CORSO DI PROGRAMMAZIONE A-L
A.A. 2016-17

Dispensa 17

Laboratorio

Dott. Mirko Ravaioli
e-mail: mirko.ravaioli@unibo.it

17 Le funzioni

Una funzione è una sezione con nome indipendente di codice C che segue un compito specifico e restituisce opzionalmente un valore al programma chiamante.

Analizziamo un esempio con di codice con una funzione:

```
1:   /*Esempio di una semplice funzione*/
2:       #include <stdio.h>
3:
4:       long cubo(long x);
5:
6:       long input, risp;
7:
8:       main()
9:       {
10:             printf("Immettere un valore intero: ");
11:             scanf("%d", &input);
12:             /*Nota: %ld è l'indicatore di convers. per un intero lungo*/
13:             risp = cubo(input);
14:             printf("\nIl cubo di %ld è %ld.\n",input, risp);
15:
16:             return 0;
17:      }
18:
19:      /*funzione cubo() – Calcola il valore al cubo della variabile*/
20:      long cubo(long x)
21:      {
22:             long x_cubo;
23:
24:             x_cubo = x * x * x;
25:             return x_cubo;
26:      }
```

Ecco l'output del programma:

Esempio 1:

```
Immettere un valore intero: 100
Il cubo di 100 è 1000000
```

Esempio 2:

```
Immettere un valore intero: 9
Il cubo di 9 è 729
```

La riga 4 contiene il prototipo della funzione, il modello che apparirà nella parte seguente del programma. Un prototipo contiene il nome della funzione, un elenco di variabili che verranno passate ed eventualmente il tipo della variabile restituita. Osservando la riga 4 si può vedere che la funzione si chiama "cubo", che richiede una variabile di tipo "long" e che restituisce un valore di tipo "long". Le variabili passate ad una funzione sono dette argomenti e sono racchiuse tra le parentesi tonde che seguono il nome della funzione stessa. In questo esempio l'unico argomento della funzione è *long x*. La parola chiave che precede il nome della funzione indica il tipo di variabile restituito. In questo caso viene restituito una variabile di tipo *long*.

La riga 13 richiama la funzione *cubo* e le passa il valore introdotto da tastiera come argomento. Il valore restituito dalla funzione viene assegnato alla variabile *risp*.

La *definizione della funzione* in questo caso è contenuta nelle riche dalla 20 alla 26. Come il prototipo anche la definizione della funzione è composta da più parti. La funzione comincia con un'*intestazione* alla riga 20, contenente il nome della funzione, il tipo e la descrizione degli eventuali argomenti. Si noti che l'istruzione della funzione è identica al prototipo (a parte il punto e virgola).

Il corpo della funzione, nelle righe dalla 21 alla 26.è racchiuso all'interno di parentesi graffe e contiene istruzioni.

Le *variabili locali* sono quelle dichiarate all'interno del corpo della funzione. Infine, la funzione termina con un'istruzione *return* alla riga 25 che indica la conclusione della routine. L'istruzione *return* serve anche per passare il valore restituito al programma chiamante.

I programmi in C eseguono le istruzione contenute all'interno delle funzioni solo al momento in cui avviene la chiamata; in questo istante, il programma ha la possibilità di inviare delle informazioni alla funzione sotto forma di argomenti, per specificare i dati richiesti dalla procedura per eseguire il proprio compito. In seguito vengono eseguite le istruzioni interne fino alla conclusione della procedura, momento in cui il flusso di esecuzione torna al programma principale nell'istruzione successiva alla chiamata. Le istruzioni possono passare al programma chiamante le informazioni elaborate attraverso una gestione particolare del valore restituito.

Una funzione può essere chiamata tutte le volte che serve; inoltre le funzioni cono richiamabili in qualsiasi ordine.

Il prototipo di una funzione fornisce al compilatore un'anteprima della funzione che verrà definita in una parte seguente del codice.

