

Fondamenti di Informatica

Allievi Automatici
A.A. 2015-16

Array e Stringhe
Struct

Gli array

- Gruppi di celle consecutive
 - Rappresentano gruppi di variabili
 - (con lo stesso nome e lo stesso tipo)
- Per riferirsi a un **elemento**, si specificano:
 - Il nome dell'array
 - La **posizione** dell'elemento (indice)
- Sintassi: $\langle nomearray \rangle [\langle posizione \rangle]$
 - Il primo elemento ha indice **0**
 - L'n° elemento dell'array **v** è **v[n-1]**

dichiarazione
`int v[12];`

nome array

v[0]	-45
v[1]	6
v[2]	0
v[3]	72
v[4]	1543
v[5]	-89
v[6]	0
v[7]	62
v[8]	-3
v[9]	1
v[10]	6453
v[11]	78

posizione elemento ₂

Gli array

- **Array**: vettori di elementi **indicizzati, omogenei, memorizzati in celle di memoria consecutive**

Dichiarazione di un array:

```
char parola[12];
```

È un array di 12 elementi di tipo **char**, vale a dire un vettore di 12 caratteri

- La lunghezza dell'array è comunque decisa durante la compilazione del programma
- Nella dichiarazione, non si usano variabili per specificare la dimensione degli array

Gli array

- Sequenza di elementi consecutivi dello stesso tipo **in numero predeterminato e costante**
- Ogni elemento della sequenza è individuato da un indice con valore da 0 a N-1 (dove N è la dimensione dell'array)

Esempio

```
int v[100];
```

```
...
```

```
v[3] = 0;
```

n.b. se $i \geq 100$, $v[i]$ è un errore!

(e il comportamento è indefinito)

Gli array

- Gli elementi di un array sono normali variabili:

```
vett[0] = 3;
```

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

```
> 3
```

```
scanf("%d", &vett[1]);
```

```
> 17          vett[1] assume valore 17
```

- Si possono usare **espressioni** come indici:

```
se x = 3
```

```
vett[5-2] è equivalente a vett[3]
```

```
ed è equivalente a vett[x]
```

Come opera il calcolatore?

- `int v[100];`
 - alloca memoria per 100 elementi interi, a partire da un certo indirizzo di memoria
 - la dimensione deve essere nota al compilatore
 - deve essere una espressione **costante**
- Per accedere all' *i*-esimo elemento di *v*
 - calcola l'indice *i* (può essere un'espressione)
 - all'indirizzo della prima cella di *v* somma il numero di celle pari allo spazio occupato da *i* elementi
 - ottiene così l'indirizzo dell'elemento cercato
 - è possibile perché gli elementi sono tutti dello **stesso tipo**, e il tipo determina la dimensione in memoria

Inizializzazione di un array

- Sintassi compatta:

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

- Inizializzazione parziale: gli elementi più a destra sono posti a 0

```
int n[5] = {13};
```

 tutti gli altri elementi sono posti a 0

- Specificare *troppi* elementi tra le graffe è un errore di sintassi

- Se la lunghezza dell'array è omessa, gli inizializzatori la determinano:

```
int n[] = {5, 47, -2, 0, 24};
```

è equivalente a:

```
int n[5] = {5, 47, - 2, 0, 24};
```

- In tal caso la dimensione è inferita automaticamente, e si avranno 5 elementi nell'array (con indici che variano tra 0 e 4)

Operazioni sugli array

- Si opera sui singoli elementi, uno per volta
- **Non è possibile** operare sull'intero array, agendo su tutti gli elementi simultaneamente

```
/* come ricopiare array1 in array2 */
int array1[10], array2[10];
int i;
. . .
array2 = array1;                                /* ERRATO */
. . .
for( i = 0; i < 10; i++ ) {
    array2[i] = array1[i];    /* CORRETTO */
}
```


Esempi sugli array (I)

- **Dichiarazione del vettore:**

```
int a[20];
```

- **Inizializzazione del vettore (omogenea):**

```
for( i = 0; i <= 19; i++ )  
    a[i] = 0;
```

(alternativa alla dichiarazione, solo per lo zero: `int a[20] = {0} ;`)

