LE FUNZIONI – PARTE II

Manuale linguaggio C

Principio di funzionamento

```
int main(void)
{
        int x,y;

        /* leggi un numero compreso tra 50 e 100 e memorizzalo in x */
        /* leggi un numero compreso tra 1 e 10 e memorizzalo in y */

        printf("%d %d\n", x, y);
}
```

Principio di funzionamento

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

Principio di funzionamento

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

Principio di funzionamento

min=50	max=100

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

Principio di funzionamento

min=50 | max=100

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

chiamante

chiamato

Principio di funzionamento

min=1 max=10

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

Principio di funzionamento

min=1 max=10

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        y=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

chiamante

chiamato

Principio di funzionamento

La definizione di una funzione delimita un frammento di codice riutilizzabile più volte

La funzione può essere **chiamata** più volte

Può ricevere **argomenti** diversi in ogni chiamata

Può restituire un **valore di ritorno** al chiamante tramite l'istruzione **return**

Principio di funzionamento

```
int main(void)
{
        int x,y;

        x=leggi(50, 100);
        x=leggi(1, 10);

        printf("%d %d\n", x, y);
}
```

```
int leggi (int min, int max)
{
        int val;

        do {
            scanf("%d", &val);
        } while (val<min || val>max);
        return val;
}
```

Le funzioni possono ricevere dei parametri dal proprio chiamante:

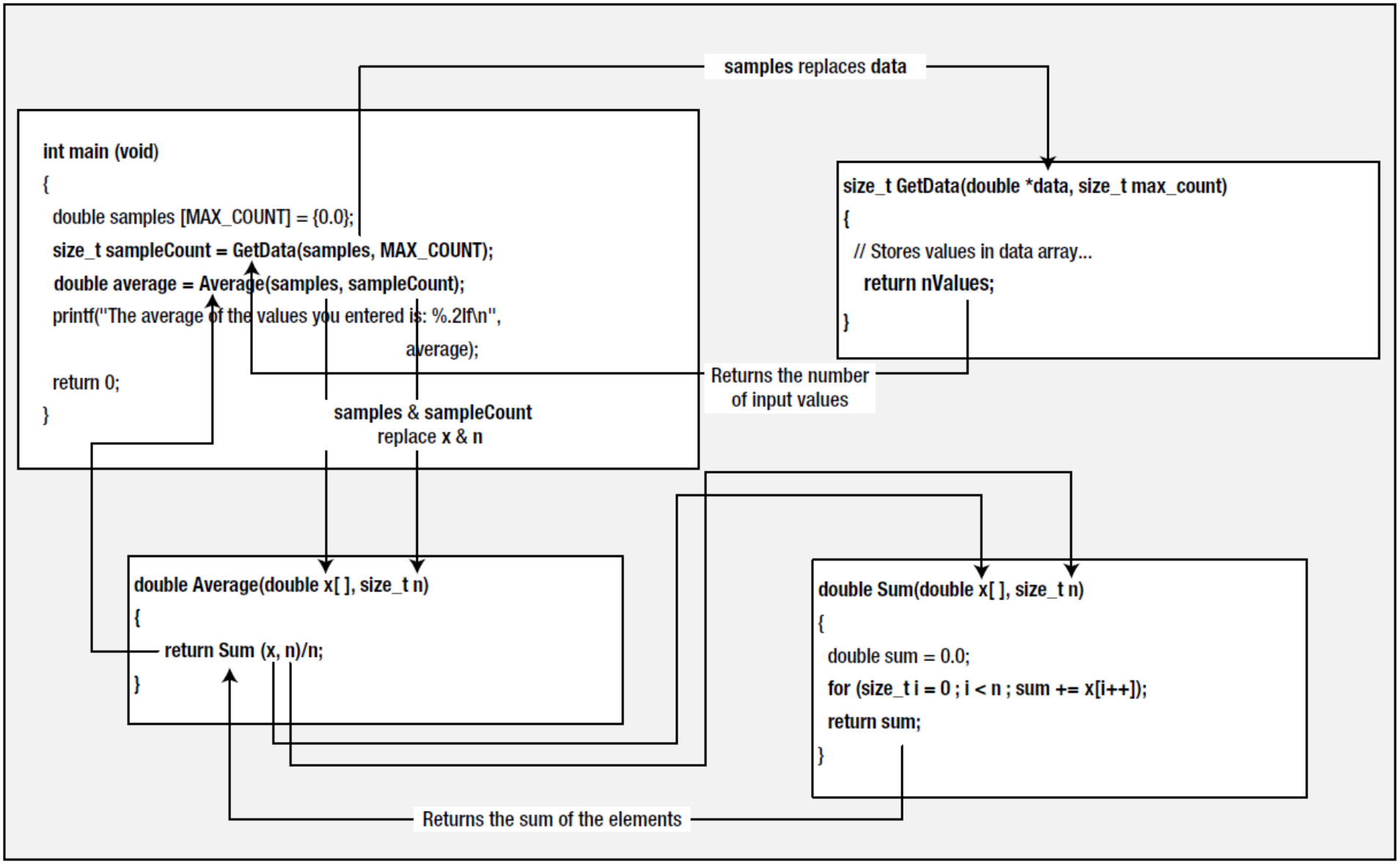
Nella **funzione**:

parametri formali: nomi interni dei parametri

Nel **chiamante**:

parametri attuali: valori effettivi (costanti, variabili, espressioni)

Principio di funzionamento - esempio



Tipi di dato restituiti dalle funzioni

Le funzioni possono restituire:

- dati di tipo semplice, come char, int, long, float, double, puntatori a void, o puntatori a qualche tipo di dato,
- strutture (struct) o puntatori a struct.