Esempi di prototipi:

```
double quadrato(double numero);

void stampa_relazione(int numero_relazione);

int get_scelta_menu(void);
```

Esempi di definizione:

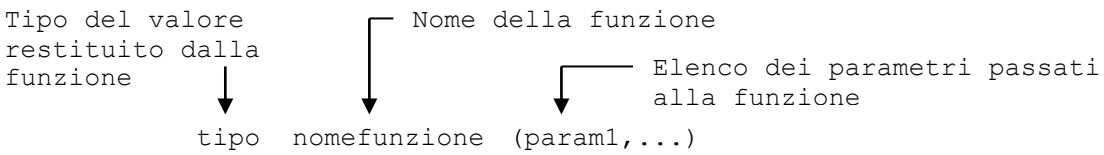
```
double quadrato(double numero)            /*intestazione*/
{                                         /*parentesi aperta*/
      return numero * numero;             /*corpo della funzione*/
}                                         /*parentesi chiusa*/


void stampa_relazione(int numero_relazione)
{
      if (numero_relazione == 1)
            printf("Stampo relazione 1");
      else
            printf("Non stampo relazione 1");
}
```

Attraverso l'impiego delle funzioni all'interno dei propri programmi in C, è possibile fare uso della *programmazione strutturata*, dove qualsiasi compito di un programma viene eseguito da una sezione indipendente.

Come si scrive una funzione

Il primo passo dello sviluppo di una funzione è la definizione del suo obiettivo.

- ***Intestazione***: la prima riga di qualsiasi funzione è l'intestazione, la quale si compone di tre parti ciascuna asservita ad uno scopo preciso.



  - *Tipo del valore restituito dalla funzione*: corrisponde al tipo di dato che la funzione restituisce al programma chiamante.

  - *Nome della funzione*: ad una funzione può essere assegnato un nome qualsiasi, a patto che segua le regole del C riguardanti i nomi di variabile. Il nome della funzione deve essere unico.

  - *L'elenco dei parametri*: molte funzioni utilizzano degli argomenti passati nella chiamata. Una funzione deve sapere che tipo di argomenti aspettarsi, cioè il tipo di dato di ciascuno. E' possibile passare a una funzione C qualsiasi tipo di dati. Per ciascun argomento passato alla funzione, l'elenco dei parametri deve contenere un oggetto distinto.

- ***Corpo della funzione***: è contenuto all'interno di una coppia di parentesi graffe e segue immediatamente l'intestazione. All'interno del corpo delle funzioni è possibile dichiarare delle variabili locali, così chiamate in quanto sono disponibili solo alla funzione di appartenenza e sono distinte dalle altre variabili con lo stesso nome dichiarate altrove. Per restituire un valore da una funzione si deve utilizzare la parola chiave *return* seguita da un'espressione del C. Una funzione può contenere più di una istruzione *return*, solo la prima ad essere eseguita è quella che conta.

```
#include <stdio.h>

int maggiore(int a, int b);

main()
{
   int x, y, z;

      puts("Inserire due valori interi diversi");
      scanf("%d%d",&x, %y) ;

      z = maggiore(x,y);

      printf("\nIl valore maggiore è %d.",z);

      return 0;
}

int maggiore(int a, int b)
{
      if (a > b)
            return a;
```

```
            else
            return b;
}
```

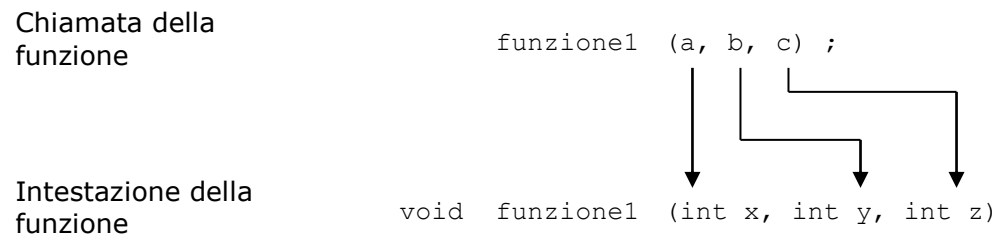
- ***I prototipi delle funzioni***: i prototipi di una funzione sono identici alle intestazioni, ma sono seguiti da un punto e virgola. I prototipi forniscono al compilatore informazioni sul tipo di valore restituito, sul nome e sui parametri.

Passaggio degli argomenti

Per passare degli argomenti a una funzione è necessario elencarli all'interno delle parentesi che seguono il nome della funzione stessa. Il numero, il tipo e l'ordine degli argomenti devono coincidere con quelli del prototipo e dell'intestazione.

Ad esempio, se una funzione viene definita con due parametri di tipo *int*, occorre passarle esattamente due argomenti *int*, ne uno di più ne uno di meno, e del tipo corretto.

