

Array

Aggragati di variabili omogenee ...

Cosa è un array ?

- È un insieme di variabili omogenee identificato da un indice, ad esempio
 - Se devo leggere 10 numeri dallo standard input e memorizzarli all'interno del mio programma è abbastanza scomodo usare 10 variabili diverse!

Array: esempio

```
double nuovo_numero (void) {
    double tmp;
    printf("inserisci un numero: \n");
    scanf("%lf", &tmp);
    return tmp;
}

int main (void) {
    double a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;
    a1 = nuovo_numero();
    a2 = nuovo_numero();
    .....
    a10 = nuovo_numero();
    /* resto del programma */

    return 0;
}
```

Cosa è un array ?

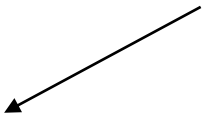
- È un insieme di variabili omogenee identificato da un indice, ad esempio
 - Se devo leggere 10 numeri dallo standard input e memorizzarli all'interno del mio programma è abbastanza scomodo usare 10 variabili diverse!
 - Posso invece utilizzare un array di 10 variabili double e scorrerle utilizzando un indice intero

Array: esempio

```
double nuovo_numero (void) {  
    double tmp;  
    printf("inserisci un numero: \n");  
    scanf("%lf", &tmp);  
    return tmp;  
}
```

```
int main (void) {  
    double a[10];  
    int i;  
    for (i=0; i<10; i++)  
        a[i] = nuovo_numero();  
}
```

Dichiarazione dell'array
Nome e numero di variabili
che servono (la *lunghezza* dell'array)



```
/* resto del programma */
```

```
    return 0;  
}
```

Array: esempio

```
double nuovo_numero (void) {  
    double tmp;  
    printf("inserisci un numero: \n");  
    scanf("%lf", &tmp);  
    return tmp;  
}
```

```
int main (void) {  
    double a[10];  
    int i;  
    for (i=0; i<10; i++)  
        a[i] ← nuovo_numero();  
  
    /* resto del programma */  
  
    return 0;  
}
```

Accesso alla variabile di indice i .
In C le variabili di una array sono numerate da 0, in questo caso 0...9

Cosa è un array ?

- È un insieme di variabili omogenee identificato da un indice, ad esempio
 - Se devo leggere 10 numeri dallo standard input e memorizzarli all'interno del mio programma è abbastanza scomodo usare 10 variabili diverse!
 - Posso invece utilizzare un array di 10 variabili double e scorrerle utilizzando un indice intero
 - Servono anche a memorizzare vettori e matrici o qualsiasi altra informazione tabellare, lo vedremo fra un attimo

Array: dichiarazione e uso

- Per dichiarare un array è necessario definire il suo nome, il tipo e il numero delle variabili (la **lunghezza**) prima del suo uso,

```
double a[10];
```

↑ ↑ ↑
tipo nome lunghezza

- La lunghezza deve essere un valore costante o comunque noto a tempo di compilazione (standard ANSI, parleremo più avanti allocazione dinamica)
- Alla dichiarazione si riserva memoria contigua per tutte le variabili nell'array. Se l'array è definito in una funzione la memoria viene riservata sullo stack.

Array: dichiarazione e uso

`double a[10];`

tipo nome lunghezza

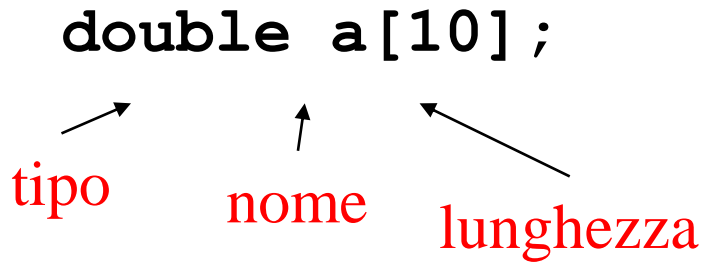
<code>a[0]</code>	?
<code>a[1]</code>	?
<code>a[2]</code>	?
<code>a[3]</code>	?
<code>a[4]</code>	?
<code>a[5]</code>	?
<code>a[6]</code>	?
<code>a[7]</code>	?
<code>a[8]</code>	?
<code>a[9]</code>	?

Come accade per le altre variabili
l'array all'inizio contiene valori
casuali che dipendono dal contenuto
precedente della memoria allocata
..e deve essere *inizializzato* prima del suo
uso

Array: dichiarazione e uso

```
double a[10];
```

tipo nome lunghezza



```
a[5] = 15;
```

```
a[0] = a[5] + 2;
```

a[0]	17
a[1]	?
a[2]	?
a[3]	?
a[4]	?
a[5]	15
a[6]	?
a[7]	?
a[8]	?
a[9]	?

- Gli elementi dell'array si usano
Esattamente come normali variabili

Array: dichiarazione e uso

```
#define N 10
double a[N];
int i;

for (i=0; i<N; i++)
    a[i]=1;
a[5] = 15;
a[0] = a[5] + 2;
a[N-1] = 4;
```

a[0]	17
a[1]	1
a[2]	1
a[3]	1
a[4]	1
a[5]	15
a[6]	1
a[7]	1
a[8]	1
a[9]	4

- È buona norma definire la lunghezza dell'array con una macro, come abbiamo già introdotto per le altre costanti

Array: non c'è l'assegnamento diretto

```
#define N 10
```

```
double a[N] = {17, 11, 4};
```

```
double b[N];
```

```
a = b;
```

- ???? Che effetto ha ????

a[0]

17

a[1]

11

a[2]

4

a[3]

0

a[4]

0

a[5]

0

a[6]

0

a[7]

0

a[8]

0

a[9]

0

Array: non c'è l'assegnamento diretto

```
#define N 10
double a[N] = {17, 11, 4};
double b[N];
/* voglio copiare tutto b in
a */
a = b; /* errato */
```

a

17
11
4
0
0
0
0
0
0
0
0

- L'errore da un messaggio un po' strano
Ma il motivo di fondo è che il nome di un array è un puntatore costante (non modificabile) che è uguale all'indirizzo di memoria dal quale è allocato l'array
- Ne riparleremo quando trattiamo i puntatori

Esempio: filtro i valori maggiori

```
#define N 10
double nuovo_numero (void) {
    double tmp;
    printf("inserisci un numero: \n");
    scanf("%lf",&tmp);
    return tmp;
}
int main (void){
    double a[N], filtro;
    int i;
    for (i=0; i<N; i++)
        a[i] = nuovo_numero();
    printf("inserisci il filtro: \n");
    scanf("%lf",&filtro);
    for (i=0; i<N; i++)
        if ( a[i] > filtro ) printf("%f\n",a[i]);
    return 0;
}
```

Esempio: istogramma ...

