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 */
```

0xA50

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

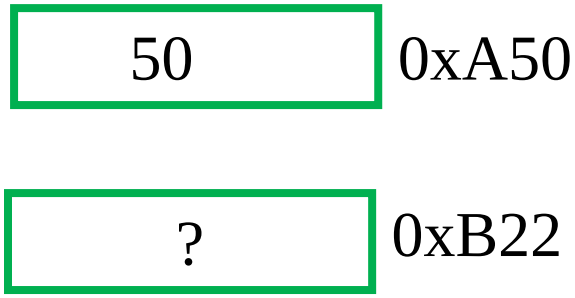
Puntatori : idea di base

- 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*/
```

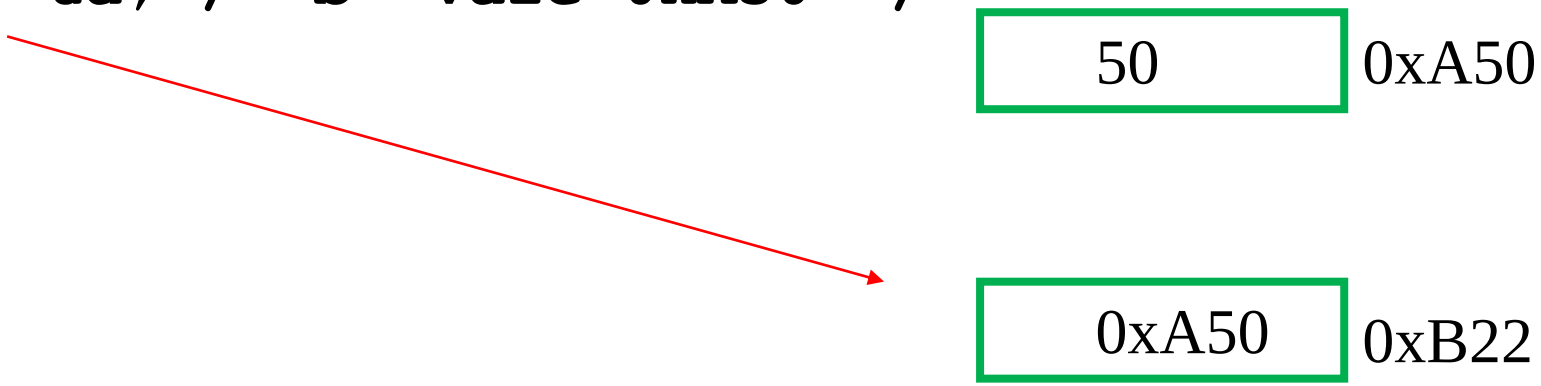


Puntatori : idea di base

- 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 */
```



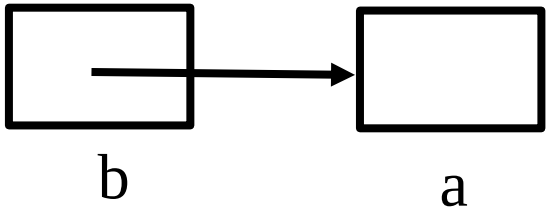
Puntatori : idea di base

- 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 */
```

In questo caso si dice che "b punta ad a"
E si usa anche questa rappresentazione visiva

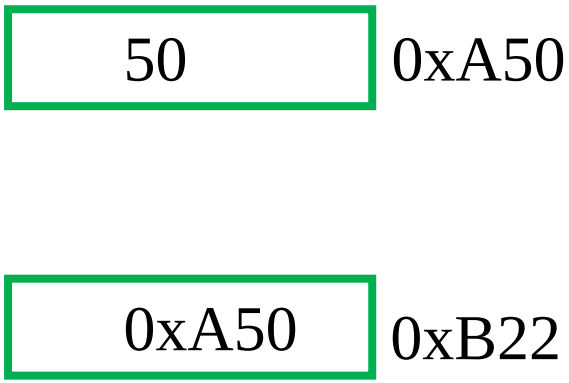


Puntatori : idea di base

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



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 */
```


```
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 */
c = &a;
/* il compilatore da un warning
"assignment from incompatible pointer
 type"*/
```

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 :

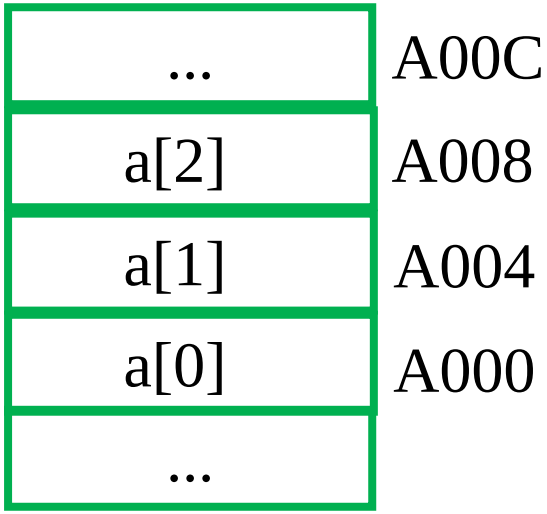
```
int a = 50; /* una var intera */
int * b, * d; /* var puntatore a intero*/
double * c; /* var puntatore a double*/
```