Se si cerca di passare a una funzione un numero o un tipo sbagliato di argomenti, il compilatore se ne accorge sulla base delle informazioni del prototipo.



Chiamata alle funzioni

Esistono due modi di chiamare una funzione. Qualsiasi funzione può essere richiamata specificandone il nome e l'elenco di argomenti come un'unica istruzione. Ad esempio:

```
areaRettangolo(5, 10);
```

Se la funzione restituisce un valore, questo viene ignorato.

Il secondo metodo può essere utilizzato con le funzioni che restituiscono un valore. Dato che queste funzioni equivalgono a un valore (quello restituito), sono a tutti gli effetti delle espressioni del C e possono essere utilizzate ovunque possa essere impiegata un'espressione. Nell'esempio seguente, *areaRettangolo()* è utilizzato come parametro di una funzione:

printf("L'area del rettangolo di base %d e altezza %d è: %d", b, h, areaRettangolo(b,h));

Viene chiamata per prima la funzione *areaRettangolo()* con i parametri *b* e *h*, quindi viene richiamata la funzione *printf()* utilizzando i valori di *b*, *h* w *areaRettangolo()*.

Chiamata alle funzioni

Il termine *ricorsione* si riferisce alle situazioni in cui una funzione richiama se stessa in maniera sia diretta che indiretta. La ricorsione indiretta avviene quando una funzione ne richiama un'altra che al suo interno richiama la funzione primaria.

Il programma presentato nell'esempio sotto, presenta un esempio di chiamata ricorsiva.

In particolare viene calcolato il fattoriale di un numero. Il fattoriale di un numero x, che ha per simbolo x! si calcola nel modo seguente:

```
x! = x *(x - 1) * (x - 2) * (x - 3) * . . . * (2) * 1
```

E' possibile calcolare x! anche nella maniera seguente:

```
x! = x * (x - 1)!
```

Avanzando di un altro passo, (x - 1)! Può essere calcolato utilizzando la stessa procedura:

```
(x - 1)! = (x - 1) * (x - 2)!
```

```
#include <stdio.h>

unsigned int fattoriale(unsigned int a);

main()
{
      unsigned int f, x;

      puts("inserire un valore intero compreso tra 1 e 8:");
      scanf("%d",&x);

      if (x > 8 || x < 1)
            printf("Solo I valori compresi tra 1 e 8 sono ammessi!!");
      else
      {
            f = fattoriale(x);
            printf("%u fattoriale valr %u",x,f);
      }

      return 0;
}

unsigned int fattoriale(unsigned int a)
{
      if (a == 1)
            return 1;
      else
      {
            a *= fattoriale(a - 1);
            return a;
      }
}
```

Ambito delle variabili

L'*ambito* di una variabile si riferisce all'estensione entro cui parti differenti di n programma hanno accesso a una determinata variabile; in altre parole, da dove la variabile risulta *visibile*. Quando si parla di variabili del C, i termini *accessibilità* e *visibilità* vengono utilizzati in modo intercambiabile. In riferimento all'ambito, con il

termine *variabile* si intende qualsiasi tipo di dato previsto dal linguaggio C; variabili scalari, array, strutture, puntatori e così via, oltre che costanti simboliche definite tramite la parola chiave *const*.

Inoltre, l'*ambito* influisce sul tempo di vita di una variabile: quanto tempo la variabile persiste nella memoria, oppure, quando viene allocato o deallocato dello spazio in memoria per la variabile stessa.

Esempio di ambito:

```
/*Versione 1 del listato*/

#include <stdio.h>

intx = 999;

void stampa_valore(void);

main()
{
      printf("%d\n",x);
      stampa_valore();

      return 0;
}

void stampa_valore(void)
{
      printf("%d\n",x);
}
```

Ecco l'output del programma:

```
999
999
```

```
/*Versione 2 del listato*/

#include <stdio.h>

void stampa_valore(void);

main()
{
      intx = 999;

      printf("%d\n",x);
      stampa_valore();

      return 0;
}

void stampa_valore(void)
{
      printf("%d\n",x);
}
```

Se si prova a compilare il listato 2, viene generato un messaggio di errore del tipo:

```
Error: undefined identifer 'x'.
```

L'unica differenza tra I due listati consiste nella posizione in cui la variabile x viene definite. Spostando la definizione di x in punti differenti del codice, se ne modifica l'ambito. Nel listato 1, x rappresenta una *variabile esterna* e il suo ambito si estende all'intero programma. Nel listato 2, x rappresenta una *variabile locale* e come tale il suo ambito è limitato all'interno della funfione *main()*.

