LE FUNZIONI IN C

Sommario

Introduzione

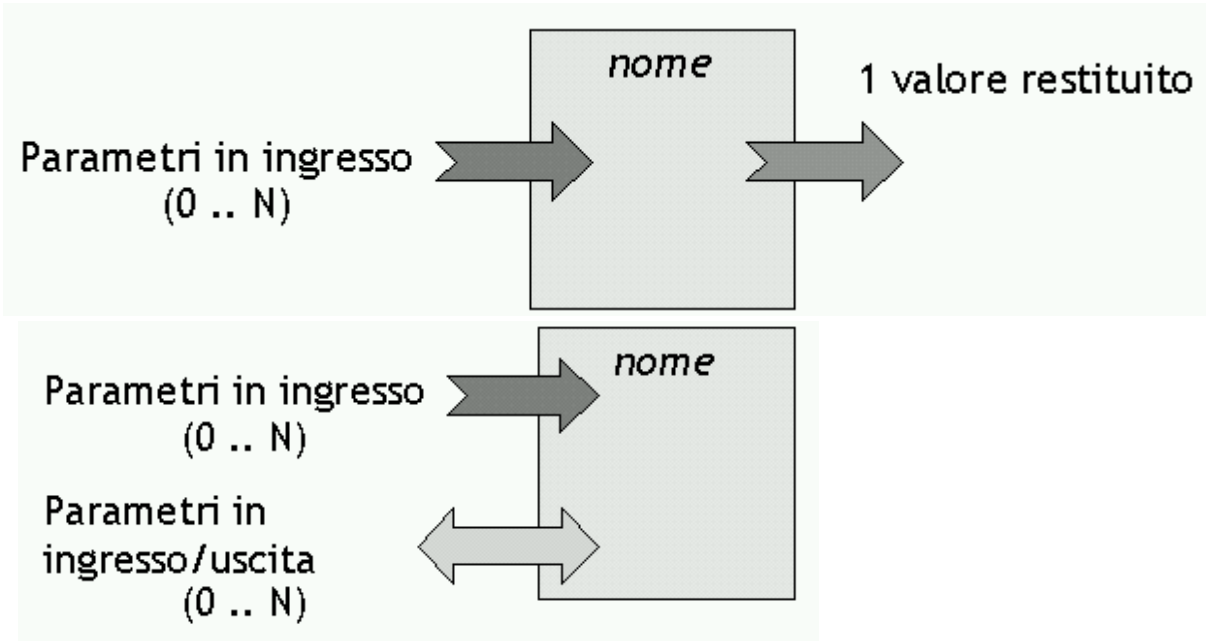
- Costruire un programma utilizzando parti o componenti più piccole
- Ogni parte è più “maneggevole” del programma originale

- Funzioni
  - Sono moduli in C
  - Tutti i programmi si scrivono combinando funzioni definite dall’utente con funzioni di libreria
    - La libreria standard del C ha una grande varietà di funzioni
    - Rendono il lavoro del programmatore più semplice (non bisogna inventare di nuovo la ruota!)

I sottoprogrammi

Un sottoprogramma è un insieme di istruzioni identificate mediante un nome, ed accessibile tramite un'*interfaccia*, che consente di far comunicare il sottoprogramma con il (sotto)programma chiamante.

In termini generali ci sono due tipi di sottoprogrammi: quelli che restituiscono un valore al (sotto)programma chiamante, e quelli che non lo fanno. Ai primi si dà il nome di ***funzioni***, ai secondi il nome di *procedure* (o *subroutine*). Quindi possiamo vedere un sottoprogramma come un ambiente che (eventualmente) riceve delle informazioni, svolge l'elaborazione richiesta ed eventualmente restituisce un valore al (sotto)programma che lo ha chiamato. Graficamente le due tipologie di sottoprogrammi sono riportate nella figura seguente.



Funzioni in C

In C solitamente la distinzione fra funzioni e procedure viene persa per ciò che concerne la terminologia, e ci si riferisce più semplicemente ai sottoprogrammi con il termine ***funzione***, presupponendo che questa restituisca o meno un valore in base alle specifiche ed al comportamento che si desidera ottenere.

Quando viene chiamato un sottoprogramma il flusso di esecuzione abbandona il programma in cui viene fatta la chiamata ed inizia ad eseguire il corpo del sottoprogramma. Quando quest'ultimo termina la propria esecuzione, con l'esecuzione dell'istruzione return o con il termine del corpo del programma, il programma chiamante continua l'esecuzione del codice che segue la chiamata al sottoprogramma.

Una funzione è una porzione di codice (sequenza di istruzioni) che vengono raggruppati e a cui viene dato un nome.

Il vantaggio di questa soluzione è che *la porzione di codice può essere utilizzata più volte semplicemente scrivendone il nome.*

Si supponga di voler scrivere una funzione che scriva a video un insieme di direttive per l'utente, che costituiscono il menù del programma, come fatto nello stralcio di codice riportato:

```
...
printf("Scegli la voce del menù:\n");
printf("1. addizione\n");
printf("2. sottrazione\n");
printf("3. moltiplicazione\n");
printf("4. divisione\n");
scanf("%d", &scelta);
 ...
```

Per rendere la parte di codice una funzione è necessario racchiudere il codice tra un paio di parentesi graffe per renderle un *blocco di codice* e dare un nome alla funzione:

```
menu()                          /*nome della funzione*/
{
printf("Scegli la voce del menù:\n");
printf("1. addizione\n");
printf("2. sottrazione\n");
printf("3. moltiplicazione\n");
printf("4. divisione\n");
}
```


A questo punto è possibile utilizzare la funzione *chiamandola*:

```
void main()
{
menu();
scanf("%d", &scelta);
 ...
}
```

Nel programma, *l'istruzione menu(); è equivalente ad aver scritto direttamente tutte le istruzioni della funzione stessa*.

A parte questo semplice esempio, le funzioni hanno lo scopo di rendere un lungo programma una collezione di porzioni di codice separate su cui lavorare in modo isolato, *suddividendo la soluzione di un problema complesso in tanti piccoli sottoproblemi*, di più facile soluzione.

Funzioni

- Le funzioni
  - Suddividono un programma in moduli
  - Tutte le variabili dichiarate all'interno di una funzione sono "locali", cioè sono note solo internamente alla funzione
  - I parametri di una funzione
    - Comunicano informazioni fra le funzioni
    - Sono variabili locali
- Vantaggi
  - "Divide et impera"
    - Rendono lo sviluppo di programmi più maneggevole
  - Costituiscono del software riutilizzabile
    - Funzioni già esistenti servono come blocchi costitutivi di nuovi programmi
    - Astrazione – nascondono i dettagli interni (ad es. le funzioni di libreria)
  - Evitano di dover ripetere più volte parti di codice uguali

Funzioni in C (cont.)

- **“Chiamata” di funzioni**
  - Sono necessari il nome della funzione ed i suoi argomenti o parametri (dati)
  - La funzione esegue operazioni o manipolazioni sui dati
  - La funzione restituisce i risultati di tali elaborazioni

Funzioni di libreria matematiche

- Le funzioni di libreria matematiche
  - Eseguono i calcoli matematici più comuni
  - **`#include <math.h>`**

- Formato della chiamata di funzioni

  `Nome_della_Funzione (argomenti);`
  - Per più di un argomento, si usa il ;
  - **`Es.: printf( "%.2f", sqrt( 900.0 ) );`**
    - Chamata della funzione **`sqrt`**, che restituisce la radice quadrata del suo argomento
    - Tutte le funzioni matematiche restituiscono variabili di tipo **`double`**
  - Gli argomenti possono essere costanti, variabili, o espressioni

Funzioni e variabili locali

Nella funzione menu, vista prima per chiarire la filosofia di fondo delle funzioni, non ci sono variabili e questo è un caso particolarmente semplice.

Una funzione è una sottounità di un programma completo: il main stesso è una funzione, chiamata dal sistema. Quindi anche una funzione avrà in generale delle variabili che verranno utilizzate al suo interno per effettuare le elaborazioni desiderate.

Le variabili dichiarate all'interno di una funzione sono significative e visibili esclusivamente all'interno della funzione stessa, ed il programma chiamante non ne ha alcuna visibilità. Si tratta di ***variabili locali*** alla funzione stessa.

Le variabili che una funzione dichiara sono create quando la funzione viene chiamata e vengono distrutte quando la funzione termina.

Si consideri il seguente **esempio**:

```
void main()
{
contastampe();                    /*1a chiamata di contastampe */
...
contastampe();                    /* 2a chiamata di contastampe */
 }
void contastampe()      /* funzione contastampe */
{
int num_stampe;
 num_stampe = 0;
 printf("xxxxxxx");
num_stampe++;
}
```

Ad ogni chiamata della funzione contastampe() la variabile num_stampe viene ricreata e distrutta al termine.