```
*d = 210 / *b ;
/* *b è intero devo usare la divisione
  intera! */
```

Aritmetica dei puntatori

- È 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

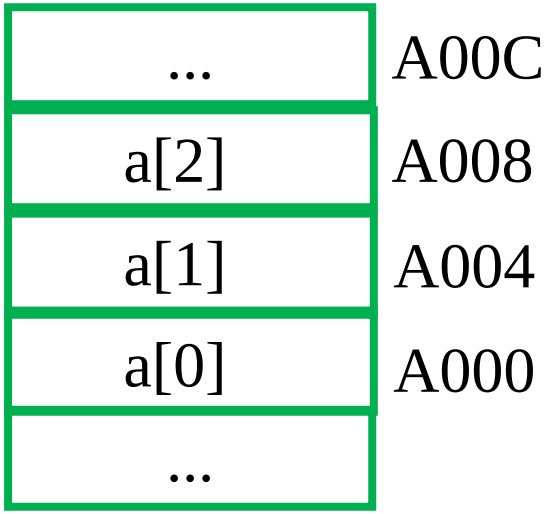
```
int a[3];
```



Aritmetica dei puntatori

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

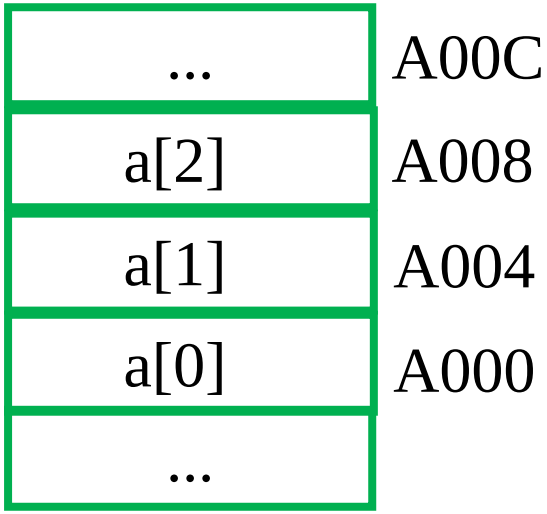
```
int a[3];
&a[0] vale 0xA000
&a[1] vale 0xA004
&a[2] vale 0xA008
```



Aritmetica dei puntatori

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

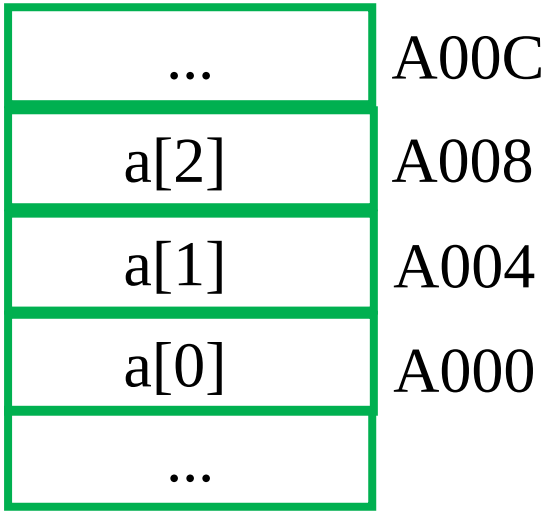
```
int a[3];
printf("%p",a);
```



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 */
```