Variabili esterne

Una variabile esterna è definita esternamente da qualsiasi funzione compresa la funzione *main()*. Le variabili esterne vengono anche definite come *variabili globali*. Se una variabile esterna no viene inizializzata al momento della definizione, il compilatore le inizializza automaticamente al valore 0.

L'ambito di una variabile esterna è rappresentata dall'intero programma. Ciò significa che una variabile esterna è visibile dalla funzione *main()* e da qualsiasi altra funzione all'interno del programma.

Variabili locali

Una variabile locale è una variabile definita all'interno di una funzione. L'ambito di una variabile locale risulta limitato alla funzione in cui è definita. Le variabili locali non vengono automaticamente inizializzata ad alcun valore. Se al momento della definizione non vengono inizializzate, contengono un valore indefinito.

Variabili statiche e variabili automatiche

Le cariabili locali sono di default *automatiche*. Ciò significa che vengono ricreate da zero ogni volta che viene richiamata la funzione in cui sono dichiarate e distrutte ogni volta che l'esecuzione del programma abbandona quella funzione. Ai fini pratici, ciò significa che una variabile automatica non mantiene il proprio calore nell'intervallo di tempo che intercorre tra due chiamate successive alla funzione in cui è definita.

E' possibile fare in modo che una variabile locale mantenga il proprio valore tra ciascuna chiamata alla funzione in cui è definita, e per fare ciò è necessario definirla come *statica* utilizzando la parola chiave *static*. Ad esempio:

```
void funzione(int x)
{
      static int a;

      /*codice della funzione*/
}
```

Esempio tra variabili locali statiche e automatiche:

```
#include <stdio.h>

void funzione1(void);

main()
{
      int conta;

      for (conta = 0; conta < 10; conta++)
      {
            printf("Alla iterazione %d: ", conta);
            funzione1();
      }

      return 0;
```

```
}

void funzione1()
{
      static int x = 0;
      int y = 0;

      printf("x = %d, t = &d\n", x++, y++) ;
}
```

Ecco l'output del programma :

```
Alla iterazione 0: x = 0, y = 0
Alla iterazione 1: x = 1, y = 0
Alla iterazione 2: x = 2, y = 0
Alla iterazione 3: x = 3, y = 0
Alla iterazione 4: x = 4, y = 0
Alla iterazione 5: x = 5, y = 0
Alla iterazione 6: x = 6, y = 0
Alla iterazione 7: x = 7, y = 0
Alla iterazione 8: x = 8, y = 0
Alla iterazione 9: x = 9, y = 0
```

Variabili di registro

La parola chiave *register* viene utilizzata per chiedere al compilatore di memorizzare una variabile locale automatica in un registro del processore, anziché in memoria. La parola chiave *register* è una richiesta e non un ordine. La parola chiave _register può essere utilizzata solo con variabile numeriche semplici e non è possibile definire un puntatore alle variabili di registro.

Passaggio di variabili a funzioni

Il modo normale di passare un argomento a una funzione è detto *passaggio per valore*. Con ciò si intende che la funzione riceve una **copia** dell'argomento.Quando una variabile è passata per valore, la funzione può accedere al valore della variabile, ma non alla variabile vera e propria. Di conseguenza, la funzione non può modificare il valore della variabile originale.

Il passaggio per valore è ammesso per i tipi di dati base (char, int, long, float e double) e per le strutture. Esiste tuttavia un secondo modo di passare argomenti a una funzione: passare un puntatore alla variabile argomento, anziché il suo valore. Questo è detto *passaggio per riferimento*. Le modifiche alle variabili passate per riferimento vengono applicate alla variabile di origine.

```
#include <stdio.h>

void per_valore(int a, int b, int c);
void per_riferimento(int *a, int *b, int *c);
```

```
main()
{
      int x = 2, y = 4, z = 6;

      printf(‘’ \nPrima della chiamata di per_valore(), x= %d, y = %d, z
= %d.‘’,x,y,z);

      per_valore(x,y,z);
      printf(‘’ \nDopo la chiamata di per_valore(), x= %d, y = %d, z =
%d.‘’,x,y,z);

      per_riferimento(&x, &y, &z);
      printf(‘’ \nDopo la chiamata di per_riferimento(), x= %d, y = %d, z
= %d.‘’,x,y,z);

      return 0;
}


void per_valore(int a, int b, int c)
{
      a = 0;
      b = 0;
      c = 0;
}


void per_riferimento(int *a, int *b, int *c)
{
      *a = 0;
      *b = 0;
      *c = 0;
}
```