Variabili globali e locali

```
int x;          /* nome globale */

f()             /*funzione f */
{
 int x;         /* x locale che nasconde x globale */
 x = 1;         /* assegna 1 a x locale */
 {
  int x;        /* nasconde il primo x locale */
  x = 2;        /* assegna 2 al secondo x locale */
 }
 x = 3;         /* assegna 3 al primo x locale */
}

scanf ("%d", &x); /* inserisce un dato in x globale */
```


Parametri

Se si desidera che un valore calcolato da una funzione resti disponibile anche dopo il termine dell'esecuzione della funzione è necessario che questo venga ***trasmesso*** al programma chiamante.

In modo analogo, se la funzione deve elaborare dei dati del programma chiamante è necessario che i dati le vengano ***passati***.

A questo scopo vengono definite variabili speciali, chiamate ***parametri*** che vengono utilizzati per passare i valori alla funzione. I parametri vengono elencati nelle parentesi tonde che seguono il nome della funzione, indicando il tipo ed il nome di ogni parametro. La lista dei parametri ha come separatore la virgola. Esempio:

```
somma(int a, int b)
{
int risultato;
risultato = a + b;
}
```

Questo codice definisce una funzione chiamata somma con due parametri a e b, entrambi di tipo intero. La *variabile risultato è dichiarata localmente alla funzione*, nel *corpo* della funzione.

I parametri a e b vengono utilizzati all'interno della funzione come normali variabili - si noti che non devono essere definite due variabili locali a e b. Inoltre, questi a e b non hanno nulla a che fare con altre variabili a e b dichiarate in altre funzioni.

PARAMETRI E VARIABILI

La differenza fondamentale tra i parametri e le variabili è che i primi hanno un valore iniziale quando la funzione viene eseguita, mentre le *variabili devono essere inizializzate*. Quindi,

somma(l, 2);

è una chiamata alla funzione somma in cui il parametro a vale 1 e b vale 2.

È anche possibile far assumere ai parametri il risultato di un espressione, come ad esempio:

somma(x+2, z*10);

che farà assumere ad a il valore pari a x+2 (in base a quanto varrà x al momento della chiamata e b pari a z*10. Più semplicemente si può fissare il valore di un parametro al valore di una variabile:

somma(x, y);

in cui a assume il valore di x e b di y.

In modo duale è anche necessario o possibile trasmettere al programma chiamante il valore calcolato dalla funzione. La via più semplice è la *restituzione* del valore attraverso il nome della funzione, ossia è come se il nome della funzione fosse una variabile dotata di un valore. Il valore viene restituito per mezzo della seguente istruzione:

return(*value*);

Questa istruzione può essere posizionata in qualunque punto della funzione, tuttavia un'istruzione return causa il termine della funzione e restituisce il controllo al programma chiamante.

È' necessario aggiungere un'informazione relativa al tipo di dato che la funzione restituisce. Con riferimento alla funzione somma, il tipo restituito è un intero; si scriverà dunque:

```
int sum(int a, int b)
{
...
 }
```

La funzione completa è:

```
int sum(int a, int b)
{
 int risultato;
risultato = a + b;
return (risultato);
}
```

La chiamata assume quindi la seguente forma:

```
r = sum(1, 2);
```

istruzione che somma 1 a 2 e memorizza il risultato nella variabile r (dichiarata int nel programma chiamante).

Ovviamente, la situazione tra ***ingressi*** ed ***uscite*** di una funzione non è uguale: è possibile passare un numero di parametri d'ingresso qualsiasi, mentre la funzione può restituire mediante l'istruzione return un singolo valore.
Si noti che una funzione può avere quante istruzioni return si desidera, ma ciò non consente di restituire più di un valore in quanto quando si esegue la prima return la funzione termina. (Per poter restituire più dati è necessario utilizzare un passaggio dell'indirizzo, come vedremo più avanti).

Per riassumere, una funzione ha la seguente **forma sintattica**:

```
tipo_restituito NomeFunzione(lista tipo-nome)
{
istruzioni
}
```

Nel caso in cui una funzione non restituisca alcun parametro, il tipo indicato è **void**, che indica appunto tale eventualità. La funzione void menu() è una funzione senza parametri d'ingresso e che non restituisce alcun valore.

Definizione di funzioni

- Formato della definizione di funzioni

  Tipo_del _valore _di _ritorno ***nome_della_funzione*** **(**
  lista_dei_parametri **)**
  {
  dichiarazioni e istruzioni
  }

  – Dichiarazioni e istruzioni: corpo della funzione (blocco)
    - Le variabili possono essere dichiarate all'interno dei blocchi (possono essere annidati)
    - Le funzioni non possono essere definite all'interno di altre funzioni
  – Restituzione del controllo alla funzione chiamante
    - Se non viene restituito alcun valore:
      – **`return;`**
      – Oppure parentesi graffa }
    - Se viene restituito qualcosa:
      – **`return`** *espressione* **`;`**

Funzioni e prototipi

Dove va scritta la definizione di una funzione, prima o dopo il main()? L'unico requisito è che la tipologia della funzione (tipo di dato restituito e tipo dei parametri) sia nota prima che la funzione venga usata.

Una possibilità è scrivere la definizione della funzione prima del main(). Con questa soluzione è però necessario prestare molta attenzione all'ordine con cui si scrivono le funzioni, facendo sempre in modo che una funzione sia sempre definita prima che qualche altra la chiami. In alternativa, la soluzione più pulita è *dichiarare la funzione prima del main, separatamente da dove viene poi definita*. Per esempio:

```
int somma(int,int);       /*prototipo o dichiarazione della funzione*/
void main()
{
...
}
```

Qui si dichiara il nome della funzione somma e si indica che restituisce un int. A questo punto la definizione della funzione può essere messa ovunque.

Per quanto riguarda i *parametri* ricevuti in ingresso è *necessario dichiararne la tipologia ma non il nome*, come mostrato nel seguente esempio:

```
int restodivisione(int, int);
```


Proptotipi di funzioni e file di intestazione

- Prototipi di funzioni (dichiarazione)
  - Nome della funzione
  - Parametri (ciò che viene passato alla funzione)
  - Tipo di ritorno – tipo dei dati che la funzione restituisce (default **int**)
  - I prototipi sono necessari solo se la definizione della funzione segue il suo uso nel programma chiamante

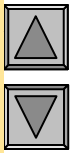
    ```
    int maximum( int, int, int );
    ```

    - Ha tre valori in ingresso di tipo **int**
    - Restituisce un **int**
- File di intestazione
  - Contiene i prototipi dielle funzioni di libreria
  - **<stdlib.h>** , **<math.h>** , etc
  - Si carica con: **#include <filename>**

    ```
    #include <math.h>
    ```


```
1
2   /* Calcola il massimo fra tre interi */
3   #include <stdio.h>
4
5   int maximum( int, int, int );   /* prototipo della funzione */
6
7   int main()
8   {
9      int a, b, c;
10
11     printf( "Enter three integers: " );
12     scanf( "%d%d%d", &a, &b, &c );
13     printf( "Maximum is: %d\n", maximum( a, b, c ) );
14
15     return 0;
16  }
17
18  /* Definizione della funzione maximum */
19  int maximum( int x, int y, int z )
20  {
21     int max = x;
22
23     if ( y > max )
24        max = y;
25
26     if ( z > max )
27        max = z;
28
29     return max;
30  }
```

```
Enter three integers: 22 85 17
Maximum is: 85
```



Outline

Prototipo della funzione maximum:

ha 3 parametri di tipo intero e restituisce un intero

Input dei dati

Chiamata della funzione

Definizione della funzione

Output del programma

Cubo di un numero

```
#include <stdio.h>
double cubo(float);      /*dichiarazione (prototipo)*/
main()
{
  float  a;
  double b;
  printf("Inserisci un numero: ");
  scanf("%f", &a);
  b = cubo(a); /*chiamata*/
  printf("%f elevato al cubo e' uguale a %f", a, b);
}
double cubo(float c)    /*definizione*/
{
  return (c*c*c);
}
```


```
#include <stdio.h>
/*dichiarazione delle funzioni x2,x3,x4,x5,potenza*/
double quad(float);
double cubo(float);
double quar(float);
double quin(float);
double pote(float, int);

main()
{
 int    base, esponente;
 double ptnz;
 printf(" Inserire base: " );
 scanf("%d", &base);
 printf(" Inserire esponente (0-5): ");
 scanf("%d", &esponente);
/*chiamata della funzione pote*/
 ptnz = pote( base, esponente);
 if (ptnz == -1)
   printf("Potenza non prevista\n");
 else
   printf("La potenza %d di %d e' %f \n", esponente, base, ptnz);
}
/*definizione delle funzioni*/
double quad(float c)
{
 return(c*c);
}
```

```
double cubo(float c)
{
 return(c*c*c);
}
double quar(float c)
{
 return(c*c*c*c);
}
double quin(float c)
{
 return(c*c*c*c*c);
}
double pote(float b, int e)
{
 switch (e) {
   case  0: return (1);
   case  1: return (b);
   case  2: return (quad( b ));
   case  3: return (cubo( b ));
   case  4: return (quar( b ));
   case  5: return (quin( b ));
   default : return (-1);
 }
}
```


