

ARRAY DI PUNTATORI

- Non ci sono vincoli sul tipo degli elementi di un vettore
- Possiamo dunque avere anche **vettori di puntatori**

Ad esempio:

```
char * stringhe[4];
```

definisce un vettore di 4 puntatori a carattere (allocata memoria per 4 puntatori)

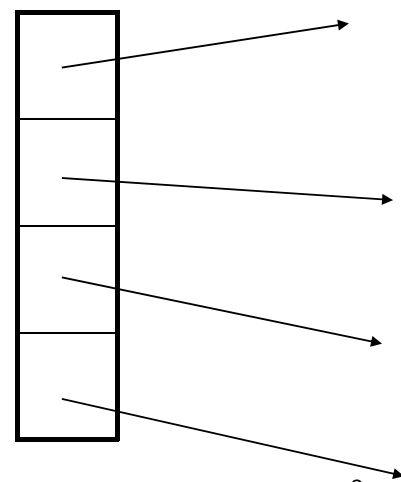
1

ARRAY DI PUNTATORI

`stringhe` sarà dunque una struttura dati rappresentabile nel modo seguente

I vari **puntatori** sono **memorizzati in celle contigue**. Possono invece **non essere contigue le celle che loro puntano**

stringhe



2

INIZIALIZZAZIONE

Come usuale, anche gli array di puntatori a stringhe possono essere *inizializzati*

```
char * mesi[] = {"Gennaio", "Febbraio",  
"Marzo", "Aprile", "Maggio", "Giugno",  
"Luglio", "Agosto", "Settembre", "Ottobre",  
"Novembre", "Dicembre"};
```

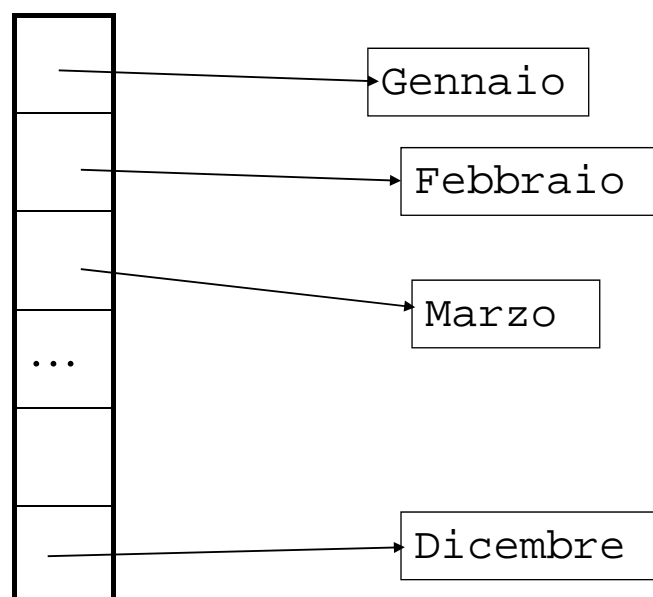
I caratteri dell'i-esima stringa vengono posti in una locazione qualsiasi e *in mesi[i] viene memorizzato un puntatore a tale locazione*

Come sempre, poiché l'ampiezza del vettore non è stata specificata, il compilatore conta i valori di inizializzazione e dimensiona il vettore di conseguenza

3

INIZIALIZZAZIONE

`mesi`



4

ARRAY MULTIDIMENSIONALI

È possibile definire variabili di tipo array con più di una dimensione

`<tipo> <identificatore>[dim1][dim2]...[dimn]`

Array con due dimensioni vengono solitamente detti **matrici**

```
float M[20][30];
```

Come sempre, **allocazione statica**:
allocazione di 20x30 celle
atte a mantenere float

	0	1	...	29
0				
1				
...				
19				

MATRICI

Per accedere all'elemento che si trova nella **riga** *i* e nella **colonna** *j* si utilizza la notazione

`M[i][j]`

Anche possibilità di vettori con più di 2 indici:

```
int C[20][30][40];
```

```
int Q[20][30][40][40];
```

MEMORIZZAZIONE

Le matrici multidimensionali sono **memorizzate per righe in celle contigue**. Nel caso di M:

M[0][0]	M[0][1]	...	M[0][29]	M[1][0]	M[1][1]	...	M[1][29]	M[19][0]	M[19][1]	...	M[19][29]
---------	---------	-----	----------	---------	---------	-----	----------	------	----------	----------	-----	-----------

E analogamente nel caso di più di 2 dimensioni: viene fatto variare prima l'indice più a destra, poi il penultimo a destra, e così via fino a quello più a sinistra

