Le funzioni in C

DD Cap.5 pp.131-160
KP Cap.5 pp. 175-199

Introduzione

La maggior parte dei programmi scritti per risolvere problemi sono (molto!) più lunghi di quelli visti fin ora.

Il modo migliore per sviluppare e amministrare grandi programmi è quello di dividerli in moduli.

In C i vari moduli si chiamano funzioni.

I programmi C si scrivono combinando le funzioni scritte dal programmatore con quelle della libreria standard.

La suddivisione in funzioni serve, oltre che per maneggiare meglio il programma, per questioni di riusabilità del software.

Abbiamo già usato la funzione `main()`; per le altre…

Funzioni

DD p.133,
KP p.175

Problema: stampa i quadrati dei primi 10 interi

```
#include <stdio.h>

int quadrato(int y);

int main()
{
   int i;
   for (i=1; i<=10; i++)
      printf("%d ", quadrato(i));
   return(0);
}

int quadrato(int y)
{
   return y*y;
}
```

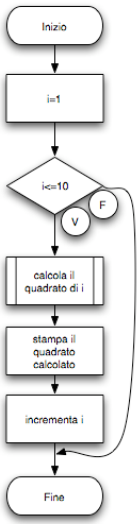
prototipo

chiamata

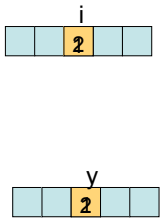
tipo ritornato

funzione

parametro



Cosa succede…




```
   #include <stdio.h>

   int quadrato(int y);

   int main()
   {
1     int i;
6 2   for (i=1; i<=10; i++)
7 3      printf("%d ", quadrato(i));
      return(0);
   }

8 4 int quadrato(int y)
   {
9 5   return y*y;
   }
```

e così via…

Chiamata di funzione

- Nell'esempio, quando la funzione viene richiamata, una copia del valore di i viene posta nel parametro y; il risultato della funzione viene ritornato alla funzione chiamante quando si giunge all'istruzione `return` dentro la funzione.
- In generale, quando il controllo del programma incontra un nome di funzione, tale funzione viene chiamata, e il controllo passa ad essa. Al termine della funzione il controllo ritorna all'ambiente chiamante, che riprende il suo lavoro.
- Le funzioni vengono chiamate scrivendo il loro nome seguito da un opportuno elenco dei parametri tra `()`. Questi parametri devono corrispondere, in numero ed in tipo, ai parametri formali presenti nella definizione della funzione.

Prototipo di funzione (1) DD p.139

- Un prototipo indica al compilatore che nel programma esiste quella funzione, ed informa sul numero, il tipo e l'ordine dei parametri. Il compilatore utilizzerà i prototipi per convalidare le chiamate di funzione.
- Non è necessario mettere nel prototipo il nome dei parametri (il compilatore li ignora) ma è meglio per ragioni di chiarezza.
- Una chiamata di funzione che non corrisponda al suo prototipo è un errore di sintassi.

Prototipo di funzione (2)

- I valori degli argomenti che non corrispondano precisamente ai tipi di parametri definiti nel prototipo saranno convertiti prima che la funzione sia richiamata (coercizione degli argomenti).
  Attenzione al troncamento! E' di solito scorretto convertire dati in un tipo più basso.
- Lo stesso accade per il valore ritornato: se il tipo non è uguale a quello dichiarato, viene riconvertito.
- Per le funzioni di libreria standard, i prototipi sono nei files inclusi nel programma dalla direttiva `#include`
- Il programmatore può inserire i suoi prototipi in un file con estensione `.h` da includere con `#include` (in tal caso "" e non `<>`!!)
- N.B. ricordare sempre il `;` dopo il prototipo, perché è quello che distingue l'intestazione di una funzione da un prototipo!

Definizione di funzione (1) KP p.182

- La definizione di funzione è la parte di programma che contiene l'intestazione e le istruzioni della funzione.
- Poiché l'intestazione è identica al prototipo (tranne il `;`), il prototipo potrebbe essere omesso, perché il compilatore deduce le stesse informazioni anche dall'intestazione, ma in tal caso il main deve essere l'ultima funzione perché tutte le funzioni vanno definite prima di essere chiamate.
- E' buona regola NON omettere i prototipi, anche se sono opzionali.

Definizione di funzione (2)

Alcuni esempi di definizioni di funzioni:

```
void nothing(void){ }
```

```
double twice(double x)
{
 return (2.0 *x);
}
```

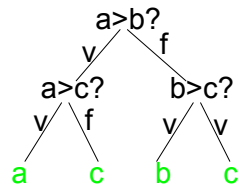
N.B. se la funzione non restituisce alcun valore, si ha il tipo void. Analogamente, se la funzione non prende alcun paramtero.
Il tipo void può essere omesso come parametro, ma non come valore ritornato. Infatti, se il tipo della funzione non è specificato nella definizione, allora esso è int.