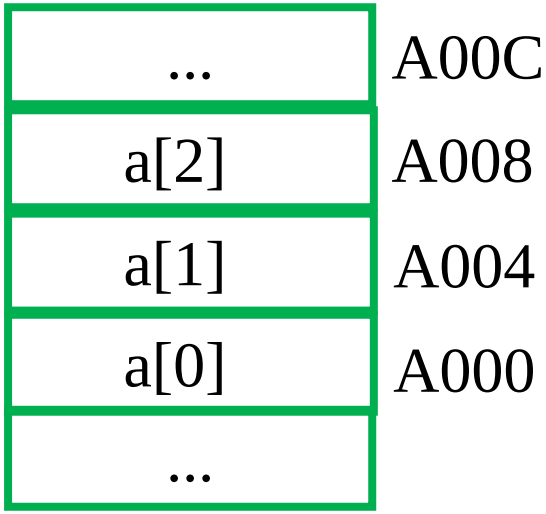


Aritmetica dei puntatori

- 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 */
```

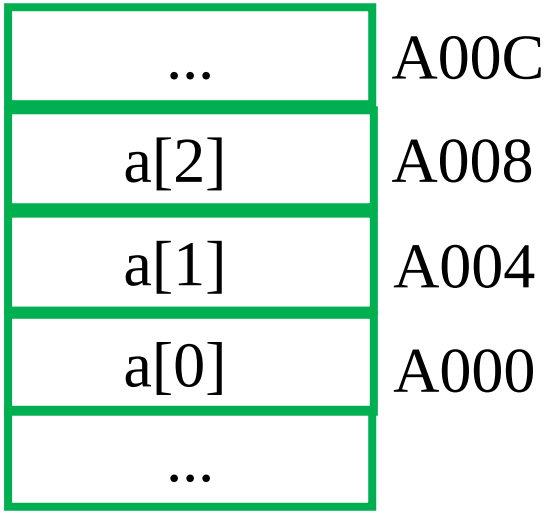
Il nome di un array è un puntatore costante (non modificabile) che ha come valore L'indirizzo del primo elemento **`&a[0]`**



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

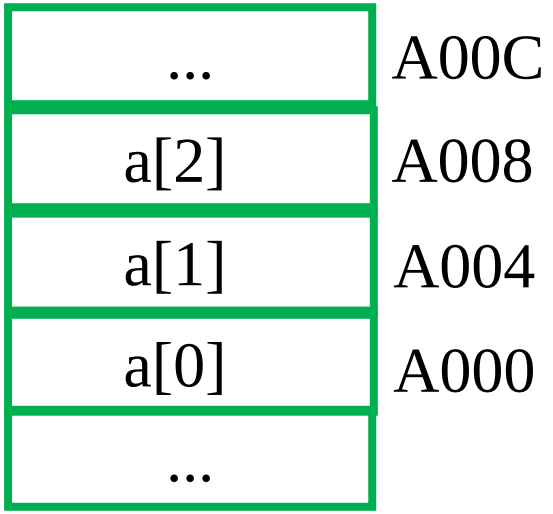
```
int a[3], *p = &a[0]; /* oppure */
int a[3], *p = a;
```



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 incremanto p di 1

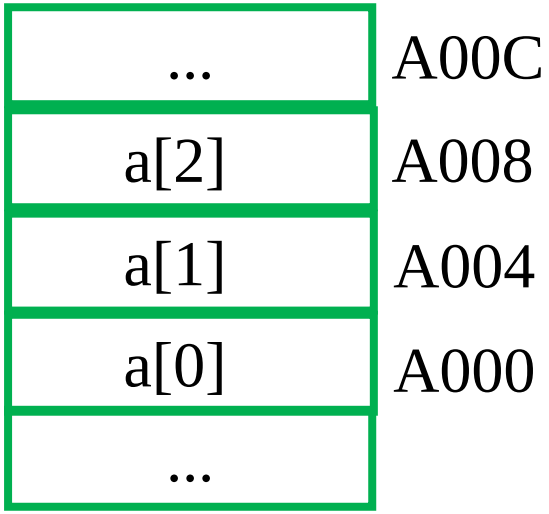
```
int a[3], *p = &a[0];


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 incremanto p di 1

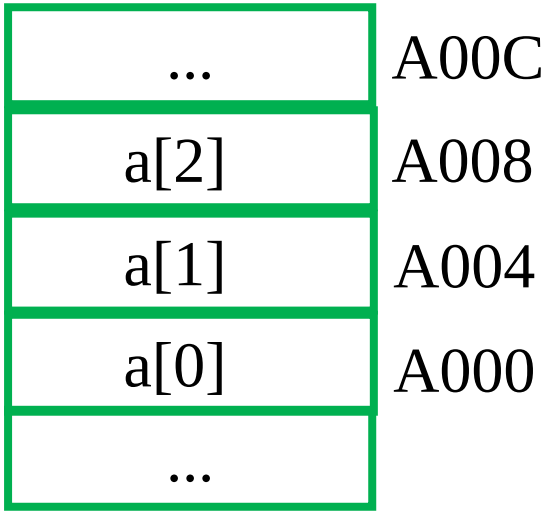
```
int a[3], *p = &a[0];


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

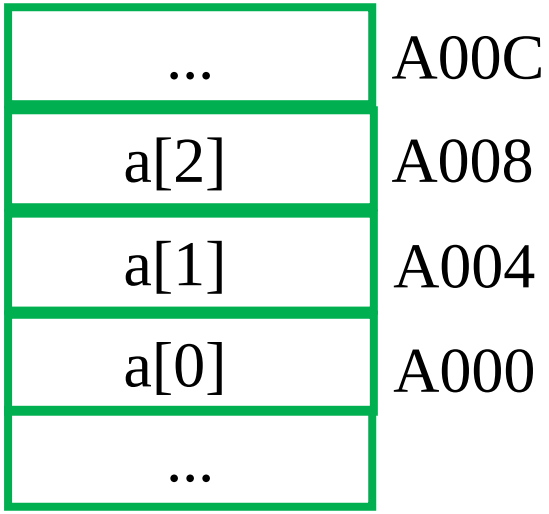
```
int a[3], *p = &a[0];


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

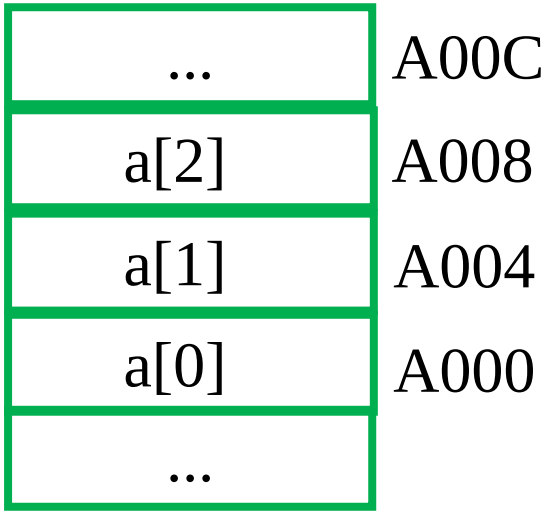
```
int a[3], *p = &a[0];
p=p+1;
p--;
printf("%p",p);
/* stampa 0xA000 */
```