Area di un poligono: triangolo o rettangolo

```
#include <stdio.h>
double area(float, float, char);
main()
{
  float b, h;
  double a;
  char p;
  printf("Inserire poligono (Triangolo/Rettangolo): “);
  scanf("%c", &p);
   printf("\nInserire base: ");
  scanf("%f", &b);
  printf("\nInserire altezza  : ");
  scanf("%f", &h);

  a = area(b, h, p);

  printf("Il poligono (b = %f, h = %f) ha area %f\n",
b, h, a);
}
```

```
double area(float base, float altezza,
char poligono)
{
 switch (poligono) {
 case 'T': return (base * altezza/2.0);
  case 'R':  return (base * altezza);
   default :     return -1;
 }
}
```


```
#include <stdio.h>
double area(float, float, char);
main()                                  /*calcolo dell'area del triangolo e del
rettangolo*/
{
  float b, h;
  double tri, ret;
  printf("Inserire base: ");
  scanf("%f", &b);
  printf("Inserire altezza: ");
  scanf("%f", &h);
  tri = area(b, h, 'T');
  ret = area(b, h, 'R');
  printf("Il triangolo (b = %f, h = %f) ha area %f\n", b, h, tri);
  printf("Il rettangolo (b = %f, h = %f) ha area %f\n", b, h, ret);
}

double area(float base, float altezza, char poligono)
{
  switch (poligono) {
    case 'T':    return (base * altezza/2.0);
    case 'R':    return (base * altezza);
    default :    return -1;
  }
}
```


```
#include <stdio.h>

char str[] = "Lupus in fabula";

int lung_string(void);      /*funzione che conta gli elementi della frase
                              (stringa) */

main()
{
  int l;
  l = lung_string();
  printf("La stringa %s ha %d caratteri\n", str, l);
}

int lung_string(void)
{
  int i;
  for (i = 0; str[i] != '\0'; i++);                    /*\0 è il carattere di 'fine stringa'*/
  return i;
}
```


```c
#include <stdio.h>
/*divisione di due numeri con messaggio
di errore */
void messErr( void );

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 diviso %d = %d\n", a, b, c);
  }
else
  messErr();
}
```

```c
void messErr( void )

/*consente di immettere nuovamente
il divisore per 20 volte */

{

  int i;

  char c;

  for (i = 0; i <= 20; i++) printf("\n");

  printf("ERRORE!
DENOMINATORE NULLO");

  printf("\n Premere un tasto per
continuare\n");

  scanf("%c", &c);

}
```


La ricorsione

La ricorsione è una tecnica di programmazione in cui la risoluzione di problemi di grandi dimensione viene fatta mediante la soluzione di problemi più piccoli *della stessa forma*. È importante capire che il problema verrà scomposto in problemi della stessa natura.

Un esempio

Per avere un'idea di cosa sia la ricorsione, si pensi ad una delle prospettive di guadagno che talvolta vengono pubblicizzate: il vostro compito è di raccogliere 1.000.000 di euro.

Visto che è praticamente impossibile pensare di trovare una persona che versi l'intera cifra si deve pensare di raccogliere l'intera cifra mediante la somma di contributi piu' piccoli. Se ad esempio si sa che ogni persona interpellata di solito è disposta a metterci 100 euro, è necessario trovare 10.000 persone e chiedere a ciascuna di queste 100 euro. Trovare 10.000 persone potrebbe però essere un pò difficile. La soluzione è nel cercare di trovare altre persone che si dedichino alla raccolta dei soldi, e di delegare a loro la raccolta di una certa cifra. Per esempio si può pensare di individuare 10 persone, ognuna delle quali raccolga 100.000 euro. Se anche queste dieci persone adottano la stessa strategia, questi recluteranno 10 persone ciascuna delle quali deve raccogliere 10.000 euro. Lo stesso ragionamento si può fare fino ad arrivare ad avere dieci persone che raccolgano per un delegato i 100 euro.

La ricorsione (cont.)

Se cerchiamo di codificare questa strategia in pseudo-codice, l'algoritmo risultante è il seguente:

```
void RaccogliDenaro(int n)
{
 if(n <= 100)
Chiedi i soldi ad una sola persona
else {
trova dieci volontari
Ad ogni volontario chiedi di raccogliere n/10 euro
 Somma i contributi di tutti i dieci volontari
}
 }
```

La cosa che è importante notare è che l'istruzione:

Ad ogni volontario chiedi di raccogliere n/10 euro

non è altro che il problema iniziale, ma su una scala più piccola. Il compito è lo stesso - raccogliere *n* euro - solo con un *n* più piccolo.

La ricorsione (cont.)

Inoltre, siccome il problema è lo stesso, lo si può risolvere chiamando il sottoprogramma originale.

Quindi, nello pseudo-codice, si può pensare di scrivere:

```
void RaccogliDenaro(int n)
{
if(n <= 100)
Chiedi i soldi ad una sola persona
else
{
trova dieci volontari
Ad ogni volontario
RaccogliDenaro(n/10)
Somma i contributi di tutti i dieci volontari
 }
}
```

Alla fine, il sottoprogramma RaccogliDenaro finisce per chiamare se stesso se il contributo da raccogliere è inferiore a 100 euro.

La ricorsione (cont.)

Nel contesto della programmazione, avere un sottoprogramma che chiama se stesso prende il nome di **ricorsione**.

Lo scopo dell'esempio è dunque quello di illustrare l'idea di affrontare la soluzione di un problema facendo riferimento allo stesso problema su scala ridotta.

I problemi che possono essere risolti tramite una forma ricorsiva hanno le seguenti caratteristiche:

•Uno o più **casi semplici** del problema hanno una soluzione immediata, non ricorsiva;

•Il caso generico può essere ridefinito in base a problemi più vicini ai casi semplici;

•Applicando il processo di redifinizione ogni volta che viene chiamato il sottoprogramma ricorsivo il problema viene ridotto al caso semplice, facile da risolvere.

L'algoritmo ricorsivo avrà spesso la seguente forma:

if(*è il caso semplice*)

risolvilo

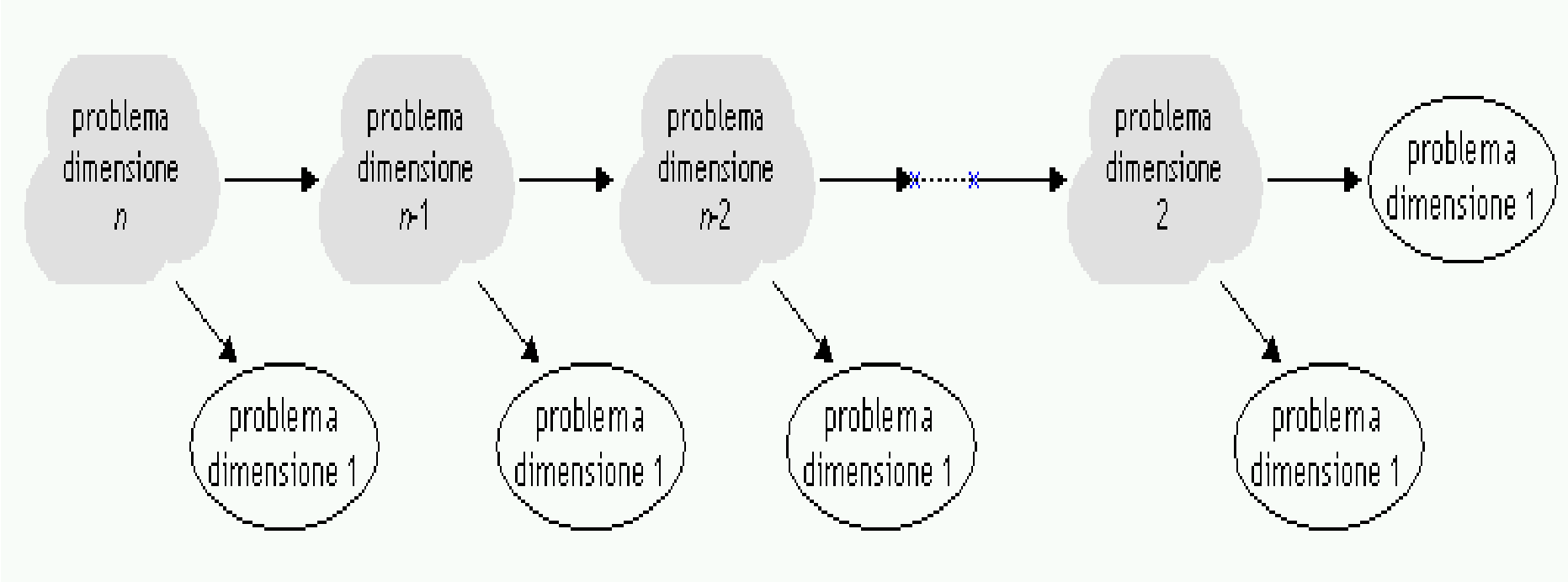
else

ridefinisci il problema utilizzando la ricorsione

Quando si individua il caso semplice, la condizione indicata nell'if per la gestione del caso semplice si chiama **condizione di terminazione**.

