Corso di Informatica A.A. 2009-2010

Lezione 10

Inside C : Puntatori

- Puntatori
- Referenziazione e Dereferenziazione

Pointers: Puntatori

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;
```

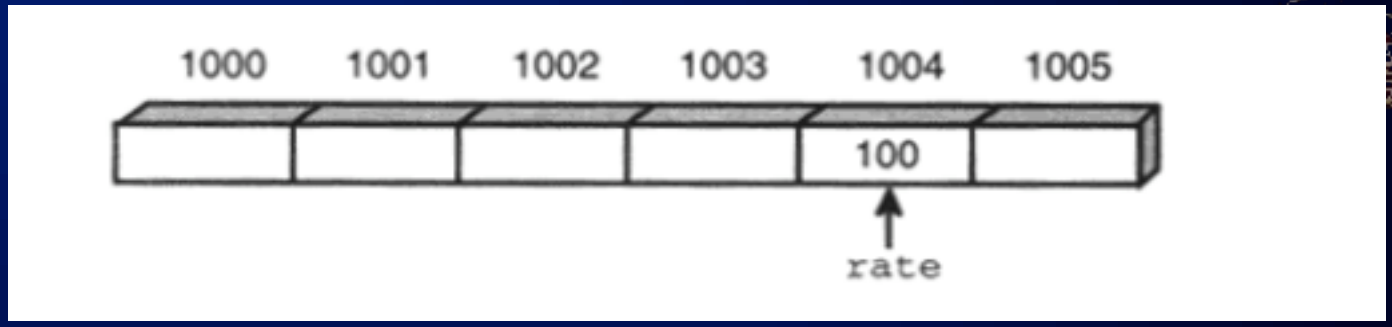
Questa istruzione significa: "preleva il valore contenuto nella cella di memoria il cui nome e' *a* e scrivilo nella cella di nome *x*".

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.

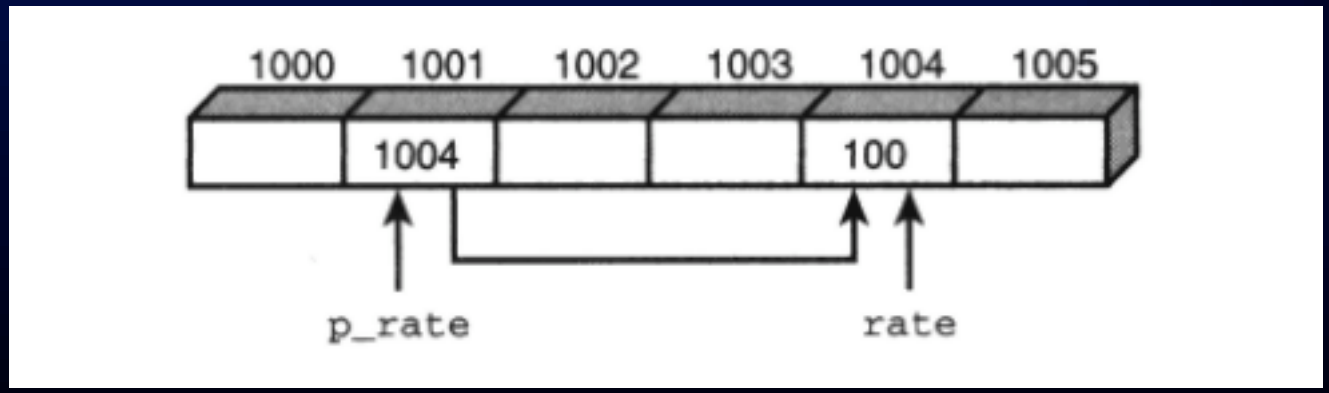
Pointers: Puntatori

Ipotizziamo di avere dichiarato e inizializzato a 100 una variabile di nome `rate`



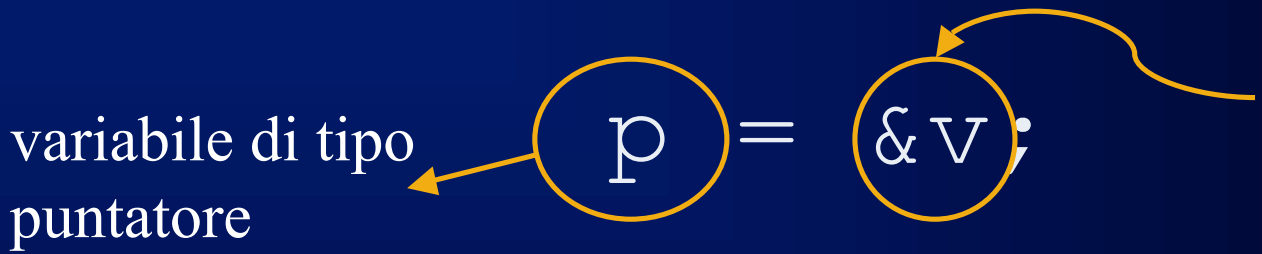
Vogliamo creare una variabile che contenga l'indirizzo di `rate`

Chiamiamo questa variabile `p_rate`



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**".

Contenuto: operatore *

Questo operatore esegue l'operazione inversa dell'operatore &: applicato ad una variabile di tipo puntatore restituisce il contenuto della memoria a cui il puntatore punta.

