Inside C : Puntatori

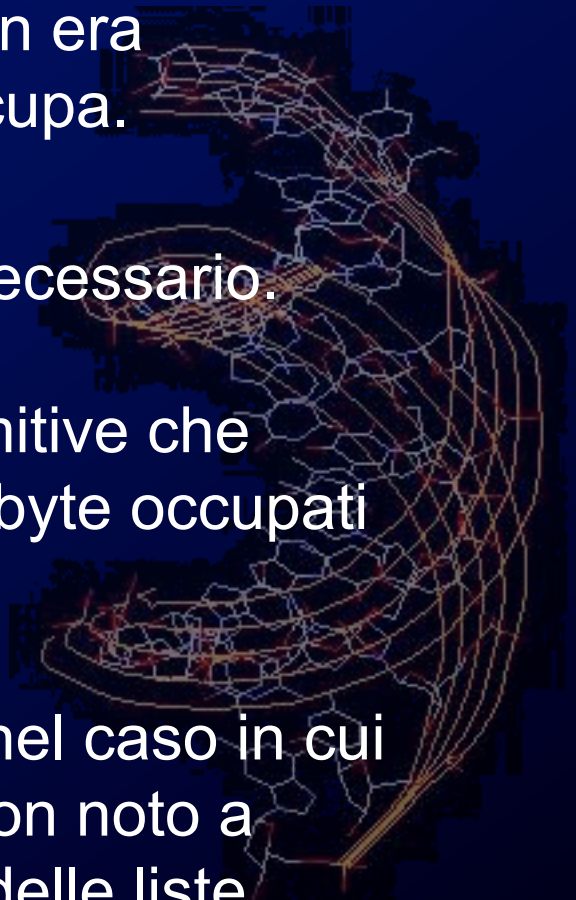
- Puntatori
- Referenziazione e Dereferenziazione
- Arrays

Nei programmi fatti fino a questo momento, non era necessario sapere quali celle una variabile occupa.

Esistono però delle situazioni in cui è invece necessario.

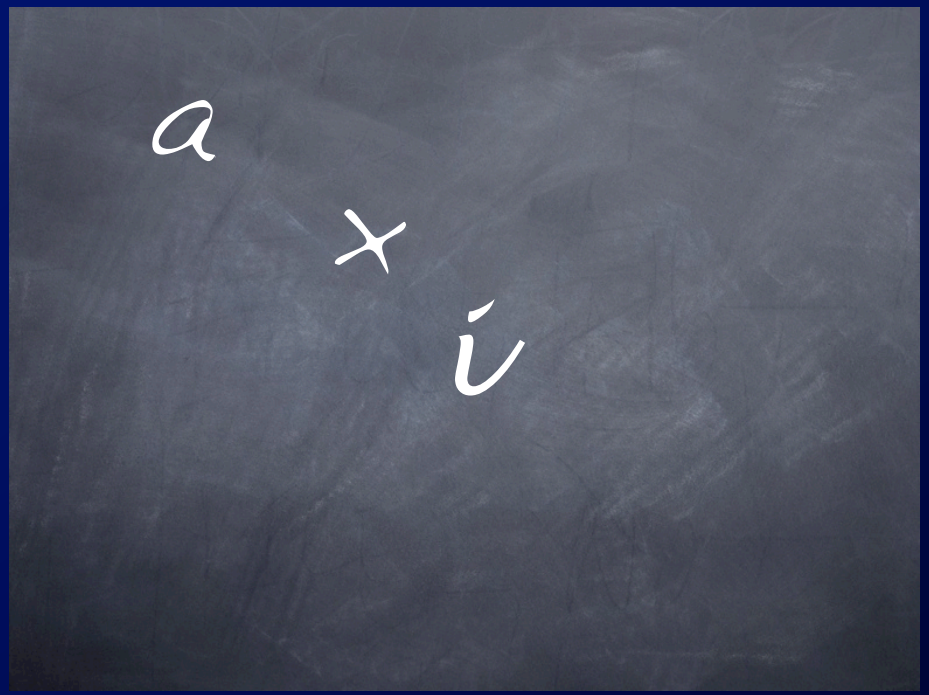
Per questo, il C mette a disposizione delle primitive che permettono di trovare l'indirizzo e il numero di byte occupati dalle variabili.

Queste nozioni sono necessarie per esempio nel caso in cui il programma deve gestire un numero di dati non noto a priori, come si vedrà a proposito degli array e delle liste.



La memoria

Per i programmi più semplici, si può pensare alla memoria come se fosse una lavagna: ogni volta che si dichiara una variabile, si disegna un quadrato; all'interno di questo quadrato si può scrivere (memorizzare) un valore



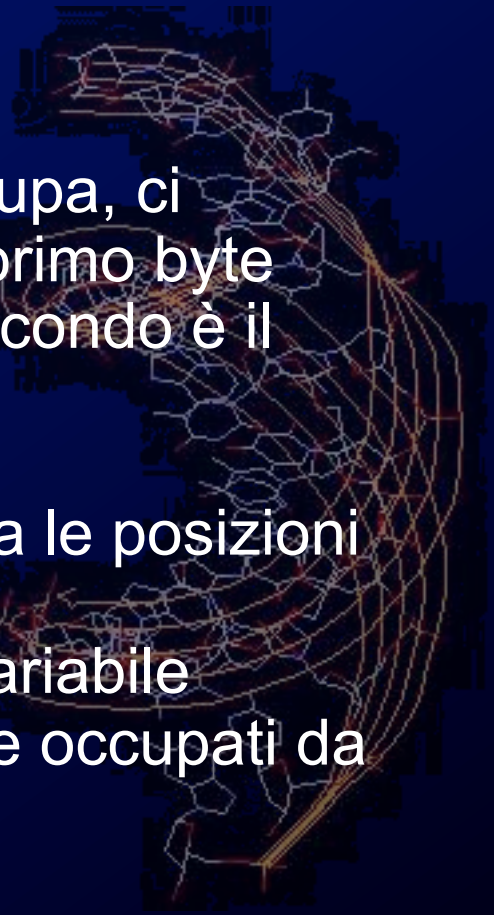
Le variabili C sono zone di memoria.