- Leggere da standard input una sequenza di caratteri (terminata da '\n')
- Stampare le frequenze delle cifre da 0 a 9

Esempio: istogramma

```
int main (void){
    int h[N], i;
    char c;
    for (i=0; i<N; i++) h[i]=0;
    c = getchar();
    while (c != '\n') {
        switch (c) {
            case '0': h[0]++; break;
            case '1': h[1]++; break;
            case '2': h[2]++; break;
            case '3': h[3]++; break;
            case '4': h[4]++; break;
            case '5': h[5]++; break;
            case '6': h[6]++; break;
            case '7': h[7]++; break;
            case '8': h[8]++; break;
            case '9': h[9]++; break;
        }
        c = getchar();
    } /* segue ... */
}
```

Esempio: istogramma

```
int main (void){
    int h[N], i; char;
    ....

    for (i=0; i<10; i++)
        printf("Frequenze di %d = %d\n",i, a[i]);
    return 0;
}
```

Esempio: istogramma

```
#define N 10
int main (void) {
    int h[N], i;
    char c ;
    for (i=0; i<N; i++) h[i]=0;

    c = getchar();
    while (c != '\n') {
        if ( c >= '0' && c <= '9' ) {
            i = c - '0';
            h[i]++;
        }
        c = getchar();
    }
    for (i=0; i<N; i++)
        printf("Frequenze di %d = %d\n",i, a[i]);
    return 0;
}
```

E gli array con 2 o più dimensioni ?

- E possibile definire in C array con due o più dimensioni

```
double a[10][5];
```

```
/* dichiara una matrice con 10 righe e 5  
colonne di double */
```

```
double c[3][3][3];
```

```
/* dichiara una cubo 3x3x3 */
```

- Vengono memorizzate in celle di memoria successive per riga
 - o comunque a partire dalla dimensione minore

Esempio: due dimensioni ...

```
#define N 2
```

```
#define M 3
```

```
double A[N][M] ;
```

```
A[0][0]
```

?

```
A[0][1]
```

?

```
A[0][2]
```

?

```
A[1][0]
```

?

```
A[1][1]
```

?

```
A[1][2]
```

?

Esempio: somma di matrici

```
#define N 10
int main (void){
    int a[N][N], b[N][N], c[N][N];
    int i, j;
    /* lettura delle due matrici a e b dallo
       standard input lo svilupperemo in laboratorio */
    .....
    /* calcolo della somma */
    for (i=0; i < N; i ++){
        for (j=0; j < N; j ++){
            c[i][j]=a[i][j]+b[i][j]

        /* eventuale stampa del risultato ... */
    }
}
```

Esempio: tre dimensioni ...

```
#define M 2
```

```
double A[M][M][M] ;
```

```
A[0][0][0]
```

```
A[0][0][1]
```

```
A[0][1][0]
```

```
A[0][1][1]
```

```
A[1][0][0]
```

```
A[1][0][1]
```

```
A[1][1][0]
```

```
A[1][1][1]
```

?
?
?
?
?
?
?
?

Stringhe

Le stringhe

- Le stringhe sono sequenze di caratteri,
 - in C le stringhe costanti vengono denotate da una successione di caratteri racchiusa fra apici
Es:

`"ciccio"`

`"n = %d"`

`"Ciao Mondo"`

- La rappresentazione interna è come un array di caratteri non modificabile terminato dal carattere `'\0'`

c	i	c	c	i	o	\0
---	---	---	---	---	---	----

Le stringhe

c	i	c	c	i	o	\0
---	---	---	---	---	---	----

- Quindi una stringa occupa un array con un carattere in più riservato al carattere terminatore
- Le stringhe variabili sono rappresentate come array di caratteri:
char parola[M] , frase[N] ;
- Per quanto detto prima bisogna sempre ricordarsi di allocare un carattere in più dei caratteri contenuti nelle stringhe che intendiamo scriverci dentro

Le stringhe

Per stampare una stringa si utilizzano `printf()` e `scanf()` con il modificatore `%s`

- As esempio:

```
int main (void) {
    char parola[M];
    printf("Inserisci una stringa:\n");
    scanf("%s",parola);
    printf("Ho letto: %s\n",parola);
    return 0;
}
```

Le stringhe

```
$ ./esegui
```

```
Inserisci una stringa:
```

Se digito `ciccio` e newline ↓

```
$ ./esegui
```

```
Inserisci una stringa: ciccio
```

```
Ho letto ciccio
```

```
$
```

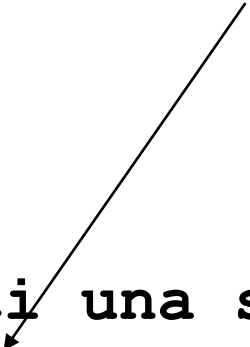
Le stringhe

Per stampare una stringa si utilizzano `printf()` e `scanf()` con il modificatore `%s`

- As esempio:

```
int main (void) {  
    char parola[M];  
    printf("Inserisci una stringa:\n");  
    scanf("%s", parola);  
    printf("Ho letto: %s\n", parola);  
    return 0;  
}
```

Non dobbiamo mettere `&`
perchè è un array
(vedi più avanti passaggio
di array a funzioni)



Le stringhe

- La libreria **string.h** contiene un insieme di funzioni predefinite per lavorare con le stringhe, ad esempio
 - **strcpy, strncpy()** per copiare una stringa in un'altra
 - **strcmp(), strncmp()** per confrontarle lessicograficamente
 - **strcat, strncat()** per concatenarle
 - **strlen()** per avere la lunghezza (senza il terminatore)
 - E molto altro

Si tratta di funzioni utili ma che possono avere dei malfunzionamenti anche gravi se non usate bene descriveremo in dettaglio caratteristiche e problemi più avanti, per il momento prima di usarle leggete attentamente il manuale (Sezione 3)

Puntatori

Puntatori : idea di base

- In C è possibile *conoscere e denotare* l'indirizzo dell'area di memoria in cui è memorizzata una variabile (il *puntatore*)

es :

```
int a = 50; /* una var intera */
```

Puntatori : idea di base

- In C è possibile *conoscere e denotare* l'indirizzo dell'area di memoria in cui è memorizzata una variabile (il *puntatore*)

es :

```
int a = 50; /* una var intera */
```



Ad ogni variabile è associata un'area di memoria la cui ampiezza in byte dipende dal tipo.

Ad ogni area è associato un indirizzo univoco (il *puntatore*)

Puntatori : idea di base

- L'indirizzo (puntatore) può essere denotato con l'operatore indirizzo (&) applicato alla variabile

es :

```
int a = 50; /* una var intera */
```

50 0xA50

&a

```
/* denota l'indirizzo di a, 0xA50  
esadecimale */
```

Puntatori : idea di base

- È anche possibile dichiarare variabili di tipo puntatore a un certo tipo (che possono contenere indirizzi di variabili di quel tipo),

- Sintassi:**

si usa il nome del tipo seguito dal modificatore asterisco (*)

es :

```
int a = 50; /* una var intera */
```

```
int * b; /* variabile puntatore a intero*/
```

50 0xA50

Puntatori : idea di base

- Sono variabili come tutte le altre e possiamo assegnare loro degli indirizzi (del tipo giusto!)

es :

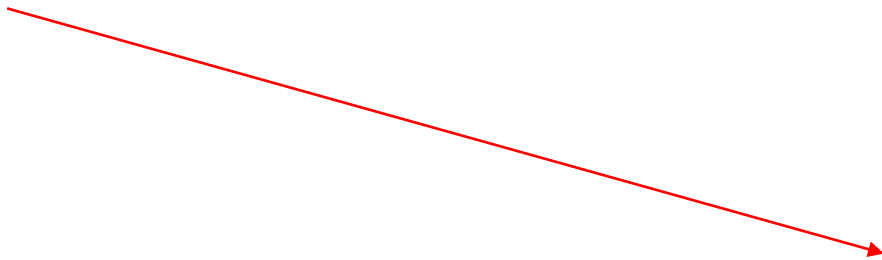
```
int a = 50; /* una var intera */
```

```
int * b; /* variabile puntatore a intero*/
```

```
b = &a; /* b vale 0xA50 */
```