Le funzioni ricorsive

Le funzioni ricorsive hanno fatto la loro comparsa sulla scena molto tempo fa, praticamente con il primo linguaggio che era dotato di funzioni. Una funzione ricorsiva in sostanza è una funzione che, per svolgere il proprio lavoro, richiama sè stessa. Ad ogni richiamo la "profondità" dell'elaborazione aumenta, finché ad un certo punto, lo scopo viene raggiunto e la funzione ritorna.

Una funzione ricorsiva "salva" il suo stato nel momento in cui richiama sé stessa, solitamente nello *stack*. Ogni volta che la ricorsione viene invocata, tutte le variabili presenti vengono inserite nello stack ed una nuova serie di variabili viene creata (sempre dallo stack). Questo significa un elevato consumo dello stack del sistema, se stiamo lavorando con uno stack limitato (come e' il caso con certi sistemi o con certe architetture di programma), rischiamo di superare i limiti dello stack e di avere un bel crash del programma. Aggiungiamo poi che, solitamente, il numero di chiamate ricorsive della funzione non e' ipotizzabile a priori ed abbiamo un possibile problema. Un altro possibile problema e' se la nostra funzione utilizza altre risorse oltre alla memoria del sistema, e tali risorse sono in quantita' limitata (connessioni a database per esempio). In questo caso, se il numero di richiami e' superiore ad un certo livello, possiamo avere un fallimento di chiamata.

Non sempre le funzioni ricorsive sono una risposta efficiente. Consideriamo che ogni chiamata ricorsiva consuma memoria ed utilizza un salto ad una funzione. Ridisegnando la funzione come non-ricorsiva, si risparmia un salto e (forse) un po' di quella preziosa memoria.

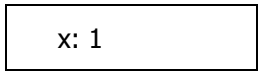
Esempi

Stampa di un elenco di numeri

Il seguente programma stampa l'elenco dei numeri all'interno di un intervallo (da 1 a 3) in modo ricorsivo:

```
1.  #include <stdio.h>
2.
3.  void stampa(int mun);
4.
5.  int main()
6.  {
7.      int x = 1;
8.      stampa(x);
9.      return 0;
10. }
11.
12. void stampa (int num)
13. {
14.     if (num > 3)
15.         return;
16.     printf("%d\n",num);
17.     stampa(num+1);
18. }
```

(chiamata A) Dopo la riga 5 la memoria si configura nel seguente modo:



dopo la chiamata alla funzione stampa() alla riga 8 nel main, la memoria è la seguente:

stampa()	num: 1	A
nel main	x: 1	

(chiamata B) Dato che num non è maggiore di 3 si procede con la stampa a video del numero 1 e alla riga 17 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 2), la memoria diventa:

stampa()	num: 2	B
stampa()	num: 1	A
nel main	x: 1	

(chiamata C) Dato che num non è maggiore di 3 si procede con la stampa a video del numero 2 e alla riga 17 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 3), la memoria diventa:

stampa()	num: 3	C
stampa()	num: 2	B
stampa()	num: 1	A
nel main	x: 1	

(chiamata D) Dato che num non è maggiore di 3 si procede con la stampa a video del numero 3 e alla riga 17 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 4), la memoria diventa:

stampa()	num: 4	D
stampa()	num: 3	C
stampa()	num: 2	B
stampa()	num: 1	A
nel main	x: 1	

dato che la variabile num è maggiore di 3 viene eseguita l'istruzione return alla riga 15, quindi la chiamata-D termina e il controllo passa alla funzione ricorsiva chiamante (chiamata-C), in particolare il controllo passa all'istruzione successiva alla chiamata ricorsiva della funzione: riga 18, quindi la funzione alla chiamata 3 termina le proprie operazioni e cede il controllo al chiamante (chiamata-B)... si procede in questo modo fino alla prima chiamata.