La ricorsione

La figura seguente illustra tale approccio: la soluzione del caso semplice è rappresentata dalla soluzione del problema di dimensione 1.



Ricorsività

- Funzioni ricorsive
  - Funzioni che richiamano se stesse
  - Possono risolvere solo un “caso base”
  - Dividono il problema in:
    - Quello che può essere fatto
    - Quello che non può essere fatto – simile al problema originale
      - Lancia una nuova copia di se stesso (passo della ricorsione)

- Quando il “caso base” è risolto
  - La funzione è attivata, svolge i vari passi e risolve il problema

Esempio: la moltiplicazione

Come altro esempio si prenda in considerazione il seguente problema: effettuare il prodotto tra due numeri a e b conoscendo l'operatore somma ma non l'operatore prodotto e sapendo solo che:

- un numero moltiplicato per uno è uguale a se stesso
- il problema del prodotto a * b può essere spezzato in due parti:

P1. il prodotto a * (b-1)

P2. somma a al risultato della soluzione del problema P1.

Esempio (cont.)

Il problema P1, per quanto non risolubile (perchè continuiamo a non conoscere l'operatore prodotto) è più vicino al caso semplice ed è possibile pensare di spezzare anche tale problema in due parti, ottenendo così la seguente cosa:

P1. Prodotto a * (b-1)

P1.1. Prodotto a * (b-2)

P1.2. Somma a al risultato del problema P1.1

P2. Somma a al risultato del problema P1.

Si procede così fino a quando:

P1.1...1. Prodotto a *1

P1.1...2. Somma a al risultato del problema P1...1

Il *caso semplice* si ha quando n == 1 è vera.

Esempio (cont.)

La forma della ricorsione prima indicata diventa quindi:

```
if(b == 1)
ris = a;                    /* caso semplice */
else
ris = a + moltiplica(a, b-1);    /* passo ricorsivo */
```

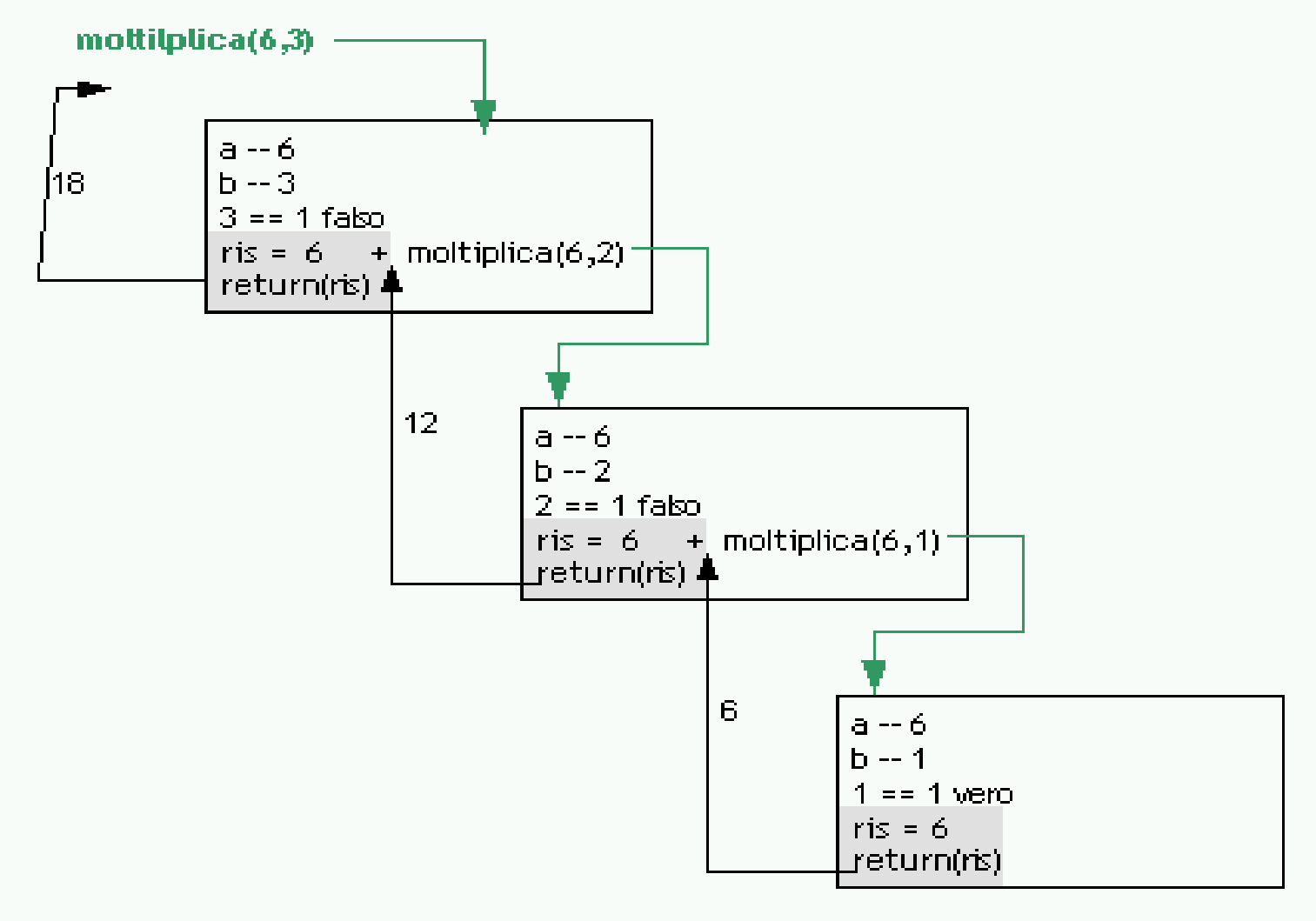
Il codice completo è quindi:

```
int moltiplica(int a, int b)
{
 int ris;
if(b == 1)
ris = a;
else ris = a + moltiplica(a, b-1);       /* passo ricorsivo */
 return(ris);
}
```

Una classe di problemi per cui la ricorsione risulta una soluzione interessante è quella che coinvolge delle sequenze (o liste) di elementi di lunghezza variabile.

Record di attivazione

Il record di attivazione mostra il valore dei parametri per ogni chiamata e l'esecuzione della funzione. Si consideri la chiamata seguente: moltiplica(6,3); la traccia dell'esecuzione è mostrata nella figura seguente.



Pila di sistema (stack)

Per maggior chiarezza nell'esempio sono stati rappresentati degli spazi di memoria distinti, anche se in realtà il compilatore mantiene un'unica **pila di sistema** (o **stack**).

Ogni volta che una funzione viene chiamata, i suoi parametri e le sue variabili locali vengono messi sullo stack, insieme all'indirizzo di memoria dell'istruzione che effettua la chiamata. Questo indirizzo serve per sapere a che punto rientrare dalla chiamata a sottoprogramma.

L'esecuzione dell'istruzione return in uscita dalla funzione svuota lo stack restituendo il valore che c'era in cima allo stack.

In questo modo, ogni chiamata a funzione, anche quelle ricorsive, riserva alla funzione spazio necessario per i parametri e per le variabili locali, cosicché tutto possa funzionare correttamente nel rispetto delle regole di visibilità.

Esempio: fattoriale

- Fattoriale:

  `5! = 5 * 4 * 3 * 2 * 1`

  Si noti che

  `5! = 5 * 4!`

  `4! = 4 * 3!` ...

  – Quindi il fattoriale può essere calcolaro ricorsivamente

  – Si risolve il caso base (1! = 0! = 1) e si "attiva" la sequenza delle chiamate

    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

Calcolo del fattoriale con una funzione ricorsiva

```
/* Calcolo del fattoriale con una funzione ricorsiva */
#include <stdio.h>
int fat(int);              /* dichiarazione */

main()
{
int n;
printf("CALCOLO DI n!\n\n");
printf("Inser. n: \t");
scanf("%d", &n);
printf("Il fattoriale di: %d ha valore: %d\n", n, fat(n));        /*chiamata*/
}

int fat(int n)             /*definizione*/
{
if(n==0)
  return(1);
else
  return(n*fat(n-1));
}
```


Esempio: serie di Fibonacci

Serie di Fibonacci : 0, 1, 1, 2, 3, 5, 8...