50 0xA50

0xA50 0xB22



Puntatori : idea di base

- Sono variabili come tutte le altre e possiamo assegnare loro degli indirizzi (del tipo giusto!)

es :

```
int a = 50; /* una var intera */
```

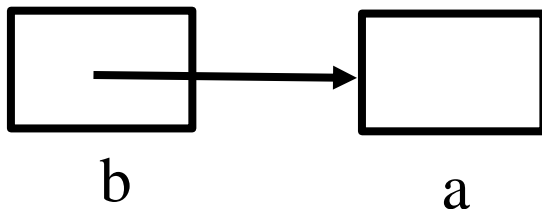
```
int * b; /* variabile puntatore a intero*/
```

```
b = &a; /* b vale 0xA50 */
```

50 0xA50

In questo caso si dice che "b punta ad a"

E si usa anche questa rappresentazione visiva



0xA50 0xB22

Puntatori : idea di base

- Inoltre conoscendo l'indirizzo di una variabile e' possibile conoscerne il valore e modificarlo! Con l'operatore di dereferenziazione (*)

es :

```
int a = 50; /* una var intera */
```

```
int * b; /* variabile puntatore a intero*/
```

```
b = &a; /* b vale 0xA50 */
```

50

0xA50

```
printf("%d %d", a, *b+1);
```

```
/* stampa "50 51" */
```

0xA50

0xB22

Puntatori : idea di base

- Inoltre conoscendo l'indirizzo di una variabile e' possibile conoscerne il valore e modificarlo! Con l'operatore *

es :

```
int a = 50; /* una var intera */
```

```
int * b; /* variabile puntatore a intero*/
```

```
b = &a; /* b vale 0xA50 */
```

12	0xA50
----	-------

```
printf("%d %d", a, *b+1); /* stampa 50 51 */
```

```
*b= *b - 38;
```

```
printf("%d %d", a, *b); /* stampa 12 12 */
```

Attenzione : i due usi di *

- Si usa nella dichiarazione come modificatore del tipo per dichiarare una variabile di tipo indirizzo

```
double *a, *b; /* ripetere '*' */  
int *a, b, c[4], **d;
```

- Si usa nelle espressioni per denotare la variabile puntata da un certo indirizzo

```
printf("%d %d", a, *b + 1);  
*b = *b - 38;  
printf("%d %d", a, *b);
```

Puntatori : costanti e stampa

- Per tutti i tipi puntatore esiste la costante "puntatore nullo" o non significativo

NULL

- È una costante predefinita (in **stdio.h**)
- Esempio:

```
int * b = NULL;
```

- **printf()**, **scanf()**, segnaposto (**%p**)

stampa il valore dell'indirizzo in notazione esadecimale, es

```
printf("ind. di a = %p\n", &a);  
/* stampa "ind. di a = 0xA50" */
```

Puntatori : perchè tanti tipi diversi ?

- Non posso usare una variabile di un tipo puntatore per contenere indirizzi di un altro tipo!

es :

```
int a = 50; /* una var intera */
```

```
int * b; /* var puntatore a intero*/
```

```
double * c; /* var puntatore a double*/
```

```
b = &a; /* b vale 0xA50 */ 50 0xA50
```

```
c = &a;
```

```
/* il compilatore da un warning
```

```
"assignment from incompatible pointer  
type"*/
```

Puntatori : perchè tanti tipi diversi ?

- Non posso usare una variabile di un tipo puntatore per contenere indirizzi di un altro tipo!

es :

```
int a = 50; /* una var intera */
```

```
int * b; /* var puntatore a intero*/
```

```
double * c; /* var puntatore a double*/
```

```
b = &a; /* b vale 0xA50 */ 50 0xA50
```

```
c = ( double * ) &a;
```

```
/* se è davvero quello che mi serve  
posso usare il cast esplicito */
```

Puntatori : perchè tanti tipi diversi ?

- Ma a che serve avere tanti tipi diversi ?
 - Serve a sapere in modo semplice che tipo ha la variabile puntata in una espressione in un punto del programma

es :

```
int a = 50; /* una var intera */
```

```
int * b, * d; /* var puntatore a intero*/
```

```
double * c; /* var puntatore a double*/
```

50

0xA50

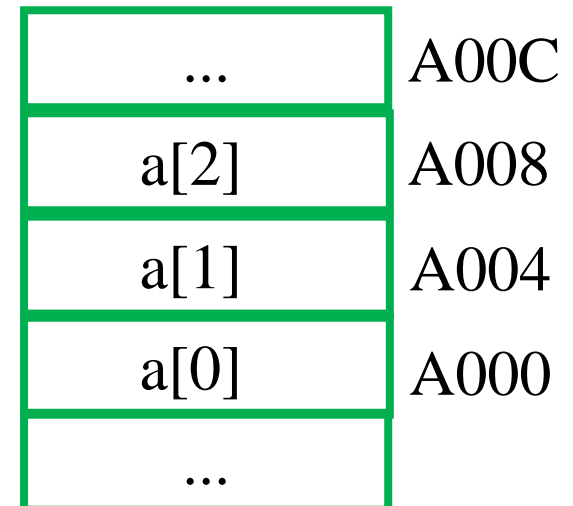
```
*d = 210 / *b ;
```

```
/* *b è intero devo usare la divisione  
intera! */
```

Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
 - Supponiamo di avere un array di 3 variabili intere
 - E supponiamo che gli interi siano memorizzati su 4 byte
 - Le situazione in memoria può essere la seguente

```
int a[3];
```



Aritmetica dei puntatori

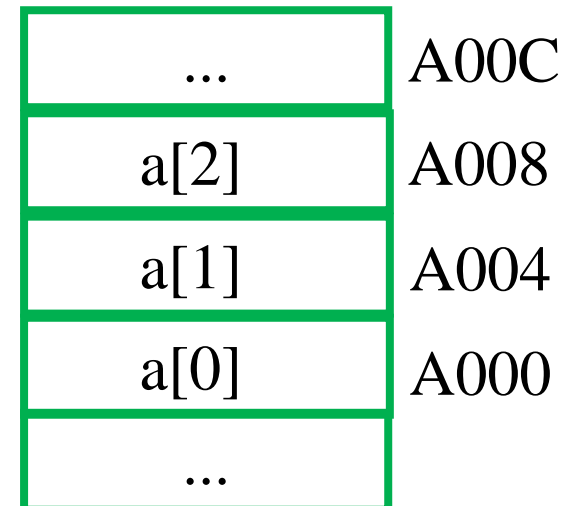
- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
- Notate che

```
int a[3];
```

```
&a[0] vale 0xA000
```

```
&a[1] vale 0xA004
```

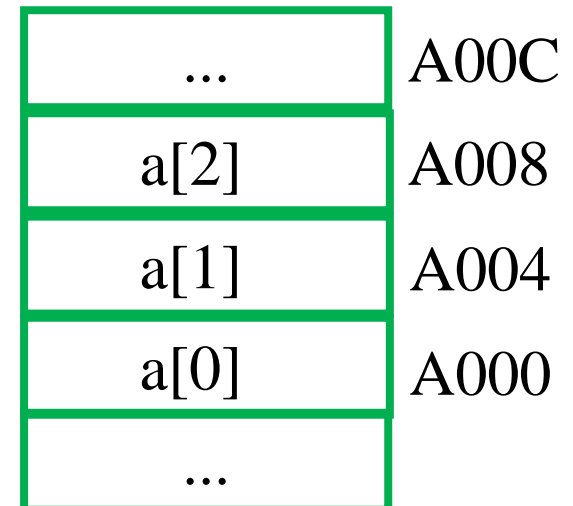
```
&a[2] vale 0xA008
```



Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
- Proviamo ora a stampare il nome dell'array a...