All'atto della chiamata, il valore restituito da una funzione:

```
double somma (double f, double g);
```

può essere utilizzato come espressione booleana:

```
if ( somma(a,b) > 100.3 )
```

può essere utilizzato come membro di destra in un'istruzione di assegnamento:

```
f = somma(a,b);
```

oppure può non essere considerato affatto:

```
somma(a,b);
```

Esempio: calcolo del fattoriale

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

void main()
{
        int n;
        double fatt;
        printf("intero---->");
        scanf("%d", &n);
        fatt=calcola_fattoriale(n);
        printf("\n Il fattoriale di %d e' %lf\n",n,fatt);
}

double calcola_fattoriale(int n)
{
        int i;
        double fatt=1;
        for (i=1; i<=n; i++)
                fatt *= i;
        return(fatt);
}
```

Provare a modificare il codice in modo tale il risultato calcolato venga restituito come secondo parametro della funzione, cioè:

void calcola_fattoriale(int n, double * risultato)

VETTORI E ARGOMENTI DI FUNZIONE

Manuale linguaggio C

Vettori e argomenti di funzione

Quando il parametro di una funzione è costituito da un vettore unidimensinale, la lunghezza del vettore può essere omessa. In questo caso, se la funzione ne ha bisogno, dovremo fornire la lunghezza come ulterirore parametro.

A causa del significato che il nome del vettore ha in C, gli array vengono **sempre passati per indirizzo**. Le funzioni non dispongono di una copia dei vettori ma lavorano sull'originale, quindi occorre prestare attenzione agli **effetti collaterali**.

Esempio di prototipo:

```
int sum(int a[], int size);
```

Con [] il compilatore riconosce il vettore

Passiamo la lunghezza del vettore

La dimensione del vettore, anche se presente, non sarebbe comunque disponibile alla funzione chiamata, tanto vale ometterla. Il compilatore non controllerà che gli argomenti abbiano veramente la dimensione specificata; inoltre specificarla fa credere che alla funzione possano essere passati solo vettori di quella lunghezza, mentre di fatto è possibile passare vettori di lunghezza qualsiasi.

Vettori e argomenti di funzione

```
int sum(int a[], int size)
{
int sum=0, i;
for(i=0;i<size;++i)
        sum+=a[i];
return(sum);
}

int main()
{
  int a[3]={1,2,3};
  sum(a,3);
  return 0;
}
```

equivalente

```
int sum(int *a, int size)
{
int sum=0, i;
for(i=0;i<size;++i)
        sum+=a[i];
return(sum);
}

int main()
{
  int a[3]={1,2,3};
  sum(a,3);
  return 0;
}
```

```
int main()
{
  int a[3]={1,2,3};
  int b[4]={4,5,6,7};
  printf("somma %d\n",sum(a,3));
  printf("somma %d\n", sum(b,4));
  return 0;
}
```

Un vettore (e quindi anche una stringa) viene sempre passato per **indirizzo**.

Vettori e argomenti di funzione

Un parametro formale del prototipo dichiarato di tipo "**array di tipo T**", completo o incompleto, viene automaticamente convertito in un parametro formale di tipo "**puntatore a tipo T**".
Cioè, le dichiarazioni seguenti della funzione es_function():

```
int es_function ( int array [10] );     /* array tipo int completo */
int es_function ( int array [ ] );      /* array tipo int incompleto */
int es_function ( int *array );         /* puntatore a tipo int */
```

prototipi

sono tutti **equivalenti** tra loro. Ma anche:

```
int es_function ( int [10] );           /* array tipo int completo */
int es_function ( int [ ] );            /* array tipo int incompleto */
int es_function ( int * );              /* puntatore a tipo int */
```

Esempio

```
#include <stdio.h>
void init_array (double array [ ] , int size);

int main ( void )
{
        int ind ;
        double a [10];
        init_array (a, 10);      // oppure: init_array(&a[0], 10);
        for (ind =0 ; ind <10 ; ++ind )
                printf ( "array[%d] = %.3f\n" , ind , a[ind] ) ;
        return 0;
}

void init_array (double *array, int size)
{
        int ind ;
        for (ind=0; ind <size; ++ind )
                array[ind] = (double) ind ;
        return ;
}
```

due dichiarazioni diverse dei parametri formali ma completamente equivalenti

```
no: init_array (a[0], 10) ;  /* passa primo elemento array */
    init_array(&a, 10) ;     /* passa puntatore ad array */
```

oppure anche:
*(array + ind) = (double) ind ;

che variabile è? Ha qualche legame con la sua omonima presente nel main?

Esempio

```
void main(void)
{
   int a[10];                                   /*Allocazione automatica*/
   float*v;
   …
   v = (float*)malloc(sizeof(float) * 20);  /*Allocazione dinamica */
   funz(a, v);
   …
}

void funz(int*b, float*w)
{
   …


   …
}
```

Usando i due parametri formali si accede rispettivamente alle aree di memoria del primo e del secondo vettore

Singole righe di vettori bidimensionali