Ogni numero è la somma dei due precedenti:

fib(n) = fib(n-1) + fib(n-2)

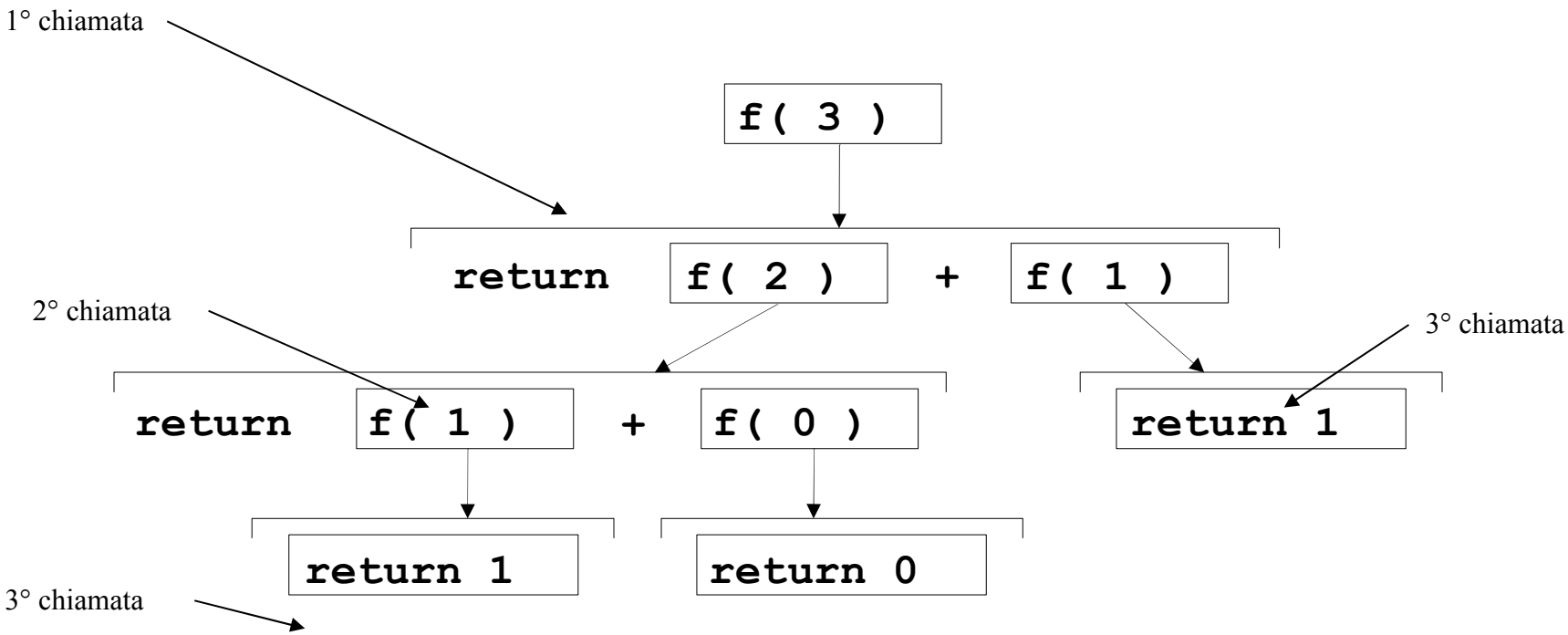
E' una formula ricorsiva

```
long fibonacci(long n)          /*long= intero lungo */
{
   if (n==0 || n==1)  /*caso base*/
   return n;
   else return fibonacci(n-1) + fibonacci(n-2);
}
```


Fibonacci

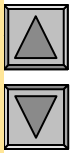
```
/* Calcolo dei numeri di Fibonacci */
#include <stdio.h>
long int fibo(int);

main()
{
int n;
printf("Successione di Fibonacci f(0)=1 f(1)=1 f(n)=f(n-1)+f(n-2)");
printf("\nInserire n: \t");
scanf("%d", &n);
printf("Termine della successione di argomento %d: %d\n", n, fibo(n));
}

long int fibo(int n)
{
if(n==0)      return(0);
else if(n==1) return(1);
    else     return(fibo(n-1)+fibo(n-2));
}
```


Fibonacci: chiamate ricorsive



```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long );

int main()
{
   long result, number;

   printf( "Enter an integer: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1
```



Outline

Prototipo della funzione

Inizializzazione delle variabili

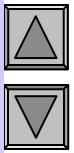
Input dei dati (un numero intero)

Chiamata della funzione Fibonacci

Output dei risultati

Definizione della funzione ricorsiva Fibonacci

Risultati



Outline

Program Output

```
Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Ricorsione ed iterazione

- Ripetizione
  - Iterazione: ciclo esplicito
  - Ricorsione: chiamate ripetute a funzione
- Terminazione
  - Itarazione: La condizione del ciclo non è più verificata
  - Ricorsione: viene riconosciuto il ciclo base
- Entrambe possono avere infiniti cicli
- Criterio di utilizzo
  - Si deve scegliere fra efficienza (iterazione) e buona ingegneria del software (ricorsione)

GLI ARRAY

Introduzione
Array
Dichiarare gli Array
Esempi di utilizzo di Array
Array multidimensionali
Passaggio di Array a Funzioni

Gli array

Le variabili semplici, capaci di contenere un solo valore, sono utili ma spesso insufficienti per numerose applicazioni.

Quando si ha la necessità di trattare *un insieme omogeneo di dati* esiste un'alternativa efficiente e chiara all'utilizzo di numerose variabili dello stesso tipo, da identificare con nomi diversi: definire un **array**, ovvero una collezione di variabili dello stesso tipo, che costituisce una *variabile strutturata*.

L'array costituisce una tipologia di dati strutturata e *statica*: la dimensione è fissata al momento della sua creazione - in corrispondenza alla dichiarazione - e non può essere mai essere variata.

Array monodimensionali

Intuitivamente un array monodimensionale - *vettore* - può essere utilizzato come un contenitore suddiviso in elementi, ciascuno dei quali accessibile in modo indipendente.

Ogni *elemento* contiene un unico dato ed è individuato mediante un *indice*: l'indice del primo elemento dell'array è 0, l''ultimo elemento di un array di N elementi ha indice N-1.

Il numero complessivo degli elementi dell'array viene detto *dimensione*, e nell'esempio utilizzato è pari a N.

Per riassumere, un array è una struttura di dati composta da un numero *determinato* di elementi dello stesso tipo, ai quali si accede singolarmente mediante un indice che ne individua la posizione all'interno dell'array.

Per ogni array, così come per ogni variabile semplice (o non strutturata) è necessario definire il **tipo** di dati; inoltre è necessario specificarne la **dimensione**, ossia il numero di elementi che lo compongono. Una dichiarazione valida è la seguente:

```
int numeri[6];
```

Che corrisponde a:

[numeri[0] numeri[1] numeri[2] numeri[3] numeri[4] numeri[5]]

Viene indicato, come sempre, prima il tipo della variabile (int) poi il nome (numeri) ed infine tra parentesi quadre la dimensione (6): l'array consente di memorizzare 6 numeri interi.

Tra parentesi quadre è necessario indicare SEMPRE un'espressione che sia un valore intero costante. In base a quanto detto, è errato scrivere:

```
int numeri[];
int i, numeri[i];
```


Array (cont.)

Gli elementi di un array sono variabili. Ad es:

```
c[0] =  3;
printf( "%d", c[0] );
```

- Si possono eseguire operazioni sui singoli elementi di un array. Ad es., se **x = 3**,

c[5-2]

corrisponde a

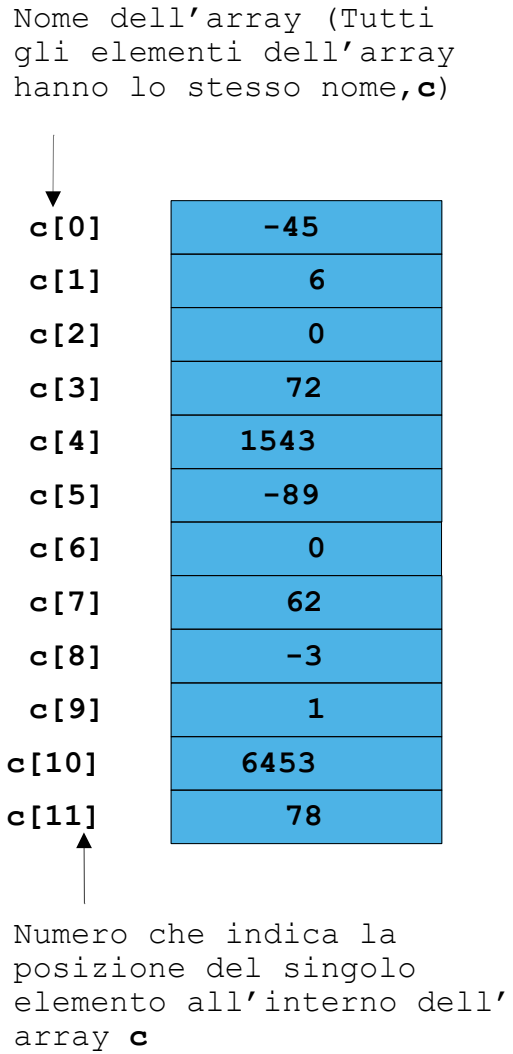
c[3]

oppure a

c[x]

Array (cont.)

- Array
  - Gruppi di locazioni di memoria consecutive
  - Tutte con lo stesso nome e tipo
- Per fare riferimento ad un elemento bisogna conoscere
  - Il nome dell'array
  - Il numero che ne individua la posizione al suo interno
- Formato: *nomearray***[numeroposizione]**
  - Il primo elemento ha indice **0**
  - Un array di n elementi di nome **c é**: **c[0]**, **c[1]**...**c[n-1]**



Dichiarare gli array

- Per dichiarare un array, si deve specificare
  - Nome
  - Tipo
  - Numero di elementi