Per non sbagliare: non omettere mai i tipi!

Esempio

```
/*massimo di 3 interi*/#include<stdio.h>
   int maggiore(int x, int y);

   main()
   {
     int a; int b; int c;
     printf("scrivi 3 interi");
     scanf("%d %d %d", &a, &b, &c);
     if (a==maggiore(a,b))
        printf("%d", maggiore(a,c));
     else printf("%d", maggiore(b,c));
   }

   int maggiore(int x, int y)
   {
   if (x>y) return x;
     else return y;
   }
```

Problema:
presi in input 3 interi, stamparne il maggiore



return (1) KP p. 178

- L'istruzione return permette di restituire un valore alla funzione chiamante.
- Può includere un'espressione facoltativa
- Esempi:
  return;
  return ++a;
  return(a+b);
- Le () sono facoltative
- Quando un'istruzione return viene raggiunta, l'esecuzione della funzione si conclude e il controllo torna alla funzione chiamante
- Una funzione può avere 0 o tante istruzioni return; se non ci sono return il controllo torna alla funzione chiamante quando si raggiunge la } che conclude la funzione

return (2)

- Quando si è in presenza dell'istruzione return all'interno della parte *then* di un if, si può omettere la parola chiave else: il flusso del programma è identico.
- Esempio:

```
int maggiore(int x, int y)
{
if (x>y) return x;
  else return y;
}
```

comunque si entra qui quando non è x>y

```
int maggiore(int x, int y)
{
if (x>y) return x;
return y;
}
```

variabili locali e globali KP p.177

Le variabili dichiarate nel corpo di una funzione vengono dette variabili locali, quelle dichiarate esternamente sono dette variabili globali.

```
#include <stdio.h>
int a=33; /*esterna al main: globale */

int main()
{
  int b=7; /* interna al main: locale */

  printf("a=%d\nb=%d\n", a,b);
  return 0;
}
```

Locali:
quando si entra nel blocco in cui sono definite vengono allocate; la memoria viene rilasciata quando si esce dal blocco. Se si rientra vengono ridefinite senza poter recuperare il valore precedente.

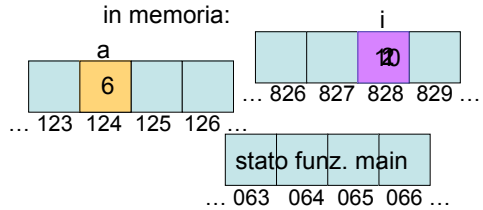
Globali:
rimangono attive per tutta la durata del programma

Un esempio

Problema: prendere in input un intero e stampare la sua tabellina

```
#include <stdio.h>
int a; /*esterna */

void stampa_tab(int x);

int main()
{
  printf("tabellina del ?");
  scanf("%d", &a);
  stampa_tab(a);
  return 0;
}
```

```
void stampa_tab(int x)
{
int i; /*interna alla funz.: locale*/

for (i=1; i<=10; i++)
  printf("%d x %d = %d \n", x,i,x*i);
}
```



Un altro esempio

Cosa stampa il seguente programma?

```
#include <stdio.h>
int z;

void f(int x)
{
  x=2;
  z+=x;
}

int main()
{
  z=5;
  f(z);
  printf("z=%d\n", z);
  return 0;
}
```

Attenzione a come si comportano le variabili globali e quelle locali…

z=7

Classi di memoria (1) DD p.154

Oltre al nome, al tipo, al valore ed all'indirizzo, le variabili hanno altri attributi: la classe di memoria, la permanenza in memoria e la visibilità.
Il C fornisce 4 classi di memoria: auto, register, extern, static.
La permanenza in memoria di una variabile è il periodo durante il quale la variabile esiste in memoria. Le quattro classi di memoria si possono dividere in due tipi di permanenza: permanenza automatica e permanenza statica.
La visibilità di una variabile è l'insieme dei punti del programma in cui è possibile usare la variabile.