```
v = *p;
```

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

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.

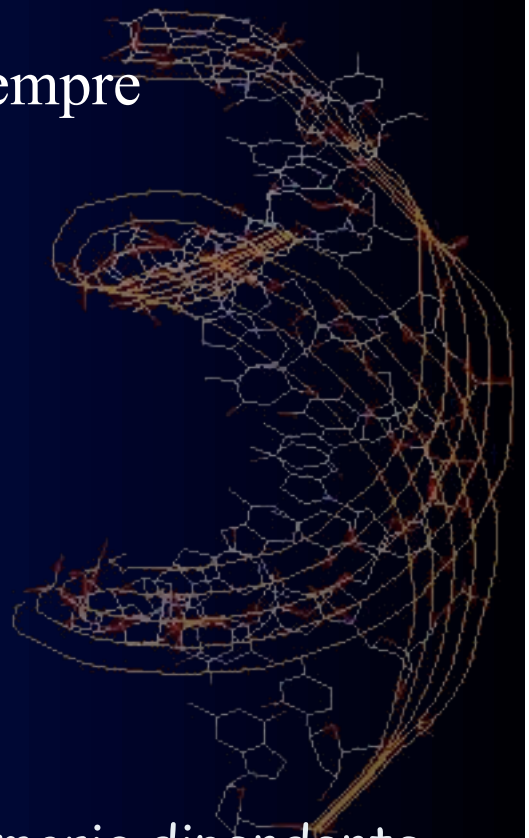
Puntatori

I puntatori sono variabili numeriche e vanno quindi sempre dichiarati prima del loro utilizzo

```
typename *ptrname;
```

Esempi

```
char *ch1, *ch2;
float *value, percent;
```



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.

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*).

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

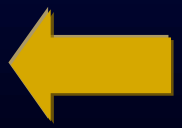
Prima di dereferenziare un puntatore e' necessario inizializzarlo:

```
int *pa, i;

. . . . .

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


```
int *pa;

. . . .

*pa = 123;
```