```
arrayType arrayName[ numberOfElements ];
int c[10];
float myArray[3284];
```

- Si possono dichiarare più array dello stesso tipo nella stessa istruzione
  - Il formato è analogo a quello delle variabili monodimensionali

```
int b[ 100 ], x[ 27 ];
```


Il vincolo di dover specificare in fase di dichiarazione la dimensione dell'array porta spesso ad un sovradimensionamento dell'array, al fine di evitare di non disporre dello spazio necessario durante l'escuzione del programma.

Nel caso in cui sia dunque richiesto l'utilizzo di un array, la specifica dell'algoritmo dovrà quindi prevedere o l'esatto numero di dati da gestire oppure un valore massimo di dati.

Nel caso in cui non sia possibile stabilire in alcun modo un limite al numero di elementi senza violare i requisiti di generalità, sarà necessario utilizzare una struttura dati diversa e più precisamente una struttura dinamica, trattata in seguito.

Per accedere al singolo elemento dell'array è necessario indicare il **nome** dell'array e **l'indice dell'elemento** posto tra parentesi quadre, ad esempio, per accedere al primo elemento si dovrà scrivere:

numeri[0]

Si noti che gli elementi dell'array vanno dall'elemento di indice 0 a quello di indice 5. Accedere all'elemento di indice 6 (o superiore) non sempre causa un errore di sintassi.

In generale il singolo elemento di un array può essere utilizzato come una semplice variabile.

Spesso l'array viene utilizzato all'interno di iterazioni per accedere uno dopo l'altro ai suoi elementi, semplicemente utilizzando un indice che viene modificato ad ogni iterazione. Ad esempio:

```
/* Inizializzazione di un array di numeri interi al valore nullo */
for (i = 0; i < 6; i++)
numeri[i] = 0;
```

L'indice i inizializzato a zero consente di accedere dal principio al primo elemento dell'array e di proseguire fino all'ultimo, con indice 5 (si noti che quando l'indice è pari a 6 la condizione è falsa ed il corpo del ciclo non viene eseguito).

Esempi di utilizzo di array

- Initializzatori

```
int n[5] = {1, 2, 3, 4, 5 };
```

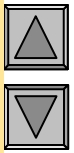
  - Se non ci sono sufficienti inizializzatori, gli elementi più a destra sono inizializzati a **0**
  - Se ce ne sono troppi, viene segnalato un errore di sintassi

```
int n[5] = {0}
```

    - Pone tutti gli elementi uguali a **0**
  - Il C non ha un controllo sui limiti degli array
- Se la dimensione del vettore è omessa, gli inizializzatori consentono di definirla in base al loro numero

```
int n[] = { 1, 2, 3, 4, 5 };
```

  - 5 inizializzatori, quindi n è un array di 5 elementi



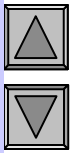
Outline

```
1  /*PROGRAMMA PER LA STAMPA DI UN ISTOGRAMMA*/
2
3  #include <stdio.h>
4  #define SIZE 10
5
6  int main()
7  {
8     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9     int i, j;
10
11    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
12
13    for ( i = 0; i <= SIZE - 1; i++ ) {
14       printf( "%7d%13d        ", i, n[ i ]) ;
15
16       for ( j = 1; j <= n[ i ]; j++ )   /* print one bar */
17          printf( "%c", '*' );
18
19       printf( "\n" );
20    }
21
22    return 0;
23 }
```

Inizializzazione di array

Cicli for annidati

Stampa risultati



Outline

Output

```
Element         Value         Histogram
      0            19         ********************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
      5             9         *********
      6            13         *************
      7             5         *****
      8            17         *****************
      9             1         *
```

```
/* Memorizza in un array di interi i voti
ottenuti da sei  studenti e ne determina il
maggiore, il minore e la media    */

#include <stdio.h>

main()
{
int voti[6];          /*array di 6 elementi interi*/
int i, max, min;
float media;

printf("VOTI STUDENTI\n\n");
/* Immissione voti nell'array (da 0 a 5)*/
for(i=0; i<=5; i++) {
  printf("Voto studente n. %d: ", i+1);
  scanf("%d", &voti[i]);
}

/* Ricerca del maggiore */
max = voti[0];
for(i=1; i<=5; i++)
  if(voti[i]>max)
    max = voti[i];

/* Ricerca del minore */
min = voti[0];
for(i=1; i<=5; i++)
  if(voti[i]<min)
    min = voti[i];

/* Calcolo della media  */
media = voti[0];
for(i=1; i<=5; i++)
  media = media + voti[i];
media = media/6;

printf("Maggiore: %d\n", max);
printf("Minore: %d\n", min);
printf("Media: %f\n", media);
}
```


```
/* legge i punteggi di n concorrenti su due prove
   Determina la classifica                    */

#include <stdio.h>

#define MAX_CONC 1000    /* massimo numero di
concorrenti */
#define MIN_PUN  1       /* punteggio minimo per
ogni prova */
#define MAX_PUN  10      /* punteggio massimo per
ogni prova */

main()
{
/*prova1, prova2, totale sono vettori di dim. 1000*/
float prova1[MAX_CONC], prova2[MAX_CONC],
totale[MAX_CONC];

int i, n;
/*Numero di concorrenti n: >1 e <=1000*/
do {
  printf("\nNumero concorrenti: ");
  scanf("%d", &n);
}
while (n<1 || n>MAX_CONC);

/* Per ogni concorrente, si richiede il punteggio
nelle due prove */
for(i=0; i<n; i++) {
  printf("\nConcorrente n.%d \n", i+1);
  do {
    printf("Prima prova: ");
    scanf("%f", &prova1[i]);
  }
  while(prova1[i]<MIN_PUN || prova1[i]>MAX_PUN);
  do {
    printf("Seconda prova: ");
    scanf("%f", &prova2[i]);
  }
  while(prova2[i]<MIN_PUN || prova2[i]>MAX_PUN);
}
/* Calcolo media per concorrente */
for(i=0; i<n; i++)
  totale[i] = (prova1[i]+prova2[i])/2;
printf("\n      CLASSIFICA\n");
for(i=0; i<n; i++)
  printf("%f  %f  %f \n", prova1[i], prova2[i], totale[i]);
}
```


```c
#include <stdio.h>
#define DIM_INT 16
/*conversione decimale-binario */
void stampaBin ( int ); /*prototipo di funzione*/

main()
{
 char resp[2];
 int  num;
 resp[0] = 's';

 while( resp[0] == 's' ) {
  printf("\nInserisci un intero positivo: ");
  scanf("%d", &num);
  printf("La sua rappresentazione binaria e': ");

  stampaBin( num );          /*chiamata della
funzione*/

  printf("\nVuoi continuare? (s/n): ");
  scanf("%s",resp);
 }
}
```

```c
void stampaBin( int v )
{
 int i, j;
 char a[DIM_INT];
/*DIM_INT è globale*/
 if (v == 0)
  printf("%d", v);
 else  {
  for( i=0; v != 0; i++)   {
   a[i] = v % 2;
   v /= 2;
  }
  for(j = i-1 ; j >= 0; j--)
   printf("%d", a[j]);
 }
}
```


Array di caratteri

- Array di caratteri
  - La stringa **"hello"** è un array di caratteri **statico**
  - Gli array di caratteri si possono inizializzare usando stringhe di caratteri:

    ```
    char string1[] = "first";
    ```

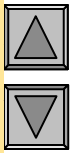
    - Il carattere nullo **'\0'** termina la stringa
    - **string1** ha 6 elementi

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

  - **string1[3]** è il carattere **'s'**
  - Il *nome dell'array è anche il suo indirizzo in memoria*, quindi per gli array il simbolo **&** non è necessario in **scanf**

    ```
    scanf( "%s", string2 ) ;
    ```

    - Legge I caratteri finchè non trova il carattere nullo



Outline

```
1  /* array di caratteri*/
2
3  #include <stdio.h>
4
5  int main()
6  {
7     char string1[ 20 ], string2[] = "string literal";
8     int i;
9
10    printf(" Enter a string: ");
11    scanf( "%s", string1 );
12    printf( "string1 is: %s\nstring2: is %s\n"
13            "string1 with spaces between characters is:\n",
14            string1, string2 );
15
16    for ( i = 0; string1[ i ] != '\0'; i++ )
17       printf( "%c ", string1[ i ] );
18
19    printf( "\n" );
20    return 0;
21 }
```

Inizializza le stringhe

Input della stringa string1

Stampa le stringhe

Ciclo for che stampa I singoli caratteri della stringa 1

```
Enter a string: Hello
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Program Output

Array bidimensionali

Gli array bidimensionali sono organizzati per righe e per colonne, come *matrici*.

