array a più dimensioni

definizioni ed esempi

Array a 1 dimensione

```
char x[20];
int w[50]
```

x ha tipo char * const
e w int * const

è facile dimenticarsi del const

```
char* p=x ;  //  OK !
```

char x[50];

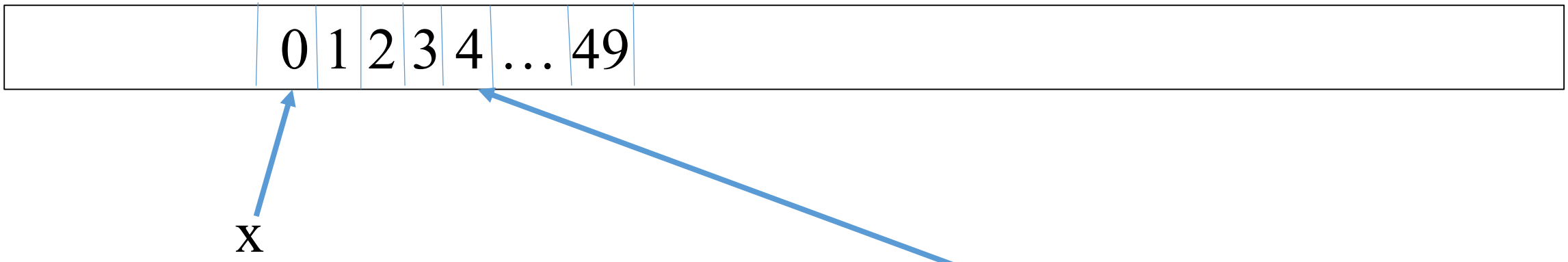
x[0] ≡ *(x+0) = *x aritmetica dei puntatori
x[1] ≡ *(x+1)
x[2] ≡ *(x+2)

e così via

e anche x[-5] e *(x-5) sono legali sebbene senza senso in questo caso

char x[50];

allocazione in memoria in 50 byte contigui della memoria



e su questo si fonda l’aritmetica dei puntatori x+4

possono servire array a 2 dimensioni e anche a 3, 4,… ecc dimensioni

- vogliamo rappresentare in modo semplice la configurazione di una scacchiera
- o abbiamo un elenco di conti correnti con delle informazioni per ciascun numero di conto: saldo, interesse, ecc
- modelli di auto con caratteristiche tecniche

- a 3 dimensioni: marche di auto e per ciascuna marca i modelli con le caratteristiche

sintassi in C e C++

int a[10][20]; ➔ array a 2 dimensioni / matrici

char r[5][10][20]; ➔ a 3 dimensioni / torte

double w[5][6][7][8] ➔ a 4 dimensioni / sequenza di torte

primo elemento= a[0][0], intermedio= a[2][5], ultimo=a[9][19]

r[0][0][0]……..r[4][9][19]