- **Inizializzazione "da terminale":**

```
for( i = 0; i <= 19; i++ ) {  
    printf("Scrivi un intero: ");  
    scanf("%d", &a[i]);  
}
```

Esempi sugli array (II)

- **Ricerca del massimo:**

```
int max = a[0];  
for( i = 1; i <= 19; i++ )  
    if( a[i] > max )  
        max = a[i];
```

- **Calcolo della media:**

```
float media = a[0];  
for( i = 1; i <= 19; i++ )  
    media = media + a[i];  
media = media / 20;
```

Esempi sugli array (III)

- Calcolo di massimo, minimo e media di un vettore
 - È sufficiente una sola scansione del vettore (un solo ciclo)

```
int a[20];
int max, min, i;
float media;
for( i = 0; i <= 19; i++ ) {
    printf("Scrivi un intero: ");
    scanf("%d", &a[i]);
}
max = min = media = a[0];
for( i = 1; i <= 19; i++ ) {
    media = media + a[i];
    if( a[i] > max )
        max = a[i];
    if( a[i] < min )
        min = a[i];
}
media = media / 20;
```

*Si noti la dinamica
delle conversioni*



Un nuovo problema

- Mostrare una sequenza di 100 interi nell'**ordine inverso** rispetto a quello con cui è stata introdotta dall'utente (stdin)
 - Con un array?
 - Senza array?

```
/* Programma InvertiSequenza */
```

```
int main() {  
    int i, a[100];  
    i = 0;  
    while( i < 100 ) {  
        printf("fornisci un valore intero");  
        scanf("%d", &a[i]);  
        i++;  
    }  
    i--;  
    while( i >= 0 ) {  
        printf("%d\n", a[i]);  
        i--;  
    }  
    return 0;  
}
```

esaminare criticamente i cicli!

N.B.: funziona SOLO per un array di 100 elementi
Che cosa possiamo fare se sono di meno?
E se (peggio) sono di più?

"Generalizziamo" con la direttiva *#define*

- In testa al programma:
`#define LUNG_SEQ 100`
- Così possiamo adattare la lunghezza del vettore alle eventuali mutate esigenze senza riscrivere la costante 100 in molti i punti del programma
 - Il preprocessore sostituisce nel codice `LUNG_SEQ` con 100 *prima* della compilazione
- La lunghezza dell'array, quindi, *anche in questo caso è decisa al momento della compilazione del programma*
- Nella dichiarazione degli array non si usano **mai** variabili per specificarne la dimensione

```
/* Programma InvertiSequenza */
```

```
#define LUNG_SEQ 100
```

```
int main() {
```

```
    int i, a[LUNG_SEQ];
```

```
    i = 0;
```

```
    while( i < LUNG_SEQ ) {
```

```
        printf("fornisci un valore intero");
```

```
        scanf("%d", &a[i]); i++;
```

```
    }
```

```
    i--;
```

```
    while( i >= 0 ) {
```

```
        printf("%d\n", a[i]);
```

```
        i--;
```

```
    }
```

```
    return 0;
```

```
}
```