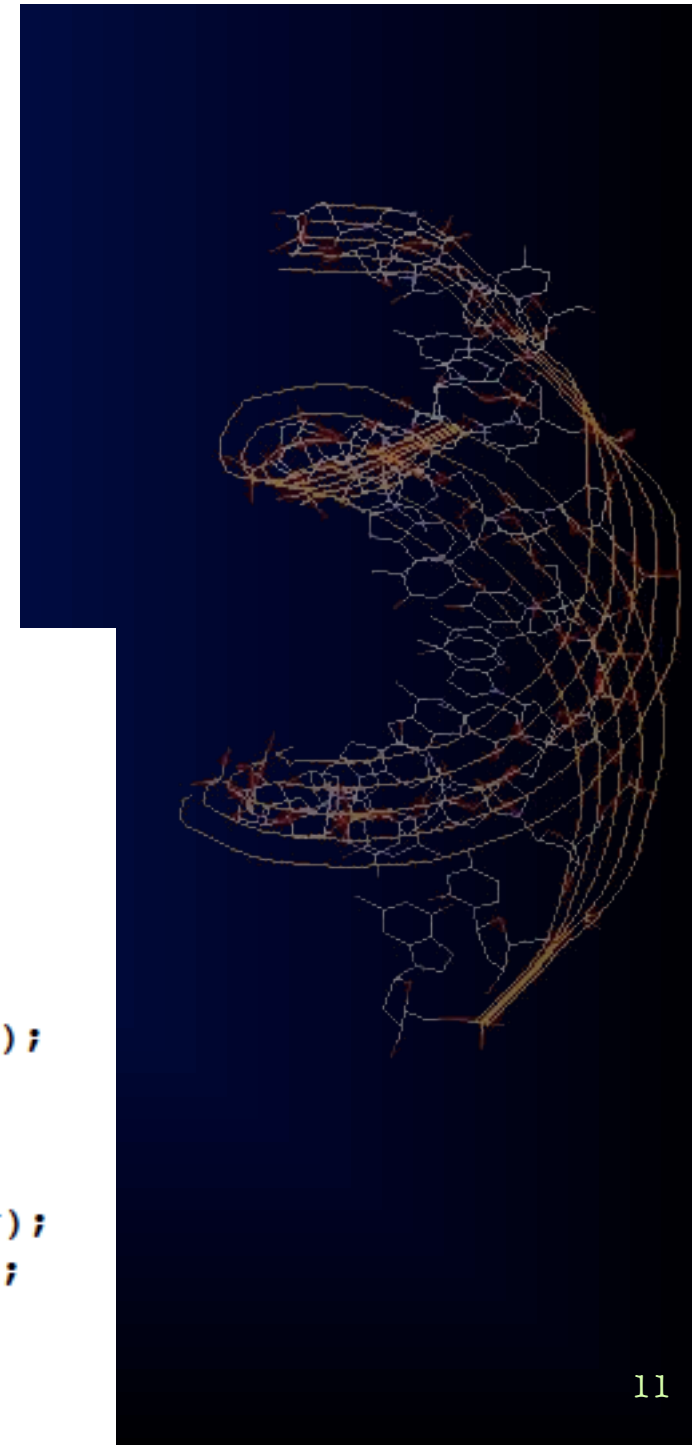
A qualunque variabile di tipo puntatore e' possibile associare il valore NULL.

```
int *pa = NULL;

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



BUONA ABITUDINE!

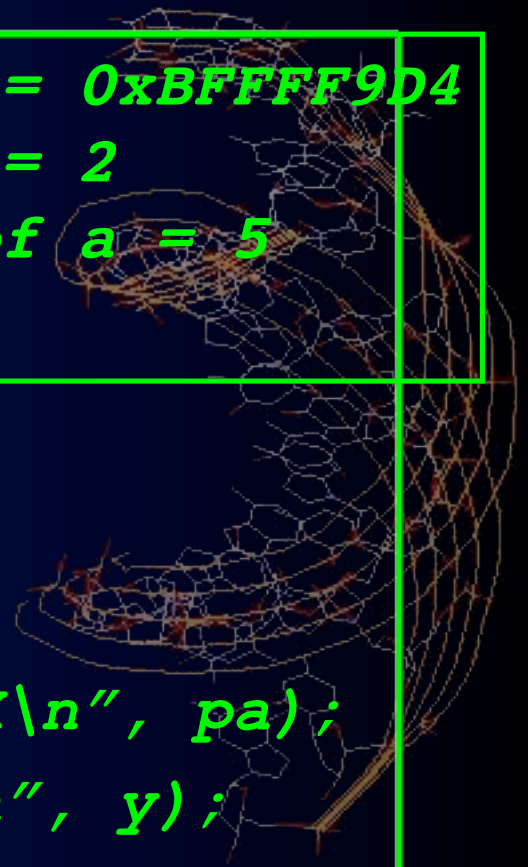
```c
/* Demonstrates basic pointer use. */