Per calcolare **l'offset** della cella di memoria dell'elemento (i,j) in una matrice bidimensionale (rispetto all'indirizzo di memorizzazione della prima cella – *indirizzo dell'array*):

$$\text{LungRiga} * i + j$$

Nel caso di M: $\mathbf{M}[i][j]$ elemento che si trova nella cella $30*i+j$ dall'inizio della matrice

Elemento: $\mathbf{*(\&M[0][0] + 30*i+j)}$

7

MEMORIZZAZIONE

In generale, se

`<tipo> mat[dim1][dim2]...[dimn]`

`mat[i1][i2]...[in-1][in]`

è l'elemento che si trova nella cella

$i_1 * \text{dim}_2 * \dots * \text{dim}_n + \dots + i_{n-1} * \text{dim}_n + i_n$ a partire dall'inizio del vettore

8

MATRICI

Si possono anche dichiarare dei tipi vettore multidimensionale

```
typedef <tipo> <ident>[dim1][dim2]...[dimn]
```

```
typedef float MatReali [20] [30];  
MatReali Mat;
```

è equivalente a

```
typedef float VetReali[30];  
typedef VetReali MatReali[20];  
MatReali Mat;
```

9

INIZIALIZZAZIONE

Come al solito, i vettori multidimensionali possono essere inizializzati con una lista di valori di inizializzazione racchiusi tra parentesi graffe

```
int matrix[4][4]={{1,0,0,0},{0,1,0,0},  
                 {0,0,1,0}, {0,0,0,1}}
```

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

10

Puntatori e Vettori MULTIDIMENSIONALI

Anche un vettore multidimensionale è **implementato in C come un puntatore all'area di memoria** da cui partono le **celle contigue** contenenti il vettore

```
int a[10][4];
```

a vettore di 10 elementi, ciascuno dei quali è un vettore a 4 interi
tipo = "puntatore a vettore di 4 interi" non "puntatore a intero"

```
int ** b; int * c;
```

b=a;
c=a; compila con warning

11

PUNTATORI E VETTORI MULTIDIMENSIONALI

Date le due definizioni

```
int a[10][4]; int *d[10];
```

la prima alloca 40 celle di ampiezza pari alla dim di un int
mentre la seconda alloca 10 celle per contenere 10 puntatori a int

Uno dei vantaggi offerti dall'uso di vettori di puntatori consiste nel fatto che si possono realizzare **righe di lunghezza variabile**

12

PUNTATORI E VETTORI MULTIDIMENSIONALI

```
char * mesi[] = {"Gennaio", "Febbraio",  
"Marzo", "Aprile", "Maggio", "Giugno",  
"Luglio", "Agosto", "Settembre",  
"Ottobre", "Novembre", "Dicembre"};
```

vengono create *righe di lunghezza variabile*

13

Esempio: PRODOTTO MATRICI QUADRATE

Programma per il **prodotto (righe x colonne) di matrici quadrate NxN** a valori interi:

$$C[i,j] = \sum_{(k=1..N)} A[i][k]*B[k][j]$$

```
#include <stdio.h>  
#define N 2  
typedef int Matrici[N][N];  
  
int main(){  
int Somma,i,j,k;  
Matrici A,B,C;  
/* inizializzazione di A e B */  
for (i=0; i<N; i++)  
for (j=0; j<N; j++)  
scanf("%d",&A[i][j]);  
for (i=0; i<N; i++)  
for (j=0; j<N; j++)  
scanf("%d",&B[i][j]);
```

14

Esempio: PRODOTTO MATRICI QUADRATE

```
/* prodotto matriciale */
for (i=0; i<N; i++)
    for (j=0; j<N; j++){
        Somma=0;
        for (k=0; k<N; k++)
            Somma=Somma+A[i][k]*B[k][j];
        C[i][j]=Somma;}

/* stampa */
for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
        printf("%d\t",C[i][j]);
    printf("\n"); }
}
```

15

PASSAGGIO DI PARAMETRI

Nel caso di passaggio come parametro di un vettore bidimensionale a una funzione, nel prototipo della funzione **va dichiarato necessariamente il numero delle colonne** (ovvero la dimensione della riga)

Ciò è essenziale perché il compilatore sappia come accedere al vettore in memoria

16

PASSAGGIO DI PARAMETRI

Esempio: se si vuole passare alla funzione `f()` la matrice `par` occorre scrivere all'atto della definizione della funzione:

```
f(float par[20][30],...)           oppure  
f(float par[][30],...)
```

perché il numero di righe è irrilevante ai fini dell'aritmetica dei puntatori su `par`

