione della funzione, lo spazio di memoria allocato ai parametri formali viene deallocato e i valori dei parametri formali non sono più accessibili
- I valori delle variabili nelle espressioni che costituiscono i parametri attuali non vengono alterati durante l'esecuzione della funzione

Esempio: passaggio dei parametri per valore

```
#include <stdio.h>
int raddoppia(int);  /* Prototipo funzione */

int main(void)
{
   int a, b;
   a = 5;
   b = raddoppia(a + 1); /* Chiamata funzione, a+1 parametro attuale */
   printf("a = %d b = %d\n", a, b);
   return 0;
}

int raddoppia(int n) {   /* Definizione funzione, n parametro formale */
   return n*2;
}
```

Quali valori sono stampati per i parametri attuali a e b?

Codice sorgente: pass_value.c

Passaggio dei parametri per riferimento

- Nel passaggio per riferimento il pf indicato come riferimento non è, come nel passaggio per valore, una variabile locale inizializzata al valore del pa, ma è un riferimento alla variabile originale (pa) nell'ambiente della funzione chiamante
  - pa e pf si riferiscono alla stessa locazione di memoria
- Ogni modifica del parametro formale viene effettuata sul parametro attuale della funzione chiamante
  - Le modifiche fatte dalla funzione chiamata si propagano alla funzione chiamante

Passaggio dei parametri per riferimento

- Il passaggio dei parametri per riferimento
  - crea aliasing: il parametro formale è un nuovo nome per una locazione di memoria già esistente
  - produce effetti collaterali: le modifiche fatte attraverso il parametro formale si ripercuotono all'esterno
- Una qualche forma di passaggio per riferimento esiste in molti linguaggi, anche se spesso è "simulato" (nel caso del C tramite l'uso dei puntatori)

Passaggio dei parametri in C

- Perchè il C adotta il passaggio per valore?
  - È una tecnica sicura: le variabili della funzione chiamante e della funzione chiamata sono completamente disaccoppiate e non ci sono effetti collaterali
    - ✓ Gli effetti della chiamata sono espliciti e ben visibili: valore restituito e azioni dirette della funzione
  - La funzione chiamata può modificare il contenuto dei parametri formali, senza impatto sui valori dei parametri attuali nell'ambiente della funzione chiamante
- Il passaggio dei parametri per valore presenta però delle limitazioni
  - Superabili in C tramite i puntatori: passando per valore un puntatore, si passa in realtà il riferimento a una variabile

Limiti del passaggio per valore

- Il passaggio per valore ha una semantica *per copia*
- La comunicazione tra funzione chiamante e chiamata è unidirezionale (da funzione chiamante a funzione chiamata), eccetto per il valore di ritorno
- Impedisce a priori di scrivere una funzione che abbia come scopo quello di modificare i dati passati dalla funzione chiamante
- Impedisce di restituire alla funzione chiamante più di un risultato
  - Ad es. quoziente e resto di una divisione

Esempio: limiti del passaggio per valore

- Funzione per scambiare il valore di due variabili

Codice sorgente: pass_value_swap.c

```
#include <stdio.h>

void swap_val(int a, int b) {
   int aux;
   aux = a;
   a = b;
   b = aux;
}

int main() {
   int x = 5, y = 14;
   printf("prima:  x = %d, y = %d\n", x, y);
   swap_val(x, y);
   printf("dopo:   x = %d, y = %d\n", x, y);
   return 0;
}
```

Nessun effetto sui parametri attuali x e y!

Regole di visibilità

- Ogni dichiarazione in C ha un suo campo d'azione o visibilità (scope)
  - Visibilità: parte del programma in cui l'identificatore dichiarato può essere usato
- Visibilità nel file
- Visibilità nel blocco

Regole di visibilità: nel file

- Visibilità nel file
  - Identificatore dichiarato all'esterno di qualsiasi funzione, nella parte dichiarativa globale
  - Detto identificatore globale
- L'identificatore globale sarà noto ("accessibile") in tutte le funzioni la cui definizione è successiva
  - I prototipi di funzione, le variabili globali e le definizioni inseriti all'esterno di una funzione hanno visibilità nel file
- Esempio

```
#include <stdio.h>
int x = 1; /* var. globale, visibile in func e main */
int func(int, double);
int main(void) { . . . }
int func(int a, double y) { ... }
```

Regole di visibilità: nel blocco

- Visibilità di un identificatore dichiarato in un blocco è limitata al blocco stesso
  - Una variabile dichiarata nel blocco istruzioni di una funzione è detta variabile locale
    - ✓ Riferibile solo nel corpo della funzione
- Esempio: frammento di codice errato (errore in fase di compilazione)

```
int a;
scanf("%d", &a);
if (a > 10) {
  int b = 10;
}
printf("%d", b);
```

error: use of undeclared identifier 'b'

Regole di visibilità: nel blocco

- Esempio: variabile locale

```
int func(int a, double y) {
    int b = 1;   /* variabile locale alla funzione func;
                    inizializzata a 1 ogni volta che func
                    è chiamata */
    …
}
```