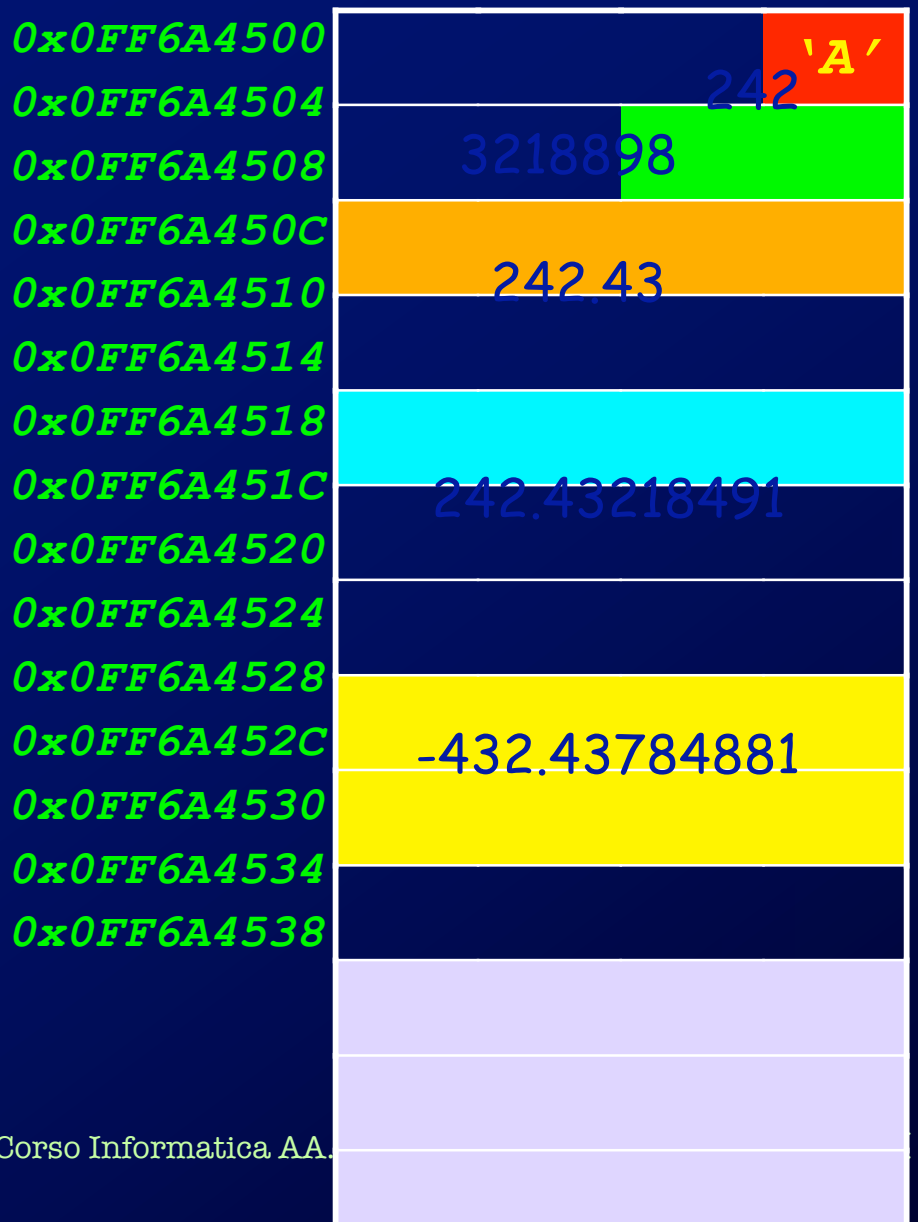
In altre parole, ogni variabile è un insieme di locazioni all'interno della memoria.

Il numero di byte occupati da una variabile dipende dal suo tipo

Se vogliamo sapere quali byte una variabile occupa, ci servono due numeri: il primo è la posizione del primo byte occupato, detto ***indirizzo*** della variabile, il secondo è il numero di byte occupati.

Se per esempio la variabile ***a*** occupa in memoria le posizioni dalla ***0x0FF6A45C*** alla ***0x0FF6A45F***,
allora il suo indirizzo è ***0x0FF6A45C*** e questa variabile occupa quattro locazioni, quindi il numero di byte occupati da a è 4.





La memoria può essere schematizzata come un array di locazioni a 32 bit (4 byte)

`char`	1 byte	(8 bits)
`short int`	2 bytes	(16 bits)
`long int`	4 bytes	(32 bits)
`float`	4 bytes	(32 bits)
`double`	8 bytes	(64 bits)
`long double`	16 bytes	(128 bits)

Le variabili finora incontrate sono caratterizzate da un nome (o identificativo), un tipo, ed occupano un'area di memoria di dimensione dipendente dal tipo.

Per accedere ad una variabile si usa il suo nome:

```
x = a;
```

Questa istruzione significa: "preleva il valore contenuto nella cella di memoria il cui nome e' *a* e scrivilo nella cella di nome *x*".

È importante notare la differenza tra il valore di una variabile e il suo indirizzo. L'indirizzo di una variabile è l'inizio della zona di memoria occupata da una variabile, mentre il valore di una variabile è il contenuto di tale zona.

Operatore `sizeof`

Il compilatore C rende disponibile un operatore, ***`sizeof()`*** , che restituisce come ***`int`*** il numero di byte occupato dal tipo di dato o dalla variabile indicati tra le parentesi.

Si noti che ***`sizeof()`*** non è una funzione, ma un operatore: esso è dunque intrinseco al compilatore e non fa parte di alcuna libreria.

```
int ciro, Tot = 0;
double peppe;
char enzo;

Tot = sizeof(ciro)+sizeof(peppe)+sizeof(enzo)+sizeof(ugo);
printf("ciro  occupa %d bytes\n", sizeof(ciro));
printf("peppe occupa %d bytes\n", sizeof(peppe));
printf("enzo  occupa %d bytes\n", sizeof(enzo));
printf("Totale: %d bytes\n", Tot);
```

Operatore `sizeof`

Il compilatore C rende disponibile un operatore, ***`sizeof()`*** , che restituisce come ***`int`*** il numero di byte occupato dal tipo di dato o dalla variabile indicati tra le parentesi.