In generale, come già detto, ***soltanto la prima dimensione di un vettore multidimensionale può non essere specificata***

17

Esempio: PRODOTTO MATRICI QUADRATE

```
#include <stdio.h>  
#define N 2  
typedef int Matrici[N][N];  
  
void prodottoMatrici(int X[][N], int Y[][N],  
                    int Z[][N]) {  
    int Somma,i,j,k;  
    for (i=0; i<N; i++)  
        for (j=0; j<N; j++){  
            Somma=0;  
            for (k=0; k<N; k++)  
                Somma=Somma+X[i][k]*Y[k][j];  
            Z[i][j]=Somma;}  
}
```

18

Esempio: PRODOTTO MATRICI QUADRATE

```
int main(void){
int Somma,i,j,k;
Matrici A,B,C;

for (i=0; i<N; i++) /* inizializ. di A e B */
    for (j=0; j<N; j++)
        scanf("%d",&A[i][j]);
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        scanf("%d",&B[i][j]);

prodottoMatrici(A,B,C); //in C verrà caricato il
    risultato del prodotto

for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
        printf("%d\t",C[i][j]);
    printf("\n"); }}
```

19

Matrici

- Un po' di esercizi sulle matrici
- Semplici
 - Lettura e scrittura
 - Calcolo della trasposta

Matrici.h – Definizione dei tipi

```
#define MAXROWS 10
#define MAXCOLS 10
#define ELEMENT int
#define ELEMENTFORMAT "%d"
```

Tipo degli elementi della matrice

Formato di stampa degli
elementi della matrice

```
#define MATRICE
```

```
typedef ELEMENT
    Matrice[MAXROWS][MAXCOLS];
```

21

Lettura di una matrice

- Leggere da console una matrice di **r** righe e **c** colonne...
- Parametri:
 - La matrice da leggere (obbligatorio riferimento – va bene così)
 - La dimensione (righe, colonne)
- Pseudocodice:
 - Leggere la matrice riga per riga – elemento per elemento → stampare un opportuno messaggio prima di richiedere i dati
 - Un ciclo per le righe
 - Un ciclo innestato nel precedente per le colonne
 - Per ogni riga della matrice
 - Indicare quale riga si sta leggendo
 - Leggere la riga con tante **scanf ()** quante sono le colonne
 - Controllare che i valori siano stati tutti convertiti correttamente, altrimenti uscire dalla funzione con un codice d'errore opportuno

22

Lettura di una matrice

```
CODICEUSCITA readMatrice(Matrice m, int righe, int colonne)
{
    int riga, colonna;
    for (riga = 0; riga < righe; riga++)
    {
        printf("Inserire la riga %d: ", riga);
        for (colonna = 0; colonna < colonne; colonna++)
        {
            if (scanf(ELEMENTFORMAT, &m[riga][colonna]) != 1)
                return VALORINONVALIDI;
        }
        while (getchar() != (char)10);
    }
    return OK;
}
```

• CODICEUSCITA è il simbolo utilizzato anche nelle slide relative al calcolo delle radici

• VALORINONVALIDI è un simbolo aggiuntivo...

Stampa di una matrice

- Stampare a console una matrice di **r** righe e **c** colonne...
- Parametri:
 - La matrice da stampare
 - La dimensione (righe, colonne)
- Pseudocodice:
 - Predisporre la stringa di formato
 - Per ogni riga **r**
 - Per ogni colonna **c**
 - Stampare l'elemento di posizione **r**, **c** usando la stringa di formato

Stampa di una matrice

```
void printMatrice(Matrice m, int righe, int colonne)
{
    int riga, colonna;
    char formato[10] = "";
    strcat(formato, ELEMENTFORMAT);
    strcat(formato, "\t");
    for (riga = 0; riga < righe; riga++)
    {
        for (colonna = 0; colonna < colonne; colonna++)
            printf(formato, m[riga][colonna]);
        printf("\n");
    }
    printf("\n");
}
```

25

Trasposta di una matrice quadrata

- La definizione di trasposta è...
- Parametri:
 - Una matrice in ingresso
 - Una matrice in uscita
 - La dimensione della matrice
- Pseudo codice:
 - Per ogni riga r , colonna c
 - Porre l'elemento di posizione r, c nella matrice in ingresso nella posizione c, r nella matrice in uscita₂₆

Trasposta di una matrice quadrata

```
void trasposta(Matrice m, Matrice tm, int
dim)
{
    int riga, colonna;
    for (riga = 0; riga < dim; riga++)
        for (colonna = 0; colonna < dim;
colonna++)
            tm[colonna][riga] =
m[riga][colonna];
}
```