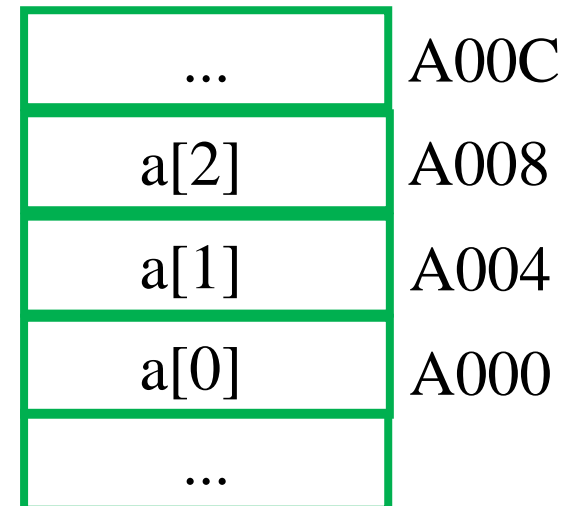
```
int a[3];  
printf("%p", a);
```



Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
- Proviamo ora a stampare il nome dell'array a...

```
int a[3];  
printf("%p", a);  
/* stampa 0xA000 */
```



Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
- Proviamo ora a stampare il nome dell'array a...

```
int a[3];  
printf("%p", a);  
/* stampa 0xA000 */
```

Il nome di un array è un puntatore costante (non modificabile) che ha come valore L'indirizzo del primo elemento **&a[0]**

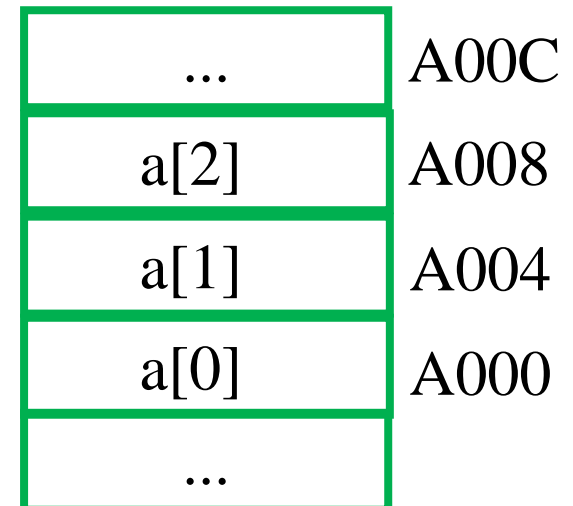
...	A00C
a[2]	A008
a[1]	A004
a[0]	A000
...	

Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
- Adesso dichiariamo una variabile di tipo puntatore a intero ed assegnamole l'indirizzo iniziale dell'array

```
int a[3], *p = &a[0]; /* oppure */
```

```
int a[3], *p = a;
```



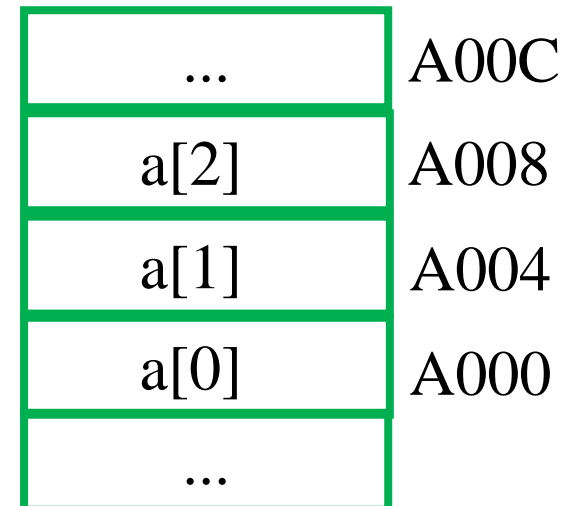
Aritmetica dei puntatori

- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
 - Adesso dichiariamo una variabile di tipo puntatore a intero ed assegnamole l'indirizzo iniziale dell'array
 - Vediamo cosa succede se incremento p di 1

```
int a[3], *p = &a[0];
```

```
p++;
```

```
printf ("%p", p);
```



Aritmetica dei puntatori

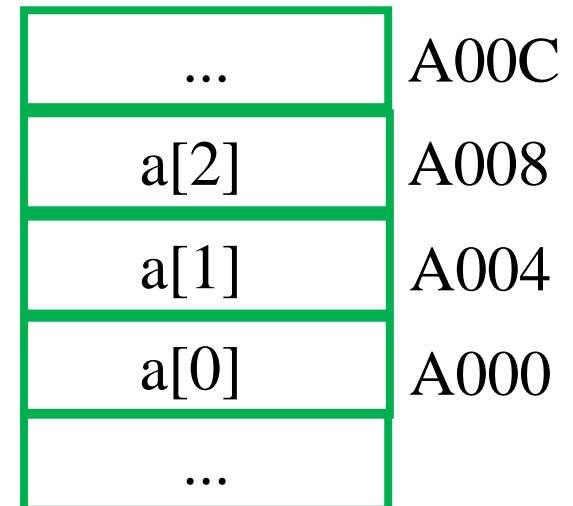
- È anche possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
 - Adesso dichiariamo una variabile di tipo puntatore a intero ed assegnamole l'indirizzo iniziale dell'array
 - Vediamo cosa succede se incremento p di 1

```
int a[3], *p = &a[0];
```

```
p++;
```

```
printf("%p", p);
```

```
/* stampa 0xA004 */
```



Aritmetica dei puntatori

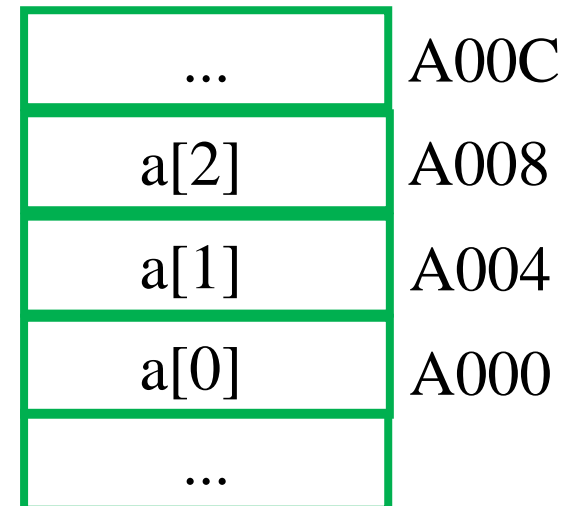
- Cosa significa sommare 1 ?
 - Significa **avanzare il puntatore di un numero di byte pari al `sizeof()` del tipo puntato**
 - In altre parole significa **avanzare un puntatore alla prossima variabile del tipo puntato**

```
int a[3], *p = &a[0];
```

```
p=p+1;
```

```
printf("%p", p);
```