Si noti che ***`sizeof()`*** non è una funzione, ma un operatore: esso è dunque intrinseco al compilatore e non fa parte di alcuna libreria.

```
ciro  occupa 4 bytes
peppe occupa 8 bytes
enzo  occupa 1 bytes
Totale: 17 bytes
```

```
int main(void) {
  int a;   char b;   float c;
  a=12;   b='a';  c=0.1243;
  printf("L'indirizzo di a e' %x, occupa %d bytes,
il suo valore e' %d\n", &a, sizeof(a), a);
  printf("L'indirizzo di b e' %x, occupa %d bytes,
il suo valore e' %c\n",&b, sizeof(b), b);
  printf("L'indirizzo di c e' %x, occupa %d bytes,
il suo valore e' %f\n",&c, sizeof(c), c);
  return 0;
}
```

l'indirizzo di una variabile, cosí come il numero di byte occupati, sono dei normali numeri, e si possono quindi

```
L'indirizzo di a è bffffa4c, occupa 4 bytes, il suo valore è 12
L'indirizzo di b è bffffa4b, occupa 1 bytes, il suo valore è a
L'indirizzo di c è bffffa44, occupa 4 bytes, il suo valore è 0.124300
```

variabile, mentre il valore di una variabile è il contenuto di tale zona.

Pointers - Puntatori

Pointers: Puntatori (II)

Nel linguaggio C e' possibile accedere alle variabili anche in modo indiretto attraverso i puntatori.

Il puntatore e' una variabile che contiene l'indirizzo della locazione di memoria di un'altra variabile.

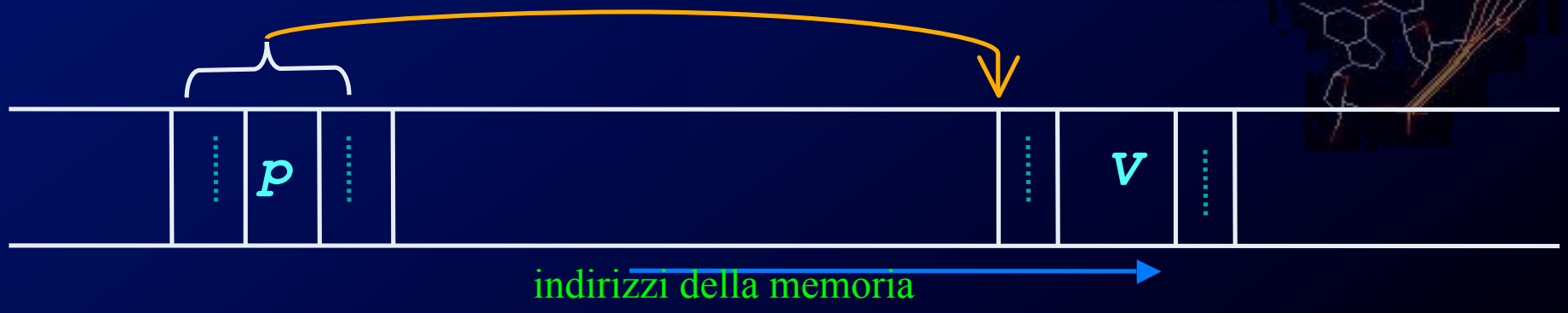
Il compilatore assegna automaticamente il numero di celle necessarie per rappresentare il tipo di dato.

Se **v** e' una variabile, e' possibile modificarne il contenuto se e' noto l'indirizzo della sua prima cella di memoria.

Indirizzo: operatore &

L'indirizzo della locazione di memoria della variabile v e' ottenuto attraverso l'operatore unario &:

In questa istruzione, l'indirizzo di **v** , **&v**, e' assegnato alla variabile **p**. Si dice che "p punta a v".



Valore ed indirizzo

- Tutte le variabili hanno un valore ed un indirizzo
- Ciò vale anche per i puntatori, solo che in questo caso il valore è un indirizzo di memoria a sua volta
- In pratica un puntatore rappresenta sempre una zona di memoria, solo che al suo interno invece di conservare un intero o un float conserva un indirizzo.

Contenuto: operatore *

Come si è visto, l'operatore & trova l'indirizzo di una variabile.

Esiste anche l'operatore inverso, che permette di accedere alla zona di memoria definita da un puntatore

Applicato ad una variabile di tipo puntatore restituisce il contenuto della memoria a cui il puntatore punta.

```
v = *p;
```

se p è una variabile di tipo puntatore a intero, si può pensare a *p come a una variabile di tipo intero.

In questa istruzione, e' assegnato a **v** il contenuto della memoria a cui **p** punta

in pratica la variabile **v** e la espressione ***p** sono esattamente equivalenti.

Contenuto: operatore *

```
#include <stdio.h>

int main(){
    int v, *p;
    v=3;
    p=&v;
    printf("v=%d *p=%d \tp=0x%x|%d \n"
                ,v,*p,p,p);
}
```

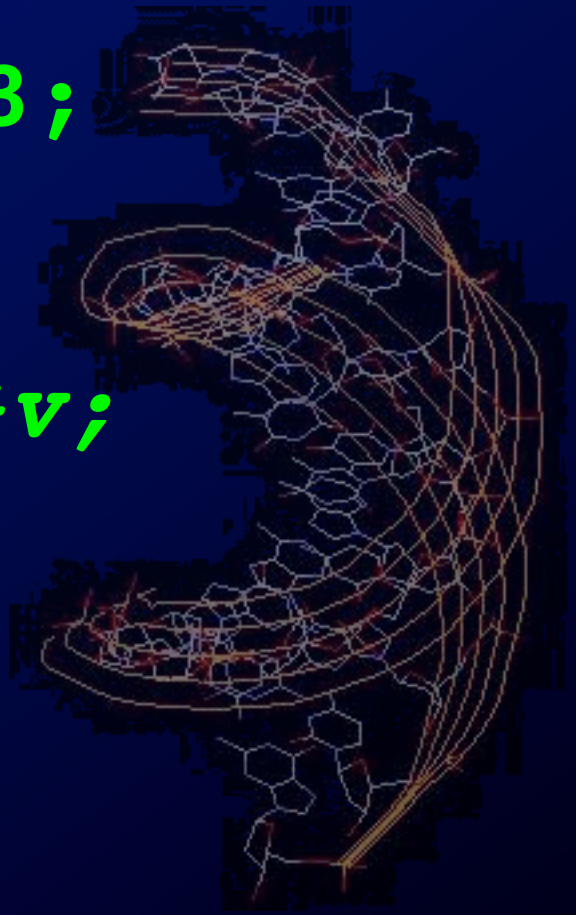
```
v=3 *p=3 p=0x0FF6450C|267797772
```

```
int v,*p;
```