Parametrizzazione



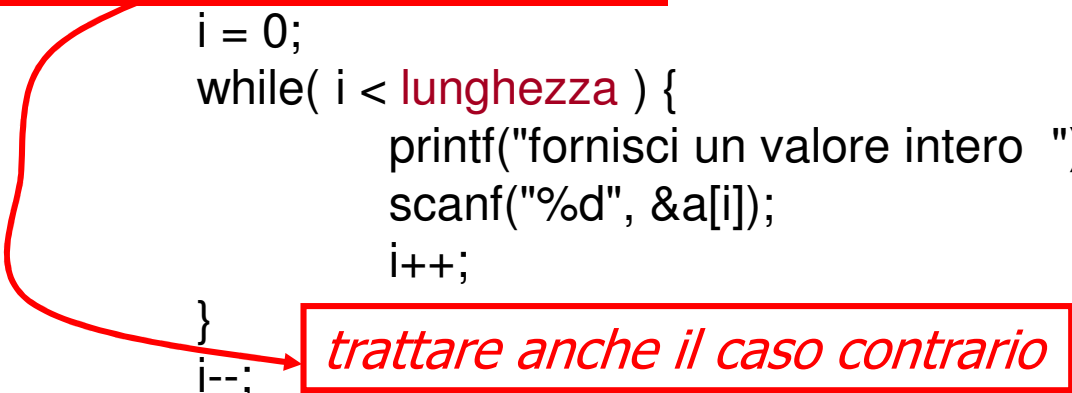
Maggiore
astrazione
del codice



```
#define LUNG_SEQ 100
```

```
// Programma InvertiSequenza di  
// lunghezza <= a un valore dato
```

```
int main() {  
    int lunghezza, i, a[LUNG_SEQ];  
    printf("lunghezza sequenza : ");  
    scanf("%d", &lunghezza);  
    if( lunghezza <= LUNG_SEQ ) {  
        i = 0;  
        while( i < lunghezza ) {  
            printf("fornisci un valore intero ");  
            scanf("%d", &a[i]);  
            i++;  
        }  
        i--;  
        while( i >= 0 ) {  
            printf("%d\n", a[i]);  
            i--;  
        }  
    }  
    return 0; }
```



trattare anche il caso contrario

Soluzione “**a sentinella**” : legge una sequenza di naturali,
terminata da -1, e la stampa **in sequenza invertita**;

Si ipotizza che la sequenza abbia lunghezza ≤ 100

```
int a[LUNG_SEQ], i=0, temp;
scanf("%d", &temp);
while( temp != -1 ) {
    a[i] = temp;
    i++;
    scanf("%d", &temp);
}
while( i > 0 ) {
    i--;
    printf("%d\n", a[i]);
}
```

*/*oppure a[i++] =temp;*/*

*/*o printf ("%d\n", a[--i]);*/*

N.B. si è trascurato il dialogo di input output!

Con una semplice modifica riusciamo a non generare errori nel caso in cui l'utente immetta più di LUN_SEQ valori [la soluzione precedente, infatti, non evita di superare il limite fisico del vettore]

```
int a[LUN_SEQ], i=0, temp;
printf("Inserire una sequenza di interi terminata da -1\n");
scanf("%d", &temp);
while( temp != -1 && i < LUN_SEQ ) {
    a[i] = temp;
    i++;
    scanf("%d", &temp);
}
if( temp != -1 && i == LUN_SEQ )
    printf("Raggiunto il limite di %d valori\n\n", LUN_SEQ);
while( i > 0 ) {
    i--;
    printf("%d\n", a[i]);
}
```

```

/* Output Strutturato: Stampa di un istogramma */
#include <stdio.h>
#define SIZE 10

int main () {
    int n[SIZE] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
    int i, j;

    printf("%s%13s%17s\n\n", "Element", "Value", "Histogram");

    for( i = 0 ; i < SIZE ; i++ ) {
        printf("%7d%13d", i, n[i]);
        for( j = 1 ; j <= n[i] ; j++ ) /* una riga di '*' */
            printf("*");
        printf("\n");
    }

    return 0;
}

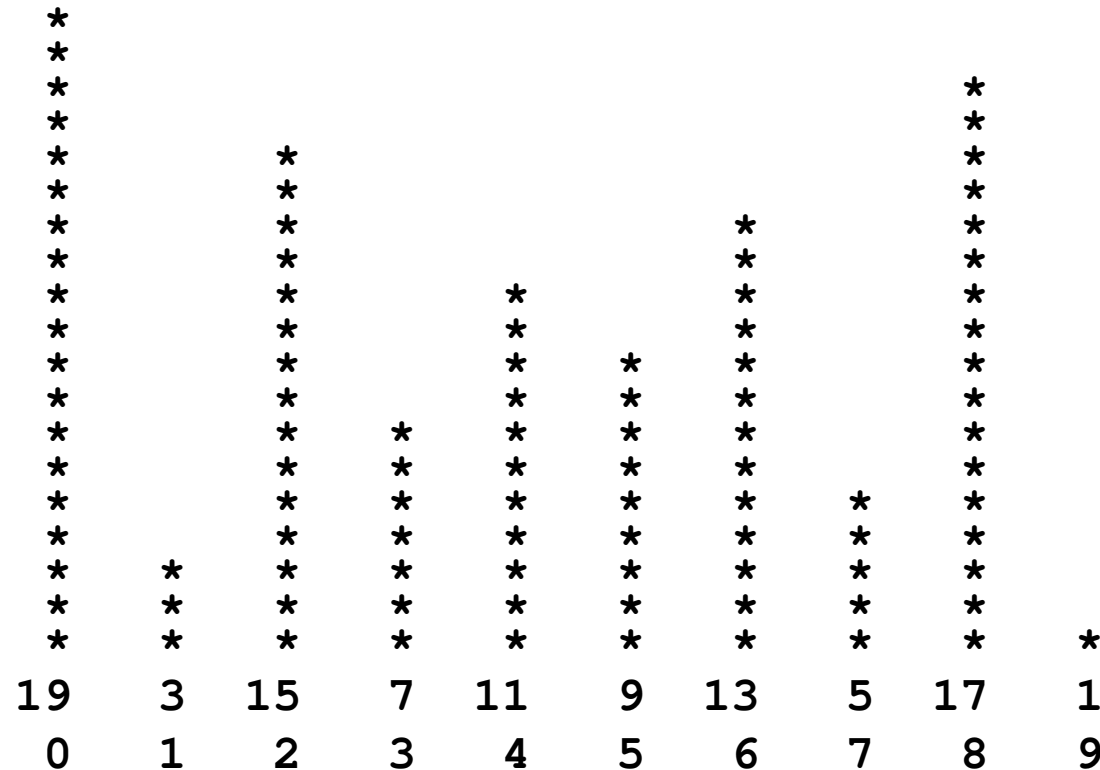
```