Aritmetica dei puntatori

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

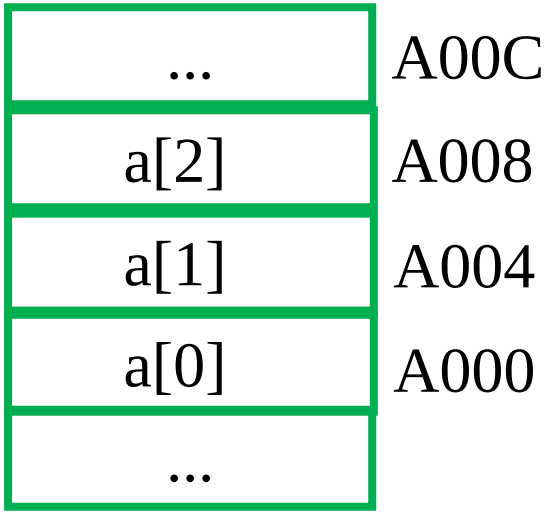
```
int a[3], *p = &a[0];
p=p+3;
printf("%p",p);
/* stampa 0xA00C */
p=p-2;
printf("%p",p);
/* stampa 0xA004 */
```



Aritmetica dei puntatori

- Possiamo sottrarre due puntatori
  - Verrà calcolato il numero di variabili di quel tipo memorizzabili fra i due puntatori (un intero!)

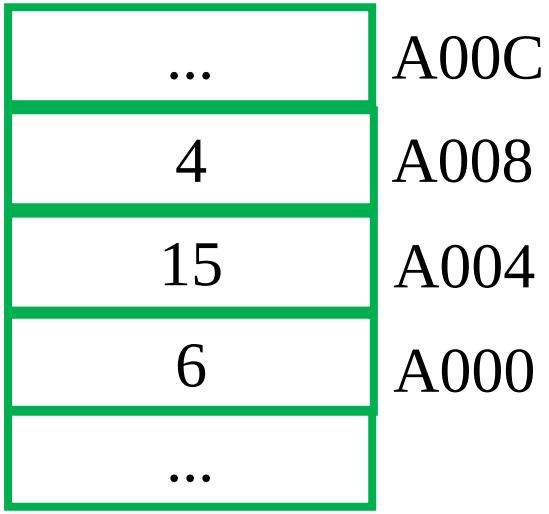
```
int a[3], *p = &a[0], *q=a, d;
p=p+3;
printf("%p",p);
/* stampa 0xA00C */
d= p-q;
printf("%d",d);
/* stampa 3 */
```



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 [...]

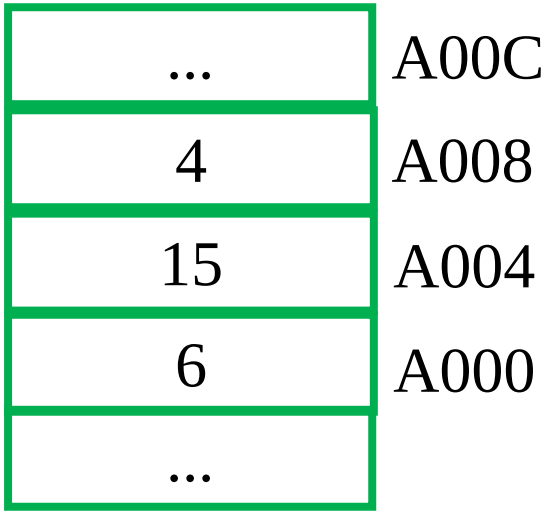
```
int a[3]={6,15,4}, tmp;
tmp = a[1];
printf("%d",tmp);
/* stampa 15 */
```



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 [...]

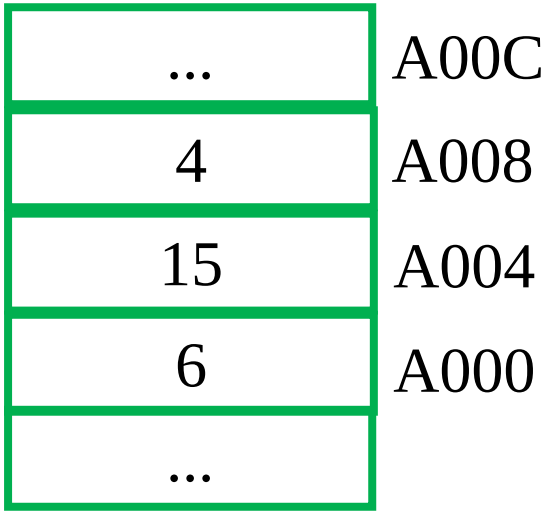
```
int a[3]={6,15,4}, tmp;
tmp = a[1];
printf("%d",tmp);
/* stampa 15 */
tmp = * (a + 1);
printf("%d",tmp);
/* stampa 15 */
```



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 */
```



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], 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….

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

```
int a[N],*p=&a[0], i; /* equivalente a *p=a */
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