0x0FF6A4500
0x0FF6A4504
0x0FF6A4508
0x0FF6A450C
0x0FF6A4510
0x0FF6A4514
0x0FF6A4518
0x0FF6A451C
0x0FF6A4520
0x0FF6A4524
0x0FF6A4528
0x0FF6A452C
0x0FF6A4530
0x0FF6A4534
0x0FF6A4538
3
0x0FF6A450C
v
p

```
v=3;
```

```
p=&v;
```



Riassumendo : operatori & e *

& operatore di referenza (the address of): applicato ad una variabile ritorna l'indirizzo della variabile.

***** operatore di dereferenza (the content of): applicato ad un puntatore di una variabile ritorna il valore della variabile.

I puntatori in C sono usati molto spesso poiche' consentono di scrivere sorgenti compatti ed efficienti e costituiscono l'unico modo per accedere a memoria allocata dinamicamente, ovvero durante l'esecuzione di un programma.

Tipo Puntatore

Gli indirizzi non sono esattamente dei numeri

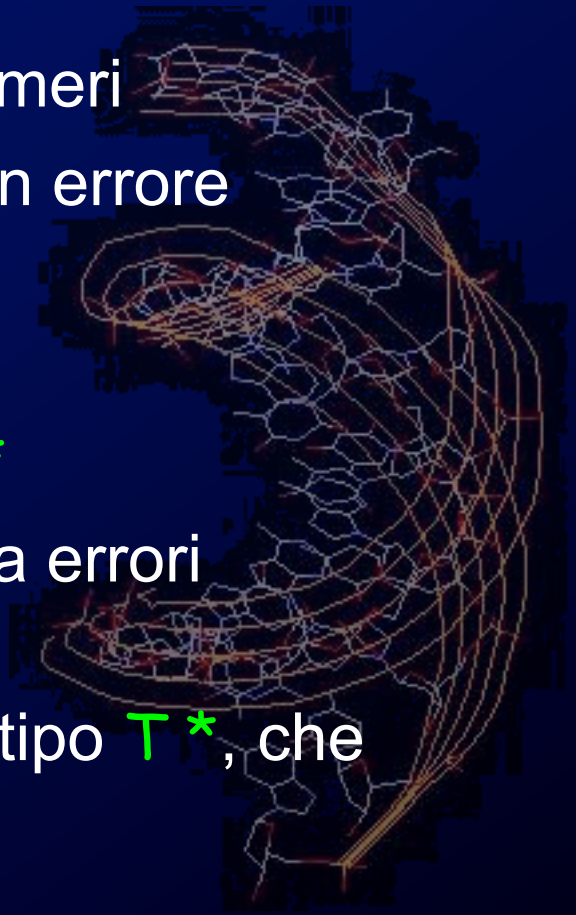
Per esempio, `int p, a;` e poi `p=&a;` genera un errore (warning)

Il tipo di un indirizzo non è `int`

L'indirizzo di una variabile `int` è di tipo `int *`

Quindi, `int *p; int a;` e poi `p=&a;` non genera errori

L'indirizzo di una variabile di un tipo `T` è di tipo `T *`, che viene detto tipo "`puntatore a T`".



Copia del valore o dell'indirizzo

Che differenza c'è tra

b=a

qui il valore in **a** viene copiato in **b** ma dopo la copia non ci sono più relazioni tra **a** e **b**

E' come fotocopiare un documento e poi fare delle modifiche solo sull'originale: chiaramente, la copia non viene modificata automaticamente

e

p=&a

qui il valore l'indirizzo di **a** viene copiato in **p**. Ogni modifica di **a** è accessibile tramite **p** anche successivamente

In pratica, ***p** e **a** sono realmente la stessa zona di memoria; quindi, da questo momento in poi, ogni modifica su **a** cambia anche ***p** e viceversa.

Variabili di tipo puntatore (I)

Ogni tipo di variabile occupa una quantita' di celle di memoria dipendente dal tipo: per poter risalire dal puntatore al valore di un tipo di variabile e' necessario che il puntatore venga definito in base al tipo.

```
main(){
        int a = 2;
        int *pa, y = 0;

        pa = &a;
        y = *pa;
        printf("address of a = 0x%X\n", pa);
        printf("value   of a = %d\n", y);
        *pa = 5;
        printf("new value   of a = %d\n", a);
      printf("old value   of a = %d\n", y);
}
```

```
address of a = 0xBFFFF9D4
value   of a = 2
new value   of a = 5
old value   of a = 2
```

Variabili di tipo puntatore (II)

I puntatori sono dichiarati tramite l'operatore unario *, che in questo caso viene chiamato "costruttore di tipo" (ossia del tipo variabile puntatore).

Quando si dichiarano più variabili di tipo puntatore * va ripetuto per tutte

Esempi:

```
int *pa, i, k, n;

float *x, *y, *z, dist;
```

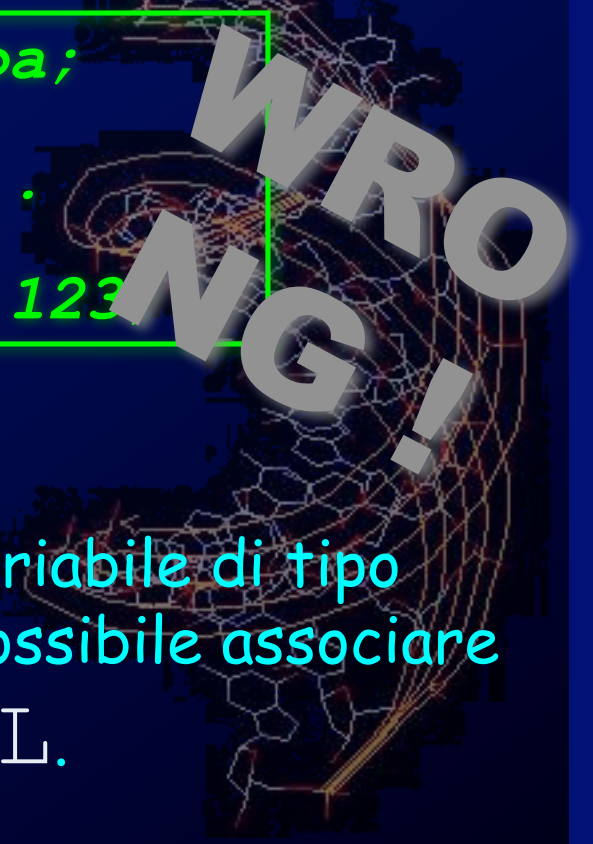
Attenzione: quando un puntatore viene dichiarato, essendo esso stesso una variabile, il suo contenuto non e' precisato: ovvero l'indirizzo a cui punta non e' un indirizzo valido!