#include <stdio.h>

/* Declare and initialize an int variable */

int var = 1;

/* Declare a pointer to int */

int *ptr;

int main()
{
    /* Initialize ptr to point to var */

    ptr = &var;

    /* Access var directly and indirectly */

    printf("\nDirect access, var = %d", var);
    printf("\nIndirect access, var = %d", *ptr);

    /* Display the address of var two ways */

    printf("\n\nThe address of var = %d", &var);
    printf("\nThe address of var = %d\n", ptr);

    return 0;
}
```



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);
}
```

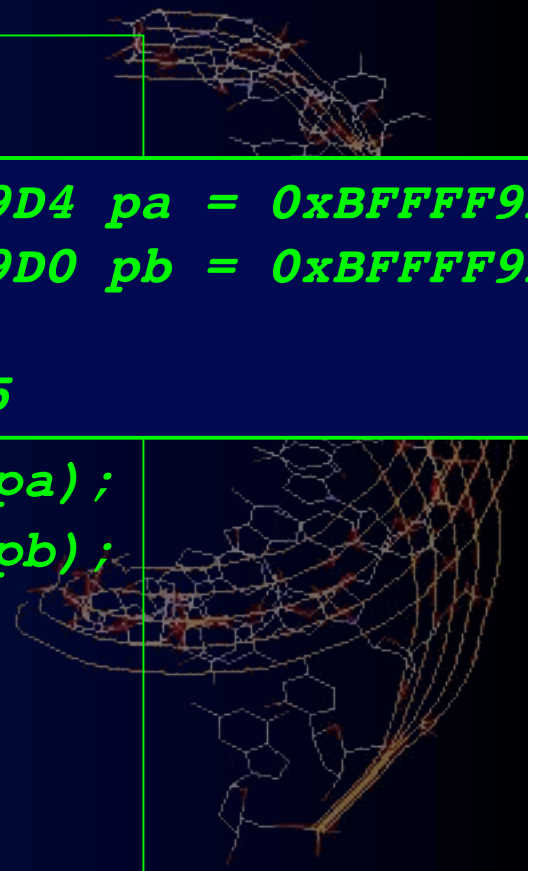
```
address of a = 0xBFFFF9D4
value   of a = 2
new value   of a = 5
```



Esempio (II)

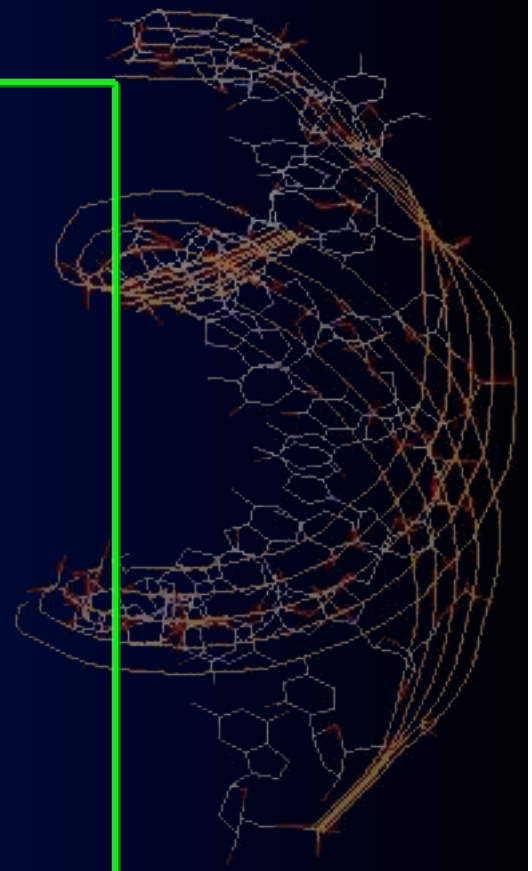
```
main(){
   int a = 3, b = 0, *pa = *pb = NULL;
   pa = &a;
   b = *pa;   /* b = 3 */
   pb = pa;
   pa = &b;
   printf("&a = 0x%X pa = 0x%X\n", &a, pa);
   printf("&b = 0x%X pb = 0x%X \n",&b, pb);
   printf("a = %d b = %d\n", a, b);
   *pa = 5;   /* b = 5 */
   *pb = 10; /* a = 10 */
   printf("a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