```
void stampa_riga(int num[][10], int riga)
{
        int *p, i;

        p=(int *) &num[riga][0]; /* determina l'indirizzo del 1° elem della riga riga */

        for (i=0;i<10;++i)
                printf("%d", *(p+i));
}
```

Un array bidimensionale può essere ridotto ad un puntatore a un array di array monodimensionali. L'uso di una variabile puntatore distinta rappresenta un modo semplice per accedere tramite i puntatori agli elementi di una riga di un arry bidimensionale.

Singole righe di vettori bidimensionali

```
void stampa_riga(int num[][10], int riga)
{
        int *p, i;

        p=(int *) &num[riga][0]; /* determina l'indirizzo del 1° elem della riga riga */

        for (i=0;i<10;++i)
                printf("%d", *(p+i));
}
int main()
{
  int mat[2][10]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  stampa_riga(mat, 0);
  stampa_riga(mat, 0);
  stampa_riga(mat, 2);
  return 0;
}
```

Vettori bidimensionali e parametri

```
void funzione(int A[][C], int n)
{
        int i, j;

        for (i=0; i<n; i++)
          for (j=0; i<C; j++)
            A[i][j]=0;
}
```

```
void funzione(int (*A)[C], int n)
{
        int i, j;

        for (i=0; i<n; i++)
          for (j=0; j<C; j++)
            A[i][j]=0;
}
```

Il **nome di un vettore** in una dichiarazione di funzione è considerato come un **puntatore** al primo elemento del vettore

Passare vettori multidimensionali a funzioni

In generale, per gli array N-dimensionali è necessario passare le N-1 dimensioni alla funzione chiamata

```
void stampaMatrice (int a[][5]);
void stampaMatrice3D (int a[][5][4]);
void stampaMatrice4D (int a[][5][4][5]);
```

Il secondo parametro di una matrice è obbligatorio, mentre il primo può essere omesso

Per matrici pluri-dimensionali, occorre specificare tutti i valori tranne, eventualmente, il primo

Passare vettori multidimensionali a funzioni

```
#define LUN 5
void stampaMatrice (int [][LUN], int);
void riempiMatrice( int [][LUN], int);
…
int main()
{
        int matr[righe][LUN];
        …
        riempiMatrice (matr, righe);
        stampaMatrice(matr, righe);
        …
}
void riempiMatrice( int m[][LUN], int dim )
{
  m[1][0] = 137;

  m = NULL;
}
```

modifica conservata fuori dalla funzione

questa modifica NON viene mantenuta

Attenzione al passaggio dei parametri (struct)

```
struct luogo {
        char nome[50];
        int x;
        int y;
        int z;
};

int print_luogo(struct luogo l)
{
        printf("Nome = %s\n ascissa = %d\n ordinata = %d\n z = %d\n",
            l.nome, l.x, l.y, l.z);
        ....
}
```

La struttura è passata per **valore**: ne viene fatta una **copia locale** e ciò può essere dispendioso in termini di occupazione di memoria

Attenzione al passaggio dei parametri (struct)

```
struct luogo {
        char nome[50];
        int x;
        int y;
        int z;
};

int print_luogo(struct luogo *l)
{
        l->nome="Nome modificato dalla funzione";
        printf("Nome = %s\n ascissa = %d\n ordinata = %d\n z = %d\n",
            l->nome, l->x, l->y, l->z);
        ....
}
```

La struttura è passata per **indirizzo**: **non** viene fatta alcuna **copia locale** della struttura ma la funzione può modificare i valori dei campi della struttura

Restituzione di una struct

Vediamo un esempio di restituzione di una struct.

```
struct point { int x; int y; };
struct point crea_point( int x, int y )
{
        struct point p, p1;
        p.x = x; p1.x=x;
        p.y = y; p1.y=y;
        return p ;
}

void main(void)
{
        struct point p1;
        int x=21 , y = -10987;
        p1 = crea_point( x, y );
        printf ( "p1.x=%d p1.y=%d \n" , p1.x, p1.y );
}
```

cosa succede alle due strutture p e p1 che la funzione crea?

Vettori e funzioni: considerazioni

Il passaggio del tipo di dato **array tramite indirizzo** è estremamente conveniente:

- la funzione può operare su un vettore di grandi dimensioni senza farne una copia

N.B.: evitare di definire funzioni che prendono come parametro strutture contenenti array di grandi dimensioni: le strutture vengono passate per valore

Attenzione: quando si lavora con i vettori (parametri di funzione e variabili locali) accertarsi di non incorrere in problem di **buffer overflow** (evitare di scrivere codice potenzialmete insicuro).

Nell'usare gli array come parametri di funzione siamo vincolati alla loro dimensione: non c'è modo di definire funzioni che lavorino con **vettori generici**, le cui dimensioni non siano fissate a priori (sia unidimensionali che multidimensionali).

- Risolveremo usando l'**allocazione dinamica**

PROGETTI DI PROGRAMMAZIONE

Manuale linguaggio C

Progetto di programmazione – funzioni

Sia dato un vettore *a* di interi e il valore *n* che rappresenta la sua lunghezza. Scrivere una funzione C che restituisca il maggiore tra gli elementi di a.