Inizializzazione dei puntatori

Prima di dereferenziare un puntatore e' necessario inizializzarlo:

Perche'?

```
int *pa;

. . . .

*pa = 123;
```




```
int *pa, i;

. . . . .

pa = &i;  *pa =123;
```

A qualunque variabile di tipo puntatore e' possibile associare il valore NULL.

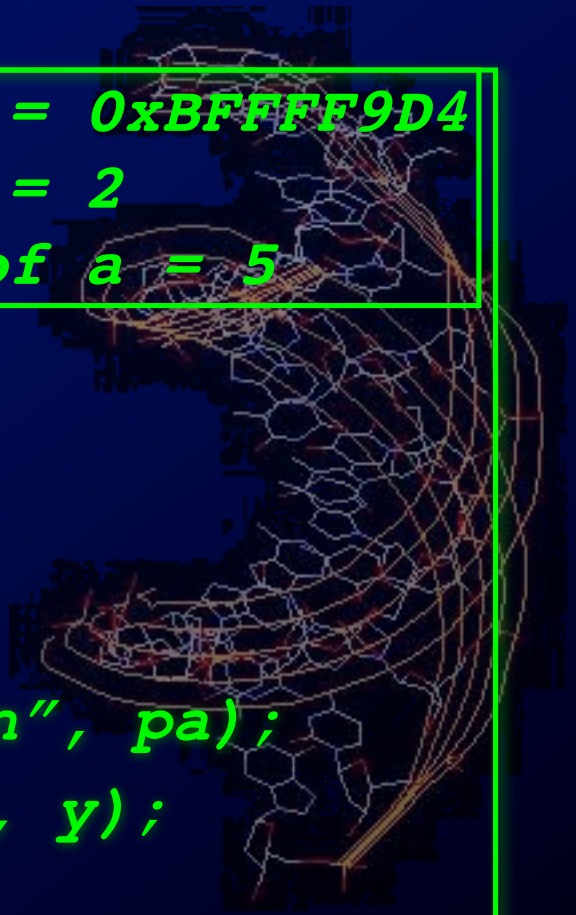
```
int *pa = NULL;

float *x = *y = NULL;
```

BUONA ABITUDINE!

Esempio (I)

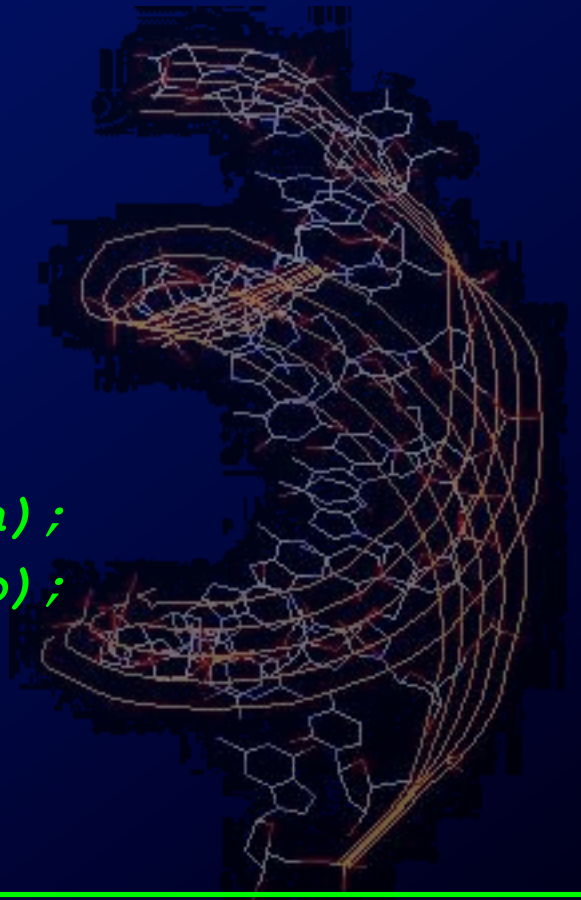
```
main(){
      int a = 2;
      int *pa = NULL
          , y = 0;

      pa = &a;
      y = *pa;
      printf("address of a = 0x%X\n", pa);
      printf("value   of a = %d\n", y);
      *pa = 5;
      printf("value   of a = %d\n", a);
}
```

```
address of a = 0xBFFFF9D4
value   of a = 2
new value   of a = 5
```



Esempio (II)