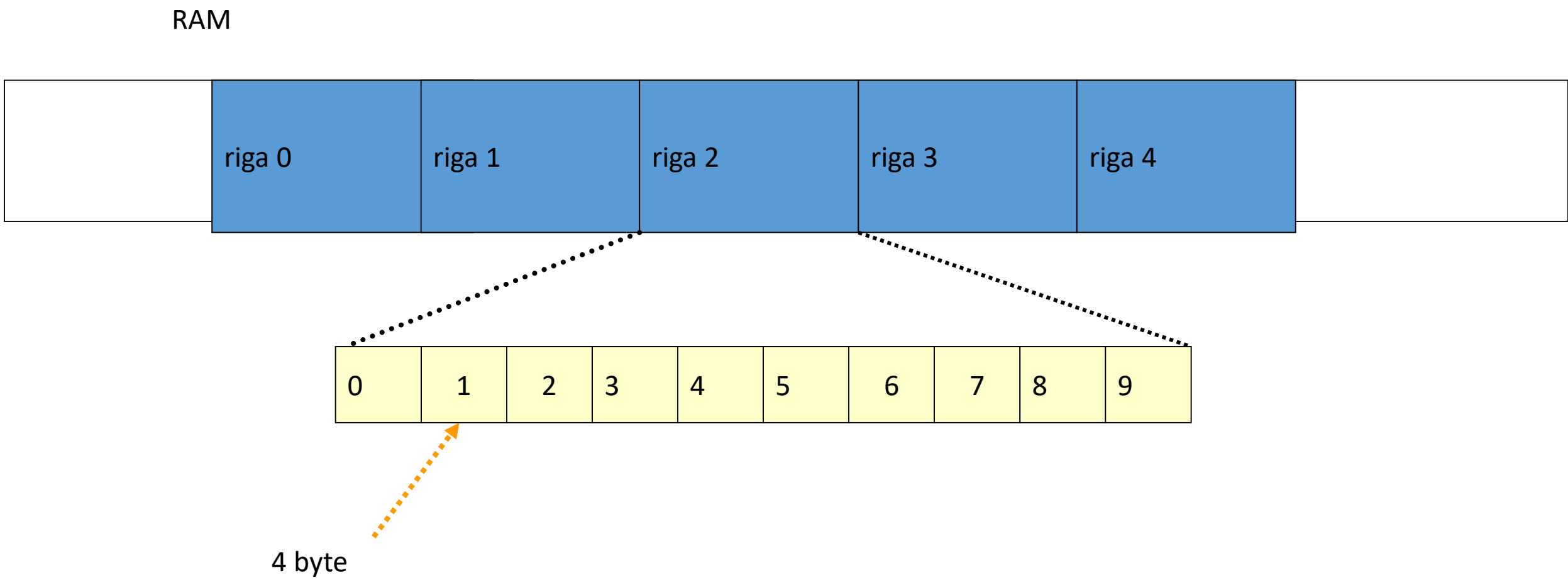
allocazione in memoria

array a 2 dimensioni (strati o matrici)= sequenza di array a 1 dimensione (righe), in memoria viene messa la prima riga, poi la seconda, la terza e così via, tutte attaccate senza spazi liberi

array a 3 dimensioni (torte)= sequenza di array a 2 dimensioni (strati), in memoria viene messo il primo strato, poi il secondo, il terzo e così via tutti attaccati

array a 4 dimensioni (seq di torte) = sequenza di torte, in memoria prima la prima torta, poi la seconda e così via.
e così per 5, 6,….dimensioni

int X[5][10]



int Y[3][4][10];

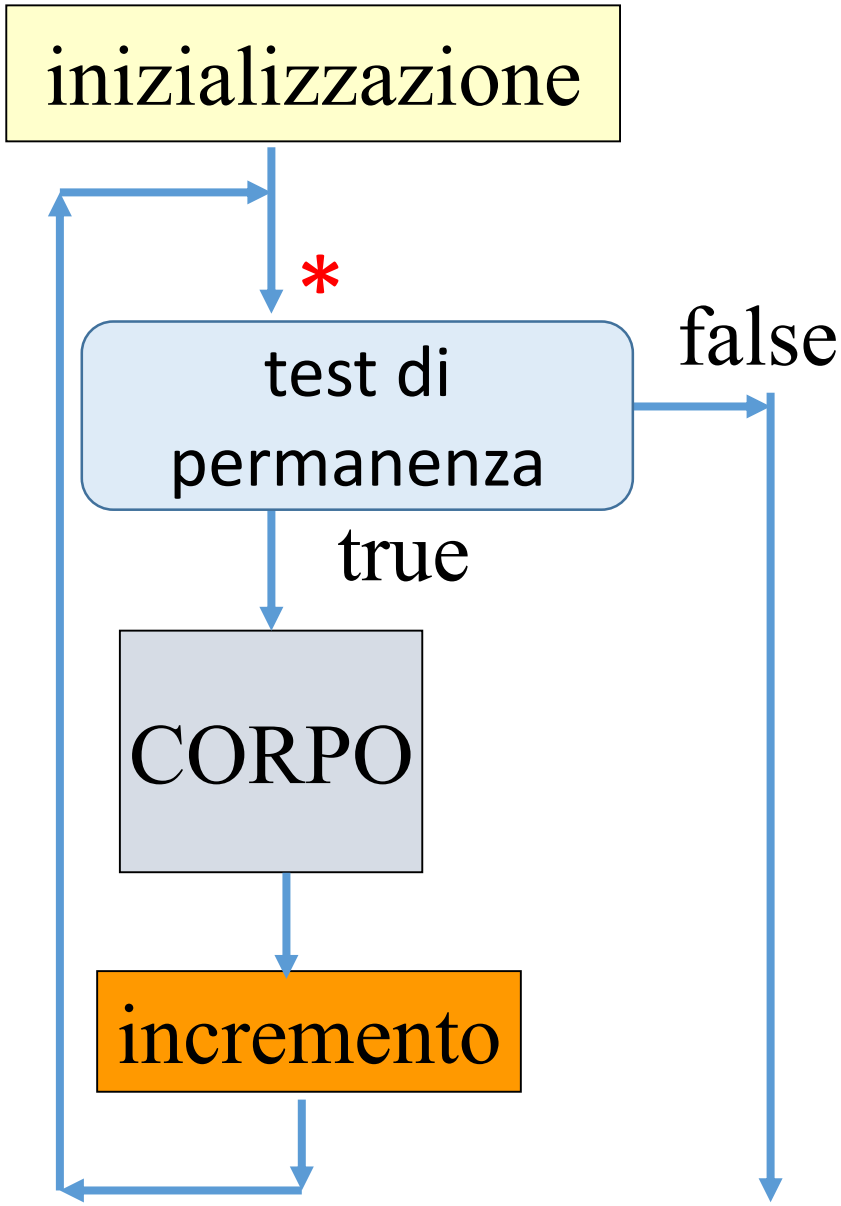
ogni strato è un array int [4][10] immagazzinato in memoria come visto prima