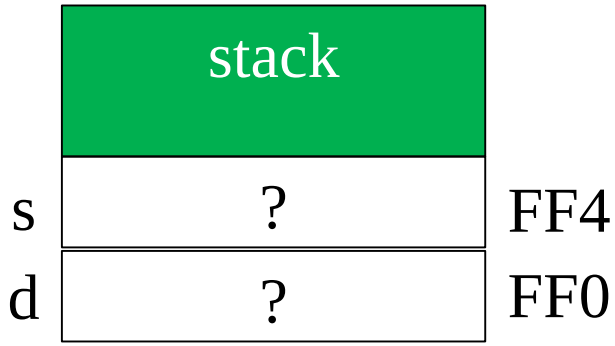
- Abbiamo visto le funzioni in C restituiscono normalmente un singolo valore.
  - E se voglio restituirne due ?
  - Ad esempio sto leggendo dei caratteri $x_o x_1 \cdots$ da input e vorrei restituire insieme sia quello con maggiori occorrenze che il numero delle occorrenze rilevate
  - Un modo di farlo 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_e_diff (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
*/
```

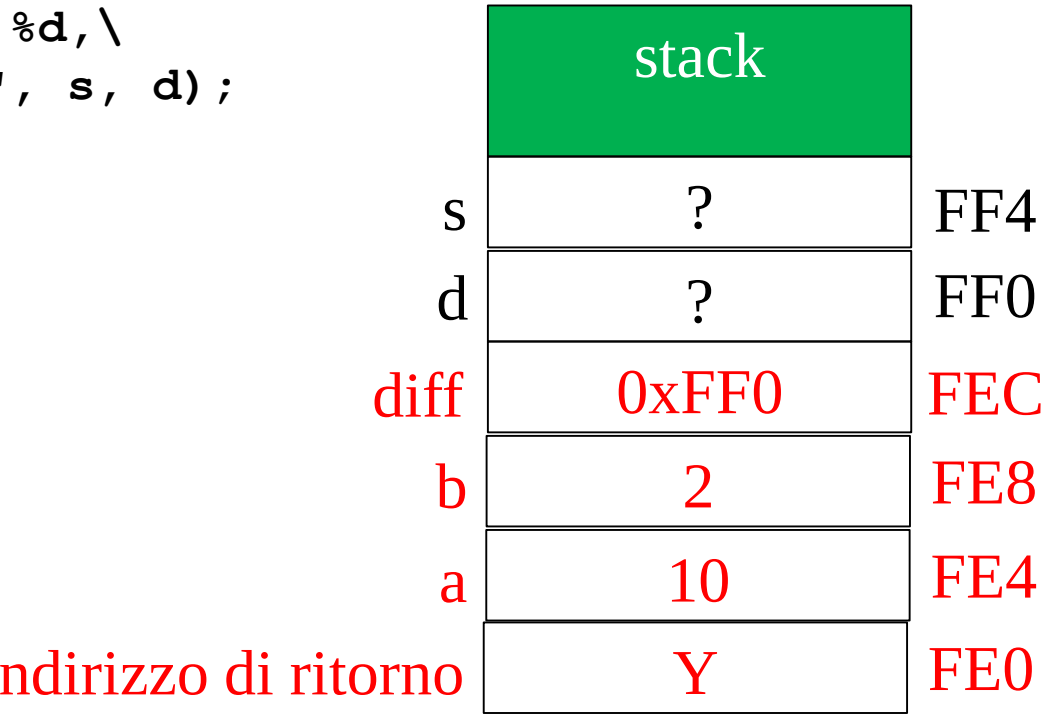
Esempio: somma e differenza

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



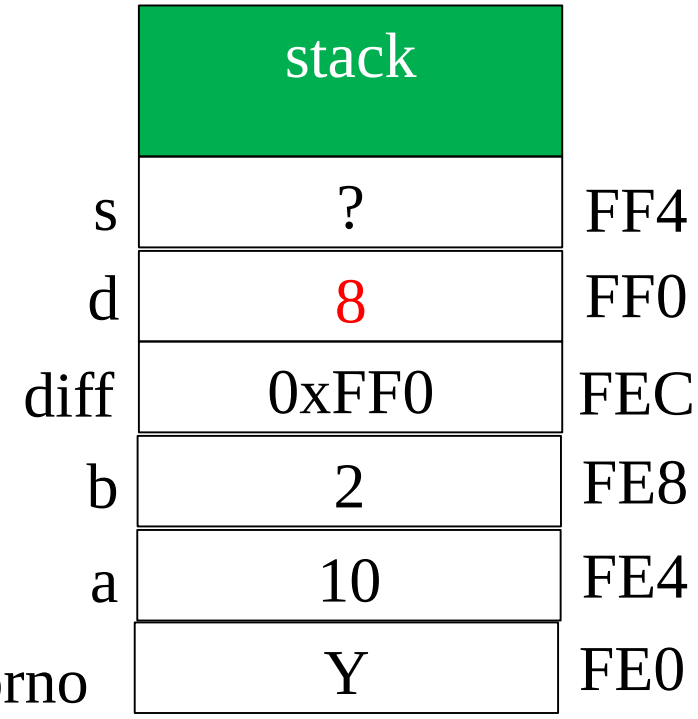
Esempio: somma e differenza

```
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_e_diff (10, 2, &d);
   printf("La somma è %d,\
   la differenza è %d", s, d);
   return 0;
}
```



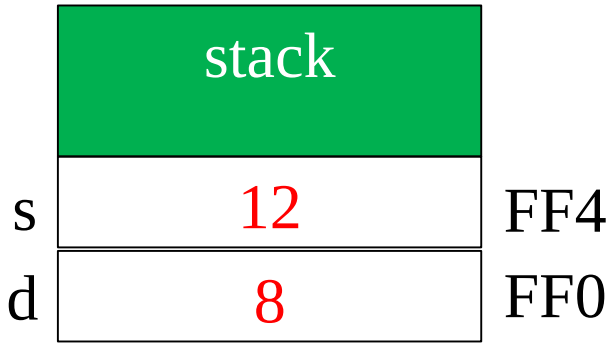
Esempio: somma e differenza