Classi di memoria (2)

auto e register sono usate per dichiarare variabili con permanenza automatica in memoria. Esse vengono create ogni qual volta si entri nel blocco in cui sono state dichiarate, esistono finché resti attivo il blocco e vengono distrutte quando vi si esca.
le variabili locali hanno per default una permanenza automatica, perciò la parola chiave auto non si usa quasi mai.
Esempio: auto int a;
la dichiarazione register suggerisce al compilatore di mettere una variabile in un registro hardware ad alta velocità, per eliminare il costo di caricamento.
N.B. il compilatore può ignorare la dichiarazione register

Classi di memoria (3)

extern e static sono usate per dichiarare variabili con permanenza statica in memoria. Le variabili con permanenza statica esistono dal momento in cui il programma inizia, ma non è detto che possano essere usate per tutto il programma, perché la permanenza e la visibilità sono cose diverse!!
le variabili globali hanno per default una permanenza statica e sono definite extern.
la parola chiave extern viene usata anche quando il programma è scritto su più file e la variabile è dichiarata altrove

Classi di memoria (4)

le variabili static sono visibili alla sola funzione in cui sono definite, ma il loro valore permane in memoria per la prossima chiamata di funzione.

```
void f(void)
{
static int cont=0;

if (cont==0)
   printf("non mi avevi mai chiamato");
else
   printf("mi avevi già chiamato");
cont++;
}
```

Può essere utile definire delle variabili locali come static quando si vuole "tramandare" un valore da una chiamata all'altra

Attenzione: l'inizializzazione a 0 viene effettuata solo la prima volta, poi viene mantenuto il valore corrente!

Passaggio dei parametri

I parametri passati vengono usati ma non modificati nella funzione chiamante, anche se vengono modificate nella funzione chiamata:
Esempio:

```
...
/* chiamata */
x=somma(a,b);
...
/* funzione */
int somma(int a, int b)
{
   a=a+b;
   return a;
}
```

✓prima della chiamata:
a=2; b=4; x=12;
✓dopo la chiamata?
a=2; b=4; x=6;
Infatti a e b sono state solo copiate (le avremmo potute chiamare x ed y: MEGLIO per evitare confusione)

Funzioni che non restituiscono valori e non prendono parametri: un esempio

```
#include <stdio.h>
void mess_err(void);

int main()
{ int a,b,c;

  printf("inserire dividendo");
  scanf("%d", &a);
  printf("inserire divisore");
  scanf("%d", &b);
  if (b!=0)
    {
      c=a/b;
      printf("%d / %d = %d", a,b,c);
    }
  else mess_err();
return 0;
}
```

```
void mess_err(void)
{
  printf("errore: denomin. nullo");
}
```

Chiamata per valore e per riferimento DD p.144

- chiamata per valore: viene fatta una copia della variabile passata.
- La chiamata di default nel C è per valore (per evitare effetti collaterali)
- chiamata per riferimento (o per variabile): viene modificata la variabile passata
- Nel C la chiamata per riferimento è ottenibile mediante l'utilizzo dei puntatori.
- A volte si può evitare la chiamata per riferimento utilizzando variabili globali (specie se i valori da modificare sono molti!), ma è -in genere- sconsigliato!

Ricorda: in un programma ben strutturato le funzioni dovrebbero comunicare solo tramite i parametri.

Un cenno sui puntatori (1)

Una variabile di tipo puntatore contiene l'indirizzo ad una zona di memoria.

Esempio: `int *x, *y;` questa dichiarazione definisce due variabili puntatore `x` ed `y` che contengono l'indirizzo di due zone di memoria la cui dimensione è tale da contenere degli interi. Per accedere a tali interi si usa l'operatore unario *. Con `*x` si fa riferimento alla variabile intera puntata da `x`.

Esempio: `*x=10;` assegna il valore 10 alla variabile puntata da `x`.

Un cenno sui puntatori (2)

Una variabile di tipo puntatore si comporta, dal punto di vista dell'assegnamento, come tutte le altre variabili, purché il suo contenuto sia un indirizzo di memoria.

Esempio: `x=10;` modifica `x` in modo che esso punti alla posizione di memoria 10, ma questa assegnazione non ha senso anche perché il tipo di `x` e di 10 sono diversi. Se proprio vogliamo fare una simile assegnazione dobbiamo usare il cast.

Esempio: `x=(int*)10;`

Un cenno sui puntatori (3)

L'operatore unario & si comporta in modo inverso rispetto a * ed, applicato ad una variabile, ne restituisce l'indirizzo in memoria, cioè il puntatore che la punta.

Esempio: `int *x, y;`

`x` è un puntatore, mentre `y` è un intero.

Esempio: `x=&y;`

Esempio: `*x=5;` è equivalente a `y=5;`

Un esempio

```
#include <stdio.h>
void try_to_change_it(int);

int main()
{
  int a=1;
  printf("%d", a); /*stampa 1 */
  try_to_change_it(a);
  printf("%d", a); /*stampa ancora 1 */
  return 0;
}

void try_to_change_it(int a)
{
  a=20;
}
```

il tipo void esprimere il fatto che la funzione non ritorna nulla

il C chiama di default per riferimento

Un altro esempio (1)

```
#include <stdio.h>
void try_to_change_it(int *);

int main()
{
  int a=1;
  printf("%d", a); /*stampa 1 */
  try_to_change_it(&a);
  printf("%d", a); /*stampa 20 */
  return 0;
}

void try_to_change_it(int *ind)
{
  *ind=20;
}
```

il parametro è l'indirizzo di memoria di una variabile intera

invece di passare a si passa l'indirizzo di a

ora la chiamata è per riferimento!

