passare array a funzioni

testo 7.3 pag. 89

ricordiamo i tipi degli array anche a più dimensioni

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 di qualunque numero di elementi

2) 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;  // ?

}
```

e ?

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

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

RICORDARE

X[0] è lo stesso di *X

e X[1] è lo stesso di *(X+1)

e così via

X[i] è lo stesso *(X+i)

---

sappiamo cosa stampa cout << X;

e

cout << (X+i); ??

passare array a più dimensioni:

ricordiamo il tipo

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

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

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

un parametro formale capace di ricevere l'array

int K[5][10];

è F(int (*A)[10]) o F(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 righe di 10 int) + 5 int

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

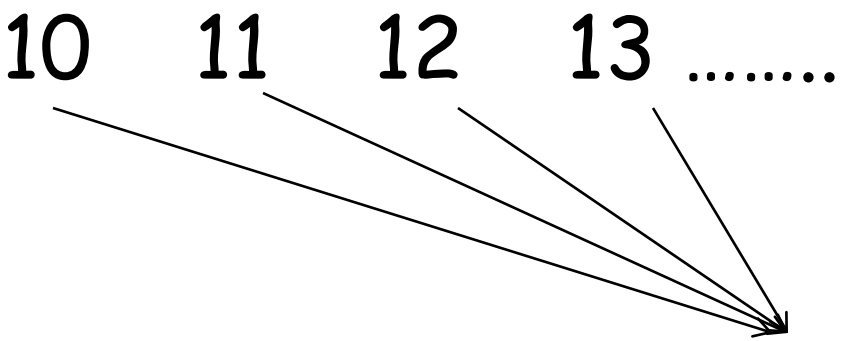
dobbiamo fare:

```
f(int A[][10], int righe)
f1(int A[][11], int righe)
f2(int A[][12], int righe)
f3(int A[][13], int righe)
```

…… e così via?

NO

usiamo l'allocazione contigua degli array in memoria per trattarli tutti come array ad una dimensione




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

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

}
```

TIPO

```
int& t3(int*A,inti,int j, int k, int ns, int nr, int nc)
{
  if(0<=i && i<=ns && 0<=j && j<=nr && 0<=k &&k<=nc)

    return *(A+(i*nr*nc)+(j*nc)+k);

  throw(1); //vedi eccezioni testo 9.5 pag.136
}
```

se voglio "vedere" int X[200] come B[2][10][10] e voglio B[1][5][8], basta invocare

```
int &z=t3(X,1,5,8,2,10,10);
```

array di char sono speciali

testo 5.5 pag. 69

Gli array di char si comportano diversamente dagli array di altri tipi

1) Inizializzazione:

```
int A[]={0,1,2,3,4,56,99};//ha 7 elementi

char B[]={'p','i','p','p','o'};// ha 5 elementi

char C[]="pippo";   //ha 6 elementi
```

C[5] contiene '\0' carattere nullo
codice ASCII = 0

2) STAMPA

il carattere nullo serve da sentinella che segnala la fine stringa e serve per molte operazioni sulle stringhe

infatti cout<<"pippo";

si comporta allo stesso modo di:

cout << C; // stampa pippo

C[3]='\0' ; cout << C; ??

e cosa stampa:

```
char B[]={'p','i','p','p','o'};
cout<< B;
```

???