La specifica di un array bidimensionale prevede l'indicazione del **tipo** di dati contenuti nell'array, del **nome** e delle **due dimensioni**, cioè numero di righe e numero di colonne, racchiusa ciascuna tra parentesi quadre.

Ad esempio, la dichiarazione che segue specifica un array bidimensionale di numeri reali, organizzati su 4 righe e 6 colonne, per un totale di 24 elementi:

```
float livelli[4][6];
```


L'accesso ai singoli elementi dell'array bidimensionale avviene in modo analogo a quanto avviene per gli array monodimensionali, specificando gli indici della riga e della colonna dell'elemento di interesse, ad esempio:

livelli[2][4]

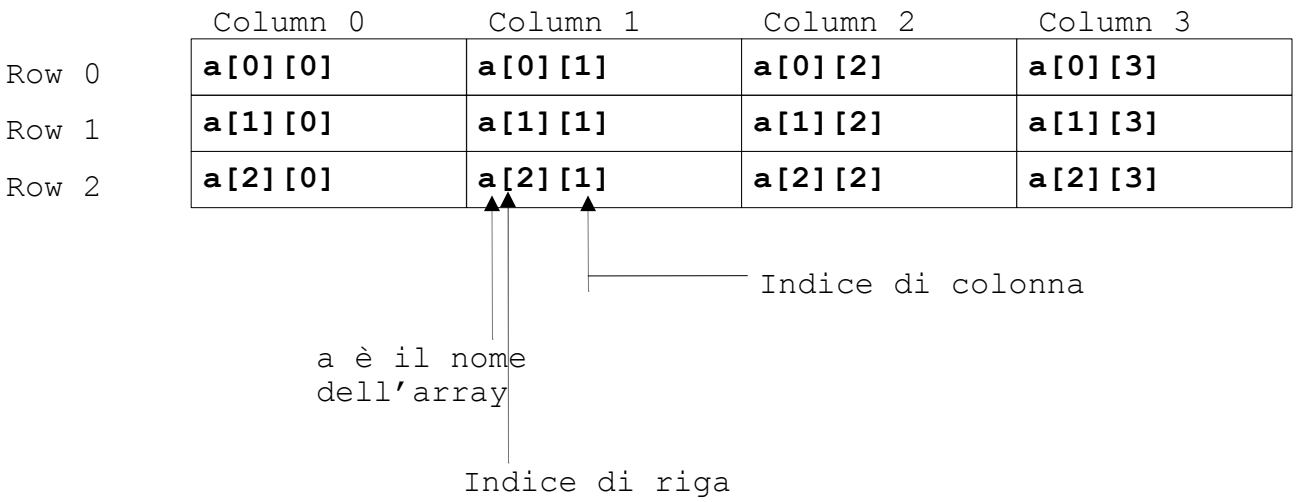
Come esempio di scansione degli elementi di un array bidimensionale si consideri lo stralcio di codice qui riportato, in cui si accede riga per riga ad ogni elemento dell'array, mediante due cicli annidati. Il ciclo più esterno scandisce le righe, quello più interno le colonne.

```
/* Acquisizione da tastiera dei valori della matrice livelli[4][6] */
 for (i = 0; i < 4; i++)
        for(j = 0; j < 6; j++)
        {
         printf("Inserisci l'elemento riga %d colonna %d: ", i, j);
        scanf("%f", &livelli[i][j]);
         }
```

Gli elementi vengono memorizzati per righe, quindi è più veloce accedere per righe ai dati memorizzati.

Array multidimensionali (cont.)

- Gli array multidimensionali
  - Sono tabelle di m righe e n colonne (*array mxn*)
  - Come per le matrici si deve specificare il numero di righe e colonne



Array multidimensionali (cont.)

- Inizializzazione

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

1	2
3	4

  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

- Riferimento ai singoli elementi
  - Specify row, then column

```
printf( "%d", b[ 0 ][ 1 ] );
```


/* Caricamento di una matrice 4x3*/

```
#include <stdio.h>

int mat[4][3];

main()
{
int i, j;

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<4; i++)
  for(j=0; j<3; j++) {
    printf("Inserisci linea %d colonna %d val: ", i, j);
    scanf("%d", &mat[i][j]);
  };

/* Visualizzazione */
for(i=0; i<4; i++) {
  printf("\n");
  for(j=0; j<3; j++)
    printf("%5d", mat[i][j]);
}
}
```


```
/* Caricamento di una matrice
   le cui dimensioni <100x100 vengono decise
dall'utente */

#include <stdio.h>
#define MAXLINEE 100
#define MAXCOLONNE 100
int mat[MAXLINEE][MAXCOLONNE];
main()
{
int n, m;
int i, j;

/* Richiesta delle dimensioni */
do {
  printf("\nNumero di linee: ");
  scanf("%d", &n);
}
while((n>=MAXLINEE) || (n<1));

do {
  printf("Numero di colonne: ");
  scanf("%d", &m);
}
while((m>=MAXCOLONNE) || (m<1));

printf("\n \n CARICAMENTO DELLA
MATRICE \n \n");

for(i=0; i<n; i++)

  for(j=0; j<m; j++) {

    printf("Inserisci linea %d colonna %d
val:", i, j);

    scanf("%d", &mat[i][j]);

  };


/* Visualizzazione */

for(i=0; i<n; i++) {

  printf("\n");

  for(j=0; j<m; j++)

    printf("%5d", mat[i][j]);

}

}
```


/* Calcolo del prodotto righe x colonne di due matrici */

```
#include <stdio.h>
#define N 4
#define P 3
#define M 5
int mat1[N][P];        /* prima matrice */
int mat2[P][M];        /* seconda matrice */
int pmat[N][M];        /* matrice prodotto */

main()
{
int i, j, k;
printf("\n \n CARICAMENTO DELLA PRIMA
MATRICE \n \n");
for(i=0; i<N; i++)
  for(j=0; j<P; j++) {
    printf("Inserisci linea %d colonna %d val:", i, j);
    scanf("%d", &mat1[i][j]);
  };
printf("\n \n CARICAMENTO DELLA
SECONDA MATRICE \n \n");
for(i=0; i<P; i++)
  for(j=0; j<M; j++) {
    printf("Inserisci linea %d colonna %d val:", i, j);
    scanf("%d", &mat2[i][j]);
  };


/* Calcolo del prodotto */
for(i=0; i<N; i++)
  for(j=0; j<M; j++) {
    pmat[i][j] = 0;
    for(k=0; k<P; k++)
      pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
  };
printf("\n \n PRIMA MATRICE \n ");
for(i=0; i<N; i++) {
  printf("\n");
  for(j=0; j<P; j++)
    printf("%5d", mat1[i][j]);
}
printf("\n \n SECONDA MATRICE \n ");
for(i=0; i<P; i++) {
  printf("\n");
  for(j=0; j<M; j++)
    printf("%5d", mat2[i][j]);
}
printf("\n \n MATRICE PRODOTTO \n ");
for(i=0; i<N; i++) {
  printf("\n");
  for(j=0; j<M; j++)
    printf("%5d", pmat[i][j]);
}
}
```


Chiamata di funzioni
Passaggio parametri per valore e per indirizzo

Passaggio per valore

- Si effettua una copia degli argomenti (parametri) passati alla funzione
- Le modifiche all'interno della funzione non alterano i valori originali
- Si usa quando non è richiesat la modifica dei parametri
  - E' utile per evitare modifiche accidentali

Passaggio per indirizzo

- Vengono passati gli argomenti originali
- Le modifiche all'interno della funzione hanno effetto sui valori originali

Per ora abbiamo parlato solo del passaggio parametri per valore

Passaggio di array a funzioni

- Passaggio di array
  - Si passa il nome dell'array senza parentesi quadre

```
int myArray[ 24 ];
myFunction( myArray, 24 );
```

    - E' bene in genere passare anche la dimensione dell'array
  - Gli array sono passati per indirizzo
  - Il nome dell'array è l'indirizzo del suo primo elemento
  - La funzione "sa" dove è memorizzato l'array
    - Può quindi effettuare modifiche sugli elementi dell'array originale
- Passaggio di elementi di un array
  - Il passaggio avviene per valore (copia dell'originale)
  - Si passa il nome e la posizione fra parentesi (**`myArray[3]`**)

Passaggio di array a funzioni (cont.)

Il passaggio di array a sottoprogrammi viene sempre fatto ***per indirizzo o riferimento***, passando per valore l'indirizzo dell'array, e lasciando quindi di fatto accessibile al sottoprogramma l'array con la possibilità di modificarne il contenuto. Per questo motivo è necessario prestare estrema attenzione durante l'accesso agli elementi dell'array.