```
int largest(int a[], int n)
{
  int i, max = a[0];

  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];

  return max;
}
```

Progetto di programmazione – funzioni

Sia dato un vettore *a* di interi e il valore *n* che rappresenta la sua lunghezza. Scrivere una funzione C che restituisca la media degli elementi di a.

```
double average(int a[], int n)
{
  int i, avg = 0;

  for (i = 0; i < n; i++)
    avg += a[i];

  return avg / n;
}
```

Progetto di programmazione – funzioni

Sia dato un vettore *a* di interi e il valore *n* che rappresenta la sua lunghezza. Scrivere una funzione C che restituisca il numero degli elementi di a che sono positivi.

```
int num_positive(int a[], int n)
{
  int i, count = 0;

  for (i = 0; i < n; i++)
    if (a[i] > 0)
      count++;

  return count;
}
```

Progetto di programmazione – funzioni

Scrivere una funzione C che cerchi all'interno di un vettore *a* di lunghezza *n* il valore più grande e il secondo valore più grande.

```
void find_two_largest(int a[], int n, int *largest, int *second_largest)
{
 int i;

 if (a[0] > a[1]) {
  *largest = a[0];
  *second_largest = a[1];
 } else {
  *largest = a[1];
  *second_largest = a[0];
 }

 for (i = 2; i < n; i++)
  if (a[i] > *largest) {
   *second_largest = *largest;
   *largest = a[i];
  } else if (a[i] > *second_largest)
   *second_largest = a[i];
}
```

Progetto di programmazione – funzioni

Scrivere una funzione C che stampi la rappresentazione binaria del valore *n* passatole come argomento assumendo che esso sia positivo.

```
#include <stdio.h>

void rappresentazione_binaria(int n);

int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);

  printf("Output: ");
  rappresentazione_binaria(n);
  printf("\n");

  return 0;
}
```

```
void rappresentazione_binaria(int n)
{
  if (n != 0) {
    rappresentazione_binaria(n / 2);
    putchar('0' + n % 2);
  }
}
```

Progetto di programmazione – funzioni

Invocare la funzione rappresentazione binaria (*br)* con argomento uguale a 53 provoca la seguente esecuzione:

```
br(53) finds that 53 is not equal to 0, so it calls
  br(26), which finds that 26 is not equal to 0, so it calls
    br(13), which finds that 13 is not equal to 0, so it calls
      br(6), which finds that 6 is not equal to 0, so it calls
        br(3), which finds that 3 is not equal to 0, so it calls
          br(1), which finds that 1 is not equal to 0, so it calls
            br(0), which finds that 0 is equal to 0, so it returns, causing
          br(1) to print 1 and return, causing
        br(3) to print 1 and return, causing
      br(6) to print 0 and return, causing
    br(13) to print 1 and return, causing
  br(26) to print 0 and return, causing
br(53) to print 1 and return.
```

Questo tipo di funzione si chiama **ricorsiva**, torneremo più avanti sull'argomento.

ARRAY MULTIDIMENSIONIONALI

Manuale linguaggio C

Esempio: allocazione matrice

Sappiamo dichiarare una matrice statica; se fosse 3x3, ad esempio:

```
double mat_stat[3][3];
```

Supponiamo di voler progettare una funzione che stampi il contenuto della matrice. La sua definizione potrebbe essere:

```
stampa_mat (double mat[3][3])
{
        int i,j;
        for (i=0; i<3; i++)
                for (j=0; i<3; i++)
                        printf("elemento %d %d della matrice: %5.0f\n", i, j, a[i][j]);
}
```

e la chiamata sarebbe:

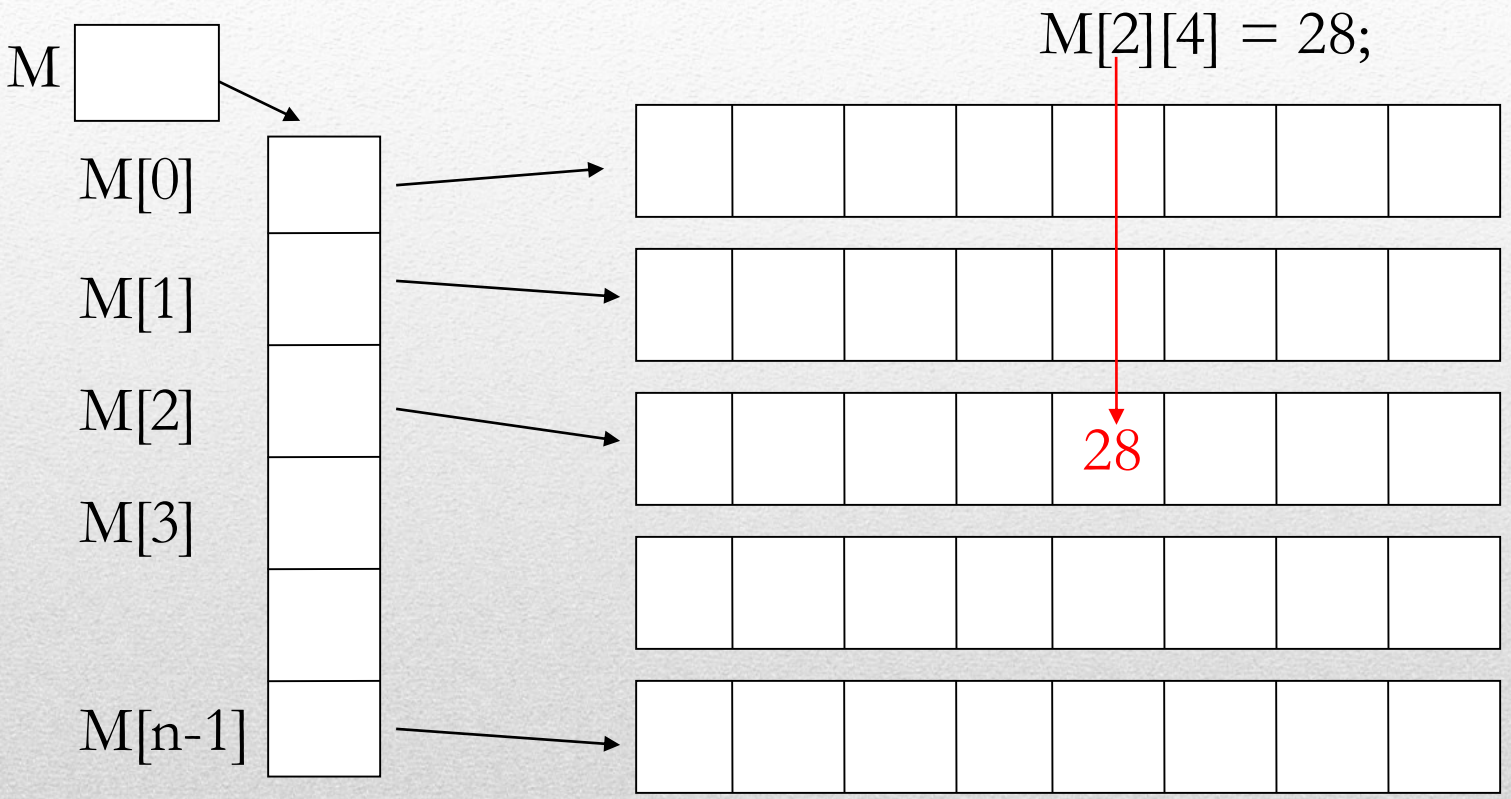
```
stampa_mat (a);
```

La funzione funzionerebbe però solo per matrici 3x3 … come definire funzioni più generali??

Vettori di Puntatori

Allocando dinamicamente le matrici possiamo lavorare in maniera più generale. Vogliamo costruire **dinamicamente** un dato M che sia un **array di n puntatori ad intero**, per poi allocare, per ciascuno dei puntatori ad int dell'array, spazio sufficiente ad **m** interi, ottenendo un array di vettori di interi. (allochiamo un array dinamico che contenga puntatori ad array dinamici)

Vettori di Puntatori



vettore dinamico
di puntatori a int

vettori dinamici di int anche di dimensioni differenti fra loro

Vettori di Puntatori

Se gli elementi dell'array **dinamico** puntato da M sono puntatori ad interi, il puntatore M sarà un puntatore di puntatore ad intero, cioè sarà un **int* *M;**

L'elemento intero che sta nella posizione c-esima dell'r-esimo vettore di interi verrà indicato mediante l'espressione: **M[r][c]**

In effetti questa struttura dati è spesso usata come matrice in C:

- **quando non sono note a priori le dimensioni della matrice da realizzare, e si preferisce non sovradimensionarla con un'allocazione statica**
- **quando si vuole realizzare una matrice in cui il numero di colonne per ciascuna riga è differente (ad es. vettori di stringhe)**

In questo caso ogni puntatore punta a un vettore di interi allocato usando la stessa dimensione. **In generale**, è possibile per ogni puntatore allocare un vettore di dimensioni diverse.

Vettori di Puntatori

```
/* alloca la struttura dati: vettore dinamico di puntatori a vettori dinamici di interi*/

int ** alloca_matrice(size_t righe, size_t colonne)
{
   int* *M;
   int i;