```
a = 0xBFFFF9D4 pa = 0xBFFFF9
b = 0xBFFFF9D0 pb = 0xBFFFF9
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);
}
```

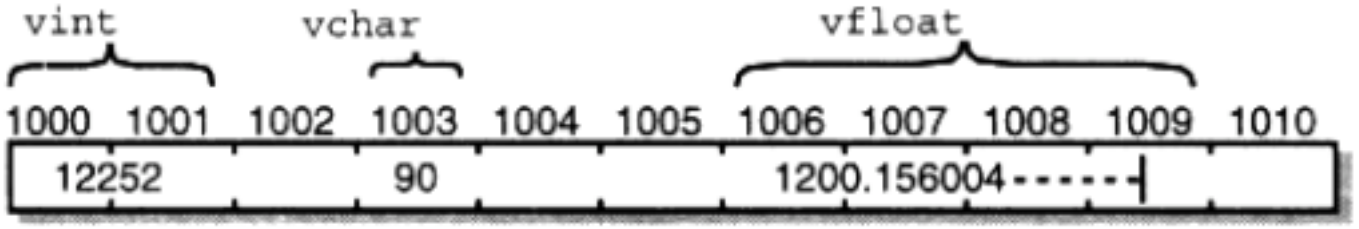


```
a = 3
b = 16 c = 16
```

Puntatori e variabili

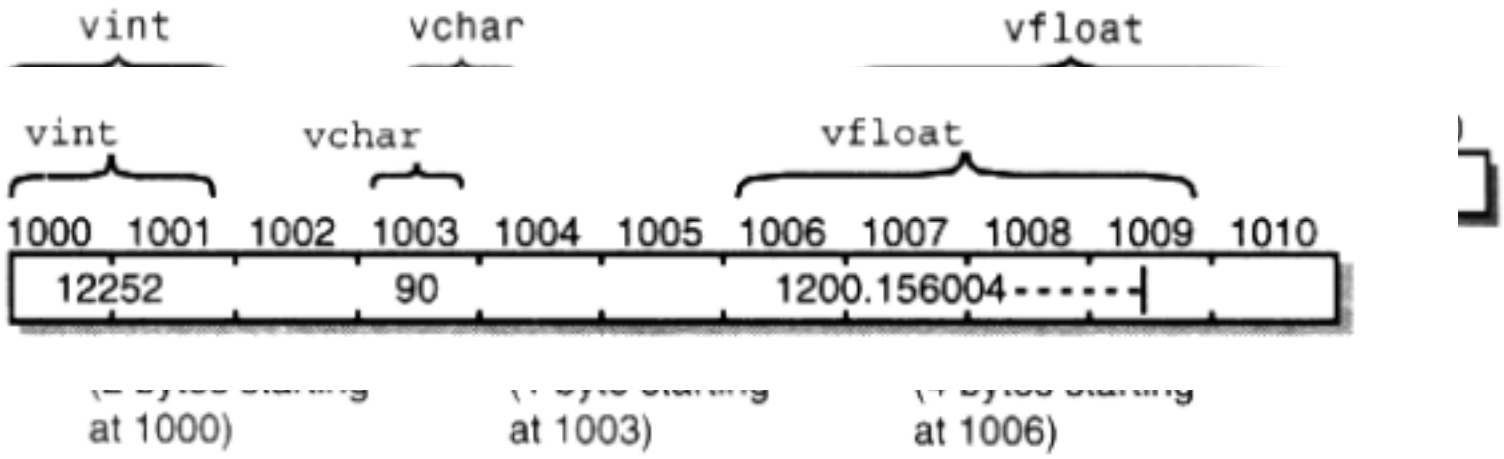
- Differenti tipi di variabili occupano differenti quantità di memoria
- Ogni locazione di memoria (byte) ha un suo indirizzo
- Si definisce indirizzo di una variabile l'indirizzo della prima locazione di memoria occupata dalla vairiabile stessa