Output del programma

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

Esercizio (semplice, ma apparentemente difficile)

- Modificare il programma per visualizzare istogrammi verticali



```

/* SOLUZIONE ESERCIZIO ISTOGRAMMI VERTICALI */
#define SIZE 10
int main () {
    int n[SIZE] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
    int i, j, max = n[0];

    for( i = 1 ; i < SIZE ; i++ ) /* Calcola max di n[] */
        if( max < n[i] ) max = n[i];

    for( j = max ; j > 0 ; j-- ) {
        for( i = 0 ; i < SIZE ; i++ )
            if( n[i] >= j ) printf("  *  ");
            else           printf("    ");
        printf("\n");
    }
    for( i = 0 ; i < SIZE ; i++ ) printf("%3d  ", n[i]);
    printf("\n");
    for( i = 0 ; i < SIZE ; i++ ) printf("%3d  ", i);
    printf("\n");
    return 0;
}

```

Ricerca di un elemento in un array (con uso del costrutto break)

/* Frammento di programma che verifica la presenza di un valore (dato) in un array. Se è presente, stampa l'indice della prima occorrenza; se no stampa -1 */

```
int n, dato, risultato = -1, array[SIZE];  
... /* acquisizione valori array */  
printf("Valore da cercare? : "); scanf("%d",&dato);  
for ( n=0; n <= SIZE-1; n++ )  
    if ( array[n] == dato ) {  
        risultato = n;  
        break;  
    }  
printf("Il valore cercato si trova in posizione %d\n", risultato);
```

**Ricerca
sequenziale**

...

Biblioteca...

*Se ne scriva una versione (più elegante...) che non usi il break,
ma una condizione di uscita dal ciclo "più raffinata"*

Ricerca binaria su array (con break)

```
int dato, risultato=-1, array[SIZE], low=0, high=SIZE-1, mid;  
... /* acquisizione valori array */  
printf("Valore da cercare? : "); scanf("%d",&dato);  
while( low <= high ) {  
    mid = (low+high)/2;  
    if( dato == array[mid] ) {  
        risultato = mid;  
        break;  
    }  
    else if( dato < array[mid] )  
        high = mid - 1;  
    else  
        low = mid + 1;  
}
```

Termina?
Perché?
È corretto?