   if ( ! (M = (int**)malloc( righe * sizeof(int*) )) )                //vettore di int *
                        exit(1);
   for ( i=0; i<righe; i++)
            if ( ! (M[i] = (int*)malloc( colonne * sizeof(int) )) )    //vettore di int
                        exit(1);
   return M;
}
```

Funzioni su matrice dinamica

Possiamo ora usare come parametro di una funzione la matrice allocata dinamicamente. La funzione *stampa_mat* sarà definita come:

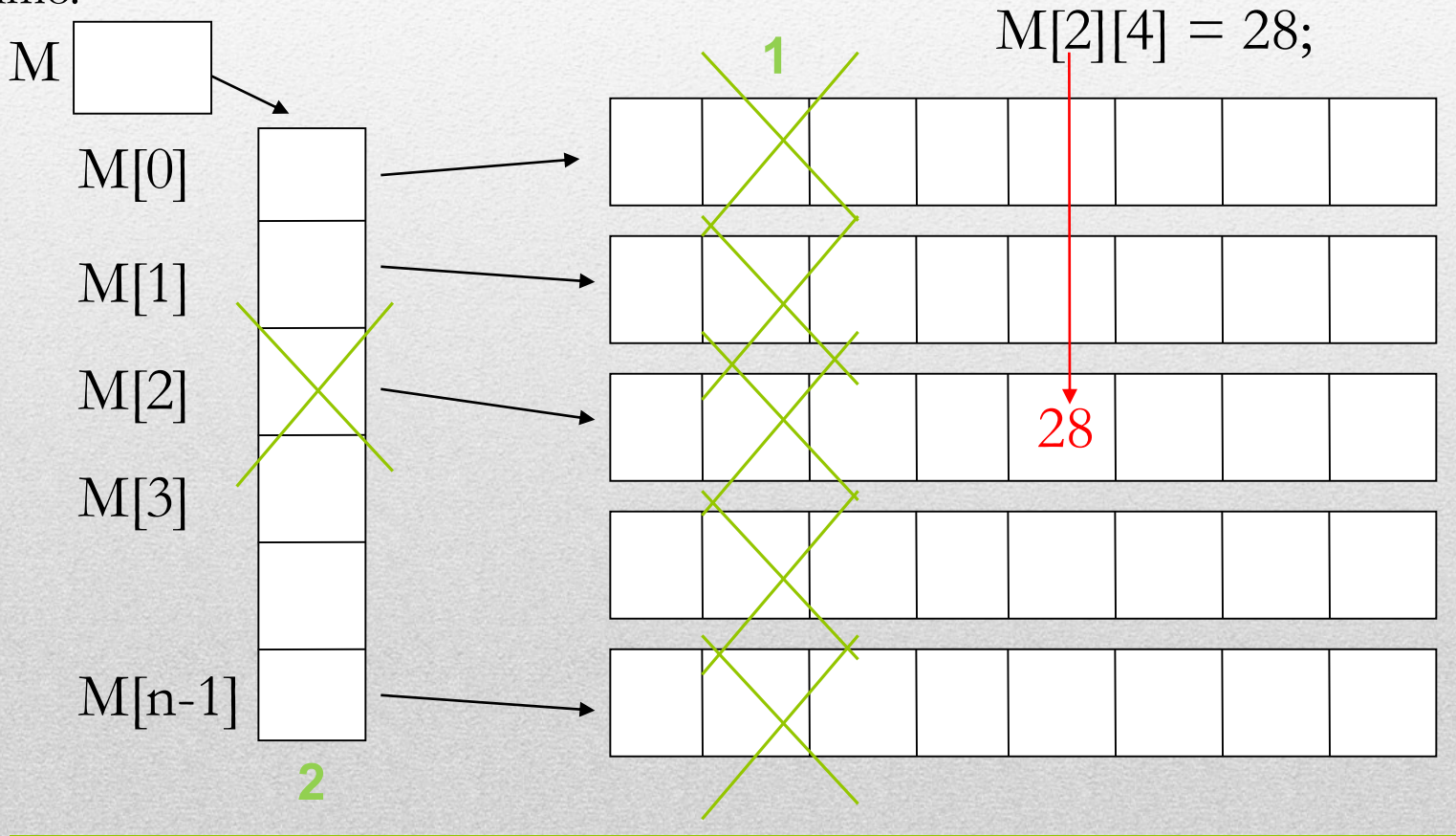
```
void stampa_mat (double **a, int righe, int colonne)
{
        int i,j;
        for (i=0; i<righe; i++)
            for (j=0; j<colonne; j++)
                printf("elemento %d %d della matrice: %5.0f\n", i, j, a[i][j]);
}
```

Abbiamo quindi scritto una sola funzione che può essere utilizzata per lavorare su matrici di differente dimensione. Una funzione di inizializzazione della matrice potrebbe essere:

```
void init_mat (double **a, int righe, int colonne, int init)
{
        int i,j;
        for (i=0; i<righe; i++)
            for (j=0; j<colonne; j++)
                a[i][j]=init;
}
```

Disallocare una matrice

Come tutta la memoria allocata dinamicamente anche quella allocata per rappresentare la matrice va deallocata. Serve deallocare prima tutti i vettori che corrispondono alle righe e poi il vettore principale che abbiamo allocato per primo.



Disallocare una matrice

Come tutta la memoria allocata dinamicamente anche quella allocata per rappresentare la matrice va deallocata. Serve deallocare prima tutti i vettori che corrispondono alle righe e poi il vettore principale che abbiamo allocato per primo.

```
/* dealloca la struttura dati: vettore dinamico di puntatori a vettori dinamici di interi*/

void dealloca_matrice(int **M, size_t righe)
{
   int i;