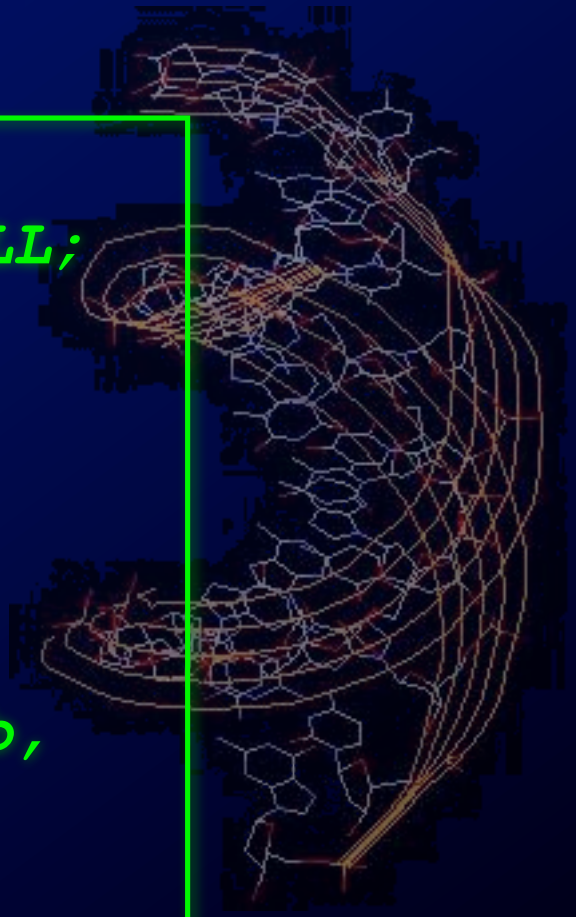
```
main(){
   int a = 3, b = 0, *pa = *pb = NULL;
   pa = &a;
   b = *pa;  /* b = 3 */
   pb = pa;
   pa = &b;
   printf("&a = 0x%X pa = 0x%X\n", &a, pa);
   printf("&b = 0x%X pb = 0x%X \n",&b, pb);
   printf("a = %d b = %d\n", a, b);
   *pa = 5;  /* b = 5 */
   *pb = 10; /* a = 10 */
   printf("a = %d b = %d\n", a, b);
}
```



```
&a = 0xBFFFF9D4 pa = 0xBFFFF9D0
&b = 0xBFFFF9D0 pb = 0xBFFFF9D4
a = 3 b = 3
a = 10 b = 5
```

Esempio (III)

```
main(){
      int a = 3, b = c = 0, *p = NULL;
      b = 2 * (a + 5);
      p = &a;
      c = 2 * (*p + 5);
      printf("\n a = %d", a);
      printf("\n b = %d c = %d\n", b,
c);
}
```



```
a = 3
b = 16 c = 16
```

Puntatori generici: puntatori void

E' possibile dichiarare una variabile di tipo puntatore senza specificarne il tipo:

```
void *pa, *pb;
```

Questa dichiarazione significa che il tipo di dato a cui il puntatore punta e' lasciato indeterminato. Ad un puntatore di tipo `void` può essere assegnato l'indirizzo di qualsiasi tipo di dato.

```
int *pa, i;
float dist;
void *p_void;
. . . . .
p_void = &i;
pa = (int *) p_void;
p_void = &dist;
```

Conversione di tipo: `cast`

La conversione da un tipo di variabile ad un altro (quindi anche da un tipo puntatore ad un altro) puo' essere "forzata" in qualunque espressione mediante l'operatore unario di ***`cast`***. Nel costrutto

`(nome tipo) espressione`

l'espressione e' convertita nel tipo di dato definito tra le parentesi.

```
int i, n;
float f;
. . . . .
n = (i+f)%4; /* errato */
n = ((int)(i+f))%4;   /* corretto */
f = sqrt((double) n); /* corretto */
```

Notate che il tipo di ***`n`*** non e' alterato: il cast produce il valore di ***`n`*** nel tipo adatto alla funzione ***`sqrt`***. Lo stesso vale per ***`(i+f)`***.

Puntatori di tipo `const`

Un puntatore puo' anche essere definito con l'attributo ***`const`***:

```
const int *pt;
```

Questa proprieta' indica che il puntatore NON puo' modificare il contenuto della variabile a cui punta.

```
const int *pa;
int i;
. . . . .
pa = &i;
*pa = 123; /* errato */
```

Come si fa?

Questa proprieta' e' molto utile quando si usano i puntatori come argomenti di funzioni.

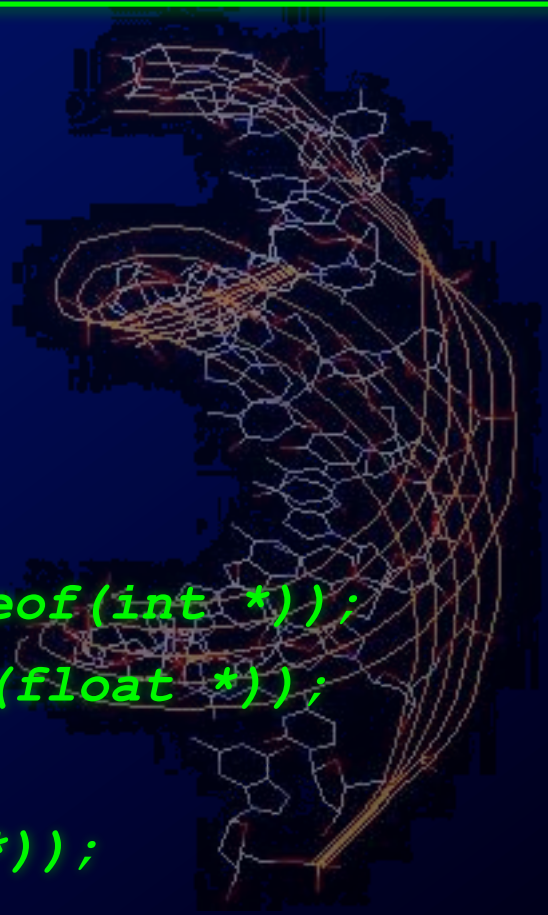
Warnings

- quando si usano i puntatori e' molto facile fare confusione fra puntatori ed oggetti puntati;
- un puntatore e' una variabile che contiene un indirizzo di memoria, mentre l'oggetto puntato e' la zona di memoria che inizia con questo indirizzo ed e' grande abbastanza da contenere l'oggetto;
- un puntatore e' una variabile, e occupa sempre lo spazio di memoria necessario a contenere l'indirizzo del dato puntato, e non il tipo di dato.