Array a più dimensioni

- Gli array a 1D realizzano i vettori, quelli a 2D realizzano le matrici, ...e così via

- Dichiarazione:

int A[20][30];

A è una matrice di 20×30 elementi interi (600 variabili distinte)

float F[20][20][30];

F è una matrice 3D di $20 \times 20 \times 30$ variabili di tipo float (12.000!)

- Oppure a quattro dimensioni, ecc...

Excursus Matematico – Le Matrici

- In matematica, una **matrice** è uno schieramento rettangolare di oggetti (di solito numeri), organizzati in **righe** (orizzontali) e **colonne** (verticali)
- In generale, una matrice ha m righe e n colonne, con m e n interi positivi fissati. Spesso una matrice è descritta indicando con a_{ij} il generico **elemento** posizionato alla riga i -esima e alla colonna j -esima
- I vettori si possono considerare matrici con una sola riga o una sola colonna (detti più precisamente **vettore riga** e **vettore colonna**)

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & -2 & 0 \\ 4.5 & 0 & 2 \\ 6 & 1 & 5 \end{bmatrix} \quad \begin{bmatrix} 7 \\ 0 \\ \pi \end{bmatrix} \quad \left[3 \quad \frac{7}{2} \quad -9 \right]$$

Excursus Matematico – Le Matrici

- Se la matrice si chiama A , l'elemento $a_{i,j}$ può essere indicato anche come $A[i,j]$. La notazione $A = (a_{i,j})$ indica che A è una matrice e che i suoi elementi sono denotati con $a_{i,j}$
- Gli elementi con i due indici di riga e di colonna uguali, cioè gli elementi della forma a_{ii} costituiscono la **diagonale principale** della matrice.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Excursus Matematico – Le Matrici

Somma

- Due matrici A e B si possono sommare se hanno le stesse dimensioni
- La loro somma $A+B$ è definita come la matrice i cui elementi sono ottenuti sommando i corrispettivi elementi di A e B . Formalmente:

$$(A+B)_{i,j} := A_{i,j} + B_{i,j}$$

Per esempio

$$\begin{bmatrix} 1 & 3 & 2 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 & 2+5 \\ 1+7 & 0+5 & 0+0 \\ 1+2 & 2+1 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 7 \\ 8 & 5 & 0 \\ 3 & 3 & 3 \end{bmatrix}.$$

Moltiplicazione per uno scalare

- È un'operazione che, data una matrice A ed un numero c (detto *scalare*), costruisce una nuova matrice cA , il cui elemento è ottenuto moltiplicando l'elemento corrispondente di A per c

$$(cA)_{i,j} := c(A_{i,j}) \quad \text{Es.:} \quad 2 \begin{bmatrix} 1 & 8 & -3 \\ 4 & -2 & 5 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 8 & 2 \times -3 \\ 2 \times 4 & 2 \times -2 & 2 \times 5 \end{bmatrix} = \begin{bmatrix} 2 & 16 & -6 \\ 8 & -4 & 10 \end{bmatrix}.$$

Excursus Matematico – Le Matrici

Prodotto riga per colonna

- La moltiplicazione tra due matrici A e B è un'operazione più complicata delle precedenti. A differenza della somma, non è definita sommando semplicemente gli elementi aventi lo stesso posto
 - La definizione di moltiplicazione che segue è motivata dal fatto che una matrice modella una applicazione lineare, e in questo modo il prodotto di matrici corrisponde alla composizione di applicazioni lineari
- La moltiplicazione è definita soltanto se il numero di righe di B coincide con il numero n di colonne di A . Il risultato è una matrice con lo stesso numero di righe di A e lo stesso numero di colonne di B
- L'elemento $(AB)_{i,j}$ è dato dalla somma dei prodotti degli elementi dell' i -esima riga di A e della j -esima colonna di B :

$$(AB)_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \dots + a_{i,n}b_{n,j}$$

Excursus Matematico – Le Matrici

Per esempio:

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & 1 & -3 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 2 & 5 & 1 \\ 0 & -2 & 1 \end{bmatrix} =$$

Moltiplicando una matrice 2×3 per una 3×3 si ottiene una matrice 2×3 .