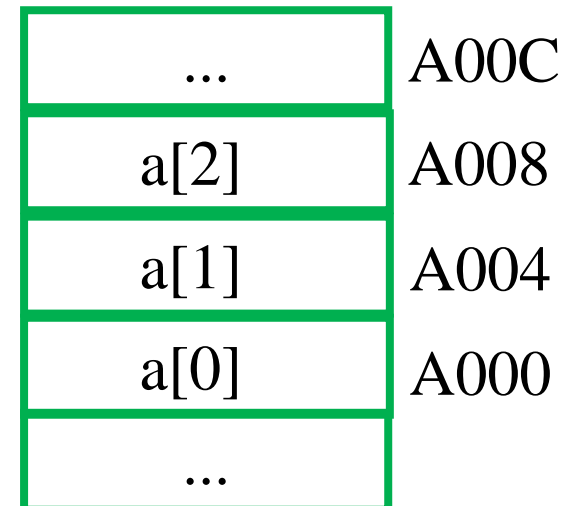
```
/* stampa 0xA004 */
```



Aritmetica dei puntatori

- E sottrarre 1?
 - Significa **diminuire il puntatore di un numero di byte pari al `sizeof()` del tipo puntato**
 - In altre parole significa **arretrare un puntatore alla precedente variabile del tipo puntato**

```
int a[3], *p = &a[0];  
p=p+1;  
p--;  
printf("%p", p);  
/* stampa 0xA000 */
```



Aritmetica dei puntatori

- Possiamo sommare e sottrarre valori interi $x \neq 1$
 - Il puntatore verrà modificato sommando $x * sizeof(type)$

```
int a[3], *p = &a[0];
```

```
p=p+3;
```

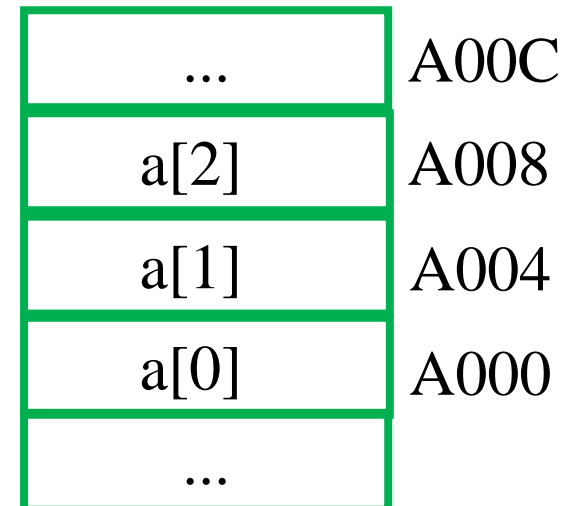
```
printf("%p", p);
```

```
/* stampa 0xA00C */
```

```
p=p-2;
```

```
printf("%p", p);
```

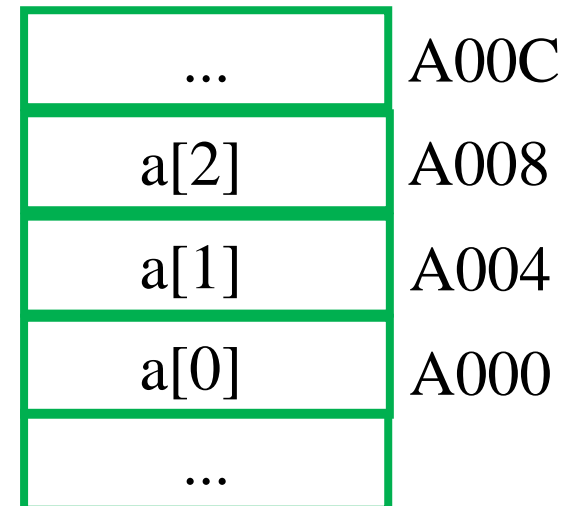
```
/* stampa 0xA004 */
```



Aritmetica dei puntatori

- Possiamo sottrarre due puntatori
 - Verrà calcolato il numero di variabili di quel tipo memorizzabili fra i due puntatori

```
int a[3], *p = &a[0], *q=a, d;  
p=p+3;  
printf("%p", p);  
/* stampa 0xA00C */  
d= p-q;  
printf("%d", d);  
/* stampa 3 */
```



Array e puntatori

- In C array e puntatori sono due concetti molto simili,
 - Abbiamo visto che il nome di un array è il puntatore costante al primo elemento dell'array
 - Vediamo adesso di capire meglio a quali operazioni su puntatori equivale l'operatore [...]

```
int a[3]={6,15,4}, tmp;
```

```
tmp = a[1];
```

```
printf("%d", tmp);
```

```
/* stampa 15 */
```

...	A00C
4	A008
15	A004
6	A000
...	

Array e puntatori

- In C array e puntatori sono due concetti molto simili,
 - Abbiamo visto che il nome di un array è il puntatore costante al primo elemento dell'array
 - Vediamo adesso di capire meglio a quali operazioni su puntatori equivale l'operatore [...]

```
int a[3]={6,15,4}, tmp;
```

```
tmp = a[1];
```

```
printf("%d", tmp);
```

```
/* stampa 15 */
```

```
tmp = * (a + 1);
```

```
printf("%d", tmp);
```

```
/* stampa 15 */
```

...	A00C
4	A008
15	A004
6	A000
...	

Array e puntatori

- Nella realtà l'operatore [...] è una abbreviazione dell'altra espressione in aritmetica dei puntatori
 - Questo significa che può essere utilizzato con qualsiasi puntatore non solo con quelli costanti! Es.

```
int a[3]={6,15,4}, p = a;
```

```
tmp = p[1];
```

```
printf("%d", tmp);
```

```
/* stampa 15 */
```

```
tmp = *(p + 1);
```

```
printf("%d", tmp);
```

```
/* stampa 15 */
```

...	A00C
4	A008
15	A004
6	A000
...	

Array e puntatori

- Quindi in C seguenti frammenti di codice sono tutti modi equivalenti di calcolare la somma degli elementi di un array :

```
int a[N], *p=a, *q, i;  
int sum = 0;
```

```
/* versione 1 */  
for(i=0; i<N; i++)  
    sum+= a[i];
```

```
/* versione 2 */  
for(i=0; i<N; i++)  
    sum+= *(a+i);
```

Puntatori e array.... (5)

- I seguenti frammenti di codice sono equivalenti (segue) :

```
int a[N], *p=&a[0], *q, i;  
int sum = 0;
```

```
/* versione 3 */  
for(i=0; i<N; i++)  
    sum+= p[i];
```

```
/* versione 4 */  
for(p=a; p<(a+N); p++)  
    sum+= *p;
```


Si ma a che serve ?

- Perchè devo preoccuparmi di andare a manipolare gli indirizzi di memoria o di capire così in dettaglio come vengono acceduti gli array ?
 - Perchè in C alcune cose importanti non si possono fare senza andare a lavorare sui puntatori o usando [...] con puntatori variabili ...
 - Perchè alcuni errori risultano incomprensibili se non si conosce cosa c'è sotto ...
 - Vediamo qualche esempio di questo (altri ne vedremo nel proseguo...)