```
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_e_diff (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_e_diff (10, 2, &d);
   printf("La somma è %d,\
   la differenza è %d", s, d);
   return 0;
}
```

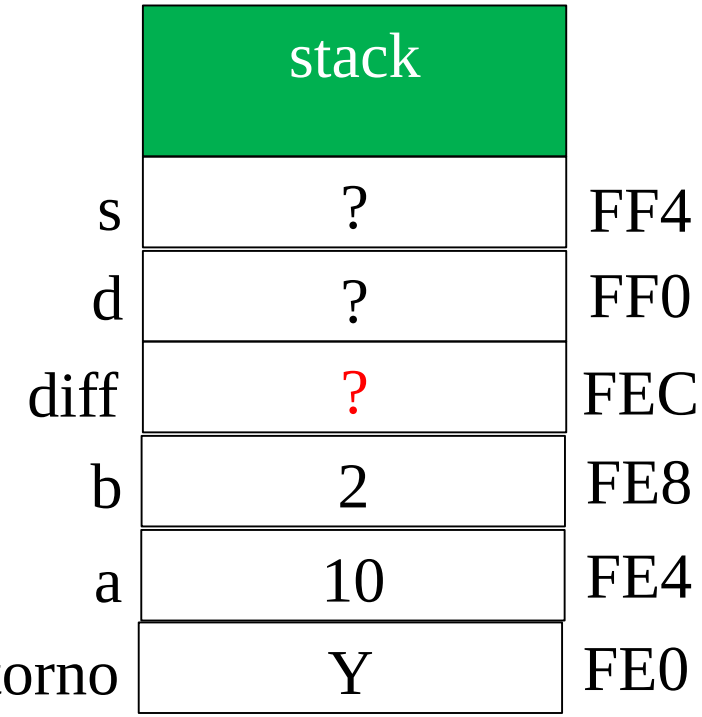


Esempio: scorretto .....

```
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 ?
*/
```

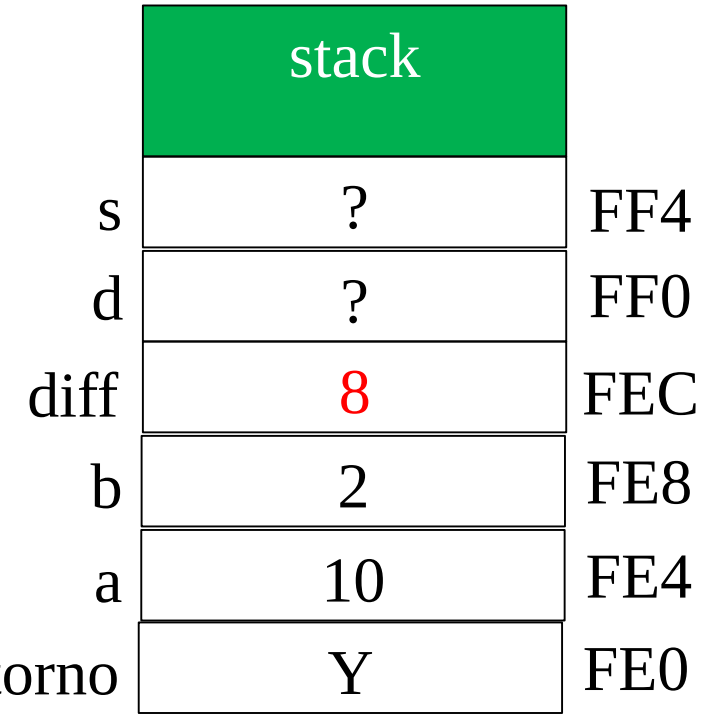
Esempio: scorretto

```
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_e_diff (10, 2, d);
    printf("La somma è %d,\
    la differenza è %d", s, d);
    return 0;
}
```



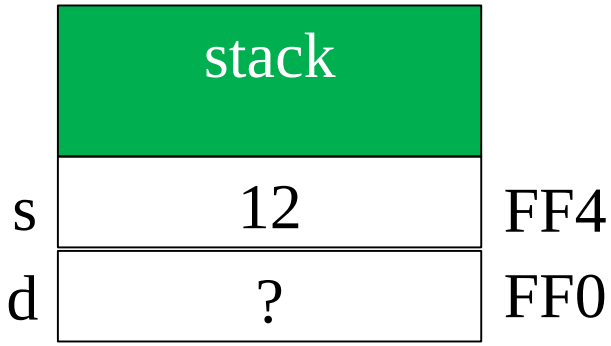
Esempio: scorretto

```
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_e_diff (10, 2, d);
    printf("La somma è %d,\
    la differenza è %d", s, d);
    return 0;
}
```



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_e_diff (10, 2, d);
    printf("La somma è %d,\
    la differenza è %d", s, d);
    return 0;
}
```



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

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 un 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
problema .... */