1° riga:

$$C_{11} = [(1 \times 1) + (1 \times 2) + (2 \times 0)] = 3$$

$$C_{12} = [(1 \times 1) + (1 \times 5) + (2 \times -2)] = 2$$

$$C_{13} = [(1 \times 1) + (1 \times 1) + (2 \times 1)] = 4$$

2° riga:

$$C_{21} = [(0 \times 1) + (1 \times 2) + (-3 \times 0)] = 2$$

$$C_{22} = [(0 \times 1) + (1 \times 5) + (-3 \times -2)] = 11$$

$$C_{23} = [(0 \times 1) + (1 \times 1) + (-3 \times 1)] = -2$$

Risultato 2×3 :

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \end{bmatrix} = \begin{bmatrix} 3 & 2 & 4 \\ 2 & 11 & -2 \end{bmatrix}$$

Excursus Matematico – Le Matrici

Matrici quadrate

- Una matrice si dice quadrata se ha lo stesso numero di righe e di colonne.
- Una matrice quadrata ha una diagonale principale, quella formata da tutti gli elementi $a_{i,i}$ con indici uguali. La somma di questi elementi è chiamata traccia. L'**operazione di trasposizione** trasforma una matrice quadrata A nella matrice A^t ottenuta scambiando ogni $a_{i,j}$ con $a_{j,i}$, in altre parole ribaltando la matrice intorno alla sua diagonale principale.
- Una matrice in cui $a_{i,j} = a_{j,i}$ è una matrice **simmetrica**. In altre parole, A è simmetrica se $A = A^t$. Se tutti gli elementi che non sono nella diagonale principale sono nulli, la matrice è detta **diagonale**.

```

/* Frammento di programma per verificare se una matrice è simmetrica */
int MatQuadra[N] [N], j, i=0;
int sim = 1;          /* sim inizialmente vale 1 */
/* ... inserimento dati nella matrice ... codice omesso ... */
while ( i < N ) {
    j = 0;
    while ( j < N ) {
        if ( MatQuadra[i] [j] != MatQuadra[j] [i] )
            sim = 0; /*Alla fine sim varrà 0 se e solo se non è simmetrica*/
        ++j;
    }
    ++i;
}

```

è corretto?
 si può migliorare?

È corretto, ma...

- Confronta `MatQuadra[i][j]` con `MatQuadra[j][i]` e, poi, anche `MatQuadra[j][i]` con `MatQuadra[i][j]`
- Confronta `MatQuadra[i][i]` con se stesso
- Se trova una dissimmetria, continua con la verifica, ma è inutile
- Esercizio: se ne scriva una versione “ottimizzata” che risolva i tre problemi

Somma di matrici

$$c[i][j] = a[i][j] + b[i][j]$$

	0	1	2	3
0	3	7	10	0
1	1	3	11	2
2	5	8	9	24

+

	0	1	2	3
0	2	1	0	5
1	7	9	6	2
2	5	1	2	4

=

	0	1	2	3
0	5	8	10	5
1	8	12	17	4
2	10	9	11	28

```
#define N 3
#define M 4
int main() {
    int a[N][M], b[N][M], c[N][M], i, j;
    for( i=0; i<N; i++ ) // leggo mat. 1
        for( j=0; j<M; j++ )
            scanf("%d", &a[i][j]);
    for( i=0; i<N; i++ ) // leggo mat. 2
        for( j=0; j<M; j++ )
            scanf("%d", &b[i][j]);
    //calcolo e stampo la somma
    for( i=0; i<N; i++ ){ //calcolo somma
        for (j=0; j<M; j++){
            c[i][j]=a[i][j]+b[i][j];
            printf("%d ", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Prodotto di matrici

	0	1	2	3
0	3	7	10	0
1	1	3	11	2
2	5	8	9	24

*

	0	1	2	3	4
0	2	1	0	4	3
1	7	9	6	1	2
2	5	1	2	5	0
3	11	0	3	8	7

0
1
2
3

$$c[1][2] = \sum (a[1][k] * b[k][2]), \text{ per } k = 0, \dots, 3$$

	0	1	2	3	4
0					
1			0+18+ 22+6		
2					