Scrivere una funzione che restituisce più di un valore

- Abbiamo visto le funzioni in C restituiscono normalmente un singolo valore.
 - E se voglio restituirne due ?
 - Ad esempio sto leggendo dei caratteri $x_0 x_1 \dots$ da input e vorrei restituire insieme sia quello con maggiori occorrenze che il numero delle occorrenze rilevate
 - Un modo di farlo è fornire il puntatore ad una variabile in cui mettere il secondo risultato
 - Vediamo come ... Analizzando prima un esempio più semplice

Esempio: somma e differenza

```
int somma_e_diff (int a, int b, int * diff) {  
    *diff = a-b;  
    return a + b;  
}
```

```
int main (void) {  
    int s, d; /* conterranno somma e differenza */  
    s = somma (10, 2, &d);  
    printf("La somma è %d, la differenza è %d", s, d);  
    return 0;  
}
```

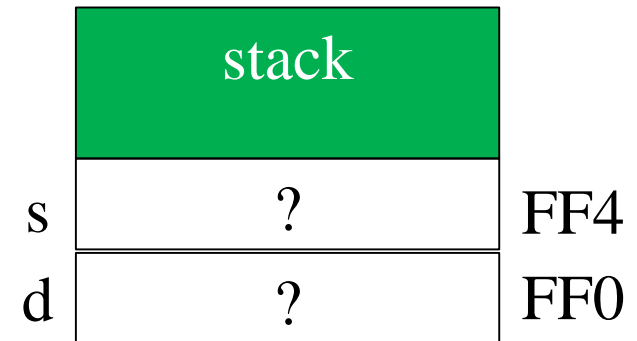
```
/* stampa correttamente
```

La somma è 12, la differenza è 8

```
Vediamo perchè funziona  
*/
```

Esempio: somma e differenza

```
int somma_e_diff (int a, int b, int * diff) {
    *diff = a-b;
    return a + b;
}
int main (void) {
    int s, d;
    s = somma (10, 2, &d);
    printf("La somma è %d,\
la differenza è %d", s, d);
    return 0;
}
```



Esempio: somma e differenza

```
int somma_e_diff (int a, int b, int * diff) {  
    *diff = a-b;  
    return a + b;  
}  
  
int main (void) {  
    int s, d;  
    Y: s = somma (10, 2, &d);  
    printf("La somma è %d,\ \  
    la differenza è %d", s, d);  
    return 0;  
}
```

	stack	
s	?	FF4
d	?	FF0
diff	0xFF0	FEC
b	2	FE8
a	10	FE4
Indirizzo di ritorno	Y	FE0

Esempio: somma e differenza

```
int somma_e_diff (int a, int b, int * diff) {
    *diff = a-b;
    return a + b;
}

int main (void) {
    int s, d;
    Y: s = somma (10, 2, &d);
    printf("La somma è %d,\
    la differenza è %d", s, d);
    return 0;
}
```

	stack	
s	?	FF4
d	8	FF0
diff	0xFF0	FEC
b	2	FE8
a	10	FE4
Indirizzo di ritorno	Y	FE0

Esempio: somma e differenza

```
int somma_e_diff (int a, int b, int * diff) {
    *diff = a-b;
    return a + b;
}
int main (void) {
    int s, d;
    Y: s = somma (10, 2, &d);
    printf("La somma è %d,\
    la differenza è %d", s, d);
    return 0;
}
```

stack		
s	12	FF4
d	8	FF0

Esempio: scorretto

```
int somma_e_diff (int a, int b, int diff) {  
    diff = a-b;  
    return a + b;  
}
```

```
int main (void) {  
    int s, d; /* conterranno somma e differenza */  
    s = somma (10, 2, d);  
    printf("La somma è %d, la differenza è %d", s, d);  
    return 0;  
}
```

```
/* Che cosa stampa questa versione (scorretta) che non  
Usa i puntatori ?  
*/
```


Esempio: scorretto

```
int somma_e_diff (int a, int b, int diff) {  
    diff = a-b;  
    return a + b;  
}  
  
int main (void) {  
    int s, d;  
    Y: s = somma (10, 2, d);  
    printf("La somma è %d,\ \  
    la differenza è %d", s, d);  
    return 0;  
}
```

	stack	
s	?	FF4
d	?	FF0
diff	?	FEC
b	2	FE8
a	10	FE4
Indirizzo di ritorno	Y	FE0

Esempio: scorretto

```
int somma_e_diff (int a, int b, int diff) {  
    diff = a-b;  
    return a + b;  
}  
  
int main (void) {  
    int s, d;  
    Y: s = somma (10, 2, d);  
    printf("La somma è %d,\ \  
    la differenza è %d", s, d);  
    return 0;  
}
```

	stack	
s	?	FF4
d	?	FF0
diff	8	FEC
b	2	FE8
a	10	FE4
Indirizzo di ritorno	Y	FE0

Esempio: scorretto

```
int somma_e_diff (int a, int b, int diff) {
    diff = a-b;
    return a + b;
}
int main (void) {
    int s, d;
    Y: s = somma (10, 2, d);
    printf("La somma è %d,\
la differenza è %d", s, d);
    return 0;
}
```

stack		
s	12	FF4
d	?	FF0

All'uscita delle funzione tutti i valori del frame vengono eliminati comprese le copie dei paramentri e Le loro eventuali modifiche ed il valore di diff è perso

Esempio: istogramma

```
char istogramma (int * n_occ) {
    int h[N], i, max, imax;
    char c = 'a';
    for (i=0; i<N; i++) h[i]=0;

    while (c != '\n') {
        if ( c >= '0' && c <= '9' ) h[c - '0']++;
        c = getchar();
    }

    max = imax = -1;
    for (i=0; i<N; i++)
        if ( max < h[i] ) {
            max = h[i];
            imax = i;
        }

    *n_occ = h[i];
    return ('0' + imax);
}
```

Esempio: istogramma

```
char istogramma (int * n_occ) {  
    .....  
    *n_occ = h[i];  
    return ('0' + imax);  
}  
  
int main (void) {  
    int b; /* conterrà il numero delle occorrenze */  
    char max; /* conterrà il carattere con  
              occorrenza massima */  
    max = istogramma(&b);  
  
    printf("Il carattere con max occorrenze è %c,\n  
con %d occorrenze.", max, b);  
    return 0;  
}
```

Passare gli array a una funzione

- Abbiamo visto che non è possibile copiare in un solo colpo tutti gli elementi di un array
- Allo stesso modo in C non è possibile passare ad una funzione una copia di una array
 - Come avviene per gli scalari
- L'unica possibilità è fornire alla funzione una copia del puntatore al primo elemento
 - Vediamo come
 - **L'array e' condiviso fra funzione chiamata e funzione chiamante!**

Esempio: il massimo di un array

- Vogliamo scrivere una funzione che prende come parametro un array e ne trova il massimo restituendolo come valore
 - Per quanto detto posso solo passare il puntotore all'inizio dell'array, quindi

```
int max_array( int * a)
```
 - Dopodichè posso usare l'operatore [...] per accedere agli elementi normalmente

```
int max_array( int * a){ .... a[i]....}
```

Esempio: massimo di un array

```
/* se però provo a scrivere la funzione ho qualche  
problemino .... */
```

```
int max_array(int * a) {  
    int i, max;  
    max= a[0];  
    for (i=0; i<???????; i++)  
        max = max < a[i] ? a[i] : max ;  
    return max  
}
```


Esempio: massimo di un array

```
/* se però provo a scrivere la funzione ho qualche  
problemino .... */
```

```
int max_array(int * a) {  
    int i, max;  
    max= a[0];  
    for (i=0; i<???????; i++)  
        max = max < a[i] ? a[i] : max ;  
    return max  
}
```

```
/* in realta' non sono nemmeno sicura che ci sia un  
elemento di indice 0..
```

```
In realtà non so quanti elementi ha l'array e con  
il puntatore dato posso solo accedere alla memoria  
vicina */
```

