

funzioni e array

testo 5.3 e 7.3

array a 1 dimensione, a 2 a 3 sono
allocati nell'array nello stesso modo,
tanti elementi attaccati uno all'altro

facciamo un gioco:

abbiamo un array X di 100 elementi e
vogliamo "vederlo" come A[5][5]

oppure

B[4][5][5]

per esempio che elemento di X è
A[2][4] ?

oppure che elemento di X è
 $B[2][0][4]$?

ma possiamo fare anche il contrario:
 $X[50]$ che elemento di A sarebbe ?
E di B ?

per passare un array ad una
funzione devo saperne il tipo

array=[]



f(?? x)

che tipo specificare per il
parametro formale?

partiamo dagli array a 1 dimensione

```
float X[10];
```

ha tipo `float *`, `float[]`, `float[10]`

quindi queste sono funzioni in grado di ricevere un array di interi di dimensione qualsiasi:

`F(int *x)` `F(int x[])` e `F(int x[10])`

queste sono invocazioni corrette per F:

```
int C[20];
```

```
F(C); // ok
```

```
int B[10], a = 2, *p=&a;
```

```
F(B);   F(p); // ATTENZIONE
```

quindi possiamo passare a questa F
indifferentemente:

1) un array ad una dimensione di
qualsunque numero di elementi

2) e anche un puntatore a int

```
//{A[0..dim_A-1] definito}
int max(int*A, int dim_A)
{
  int max=A[0];
  for(int i=1; i< dim_A; i++)
    if(max < A[i])
      max=A[i];
  return max;
} {max è massimo di A[0..dim_A-1]}
```



```
int pippo[10]={.....}, pluto[20]={.....};
```

```
int max_pippo=max(pippo,10);
```

```
int max_pluto=max (pluto,20);
```

max accetta array interi con diverso
numero di elementi
10, 20, 30,....10000,.....

2 cose da
capire

1) i tipi `int[10]` e `int[20]` sono lo stesso
tipo → `int*` o `int[]`

se non fosse così:

`max10(int[10],...)` `max20(int[20],...)` e
così via

INACCETTABILE

il primo PASCAL ('70) era così !

2) il fatto che

`f(int *,...)` permetta a `f` di ricevere un array di interi

mostra che viene passato solo il puntatore al primo elemento dell'array e **NON una copia** dell'array

c'è un SOLO array : quello del chiamante

```
void F(int *A)
{A[0]=A[1];}

main()
{
int x[]={0,1,2,3,4};
F(x);
cout<<x[0]<<endl; // ?
}
```

c'è solo l'array x, in F A punta a x[0]

quindi quando le
funzioni ricevono
array, in generale
producono
side-effect

se vogliamo che la funzione non
cambi l'array che riceve:

`F(const int A[],...)`

se *F* cerca di modificare qualche
elemento di *A* il compilatore dà
errore

passare array a più dimensioni:

il tipo degli array a più dimensioni

`int K[5][10];` tipo = `int (*) [10]` o `int[][10]`

`char R[4][6][8];` tipo = `char (*) [6][8]` o
`char[][6][8]`

`double F[3][5][7][9];` tipo = `double (*) [5][7][9]`
o `double[][5][7][9]`

un parametro formale capace di ricevere l'array

```
int K[5][10];
```

è $F(\text{int } (*A)[10])$ o $F(\text{int } A[][10])$

riceve anche

```
int B[10][10] e C[20][10]
```

insomma la seconda dimensione è **fissa**,
solo la prima è **variabile**

```
char R[4][6][8]; tipo = char (*) [6][8]
```

la riceviamo con:

```
...F(char (*A)[6][8]) o F(char A[][6][8])
```

di nuovo solo la prima dimensione è libera
le altre sono fisse

fissato un tipo T , per array di una dimensione di tipo T

una stessa funzione può ricevere ogni array di tipo T indipendentemente dal numero degli elementi

è BENE

ma per array a più dimensioni NON è così:

la funzione che accetta $K[5][10]$, accetta anche $K[10][10]$, ma non $K[5][11]$.

perché?

nel corpo di `F(int A[][10])` si accede per esempio a `A[3][5]`

`F` riceve in `A` il puntatore ad `A[0][0]` e deve calcolare l'indirizzo di `A[3][5]`

$A + (3 \text{ righe di } 10 \text{ int}) + 5 \text{ int}$

insomma **[10]** nel tipo di `A` serve `A[][]`
non basterebbe

visto che $A[][10]$, $A[][11]$, $A[][12]$,...sono
tipi diversi rischiamo :

$f(\text{int } A[][10], \text{int righe})$

$f1(\text{int } A[][11], \text{int righe})$

$f2(\text{int } A[][12], \text{int righe})$

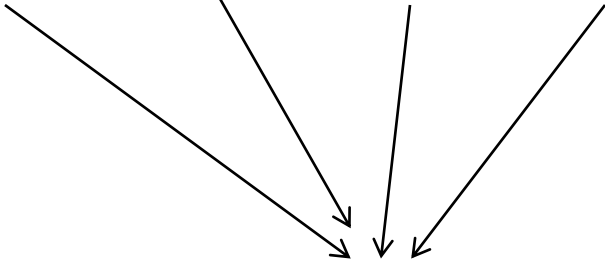
.....

meglio di NO

li trattiamo tutti come array ad una
dimensione

```
int A[....][colonne]
```

10 11 12 13



```
void f(int * p, int colonne)  
{
```

```
    *(p + 3*colonne + 5) = p[3][5]
```

```
}
```

e il numero delle righe?

```
void stampa(int*p,int righe,int colonne)
{
    for(int i=0; i<righe*colonne; i++)
        cout<<p[i]<<' ';
    cout<<endl;
}
```

ma se volessi andare a capo dopo ogni riga ?