Ogniqualvolta si passa un array pluridimensionale è necessario indicare - tra parentesi quadre - *tutte le dimensioni dell'array ad eccezione della prima*. Ne consegue che per un array monodimensionale la dichiarazione dell'array come parametro viene fatta così:

```
void funzionex(int numeri[], ...)
{
...
}
```

Nel caso di array pluridimensionali, o più comunemente i bidimensionali, la dichiarazione deve essere fatta come segue:

```
void funzioney(int livelli[][6], ...)
{
...
}
```


La specifica del numero di colonne è un'informazione di servizio e non può essere utilizzata dal programmatore come specifica del numero di colonne dell'array.

In tal senso, **è buona norma**, passare sempre, mediante ulteriori parametri, il numero di elementi dell'array monodimensionale, o il numero di righe e di colonne in array bidimensionali, come mostrato nei seguenti stralci di codice:

```
void funzionex(int numeri[], int num_elem)
{
 ...
}
void funzioney(int livelli[][6], int num_righe, int num_colonne)
{
...
}
```


È possibile fare alcune considerazioni che motivano l'opportunità di passare ad un sottoprogramma anche le dimensioni dell'array, e fanno concludere che la prima soluzione costituisce un codice migliore.

In primo luogo, *ad un sottoprogramma potrebbe venire passata solo una porzione ridotta dell'intero array*. Si ricordi infatti che dovendo specificare a priori la dimensione di un array in fase di dichiarazione, spesso si deve sovradimensionare l'array onde evitare di avere problemi di memoria. Non necessariamente tutti gli elementi verranno quindi utilizzati. In tal caso, quando si passa l'array come parametro, è importante specificare le dimensioni su cui lavorare, anche tenendo presente che è spesso impossibile distinguere in base al contenuto dell'array quali elementi siano "validi" e quali non lo siano.

Una seconda considerazione a favore del passaggio esplicito delle dimensioni dell'array mediante opportuni parametri, è la *possibilità di riutilizzare i sottoprogrammi* senza doverli "aggiustare" di volta in volta.

Nel primo caso, indipendentemente dalla dimensione dell'array utilizzato, il sottoprogramma funziona perfettamente così come è.

Nella seconda soluzione invece, sarà necessario accertarsi che l'array sia effettivamente di dimensione N, e nel caso in cui ci siano array di dimensione diverse, sarà necessario scrivere funzioni diverse. La prima soluzione costituisce dunque una soluzione più flessibile e riutilizzabile.

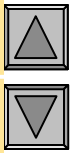
Questa considerazione purtroppo non è universale. Il fatto che nel passaggio di array pluridimensionali sia necessario specificare le dimensioni ad eccezione della prima fa sì che una funzione possa essere riutilizzata solo per array che hanno tutti le stesse dimensioni - ad eccezione della prima!

Passaggio di array a funzioni (cont.)

- Prototipo della funzione

```
void modifyArray( int b[], int arraySize );
```

  - Nei prototipi i nomi dei parametri sono opzionali
    - **`int b[]`** può essere semplicemente **`int []`**
    - **`int arraySize`** può essere semplicemente **`int`**



Outline

```c
/* Fig. 6.13: fig06_13.c
   Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

void modifyArray( int [], int );  /* appears strange */
void modifyElement( int );

int main()
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;

   printf( "Effects of passing entire array call "
           "by reference:\n\nThe values of the "
           "original array are:\n" );

   for ( i = 0; i <= SIZE - 1; i++ )
      printf( "%3d", a[ i ] );

   printf( "\n" );
   modifyArray( a, SIZE );  /* passed call by reference */
   printf( "The values of the modified array are:\n" );

   for ( i = 0; i <= SIZE - 1; i++ )
      printf( "%3d", a[ i ] );

   printf( "\n\n\nEffects of passing array element call "
           "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
   modifyElement( a[ 3 ] );
   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
   return 0;
}
```

Dichiarazione delle funzioni

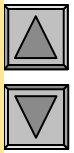
Inizializzazione di a

L'array è passato per indirizzo e può essere modificato

Passaggio di array alla funzione

L'elemento dell'array è passato per valore e non può essere modificato

Passaggio di un elemento dell'array alla funzione



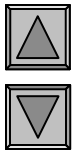
Outline

```
33
34 void modifyArray( int b[], int size )
35 {
36    int j;
37
38    for ( j = 0; j <= size - 1; j++ )
39       b[ j ] *= 2;
40 }
41
42 void modifyElement( int e )
43 {
44    printf( "Value in modifyElement is %d\n", e *= 2 );
45 }
```

Definizione delle funzioni

```
Effects of passing entire array call by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call by value:


The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Output del programma



Outline

```
/* Fig. 6.22: fig06_22.c
   Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

int minimum(int [][ EXAMS ], int, int );
int maximum(int [][ EXAMS ], int, int );
double average(int [], int );
void printArray(int [][ EXAMS ], int, int );

int main()
{
   int student;
 int studentGrades[ STUDENTS ][ EXAMS ] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   printf( "The array is:\n" );
   printArray( studentGrades, STUDENTS, EXAMS );
   printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
           minimum( studentGrades, STUDENTS, EXAMS ),
           maximum( studentGrades, STUDENTS, EXAMS ) );

   for ( student = 0; student <= STUDENTS - 1; student++ )
      printf( "The average grade for student %d is %.2f\n",
              student,
              average( studentGrades[ student ], EXAMS ) );

   return 0;
}
```

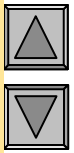
Definizione delle funzioni

Ogni riga è uno studente, ogni colonna è il voto per ciascun esame

Inizializzazione della matrice studentGrades

Chiamata della funzione printArray

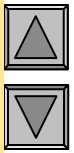
Chiamate delle funzioni maximum,minimum,average



Outline

Definizione della funzione minimum

```
33
34 /* Find the minimum grade */
35 int minimum(int grades[][ EXAMS ],
36              int pupils, int tests )
37 {
38    int i, j, lowGrade = 100;
39
40    for ( i = 0; i <= pupils - 1; i++ )
41       for ( j = 0; j <= tests - 1; j++ )
42          if ( grades[ i ][ j ] < lowGrade )
43             lowGrade = grades[ i ][ j ];
44
45    return lowGrade;
46 }
47
48 /* Find the maximum grade */
49 int maximum(int grades[][ EXAMS ],
50             int pupils, int tests )
51 {
52    int i, j, highGrade = 0;
53
54    for ( i = 0; i <= pupils - 1; i++ )
55       for ( j = 0; j <= tests - 1; j++ )
56          if ( grades[ i ][ j ] > highGrade )
57             highGrade = grades[ i ][ j ];
58
59    return highGrade;
60 }
61
62 /* Determine the average grade for a particular exam */
63 double average(int setOfGrades[], int tests )
64 {
```



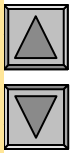
Outline

```
33
34 /* Find the minimum grade */
35 int minimum(int grades[][ EXAMS ],
36              int pupils, int tests )
37 {
38    int i, j, lowGrade = 100;
39
40    for ( i = 0; i <= pupils - 1; i++ )
41       for ( j = 0; j <= tests - 1; j++ )
42          if ( grades[ i ][ j ] < lowGrade )
43             lowGrade = grades[ i ][ j ];
44
45    return lowGrade;
46 }
47
48 /* Find the maximum grade */
49 int maximum(int grades[][ EXAMS ],
50             int pupils, int tests )
51 {
52    int i, j, highGrade = 0;
53
54    for ( i = 0; i <= pupils - 1; i++ )
55       for ( j = 0; j <= tests - 1; j++ )
56          if ( grades[ i ][ j ] > highGrade )
57             highGrade = grades[ i ][ j ];
58
59    return highGrade;
60 }
61
62 /* Determine the average grade for a particular exam */
63 double average(int setOfGrades[], int tests )
64 {
```

Definizione della funzione minimum

Definizione della funzione maximum

Definizione della funzione average

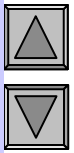


Outline

```
65    int i, total = 0;
66
67    for ( i = 0; i <= tests - 1; i++ )
68       total += setOfGrades[ i ];
69
70    return total / tests;
71 }
72
73 /* Print the array */
74 void printArray(int grades[][ EXAMS ],
75                 int pupils, int tests )
76 {
77    int i, j;
78
79    printf( "                  [0]   [1]   [2]   [3]" );
80
81    for ( i = 0; i <= pupils - 1; i++ ) {
82       printf( "\nstudentGrades[%d] ", i );
83
84       for ( j = 0; j <= tests - 1; j++ )
85          printf( "%-5d", grades[ i ][ j ] );
86    }
87 }
```

Definizione della funzione printArray

Allinea a sinistra in un campo di 5 cifre



Outline

Program Output

```
The array is:
                  [0]   [1]   [2]   [3]
studentGrades[0] 77    68    86    73
studentGrades[1] 96    87    89    78
studentGrades[2] 70    90    86    81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```