per scandire gli elementi degli array
è comodo il comando iterativo for

```
for ( iniz ; *test di perm ; increm)

{ CORPO}
```

è equivalente al while



```
//riempimento di un array a 1 dimensione
int A[20];
for(int i=0; i<20; i=i+1)
   cin >> A[i];

//riempimento di array a 2 dimensioni per righe
char X[5][10];
for(int i=0; i<5; i=i+1)
   for(int j=0; j<10; j=j+1)
      cin >> X[i][j];
```

e a 3 dimensioni? 3 for annidati e così via

for (iniz ; test di permanenza ; increm)
{CORPO}

è equivalente a

{ iniz;

  while (test-perm)
        {CORPO; increm}

}

anche per il for, correttezza in 3 parti come per il while

Tipi degli array

- int x[10]; x ha tipo int * e R-valore = &x[0]

- int y[5][10]; y è un array di 5 array di 10 interi

quindi y è un puntatore al primo dei 5 array di 10 int, e quindi:

1. y ha tipo int (*) [10] o int [][10]
2. il suo valore è & y[0][0] che è il primo elemento del primo array di 10 interi

y è l'oggetto puntato da y e quindi è un array di 10 interi che ha tipo int, quindi *y è un valore di tipo int * e punta a y[0][0]

e quindi y e *y hanno lo stesso valore &y[0][0], ma hanno tipi diversi !!!

cout << y <<‘ ‘<<*y;

stampa 2 volte &y[0][0] !!

consideriamo ora double D[4][5][6] il tipo di D è double (*) [5][6] , cioè un puntatore al primo strato 5x6 di D che chiamiamo una torta

se stampiamo D otteniamo &D[0][0][0]
*D è l'oggetto puntato da D, cioè il primo strato 5x6 che ha tipo double (*)[6], insomma *D punta alla prima riga del primo strato

**D è l'oggetto puntato da *D, cioè la diga di 6 elementi che ha tipo double*
finalmente ***D=D[0][0][0] ha tipo double

quindi,

cout << D << *D << **D <<endl;

stampa 3 volte l'indirizzo &D[0][0][0], infatti questo è il primo elemento del primo strato di D, ma anche della prima riga di D

D ha tipo double (*) [5][6]
*D ha tipo double (*) [6]
**D ha tipo double*

con 4 dimensioni, double Q[5][3][5][6] è una sequenza di 5 torte 3x5x6
Q punta alla prima torta e il suo tipo è double (*) [3][5][6]

*Q punta al primo strato della prima torta e quindi ha tipo double (*) [5][6]

**Q punta alla prima riga del primo strato della prima torta e quindi ha tipo, double (*) [6]

***Q punta al primo elemento della prima riga del primo strato della prima torta e ha tipo, double*

cout << Q << *Q << **Q << ***Q<<endl;
stampa 4 volte lo stesso indirizzo che è &Q[0][0][0][0]

il tipo dei puntatori ci dice la dimensione dell'oggetto puntato:
double Q[5][3][5][6]