```
#define N 3
#define M 4
#define L 5
.....
for( i=0; i<N; i++ ){
    for( j=0; j<L; j++ ){
        c[i][j] = 0;
        for( k=0; k<M; k++ )
            c[i][j] = c[i][j] + (a[i][k] * b[k][j]);
        printf("%d", c[i][j]);
    }
    printf("\n");
}
```

Le stringhe

- Array di caratteri: spesso chiamati ***stringhe***
 - Quando rappresentano “caratteri da leggersi in fila”
- Dichiarazione+inizializzazione di una stringa:
`char stringa[] = "word";`
- Il carattere nullo '`\0`' **termina le stringhe**
- Perciò l'array `stringa` ha **5** elementi (non 4):



- Dichiarazione equivalente:
`char stringa[] = {'w', 'o', 'r', 'd', '\0'};`

Stringhe e caratteri

- Qual è la differenza tra 'x' e "x"?
 - 'x' è una costante di tipo char, rappresentata in memoria da un intero
 - "x" è una stringa costante, rappresentata in memoria da un array contenente i caratteri: 'x' e '\0'

ATTENZIONE

Le stringhe **non** sono propriamente un **tipo** di dato
(non sono un tipo base!)

Non hanno operatori nativi, ma una serie di funzioni nella libreria standard che permettono di manipolarle

Operazioni su stringhe

```
char str1[32]; /* str1 ha spazio per 32 char. */
char str2[64]; /* str2 ha spazio per 64 char. */

/* inizializza str1 con la stringa "alfa" */
strcpy(str1, "alfa"); /* str1 contiene "alfa" */

/* copia str1 in str2 */
strcpy(str2, str1); /* str2 contiene "alfa" */

/* lunghezza di str1 */
x = strlen(str1); /* x assume valore 4 */

/* scrivi str1 su standard output */
printf("%s", str1); /* scrive str1 su stdout */

/* leggi str1 da standard input */
scanf("%s", str1); /* str1 "riceve" da stdin */
```

Operazioni su stringhe

```
char str1[32];
```

```
char str2[64];
```

```
scanf("%s", str1);
```

```
> ciao ↵    /* ora str1 contiene "ciao" */
```

```
strcpy(str2, str1);    /* str2 riceve "ciao"*/
```

```
val = strlen(str2);    /* val  = 4          */
```

```
printf("%s\n", str2);
```

```
> ciao      /* stampa "ciao" */
```

Attenzione: **strlen**("") vale 0 !

Particolarità delle stringhe

- Inizializzazione e accesso ai singoli caratteri:

```
char stringa[] = "word";
```

stringa[3] è un'espressione di valore **'d'**

- Il nome dell'array rappresenta l'indirizzo del suo primo elemento, perciò quando ci si vuole riferire all'intero array nella **scanf** non si mette il simbolo **&**!

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

- Questa **scanf** legge in input i caratteri fino a quando trova il carattere “blank” (lo spazio), o l'invio
- Se il buffer contiene una stringa “troppo lunga”, essa è memorizzata oltre la fine dell'array !!! Ed è un errore grave !!!


```
/* Stringhe e array di caratteri */
```

```
int main () {  
    char str1[20], str2[] = "string literal";  
    int i;  
    printf("\n Enter a string: ");  
    scanf("%s", str1);  
    printf("str1: %s\n str2: %s\n", str1, str2);  
    printf("str1 with spaces is: \n");  
    i = 0;  
    while( str1[i] != '\0' ) {  
        printf ("%c ", str1[i]);  
        i++;  
    }  
    printf ("\n");  
    return 0;  
}
```

**scanf() interrompe
la scansione quando
incontra uno spazio**

```
> Enter a string: Hello guys  
> str1: Hello  
> str2: string literal  
> str1 with spaces is:  
> H e l l o
```

strcpy(s1, s2)

- E se non ci fosse la funzione `strcpy()` ?
 - Assegneremmo sempre un carattere alla volta!