Esempio: regole di visibilità

```
#include <stdio.h>
int a = 1; /* Dichiarazione di variabile globale */

void func(void) {
   printf("Func: a = %d\n", a);
   // printf("b = %d", b); /* errore di compilazione, b è
         una variabile locale a main */
}

int main(void) {
   int b = 5; /* Dichiarazione di variabile locale */
   func();
   a = b;
   printf("Main: a = %d\n", a);
   return 0;
}
```

error: use of undeclared identifier 'b'

Codice sorgente: var_scope.c

Regole di visibilità: mascheramento

- Il mascheramento (shadowing) si ha quando all'interno di un blocco si dichiara un identificatore già dichiarato fuori dal blocco ed in esso visibile
  - La dichiarazione interna maschera quella esterna
  - La dichiarazione esterna è ripristinata all'uscita dal blocco
- Esempio

```
#include<stdio.h>
int x;
int main(void) {
  double x;
  ...
  return 0;
}
```

Esempio: mascheramento

```
#include <stdio.h>
int i = 1, j = 2; /* variabili globali */
int fm(void) {
   long i = 10; /* var. locale, maschera la var. globale i */
   int k = 3;
   { /* blocco interno a fm */
      float j = 8.5; /* var. locale, maschera la var. globale j */
      printf("blocco di fm: i = %ld, j = %f, k = %d \n", i, j, k);
   }
   printf("fm: i = %ld, j = %d, k = %d \n", i, j, k);
   return k*i;
} /* end f */

int main(void) {
   j = fm();
   printf("main: i = %d, j = %d\n", i, j);
   return 0;
}
```

Codice sorgente: shadowing.c

Tempo di vita di una variabile

- Le variabili locali (le locazioni di memoria associate) vengono:
  - Create al momento dell'attivazione di una funzione
  - Distrutte al momento dell'uscita dall'attivazione
- Quindi:
  - La funzione chiamante non può riferirsi ad una variabile locale alla funzione chiamata
  - Ad attivazioni successive di una stessa funzione corrispondono variabili locali allocate in locazioni di memoria diverse
- Nota: il tempo di vita di una variabile è un concetto rilevante a tempo di esecuzione (run-time)

Gestione della memoria a run-time

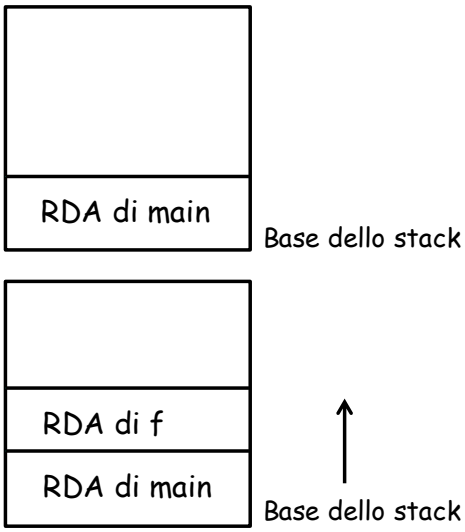
- Codice eseguibile e dati entrambi in memoria RAM, ma in aree separate
  - Memoria per il codice eseguibile determinata a tempo di compilazione
  - Memoria per dati locali alle funzioni (parametri formali e variabili) allocata e deallocata dinamicamente durante l'esecuzione
    - ✓ Spazio di memoria gestito a pila
    - ✓ Stack (pila): struttura dati con accesso LIFO (Last In First Out), operazioni di push e pop
- Durante l'esecuzione del programma viene gestito lo stack dei record di attivazione (RDA) in memoria
  - Per ogni chiamata di funzione viene allocato (push) un RDA in cima allo stack
  - Al termine della funzione chiamata il RDA viene deallocato (pop) dallo stack

Record di attivazione

- Ogni RDA (detto anche stack frame) contiene tutto quanto è necessario tenere in memoria per gestire la chiamata di una funzione:
  - Locazioni di memoria per i parametri formali (se presenti)
  - Locazioni di memoria per le variabili locali (se presenti)
  - Valore di ritorno della funzione
  - Indirizzo di ritorno (IDR): indirizzo di memoria della prossima istruzione della funzione chiamante

Record di attivazione

- Quando il programma viene lanciato, l'unica funzione attiva è main; nello stack è riservato spazio per le variabili locali a main
- Se durante l'esecuzione di main viene chiamata la funzione f, viene allocato un nuovo RDA per f
- Il record di attivazione di f contiene anche l'IDR all'istruzione del main successiva alla chiamata



Esempio: record di attivazione

Codice sorgente: rda.c

```
#include <stdio.h>
void f(int q) {
   printf("f(): il parametro q vale %d\n", q);
   return;
}
void g(int p) {
   int loc;
   printf("g(): il parametro p vale %d\n", p);
   loc = 5;
   printf("g(): chiamo f() con parametro attuale pari a %d\n", loc);
   f(loc);
   printf("g(): di nuovo in esecuzione\n");
   return;
}
int main(void) {
   int s;
   printf("main(): in esecuzione\n");
   s = 2;
   printf("main(): chiamo g() con parametro attuale pari a %d\n",s);
   g(s);
   printf("main(): di nuovo in esecuzione\n");
   return 0;
}
```