Q ha tipo double (*) [3][5][6] e punta a una torta di 3*5*6*8 byte
*Q ha tipo double (*) [5][6] e punta ad uno strato di 5*6*8 byte
**Q ha tipo double (*) [6] e punta a una riga di 6*8 byte
***Q ha tipo double* e punta a un elemento di 8 byte

questa informazione sarà determinante per l'aritmetica dei puntatori

abbiamo visto come leggere valori in un array a 2 dimensioni:

```
int x[5][6];
for(int i=0; i<5;i++)//scorre le righe i =[0..4]
 for(int j=0; j<6; j++)//legge la riga i di 6 elementi
   cin >> x[i][j];
```

x viene riempito per righe

prima la riga 0, poi la 1, poi la 2, fino alla 4

ma visto che le righe sono in memoria proprio nello stesso ordine, prima la riga 0, poi la 1, poi la 2, ecc,
possiamo vedere x come un array ad una dimensione con 30 elementi
per cui

```
int x[5][6];
int* y= *x; // o anche &x[0][0]
for(int i=0; i<30; i++)
   cin >> y[i];
```

ha lo stesso effetto del precedente programma

quindi possiamo vedere un array a 2 dimensioni come se ne avesse 1 sola

lo stesso vale per 3, 4, … dimensioni

indipendentemente dal numero delle dimensioni, ogni array è una sequenza di elementi in memoria (tutti di uguale tipo e quindi uguale n. di byte)

esempio: dato double Q[5][3][5][6] lo possiamo vedere come double [5*3*5*6] e possiamo leggerci dentro 195 interi senza preoccuparci di quale parte dell'array viene riempita

```
int *x=***Q;
for(int i=0; i<195; i++)
   cin >> x[i];
```

ma quale parte di Q viene definita in questo modo?
ogni torta conta 90 posti, ne avanzano 15 per la terza torta che avrà 2 righe del primo strato piene e 3 valori nella terza riga. Il resto di Q resta indefinito. Sarebbe più complicato fare questa lettura direttamente in Q

quindi abbiamo 2 visioni degli array multidimensionali
double Q[5][3][5][6] o unidimensionali double *x=***Q

a seconda del caso possiamo usare una visione o l'altra

possiamo passare da una all'altra :
Q[2][2][3][1] ➔ 2torte + 2strati + 3righe + 1elemento =
y[2*(3*5*6)+2*(5*6)+3*6+1]=y[439]

y[197] ➔ Q[197/torte][(197%torte)/strati]
[((197%torte)%strati)/righe][((197%torte)%strati)%righe]
=Q[2][0][2][5]

passare array alle funzioni:

array a una dimensione: min(int*A, int dim) //bene!
array a 2 dimensioni: min(int (*A)[10], int righe) //ahi!
array a 3 dimensioni: min(int (*A)[8][10], int strati)//ahi ahi!!

e così via

più dimensioni hanno gli array e più vincolate diventano le funzioni

perché è così?

```
double f(double (*X)[8][10], int strati)
{
    y=X[2][3][4];   //come fa il compilatore a calcolare
                              //l'indirizzo di questo elemento di X?

}
```

X è l'indirizzo del primo elemento di X

X+2strati+3righe+4double = 2*(8*10)*8+3*10*8+4*8

insomma i limiti delle dimensioni 2, 3, ecc sono necessari, ma limitano la generalità delle funzioni

soluzione:

passiamo l'array come fosse unidimensionale e passiamo come parametri extra il numero di strati, righe e colonne, ecc.

```
double min(double*x, int strati, int righe, int colonne)
{
//al posto di y=X[2][3][4];
    y=x[2*righe*colonne+3*righe+4]; //il calcolo lo facciamo a mano
}
```

più fatica, ma più generalità

esercizio: dato l'array int x[5][6], vogliamo calcolare l'indice della colonna a somma massima

Vogliamo farlo con 2 diverse funzioni,

- la prima riceve x come array a 2 dimensioni,
- la seconda lo riceve come un array int y[30], dove y=*x

a volte mantenere le dimensioni conviene.
esercizio: trovare l'indice della riga che ha somma degli elementi massima (in caso di parità, vogliamo l'indice minimo)

```
int x [5][6];  ……//lettura di x implicita
bool prima=true; int maxv, maxindice;
for(int i=0; i < 5; i++)
{  int somma=0;
    for(int j=0; j<6; j++)
        somma=somma+x[i][j];
   if (prima || somma > maxv)
      {prima=false; maxv=somma; maxindice=i;}
}
```

ma sfruttando che l'elemento &x[i][j] =*x+(i*6)+j

```
int x[5][6]=;  ……//lettura di x implicita
bool prima=true;  int *y =*x;
int maxv, maxindice;
for(int i=0; i < 5; i++)
{  int somma=0;
    for(int j=0; j<6; j++)
        somma=somma+ *(y+(i*6)+j);
   if (prima || somma > maxv)
      {prima=false; maxv=somma; maxindice=i;}
}
```