Consideriamo una variante del programma visto precedentemente, inserendo la stampa dopo la chiamata ricorsiva:

```
1. #include <stdio.h>
2.
3. void stampa(int mun);
4.
5. int main()
6. {
```

```
7.      int x = 1;
8.      stampa(x);
9.      return 0;
10. }
11.
12. void stampa (int num)
13. {
14.     if (num > 3)
15.         return;
16.     stampa(num+1);
17.     printf("%d\n",num);
18. }
```

Dopo la riga 5 la memoria si configura nel seguente modo:

nel main	x: 1

(chiamata A) dopo la chiamata alla funzione stampa() alla riga 8 nel main, la memoria è la seguente:

stampa()	num: 1	A
nel main	x: 1	

(chiamata B) Dato che num non è maggiore di 3 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 2), la memoria diventa:

stampa()	num: 2	B
stampa()	num: 1	A
nel main	x: 1	

(chiamata C) Dato che num non è maggiore di 3 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 3), la memoria diventa:

stampa()	num: 3	C
stampa()	num: 2	B
stampa()	num: 1	A
nel main	x: 1	

(chiamata D) Dato che num non è maggiore di 3 viene eseguita la chiamata ricorsiva passando il valore num+1 (cioè il valore 4), la memoria diventa:

stampa()	num: 4	D
stampa()	num: 3	C
stampa()	num: 2	B

stampa()	num: 1	A
nel main	x: 1	

dato che la variabile num è maggiore di 3 viene eseguita l'istruzione return alla riga 15, quindi la chiamata-D termina e il controllo passa alla funzione ricorsiva chiamante (chiamata-C), in particolare il controllo passa all'istruzione successiva alla chiamata ricorsiva della funzione: riga 17 quindi alla stampa della variabile num all'interno della chiamata-C quindi la stampa del numero 3. Questo perchè all'interno della chiamata-C il valore della variabile num è pari a 3. Dopo aver stampato il numero la funzione alla chiamata-C termina e il controllo passa al chiamante e cioè chiamata-B, in particolare il controllo passa all'istruzione successiva alla chiamata ricorsiva della funzione: riga 17 quindi alla stampa della variabile num all'interno della chiamata-B quindi la stampa del numero 2... si procede in questo modo fino alla prima chiamata.

Nella prima versione dell'esempio l'output sarà: 1, 2, 3. Nel secondo esempio invece avremo: 3, 2, 1

Fattoriale

Il seguente programma calcola il fattoriale di un numero in modo ricorsivo:

```
1.  #include <stdio.h>
2.
3.  int fattoriale(int mun);
4.
5.  int main()
6.  {
7.      int x, fat;
8.      scanf("%d",&x);
9.      fat = fattoriale(x);
10.     printf("fattoriale: %d", fat);
11.     return 0;
12. }
13.
14. int fattoriale(int num)
15. {
16.     if (n == 0)
17.         return 1;
18.
19.     ris = num * fattoriale(num - 1);
20.     rerturn ris;
21. }
```

Supponiamo che venga inserito il valore 3 come input. Dopo la lettura della variabile x la memoria si configura così:

nel main	x: 3

(chiamata A) Dopo la chiamata della funzione fattoriale() passando il valore num-1 quindi 3 (riga 19), la memoria è la seguente:

fattoriale()	num: 3	A
nel main	x: 3	

(chiamata B) Dopo la prima chiamata ricorsiva della funzione fattoriale()passando il valore num-1 quindi 2 (riga 19), la memoria è la seguente:

fattoriale()	num: 2	B
fattoriale()	num: 3	A
nel main	x: 3	

(chiamata C) Dopo la prima chiamata ricorsiva della funzione fattoriale() passando il valore num-1 quindi 1 (riga 19), la memoria è la seguente:

fattoriale()	num: 1	C
fattoriale()	num: 2	B
fattoriale()	num: 3	A
nel main	x: 3	

(chiamata D) Dopo la prima chiamata ricorsiva della funzione fattoriale() passando il valore num-1 quindi 0 (riga 19), la memoria è la seguente:

fattoriale()	num: 0	D
fattoriale()	num: 1	C
fattoriale()	num: 2	B
fattoriale()	num: 3	A
nel main	x: 3	