non si modifica il parametro ind, che è un indirizzo, ma ciò che la cella corrispondente contiene

Un altro esempio (2)

```
#include <stdio.h>
void try_to_change_it(int *);

int main()
{
  int a=1;
  printf("%d", a); /*stampa 1 */
  try_to_change_it(&a);
  printf("%d", a); /*stampa 20 */
  return 0;
}

void try_to_change_it(int *ind)
{
  *ind=20;
}
```

leggi: indirizzo di a

leggi: cella indirizzata da ind

a

20

00101

(parametro passato: 00101)

N.B. NON abbiamo modificato il parametro!

ind

00101

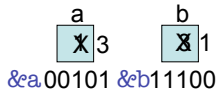
11100

(si modifica il contenuto della cella indirizzata da ind, cioè a)

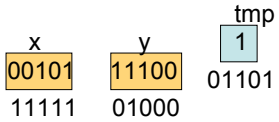
funzione di scambio

```
#include <stdio.h>
void scambia(int *, int *)

int main()
{
   int a=1, b=3;
   scambia(&a,&b);
   printf("%d %d", a, b);
   return 0;
}

void scambia(int *x, int *y)
{
   int tmp;
   tmp=*x; *x=*y; *y=tmp;
}
```

Problema: date due variabili locali al main, scrivere una funzione che le scambi



(parametri passati: 00101 e 11100)



(si modifica il contenuto delle celle indirizzate da x e y, cioè a e b)

Un altro esempio (1)

```
#include <stdio.h>
int cuboPerVal(int n);

int main()
{
   int num=5;
   printf("vecchio num %d", num);
   num=cuboPerVal(num);
   printf("nuovo num %d", num);
   return 0;
}

int cuboPerVal(int n)
{
   return n*n*n;
}
```

Problema: elevare al cubo una variabile usando una chiamata per valore

Cosa stampa questo programma?

vecchio num 5
nuovo num 125

Un altro esempio (2)

```
#include <stdio.h>
void cuboPerRif(int *nPtr);

int main()
{
   int num=5;
   printf("vecchio num %d", num);
   cuboPerRif(&num);
   printf("nuovo num %d", num);
   return 0;
}

void cuboPerRif(int *nPtr)
{
   *nPtr= *nPtr * *nPtr* *nPtr;
}
```

Problema: elevare al cubo una variabile usando una chiamata per riferimento

Cosa stampa questo programma?

vecchio num 5
nuovo num 125

Esercizi (funzioni) (1)

Dopo aver costruito il diagramma di flusso per ciascun problema:

- Scrivere un programma che prenda in input un insieme di interi terminati da 0 e, per ciascuno, ne stampi il quadrato.
  Il programma deve contenere una funzione di lettura intero, una di calcolo quadrato ed una di stampa intero.
- Scrivere un programma che prenda in input un insieme di interi terminati da 0, e ne stampi la media. Il programma deve contenere una funzione lettura intero, una di somma di due interi ed una funzione che conta gli interi in input in una variabile locale al main.
- Sia n_0 un numero intero positivo. Per i>0 si ponga:
  n_(i+1)=n_i/2 se n_i pari
  n_(i+1)=3n_i+1 se n_i dispari
  La sequenza termina quando n_i vale 1.Scrivere un programma che scriva tutta la sequenza a partire da n_0 dato in input.

Esercizi (funzioni) (2)

Dopo aver costruito il diagramma di flusso per ciascun problema:

- Scrivere un programma che:
  - prenda in input 5 reali che rappresentano i coefficienti di un polinomio di quarto grado;
  - prenda in input un valore reale x;
  - restituisca in output il valore del polinomio in x.

  Il programma deve contenere una funzione che calcola il valore a cui siano passati tutti i valori di cui ha bisogno (no variabili globali).
- Scrivere un programma che:
  - prenda in input 10 interi che rappresentano i coefficienti di due polinomi di quarto grado;
  - restituisca in output i coefficienti del polinomio somma.

  Il programma deve contenere una funzione somma che viene richiamata più volte ed i coefficienti del polinomio risultante NON devono essere salvati in alcuna variabile.
- Scrivere un programma che:
  - prenda in input 6 interi che rappresentano i coefficienti di due polinomi di secondo grado;
  - restituisca in output i coefficienti del polinomio prodotto.

  Il programma deve usare la funzione somma dell'esercizio precedente e i coefficienti del polinomio risultante NON devono essere salvati in alcuna variabile.