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
}

/* 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

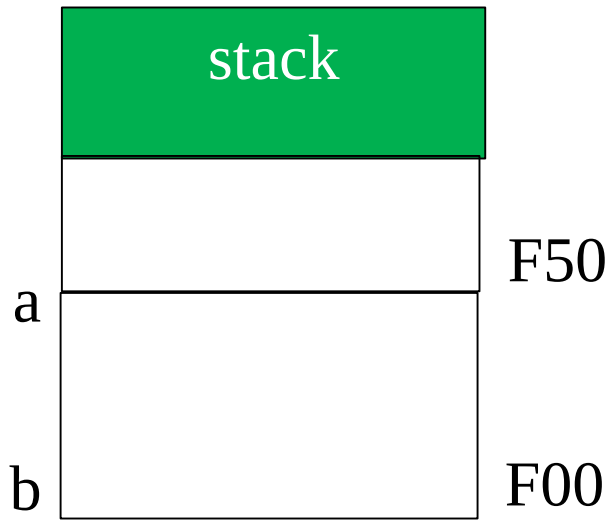
- È fondamentale capire che questo meccanismo di passaggio non crea una copia dell'array ma solo del suo indirizzo (si chiama infatti passaggio *per indirizzo*)
- Quindi se modifico l'array dentro la funzione tutte le modifiche vengono effettuate sull'unica copia dell'array presente nell'ambiente di 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 valoro letti sono in a e
b */
  ....
  return 0;
}
```

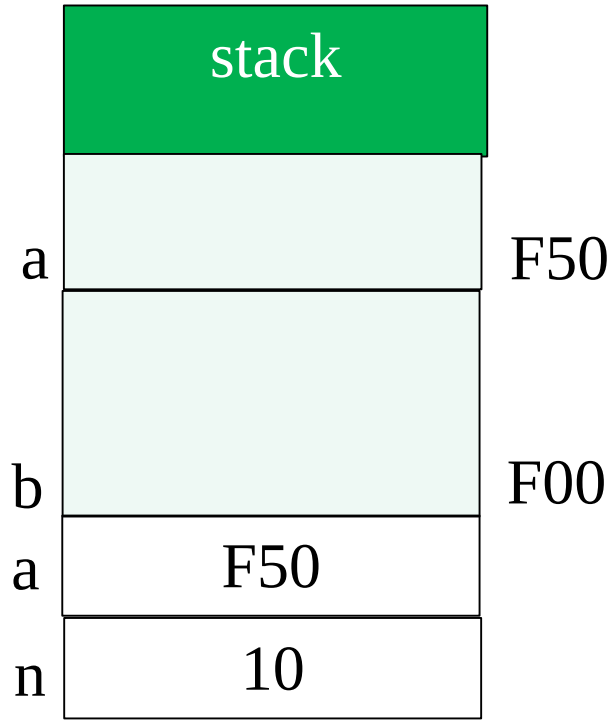
Array: lettura di una array da stdin

```
#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 ...*/
  ....
  return 0;
}
```



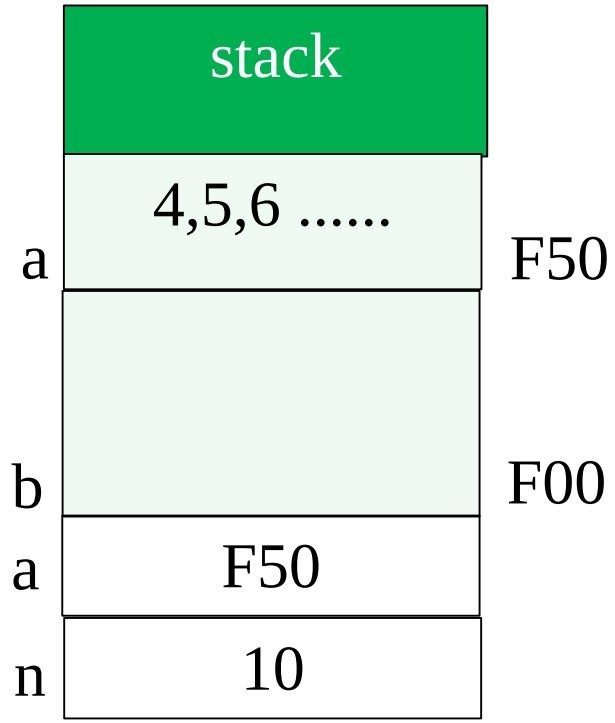
Array: lettura di una array da stdin

```
#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 ...*/
   ....
   return 0;
}
```



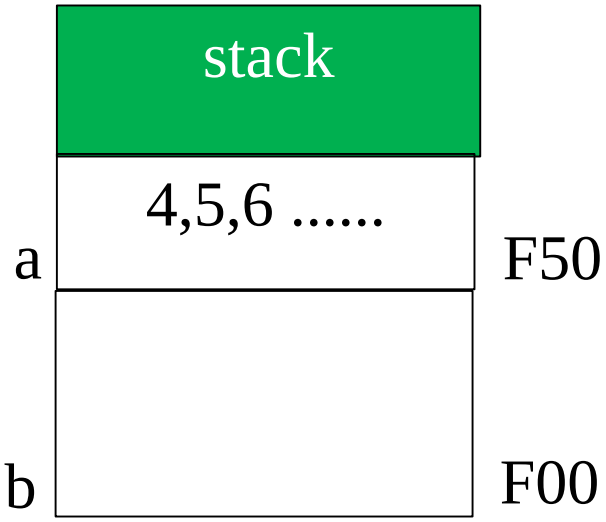
Array: lettura di una array da stdin