   if (M != NULL )
      for (i=0; i<righe; i++)
            free(M[i]);              //dealloca i vettori di interi
   free(M);                          //dealloca il vettore di puntatori ad intero
}
```

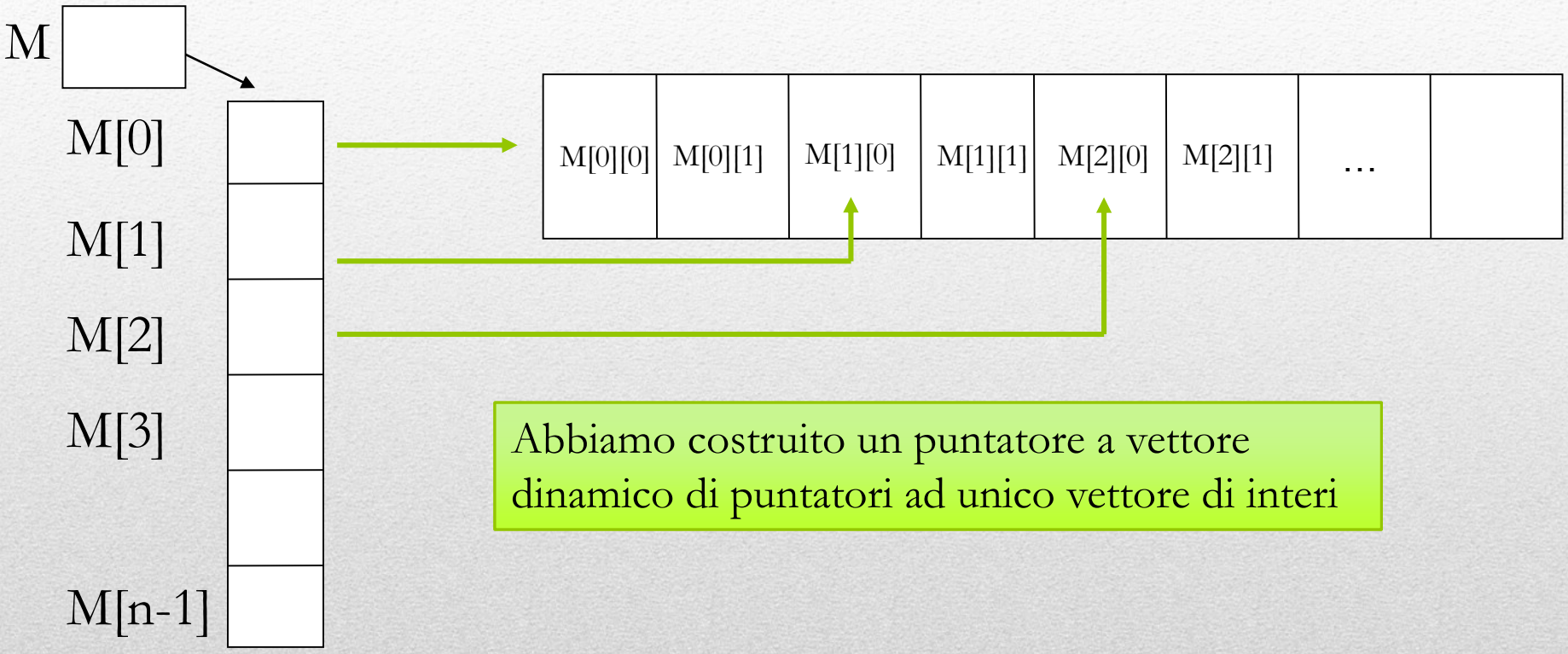
Vettori di Puntatori

Se avessimo scritto:

```
/* alloca la struttura dati: ??*/

int ** alloca_matrice(size_t righe, size_t colonne)
{
   int* *M;
   int i;

   if ( ! (M = (int**)malloc( righe * sizeof(int*) )) )          //vettore di (int *)
                        exit(1);
   M[0]= (int *) malloc(righe*colonne, sizeof(int));

   for ( i=1; i<righe; i++)
            M[i] = &M[0][i*colonne];
   return M;
}
```

Che tipo di struttura avremmo allocato?

Vettori di Puntatori



Abbiamo costruito un puntatore a vettore dinamico di puntatori ad unico vettore di interi

Funzioni su matrice dinamica

Possiamo accedere agli elementi della matrice usando la notazione vettoriale così come abbiamo fatto nel precedente caso:

```
void stampa_mat (int **a, int righe, int colonne)
{
        int i,j;
        for (i=0; i<righe; i++)
            for (j=0; j<colonne; j++)
                printf("elemento %d %d della matrice: %d\n", i, j, a[i][j]);
}
```

```
void init_mat (int **a, int righe, int colonne, int init)
{
        int i,j;
        for (i=0; i<righe; i++)
            for (j=0; j<colonne; j++)
                a[i][j]=init;
}
```

Disallocare una matrice

Per deallocare la matrice ci serve conoscere l'indirizzo del vettore di puntatori:

```
/* dealloca la struttura dati: vettore dinamico di puntatori a unico vettore di interi*/

void dealloca_mat(int **M)
{

   if  (M != NULL )
       free(M[0]);                    //dealloca il vettore di interi
   free(M);                           //dealloca il vettore di puntatori ad intero
}
```

Esercizio

Scrivere una funzione per inizializzare i valori della matrice con un solo ciclo:

```
#include <stdio.h>
#define R 2
#define C 3

//utilizzo unico puntatore che passa tutti gli elementi della matrice
//n mantiene il numero di righe, init il valore di inizializzazione
void init_matrice(int A[][C], int n, int init)
{
        int *p, *q;

        //p punta al primo elemento di A[][], q all'ultimo
        for(p=*A, q=*(A+n-1)+C-1; p<=q; p++)
          *p=init;
}
// CONTINUA
```

Esercizio

```
int main(void) {
  int M[R][C]={{1,2,3},{4,5,6}};
  int r,c;
  int (*p)[C] = M;          //puntatore per accedere agli elementi della matrice M

  for(r=0;r<R;r++)
    for(c=0;c<C;c++)
      printf("%d = %d\n",p[r][c], M[r][c]);

   init_matrice(M, R, 0);

   for(r=0;r<R;r++)
    for(c=0;c<C;c++)
      printf("%d = %d\n",p[r][c], M[r][c]);

   init_matrice(M, R, 10);

   for(r=0;r<R;r++)
    for(c=0;c<C;c++)
      printf("%d = %d\n",p[r][c], M[r][c]);
   return 0;
}
```