```
int vint = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```



Puntatori e variabili

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```



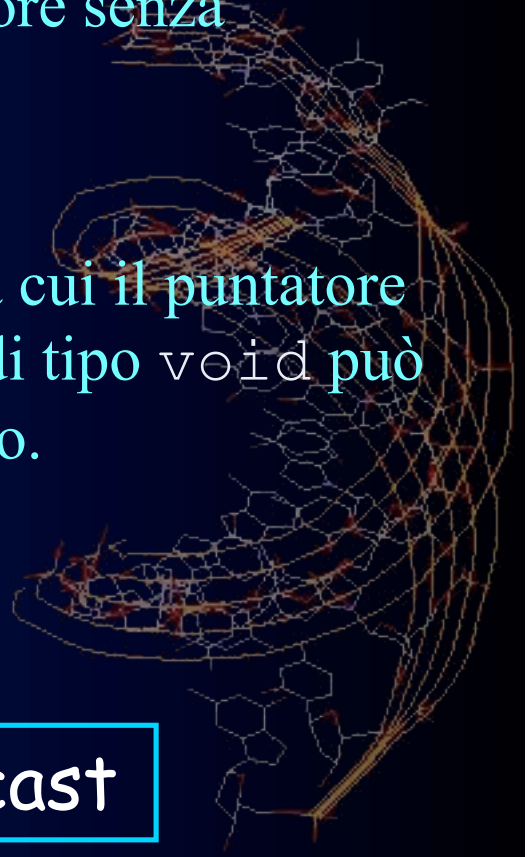
Puntatori void

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 può 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 */
```

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

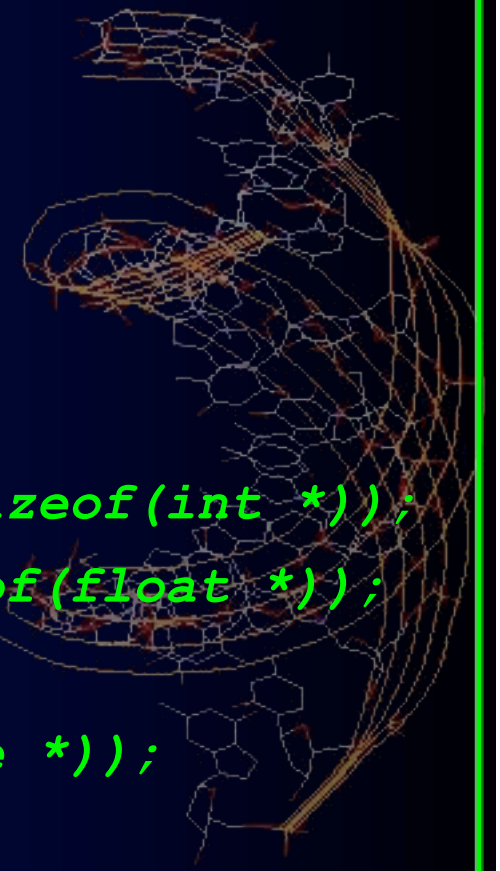
Warnings

- 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:

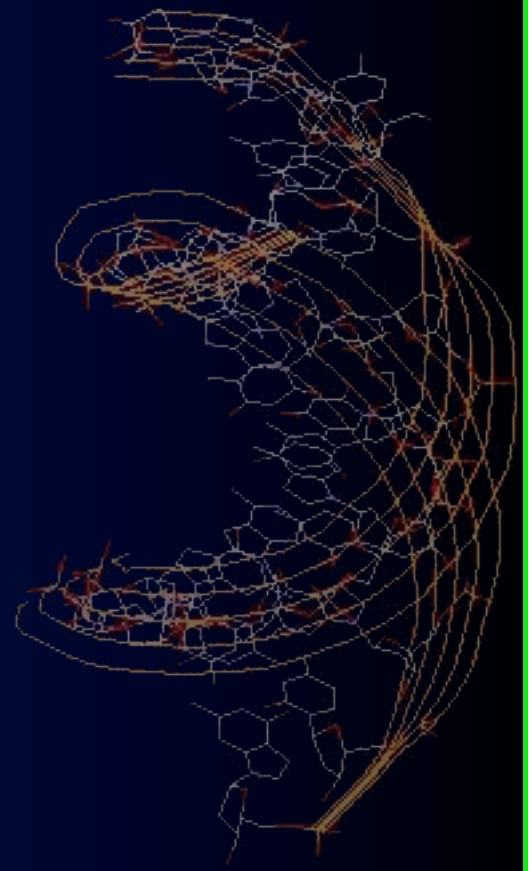
Get the size: `sizeof`