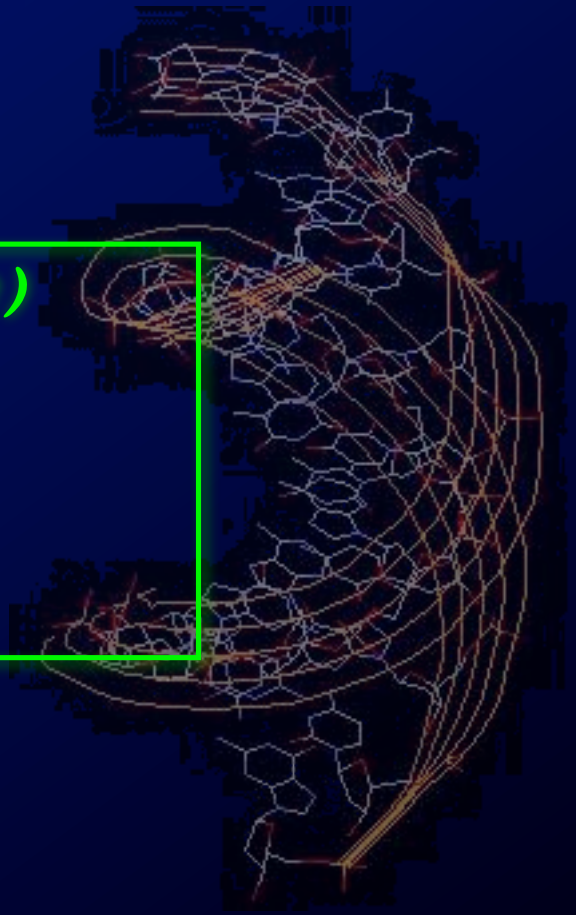
A verifica di quest'ultima osservazione studiamo il seguente programma:

Get the size: `sizeof`

```
main(){
int i, *pi;
float f, *pf;
double d, *pd;
struct Point{double x,y;} t, *pt;

printf("\n Tipo   Dim   *Dim   byte\n");
printf("int    : %2d %2d\n", sizeof(int), sizeof(int *));
printf("float  : %2d %2d\n", sizeof(f),sizeof(float *));
printf("double: %2d %2d\n", sizeof(double),
                            sizeof(double *));
printf("Point  : %2d %2d\n", sizeof(t),
                          sizeof(struct Point *));
}
```



Get the size: `sizeof`



```
Tipo    Dim          *Dim (byte)
int    :   4            4
float  :   4            4
double:    8            4
Point  :  16            4
```

Array

Array (I)

Un array e' un insieme contiguo di celle di memoria tutte contenenti dati dello stesso tipo. Queste celle, che costituiscono in pratica un gruppo di variabili, hanno tutte lo stesso nome ed sono accessibili attraverso un indice.

```
int a[10], b[100];
float f[30];
char c[200];
struct Coord A[10];
```

La dichiarazione di un array (statico) è analoga a quella di una variabile, ad eccezione del fatto che il nome dell'array è seguito dal numero di elementi che lo compongono, racchiuso tra parentesi quadre.

Array (II)

In C sono possibili due categorie di array:

Array Statici: sono quelli la cui dimensione viene definita all'interno del programma nella dichiarazione.

```
int a[10], mat[10][20];
float f[30];
struct Coord A[10];
```

Array Dinamici: sono quelli la cui dimensione viene determinata, ed eventualmente anche modificata, solo durante l'esecuzione del programma (in run-time).

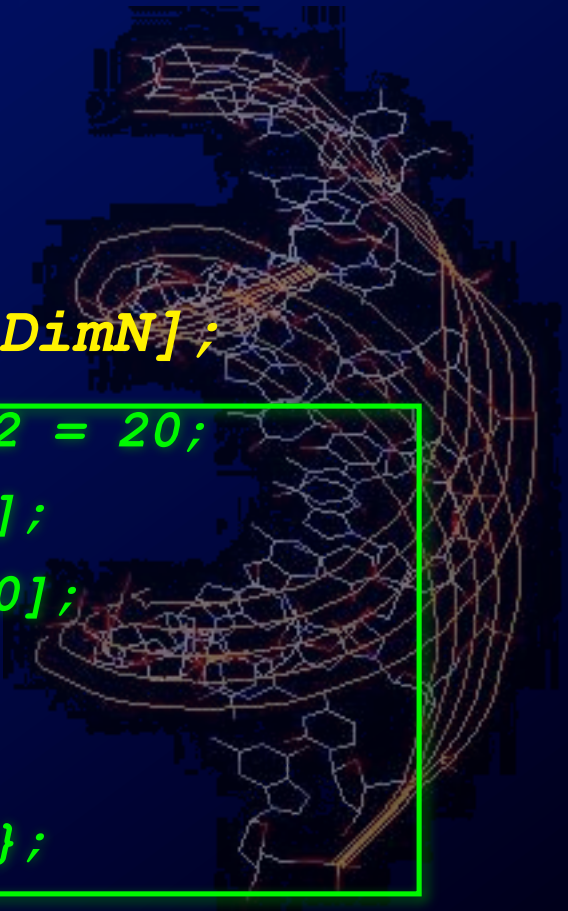
Array Statici (I)

Dichiarazione di array statici:

```
const int Dim, Dim1,. . DimN;
TipoDato nome_array[Dim][Dim1]..[DimN];
```

Esempi di dichiarazioni

```
const int Dim1 = 5, Dim2 = 20;
int a[Dim1], mat[10][20];
unsigned int Coeff[3][10];
float f[Dim2];
struct Coord A[10];
int B[] = {10, 20, 2, 8};
```



Gli array unidimensionali sono anche detti vettori; gli array bidimensionali sono detti matrici.

Array Statici (II)

Reminder: se int `a[10]` e' un array, il primo elemento e' `a[0]` e l'ultimo e' `a[9]`.

Cosa accade se si tenta di referenziare un elemento che non fa parte dell'array, ad esempio `a[10]` ?

Il compilatore non puo' rivelare un tale errore: la memoria relativa ad a[10] può essere sempre letta e scritta, ma non fa parte dell'array!!
In caso di lettura, il valore letto non ha alcun significato logico ai fini del programma, mentre in caso di scrittura si rischia di sovrascrivere qualche altro dato importante che risiede nella locazione di memoria subito dopo le locazioni assegnate all'array. Per il compilatore, a[10] è semplicemente la word che si trova a 320 byte dall'indirizzo al quale l'array è memorizzato.

Se la locazione di memoria referenziata non appartiene al programma, il programma stesso termina con un errore (segmentation fault).

Array Statici e Puntatori (I)

Tutti gli elementi di un array, come qualsiasi altro oggetto in memoria, hanno un indirizzo scelto dal compilatore.