Esempio: il massimo di un array

Conoscendo solo il puntatore non è possibile sapere quanto è lungo l'array, quindi devo fornire anche questa informazione alla funzione, ad esempio

```
int max_array(int * a, int n) {
    int i, max;
    if ( n <= 0) return 0;
    max= a[0];
    for (i=0; i<n; i++)
        max = max < a[i] ? a[i] : max ;
    return max
}
```

Passare gli array a una funzione

Quindi per array unidimensionali

- il modo standard di passarli ad una funzione è con la coppia di parametri
 - Il puntatore al primo elemento dell'array
 - La sua lunghezza
- Ci sono diverse sintassi alternative per passare il puntatore, ad esempio:

```
int max_array(int * a, ...) {...  
int max_array(int a[], ...) {...  
int max_array(int a[5], ...) {...
```

```
/* sono tutte equivalenti meglio secondo me  
usare la prima per ricordarsi bene che stiamo  
manipolando solo un puntatore */
```

Passare gli array a una funzione

Quindi per array unidimensionali

- È fondamentale capire che questo meccanismo di passaggio **non crea una copia dell'array ma solo del suo indirizzo** (si chiama infatti passaggio *per indirizzo*)
- Quindi se modifico l'array dentro la funzione **tutte le modifiche vengono effettuate sull'unica copia dell'array presente nell'ambiente di chi ha chiamato la funzione**
- In alcuni casi questo può essere utile per non fare molte copie dei dati ma in altri può essere necessario crearsi (con un ciclo) una copia locale per non sporcare quella originale

Array: lettura di una array da stdin

```
#define N 10
#define M 20
void nuovo_array (double* a, int n) {
    int i;
    for (i=0; i< n; i++) {
        printf("inserisci un numero: \n");
        scanf("%lf", &a[i]);
    }
}
int main (void) {
    double a[N], b[M];
    nuovo_array(a, N);
    nuovo_array(b, M);
    /* resto del programma i valori letti sono in a e
b */
    ....
    return 0;
}
```

Passare gli array a una funzione

Quindi per array unidimensionali

- È fondamentale anche fare buon uso della lunghezza per evitare di uscire dai limiti dell'array, controllando di non generare indirizzi oltre il limite

a[i] viene **sempre** trasformato in ***(a+i)**

anche se $i=100$ e l'array ha solo 10 elementi

- Questo è estremamente pericoloso, potete accedere ad altre parti della memoria e leggere e scrivere questa memoria in modo incontrollato

Array: lettura di una array da stdin

```
#define N 10
#define M 20
void nuovo_array (double* a, int n) {
    int i;
    for (i=0; i< n+2; i++) {
        printf("inserisci un numero: \n");
        scanf("%lf", &a[i]);
    }
}
```

```
int main (void) {
    double a[N], b[M];
    nuovo_array(a, N);
    nuovo_array(b, M);
    /* resto del programma i valori letti sono in a e
b */
    ....
    return 0;
}
```

Scorretto ma il compilatore non lo rileva

Passare array bidimensionali

- Per gli array con più di una dimensione il passaggio appena descritto non funziona
- Vediamo perchè su una matrice bidimensionale:

Esempio: due dimensioni ...

```
#define N 2  
#define M 3  
double A[N][M] ;
```

$\&A[1][2] = A + M * 1 + 2$

A[0][0]	?	0xAA000
A[0][1]	?	0xAA004
A[0][2]	?	0xAA008
A[1][0]	?	0xAA00C
A[1][1]	?	0xAA010
A[1][2]	?	0xAA014

Passare array bidimensionali

- In generale:
 - L'indirizzo di `a[i][j]` è
$$a + m * i + j$$

Quindi per essere in grado di calcolarlo dentro una funzione devo conoscere il valore di m , ovvero il numero di colonne della matrice

- Questo significa che devo specificare il numero delle colonne esplicitamente con una **costante** nota a tempo di compilazione quando dichiaro la funzione

```
void stampa(int mat[][5], int righe) {  
    ...  
}
```

Passare array ≥ 3 dimensioni ?

- Funziona esattamente allo stesso modo, devo specificare i valori di tutte le dimensioni eccetto la prima:

- Esempio in 3 dimensioni

```
void stampa(int mat[][5][3], int righe) {  
    ... }
```

L'indirizzo di `mat[i][j][k]` è

$$mat + 5 * 3 * i + 3 * j + k$$

- Questo ha come effetto spiacevole che devo definire una funzione diversa per ogni combinazione di valori
- Altrimenti il compilatore non ce la fa a generare il codice corretto

Array multidimensionali, che fare ?

- Per passare un array multidimensionale ad una funzione in modo meno rigido ci sono varie strategie:
- **Collassarlo in un array unidimensionale**, in questo caso avrei

```
void stampa(int mat_1*, int nrow, int ncol){  
    ...}
```

Ma in questo caso ogni volta che accedo ad un elemento devo calcolarmi esplicitamente l'indice, ad esempio per calcolare l'elemento **mat[i][j]** calcolo prima l'indice

$$k = ncol * i + j$$

E poi accedo a **mat_1[k]**

Array multidimensionali, che fare ?

- Per passare un array multidimensionale ad una funzione in modo meno rigido ci sono varie strategie:

- **Dichiararlo globale**, in questo caso avrei

```
#define N 2  
#define M 3  
double A[N][M] ;
```

```
void stampa(void) { extern A; ... }
```

Ma in questo caso la funzione lavora sempre sulla stessa matrice A, non posso passare la matrice come parametro!

Array multidimensionali, che fare ?

- È però possibile costruire una rappresentazione interna **non contigua** che permette di passare solo il puntatore
 - Ne parleremo quando discuteremo l'allocazione dinamica degli array con le varie funzioni della famiglia malloc()

.....

Algoritmi di ordinamento

Il problema

- Vogliamo ordinare un array monodimensionale in base in modo crescente
 - per il caso decrescente valgono le stesse considerazioni
- Vari algoritmi possibili
 - Diverse caratteristiche di complessità computazionale

Un primo algoritmo: selection sort

- Abbiamo un array a di lunghezza N
 - Al primo passo scegliamo il valore minimo fra tutti quelli presenti nell'array e lo spostiamo nella posizione 0
 - Al passo 1 scegliamo il minimo fra tutti i valori in posizione $1, 2, \dots, N-1$ e lo scambiamo con il valore in posizione 1
 - Al passo i scegliamo il minimo fra tutte le posizioni $i, i+1, \dots, N-1$
e lo scambiamo con il valore in posizione i

Selection sort: esempio

5	2	4	6	1	3
1	2	4	6	5	3
1	2	4	6	5	3
1	2	3	6	5	4
1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6

Selection sort: codifica

```
/* calcola la posizione del minimo elemento di
v nella porzione [from,to] */
int minPos(int * v, int from, int to);
/* scambia il valore delle variabili puntate da p e q */
void swap(int *p, int *q);
void selectionSort(int * v, int dim) {
    int i, min;
    for(i=0; i<dim-1; i++){
        min = minPos(v, i, dim-1);
        swap(v+i, v+min);
    }
}
/* swap, minPos le implementiamo in laboratorio */
```