alla chiamata-D il valore della variabile num è uguale a 0 quindi al controllo nella riga 16 viene eseguita l'istruzione return passando il valore 1 al chiamante, quindi alla chiamata-C.
Alla chiamata-C troviamo nella riga 19 il valore restituito dalla chiamata-D e cioè 1 e il valore di num all'interno di questa chiamata e cioè 1. Dopo aver calcolato il prodotto il risultato viene restituito alla chiamata-B. Alla chiamata-B troviamo nella riga 19 il valore restituito dalla chiamata-C e cioè 1 e il valore di num all'interno di questa chiamata e cioè 2. Dopo aver calcolato il prodotto il risultato viene restituito alla chiamata-A. Alla chiamata-A troviamo nella riga 19 il valore restituito dalla chiamata-B e cioè 2 e il valore di num all'interno di questa chiamata e cioè 3. Dopo aver calcolato il prodotto il risultato viene restituito alla main e stampato a video (viene visualizzato il valore 6).

Torri di Hanoi

La Torre di Hanoi è un rompicapo matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti.
Il gioco inizia con tutti i dischi incolonnati su un paletto in ordine decrescente, in modo da formare un cono. Lo scopo del gioco è portare tutti dischi sull'ultimo paletto, potendo spostare solo un disco alla volta e potendo mettere un disco solo su un altro disco più grande, mai su uno più piccolo.

La proprietà matematica base è che il numero minimo di mosse necessarie per completare il gioco è $2^n - 1$, dove n è il numero di dischi. Ad esempio avendo 3 dischi, il numero di mosse minime è 7. Di conseguenza, secondo la leggenda, i monaci di Hanoi dovrebbero effettuare almeno 18.446.744.073.709.551.615 mosse prima che il mondo finisca, essendo $n = 64$.

La soluzione generale è data dall'algoritmo seguente:

- Si etichettano i paletti con le lettere A, B e C
- Dato n il numero dei dischi
- Si numerano i dischi da 1 (il più piccolo, in alto) a n (il più grande, in basso)

Per spostare i dischi dal paletto A al paletto B:

1. Sposta i primi n-1 dischi da A a C. Questo lascia il disco n da solo sul paletto A
2. Sposta il disco n da A a B
3. Sposta n-1 dischi da C a B

Questo è un algoritmo ricorsivo, di complessità esponenziale, quindi per risolvere il gioco con un numero n di dischi, bisogna applicare l'algoritmo prima a n-1 dischi. Dato che la procedura ha un numero finito di passi, in un qualche punto dell'algoritmo n sarà uguale a 1. Quando n è uguale a 1, la soluzione è banale: basta spostare il disco da A a B.

Esercizio 1. Passaggio per valore/per indirizzo

Scrivere un programma che esegue lo swap di due variabili intere x e y.
Prevedere due funzioni di swap, la prima (swap_val) che utilizzi un passaggio per valore e la seconda (swap_rif) che utilizzi, invece, un passaggio per indirizzo.
Visualizzare il valore delle variabili x e y in seguito alla chiamata di ciascuna funzione e commentare il risultato…

Esercizio 2. Ricerca di un pattern in una stringa

Scrivere un programma che cerca le occorrenze di un pattern all'interno di un insieme di stringhe.
Il programma prevede che l'utente possa inserire più stringhe; si considera la terminazione di una stringa con l'andata a capo (si legge cioè una linea per volta). L'input termina quando viene inserita una stringa vuota (nessun carattere+invio).
Si implementino e utilizzino due funzioni:
int getline(char s[], int lim):
legge una linea di input nel vettore s, fino a un massimo di lim caratteri, e ritorna la lunghezza della stringa

int ricerca_pattern(char s[])
cerca il pattern dentro la stringa s e ritorna -1 se non ha trovato il pattern, un numero >=0 altrimenti.

Per ogni input, il programma stampa il numero di occorrenze trovate fino a quel momento e, nel caso in cui sia stato trovato il pattern, anche la stringa.