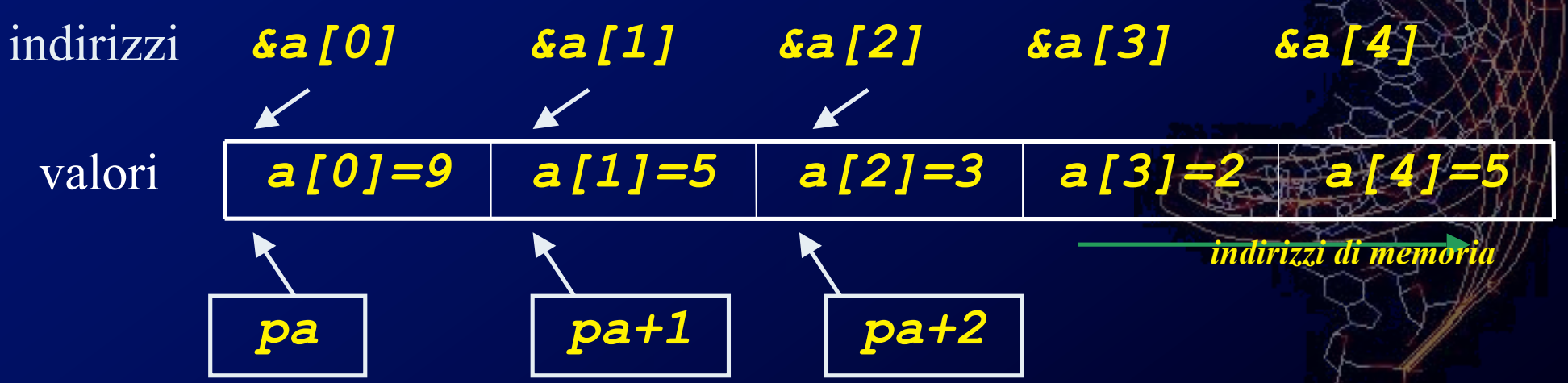
Il programmatore non può modificarlo, ma può conoscerlo ?

In C, il nome di un array e' una variabile che contiene l'indirizzo del primo elemento dell'array stesso, ovvero e' un puntatore alla prima locazione di memoria assegnata all'array.

```
int *pa, a[5];

. . . . .

pa = a; /* corretto */

pa = &a; /* equivalente */
```

Array Statici e Puntatori (II)

```
int *pa, a[]= {9, 5, 3, 2, 5};

. . . . .

pa = a;  /* corretto */

pa = &a; /* equivalente */
```

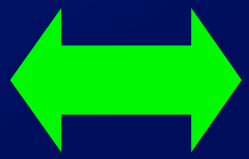


Se ***pa*** punta al primo elemento dell'array, ***pa+1*** punta a quello successivo: ***pa+n*** punta all'n-esimo elemento dopo il primo, indipendentemente dal tipo o dimensione delle variabili nell'array.

Array Statici e Puntatori (III)

Dereferenziando pa:

***pa** ha valore a[0]*	*a ha valore a[0]
***(pa+1) ha valore a[1]*	*(a+1) ha valore a[1]
. . . . .	. . . . .
***(pa+n) ha valore a[n]*	*(a+n) ha valore a[n]



Equivalentemente, l'indice di un elemento di un array ne esprime l'*offset* (in termini di numero di elementi) dal primo elemento dell'array stesso: il primo elemento di un array ha offset 0 rispetto a se stesso; il secondo ha offset 1 rispetto al primo; il terzo ha offset 2, cioè dista 2 elementi dal primo...
Il compilatore "ragiona" sugli arrays in termini di elementi, e non di byte.

Array Statici e Puntatori (IV)

Attenzione: rimane comunque una differenza sostanziale tra nome dell'array e puntatore: il valore di un puntatore puo' variare, il valore del puntatore dichiarato come array non puo' variare.

```
int *pa,a[5];

. . . . .

pa = a;

pa++; /* corretto */

x = a[3]; /*corretto */

x = *(pa+2);/*corretto*/
```

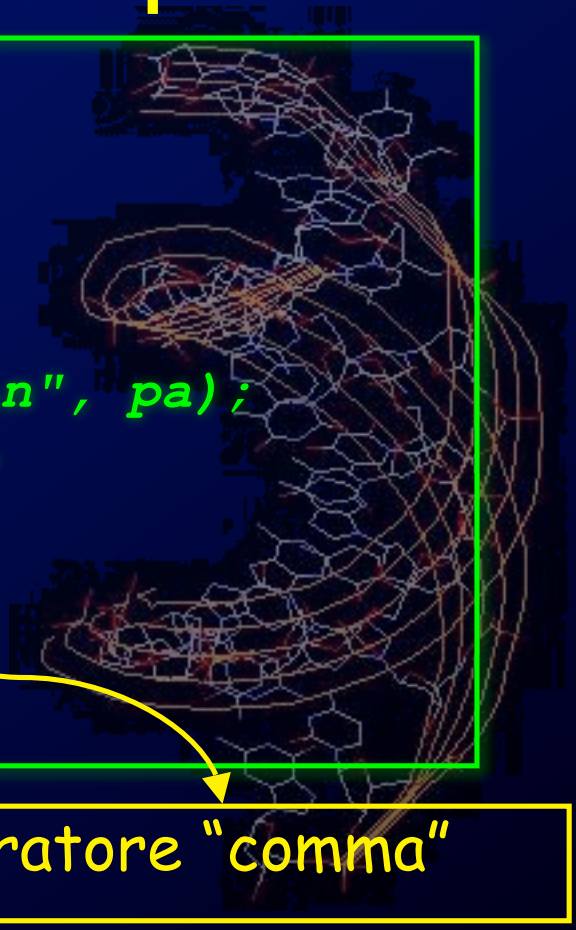
OK !

```
int *pa,a[5], b[10];

. . . . .

pa = b;

a = pa; /* l'indirizzo di un
array statico non puo' essere
modificato: errato */

a++;  /*errato */
```

Wrong!

Array & Puntatori: Esempio

```
main(){
    int i, *pa, a[]= {8,12,56,3,9};

    printf("indirizzo di a: 0x%X\n", a);
    printf("pa punta all'indirizzo: 0x%X\n", pa);
    for(i = 0, pa = a; i < 5; i++, pa++){
        printf("a[%d]= %d\n", i, *pa);
    }
}
```



operatore "comma"

Osservazione: l'accesso agli elementi di un array mediante puntatori produce un eseguibile che e' in generale piu' veloce.