```
#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 ...*/
   ....
   return 0;
}
```



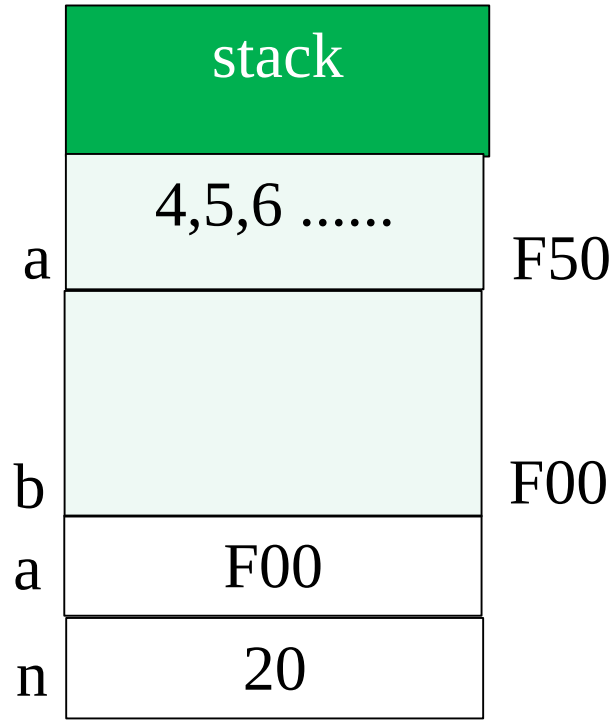
Array: lettura di una array da stdin

```
#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 ...*/
   ....
   return 0;
}
```



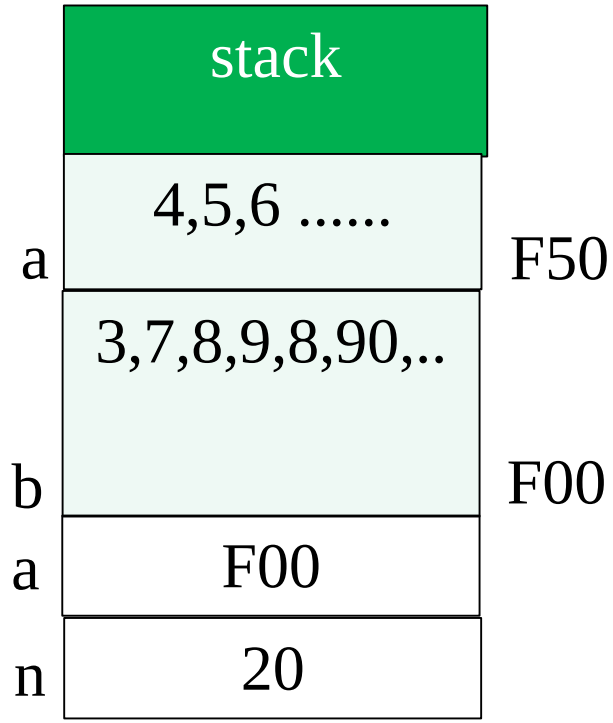
Array: lettura di una array da stdin

```
#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 ...*/
   ....
   return 0;
}
```



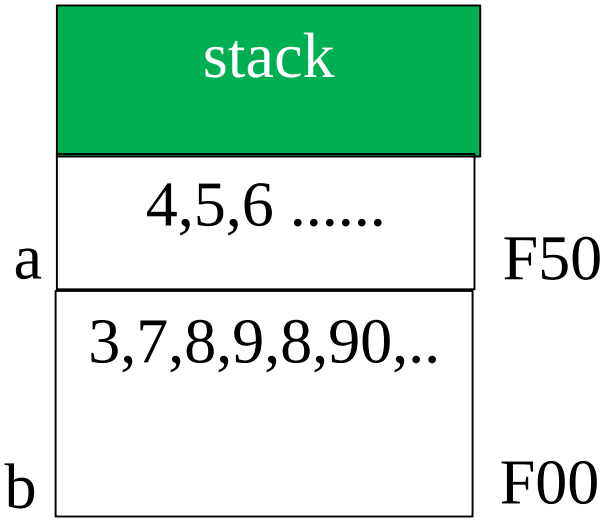
Array: lettura di una array da stdin

```
#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 ...*/
   ....
   return 0;
}
```



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 ...*/
   ....
   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 valoro letti sono in a e
b */
  ....
  return 0;
}
```

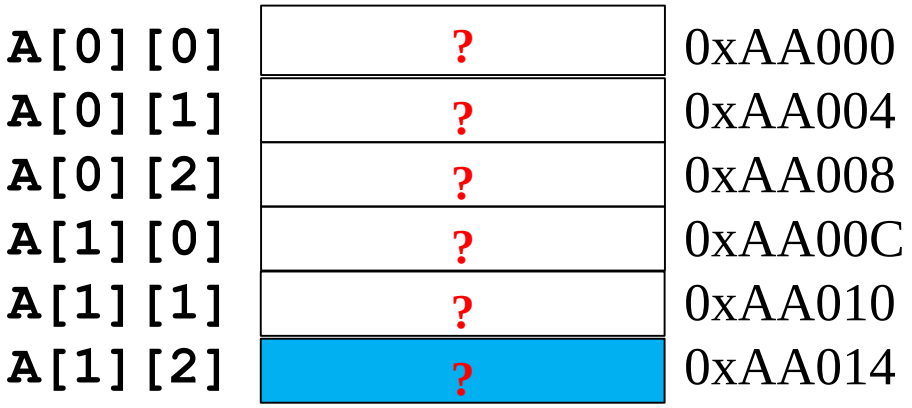
Scorretto ma il compilatore non lo rileva

Passare array bidimensionali

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

Esempio: due dimensioni ...

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

&A[1][2] = A + M * 1 + 2
```



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 più matrici diverse come parametro in diverse attivazioni!

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()

    .....