Soluzione esercizio 1

```
#include<stdio.h>

void swap_val(int,int);
void swap_rif(int*,int*);

main()
{
   int x,y;
   printf("Inserisci x e y:\n");
   scanf("%d %d",&x,&y);
   swap_val(x,y);
   printf("Dopo lo swap con passaggio per valore: x vale %d, y vale %d\n",x,y);
   swap_rif(&x,&y);
   printf("Dopo lo swap con passaggio per indirizzo: x vale %d, y vale
%d\n",x,y);
}

void swap_val(int x, int y)
// swap con passaggio per valore
{
      int tmp;
      tmp = x;
      x = y;
      y = tmp;
}

// swap con passaggio per indirizzo
void swap rif(int *px, int *py)
{
      int tmp;
      tmp = *px;
      *px = *py;
      *py = tmp;
}
```

Soluzione esercizio 2

```
#include <stdio.h>
#define LIM 100

int getline(char[], int);
int ricerca_pattern(char[]);

char pattern[] = "ando";
int trovato=0; // inizializzazione ridondante

main()
{
        // cerca le occorrenze di un pattern dentro una stringa
        char line[LIM];
        printf("Inserisci il testo in cui ricercare il pattern ");
        printf("una linea per volta).\nStringa vuota per terminare.\n");
        while (getline(line,LIM) > 0)
```

```
        {
                if (ricerca_pattern(line) >= 0)
                        printf("%s\n", line);
                printf("\n Finora ho trovato”);
                printf(“%d occorrenze di %s\n",trovato,pattern);
        }
}

int getline(char s[], int lim)
{

        /* legge una linea di input nel vettore s,
        fino a un massimo di lim caratteri;
        ritorna la lunghezza della stringa */

        int c, i = 0;

        while(lim > 1 && (c=getchar()) != '\n')
        {
                s[i] = c;
                ++i;
                --lim;
        }
        s[i] = '\0';
        return i;
}

int ricerca_pattern(char s[])
{
        /* cerca la stringa pattern dentro s */

        int i, j, k, ret=-1;

        for (i = 0; s[i] != '\0'; ++i) {
                for (j = i, k = 0; pattern[k] != '\0'; ++j, ++k)
                        if (s[j] != pattern[k])
                                break;

        if (pattern[k] == '\0')
        {
                ++trovato;
                ret=i;
        }
   }
   return ret;
}
```

ESEMPI DI SCOPE

```
#include <stdio.h>

int counter = 0;       /* variabile globale */

void f(void);
void f2(void);
```

```
int main()
{
        ++counter;

        printf("counter is %2d before the call to f\n", counter);
        f();
        printf("counter is %2d after the call to f\n", counter);
        f2();

        return 0;
}

void f(void)
{
        int counter = 10;              /* variabile locale    */

        printf("counter is %2d within f\n", counter);
}

void f2(void)
{
        printf("counter is %2d within f2\n", counter);
}
```

Occorre sempre fare attenzione ai nomi e alla visibilità delle variabili: la variabile globale counter, definita all'inizio del programma, è visibile e modificabile in tutto il programma, eccetto per la funzione f, dove il nome counter è usato per identificare una variabile locale. Nota bene: quello visto sopra non è un buon esempio di programmazione, in quanto genera confusione per il lettore.

```
#include <stdio.h>

void f1(int);
int f2(void);

void f1(int i)
{
      extern int pippo;
      pippo = pippo + i;
}

int f2()
{
      static int pluto; // pluto è inizializzato solo alla prima chiamata a f2
      pluto = pluto + 1;
      return pluto;
}

int pippo;

int main()
{
      int j;
      int i = 2;
      printf("%d\n",pippo);
      f1(i);
      printf("%d\n",pippo);
```

```
        j = f2();
        printf("%d\n",j);
        j = f2();
        printf("%d\n",j);

        return 0;
}
```

- Lo scope di una variabile locale (automatica) è la funzione dove è stata definita.
- Lo scope di una variabile globale (esterna) va dal punto in cui essa è definita al termine del file sorgente in cui si trova.
- Se è necessario riferire una variabile esterna prima che essa sia stata definita, oppure se essa è definita in un file sorgente diverso da quello in cui viene utilizzata, allora è necessaria una dichiarazione di extern.
- La dichiarazione static, applicata ad una variabile esterna, ne limita lo scope al file sorgente nel quale essa si trova.
- La dichiarazione static, applicata ad una variabile non esterna, consente alla variabile di mantenere il proprio valore anche fra due chiamate successive.
- Le variabili esterne e static vengono inizializzate a zero, mentre le variabili automatiche hanno valori iniziali indefiniti.