Esempio: record di attivazione

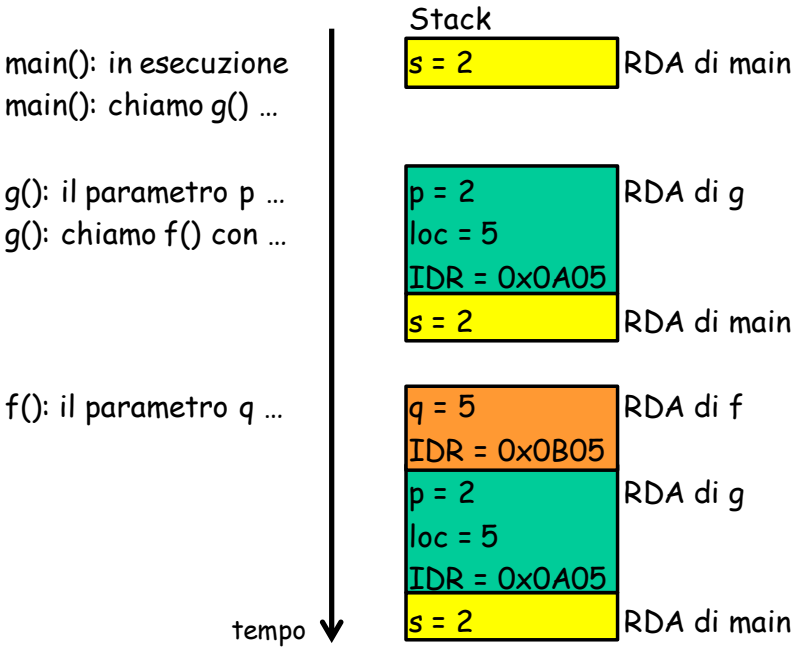
- Codice in linguaggio macchina in memoria:

main()		g()			f()	
0x0A00	m1	0x0B00	g1		0x0C00	f1
0x0A01	m2	0x0B01	g2		0x0C01	f2
0x0A02	m3	0x0B02	g3		0x0C02	
0x0A03	m4	0x0B03	g4			
0x0A04	m5 Chiamata g(s)	0x0B04	g5	Chiamata f(loc)		
0x0A05	m6	0x0B05	g6			
0x0A06	m7	0x0B06	g7			

- L'esecuzione del programma richiede:
  - Program Counter (PC)
  - Stack di RDA

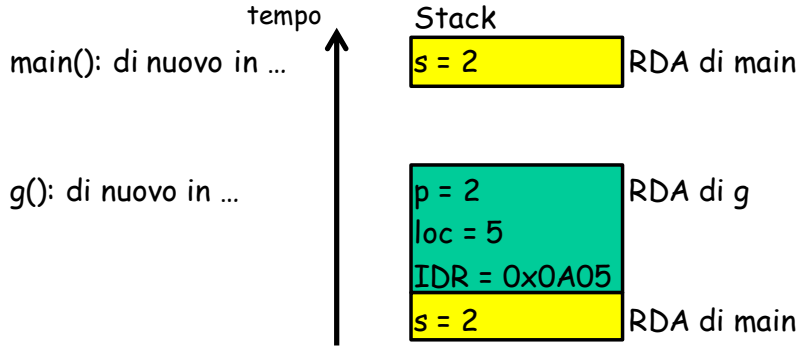
Esempio: record di attivazione

- Esecuzione del programma:



Esempio: record di attivazione

- Esecuzione del programma:



Continua da slide precedente

Esempio: RDA e passaggio dei parametri

- Alla chiamata della funzione g()
  - viene creato un nuovo RDA per la funzione
  - viene copiato il valore dei parametri attuali nelle locazioni di memoria riservate ai parametri formali della funzione (passaggio di parametri)

RDA di g

p = 2	Locazione riservata al parametro formale p
loc = 5	Locazione per la variabile locale loc
IDR = 0x0A05	Indirizzo di ritorno

Variabili automatiche e statiche

- Tempo di vita di una variabile: periodo in cui esiste la cella di memoria associata alla variabile
  - Coincide con il periodo in cui esiste il RDA relativo (variabili locali automatiche)
- Una variabile statica esiste per tutto il tempo di esecuzione del programma:
  - se è dichiarata all'esterno di ogni funzione
  - se è locale ad una funzione ma lo specificatore static precede la dichiarazione
  - Es. `void f() { static int x; … }`
    - ✓ La variabile x viene inizializzata alla prima attivazione della funzione
    - ✓ Conserva il valore tra attivazioni successive
    - ✓ E' locale (visibile solo nella funzione in cui è dichiarata)

Esempio: variabile statica

```
#include <stdio.h>
void f() {
   int a = 0;
   static int b = 0;
   printf("\n a:%d b:%d", a, b);
   a++;
   b++;
}

int main(void) {
   int j;
   for (j = 0; j < 3; j++)
      f();
   return 0;
}
```

Il programma stampa
a:0 b:0
a:0 b:1
a:0 b:2