```
1 = 1
2 = 2
3 = 3
4 = 4
5 = 5
6 = 6
0 = 0
0 = 0
0 = 0
0 = 0
0 = 0
0 = 0
10 = 10
10 = 10
10 = 10
10 = 10
10 = 10
10 = 10
```

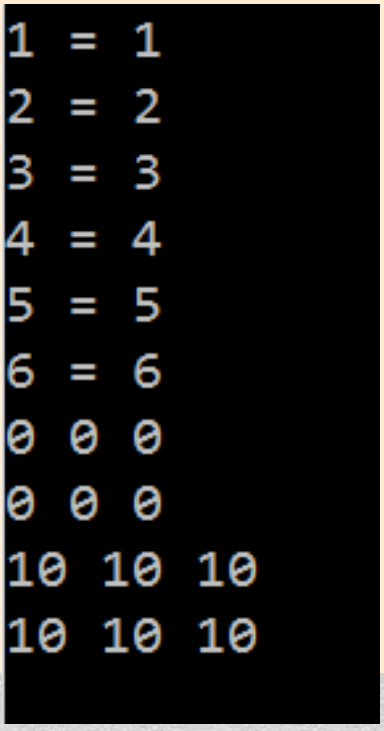
Esercizio

Scrivere una funzione per la stampa dei valori della matrice usando l'aritmetica dei puntatori:

```
void stampa_matrice(int A[][C], int n)
//n numero di righe
{
        int (*p)[C], (*q)[C];
        int *r, *s;

        //p punta al primo vettore di A[][], q all'ultimo
        for(p=A, q=A+n-1; p<=q; p++) {
                //r punta al primo elemento della riga corrente, s all'ultimo
                for(r=*p, s=*p+C-1; r<=s; r++)
                        printf("%d ", *r);
                printf("\n");
   }
 }
// CONTINUA
```

Esercizio

```
int main(void) {

  int M[R][C]={{1,2,3},{4,5,6}};
  int r,c;
  int (*p)[C] = M;          //puntatore per accedere agli elementi della matrice M

  for(r=0;r<R;r++)
    for(c=0;c<C;c++)
      printf("%d = %d\n",p[r][c], M[r][c]);


  init_matrice(M, R, 0);
  stampa_matrice(M,R);
  init_matrice(M, R, 10);
  stampa_matrice(M,R);

   return 0;
}
```



Esercizio

Scrivere una funzione analoga alla funzione di libreria *strcpy()* usando l'aritmetica dei puntatori e mantenendo lo stesso prototipo della *strcpy()*:

```
char * strcpy1(char *s1, const char *s2)
{
        int i=0;

        for (i=0; i<strlen(s2); i++)
          s1[i]=s2[i];
        s1[i]='\0';
        return s1;
}

char * strcpy2(char *s1, const char *s2)
{
        int i=0;

        while ((s1[i]=s2[i])!='\0')
          i++;
        return s1;
}
```

La lunghezza viene ricalcolata ad ogni ciclo

Non serve contatore e controllo su terminatore

Esercizio

```
char * strcpy3(char *s1, const char *s2)
{
        char *s=s1;

        while (*s1 = *s2) {
          s1++;
          s2++;
        }
        return s;
}

char * strcpy4(char *s1, const char *s2)
{
        char *s=s1;

        while (*s1++ = *s2++)
          ;
        return s;
}
```

Usiamo l'aritmetica dei puntatori; possiamo compattare ulteriormente il codice

l'incremento viene effettuato sull'indirizzo di memoria e non sul valore puntato dal puntatore. Essendo un post-incremento prima assegno, poi incrementato

Esercizio

```
void stampa_stringa(char *s)
{
   int i=0;
   while (s[i]!='\0') {
      printf("%c",s[i]);
      i++;
   }
   printf("\n");
}
```

```
void stampa_stringa1(char *s)
{
   printf("%c",*s);
   while (*s++)
     printf("%c",*s);
   printf("\n");
}
```

```
int main(void) {

  char string1[30]= "Prima stringa";
  char string2[30]= "Seconda stringa";


  strcpy1(string1,string2);
  stampa_stringa(string1);

  strcpy(string1,"Prima stringa");
  strcpy2(string1,string2);
  stampa_stringa(string1);

  strcpy(string1,"Prima stringa");
  strcpy3(string1,string2);
  stampa_stringa1(string1);

  strcpy(string1,"Prima stringa");
  strcpy4(string1,string2);
  stampa_stringa1(string1);

  return 0;
}
```

Esercizio

```
void stampa_stringa(char *s)
{
   int i=0;
   while (s[i]!='\0') {
      printf("%c",s[i]);
      i++;
   }
   printf("\n");
}
```

```
int main(void) {

  char string1[30]= "Prima stringa";
  char string2[30]= "Seconda stringa";


  strcpy1(string1,string2);
  stampa_stringa(string1);

  strcpy(string1,"Prima stringa");
  strcpy2(string1,string2);
  stampa_stringa(string1);

  strcpy(string1,"Prima stringa");
  strcpy3(string1,string2);
  stampa_stringa1(string1);

  strcpy(string1,"Prima stringa");
  strcpy4(string1,string2);
  stampa_stringa1(string1);

  return 0;
}
```

```
Seconda stringa
Seconda stringa
Seconda stringa
Seconda stringa
```