Altro algoritmo: bubblesort

- Si fanno salire gli elementi piu piccoli (piu *leggeri*) verso l'inizio del vettore (*verso l'alto*), scambiandoli con quelli adiacenti
- L'ordinamento e suddiviso in $N-1$ fasi:

bubblesort: esempio

5 2 4 6 1 3

5 2 4 6 1 3

5 2 4 1 6 3

5 2 1 4 6 3

5 1 2 4 6 3

1 5 2 4 6 3

1 5 2 4 3 6

1 5 2 3 4 6

1 5 2 3 4 6

1 2 5 3 4 6

bubblesort: esempio

1 2 5 3 4 6

1 2 5 3 4 6

1 2 5 3 4 6

1 2 3 5 4 6

1 2 3 5 4 6

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6

bubblesort: codifica

```
/* scambia il valore delle variabili puntate da p e q */
```

```
void swap(int *p, int *q);
```

```
void bubbleSort(int * v, int dim) {
```

```
    int temp,i,j;
```

```
    for (i = 0; i < dim-1; i++) /* fase i-esima */
```

```
        /* bolla piu' leggera in posizione i */
```

```
            for (j = dim-1; j > i; j--)
```

```
                if (v[j] < v[j-1]) swap(v+j, v+j-1);
```

```
    }
```

```
/* lo implementiamo in laboratorio */
```

Selection sort ricorsivo

Il metodo del selection sort puo essere facilmente realizzato in modo ricorsivo:

- si definisce una funzione che ordina (ricorsivamente) la porzione di array individuata da due indici **from** e **to**
- il minimo della porzione viene messo in posizione **from**
- Poi si ordina ricorsivamente la porzione tra **from+1** e **to**
- Il caso base corrisponde all'ordinamento di una porzione fatta da un solo elemento (è già ordinata!)

Selection sort rec: codifica

```
void selectionSort_r (int* v, int from, int to) {
    if (from < to) {
        int min = minPos(v, from, to);
        swap(v+from, v+min);
        selectionSort_r (v, from+1, to);
    }
}

void selectionSort (int v[], int dim) {
    selectionSort_r (v, 0, dim-1);
}

/* lo implementiamo in laboratorio */
```

Algoritmo Merge-Sort

- Utilizza uno schema di soluzione *divide et impera*
 - *Decomposizione*: Si divide il problema in piu' sottoproblemi
 - *Ricorsione*: Si risolvono i sottoproblemi
 - *Combinazione*: Si combina la soluzione dei sottoproblemi nella soluzione complessiva del problema di partenza

Algoritmo Merge-Sort

- Per l'ordinamento di un array
 - *Decomposizione*: Si divide l'array in due parti di lunghezza simile
 - *Ricorsione*: Si ordinano le due parti
 - *Combinazione*: Si combinano i due array ordinati

Algoritmo Merge-Sort

- Per l'ordinamento di un array
 - *Decomposizione*: Si divide l'array in due parti di lunghezza simile
 - *Ricorsione*: Si ordinano le due parti
 - *Combinazione*: Si combinano i due array ordinati

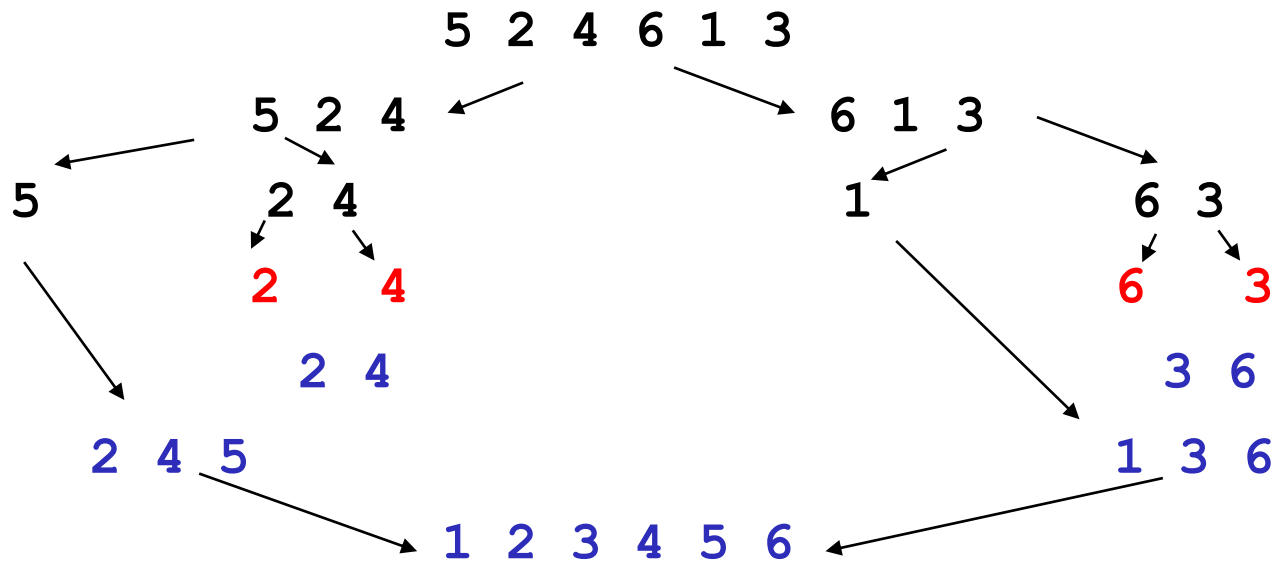
Merge sort: codifica

```
void mergeRicorsivo(int * a, int from, int to) {
    int mid;
    if (from < to) { /* l'intervallo da mid a to, estremi
        inclusi, comprende almeno due elementi */
        mid = (from + to) / 2;
        mergeRicorsivo(a, from, mid);
        mergeRicorsivo(a, mid+1, to);
        merge(a, from, mid, to); /* fonde le due porzioni
            ordinate [from, mid], [mid+1, to] nel sottovettore
            [from, to] */
    }
}
```

Merge sort: codifica

```
void sort(int * v, int dim) {  
    mergeRicorsivo(v, 0, dim-1);  
}
```

Esempio:



```

void merge(int* a, int from, int mid, int to) {
    int b[LUNG]; /* vettore di appoggio */
    int i = from, j = mid + 1; /* scorrono la prima e la seconda
        parte di a */
    int k= from; /* scorre b */
    /* copio il minore dalla prima o dalla seconda porzione */
    while (i <= mid && j <= to) {
        if (a[i] <= a[j]) {
            b[k] = a[i];
            i++; /* copia dalla prima porzione ed avanza l'indice */
        }
        else {
            b[k] = a[j] ;
            j++; /* copia dalla seconda porzione ed avanza l'indice */
        }
        k++;
    } /* fine while */
} /* ... Segue ... */

```

```

void merge(int* a, int from, int mid, int to) {
    .....
    /* controllo quale porzione è finita prima e copio quello
    che rimane dell'altra nel vettore di appoggio */
    if ( i > mid )
        for ( ; j <= to ; j++, k++ )
            b[k] = a[j] ;
    else
        for ( ; i <= mid ; i++, k++ )
            b[k] = a[i] ;

    /* ricopia tutti gli elementi ordinati da b ad a */
    for ( k = from ; k <= to ; k++ )
        a[k] = b[k] ;
} /* fine merge */

```



```
/* un possibile main */  
#define LUNG 10  
  
int main (void) {  
    int a[LUNG];  
    leggi(a, LUNG);  
    mergeSort(a, LUNG);  
    stampa(a, LUNG);  
    return 0;  
} /* fine main */
```