```
char s1[N], s2[M];
```

```
/* assegnamento di s2, omissso */
```

```
int i = 0;
```

```
while( i <= strlen(s2) && i < N ) {
```

```
    s1[i] = s2[i];
```

```
    ++i;
```

```
}
```

N.B. funziona correttamente se s2 è una stringa ben formata (cioè terminata da '\0') e se s1 è sufficientemente grande da contenere i caratteri di s2 (N >= strlen(s2))

QUIZ

- Che cosa stampano le seguenti printf()?

```
int a = 2;
```

```
char amac[] = "amac";
```

```
amac[strlen(amac)] = amac[a];
```

```
printf("%s\n", amac);
```

```
printf("%d\n", strlen(amac));
```

Morale: mai dimenticare che c'è anche il carattere '\0'

Confrontare due stringhe

- Una funzione apposita: **strcmp(s1, s2)**
 - restituisce un intero
 - confronta le due stringhe fino al '\0'

```
char s1[32], s2[64];  
int diverse;
```

```
/*... acquisizione di valori per le stringhe ... (codice omesso)*/
```

```
diverse = strcmp(s1, s2);
```

```
if( diverse == 0 )      printf("UGUALI\n");  
else if( diverse < 0 ) printf("%s PRECEDE %s\n", s1, s2);  
else                   printf("%s SEGUE %s\n", s1, s2);
```

strcmp(s1, s2)

- E se non ci fosse?
 - controlliamo un carattere alla volta
 - interrompiamo il controllo appena sono diverse

```
char s1[N], s2[M];  
int i = 0, uguali = 1;  
while( s1[i] == s2[i] && i < N && i < M )  
    ... i++ ... uguali = 0; ...
```

La riconsidereremo più avanti,
studiando funzioni e puntatori

Aggregazione di variabili

- "Gruppi" di variabili **omogenee**
 - ARRAY (vettori)
- Gruppi di variabili **eterogenee**
 - STRUCT (record)

Dichiarazione dei dati: dati complessi o strutturati

- **Record (o struct):** memorizzano aggregazioni di dati (ciascun dato è chiamato “campo”) di diversa natura:

```
struct {  
    char via[20];           /* 1° campo:  stringa  */  
    int  numero;           /* 2° campo:  intero   */  
    int  CAP;              /* 3° campo:  intero   */  
    char citta[20];        /* 4° campo:  stringa  */  
} indirizzo;
```

nome del record

- **indirizzo:** un record con 4 campi di vario tipo

Uso dei record

- Il record (o struct) è una sorta di “**contenitore**” di campi di tipo eterogeneo
- Il record **raggruppa** dati più semplici
 - Ne rende più ordinata la gestione, evitando confusioni
- I campi del record non sono visibili direttamente
 - Il loro nome deve essere preceduto da quello del record a cui appartengono (interponendo ‘.’ come separatore)
 - Non sono identificatori
 - Sono però identificatori “locali” all’interno di una variabile di tipo strutturato, da usarsi come **suffissi**

Operazioni su record

- **Assegnamento** ai campi del record:

```
strcpy(indirizzo.via, "Ponzio");  
indirizzo.numero = 34;  
indirizzo.CAP = 20133;  
strcpy(indirizzo.citta, "Milano");
```

- **Accesso** (leggere, scrivere...):

```
printf("%d\n", indirizzo.numero);  
> 34  
printf("%d\n", strlen(indirizzo.citta));  
> 6  
printf("%s\n", indirizzo.citta);  
> Milano  
scanf("%s", indirizzo.via);  
> Ponzio↵  
scanf("%d", &indirizzo.CAP);  
> 20133↵
```

Ancora operazioni su record

- Dati due record **identici** (cioè dichiarati insieme) è lecito **assegnare** globalmente il primo al secondo

```
struct { ... /* campi */ } rec1, rec2;
```

è lecito scrivere:

```
rec2 = rec1;
```

e *tutti i campi* di **rec1** sono **ordinatamente copiati** nei campi corrispondenti di **rec2**.

- Se i due record sono **diversi** (anche solo per l'ordine dei campi) l'assegnamento è *privo di senso* !
- Memento: l'assegnamento “diretto” tra array è *vietato*
 - deve avvenire elemento per elemento