```
main(){
int i, *pi;
float f, *pf;
double d, *pd;

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 *));
}
```



Get the size: `sizeof`

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



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, *ugo;
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("ugo   occupa %d bytes\n", sizeof(ugo));
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
ugo   occupa 4 bytes
enzo  occupa 1 bytes
Totale: 17 bytes
```

Array Statici e Puntatori

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

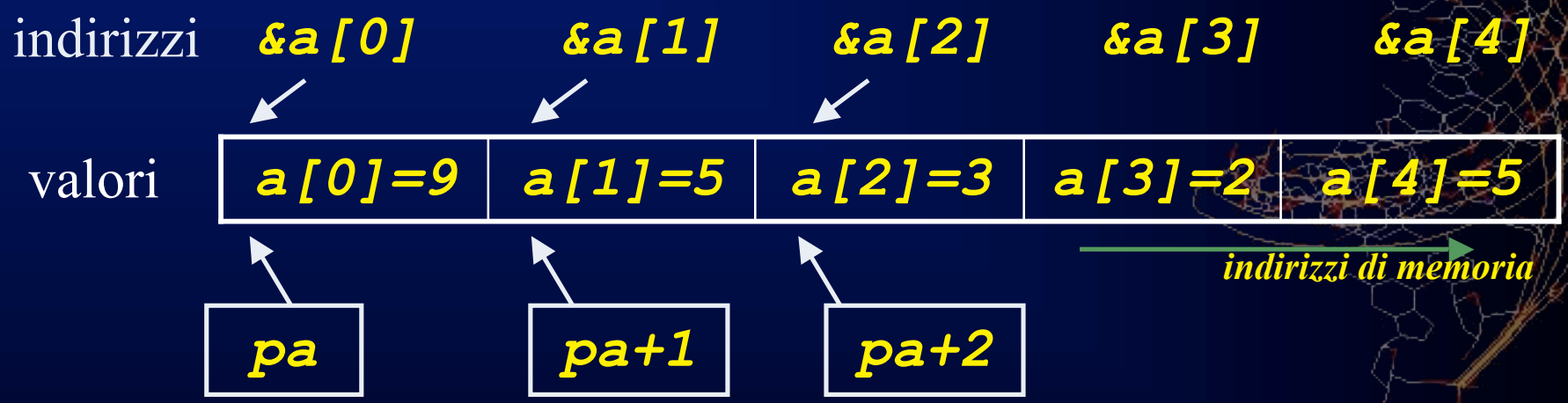
Il programmatore non può modificarlo, ma può conoscerlo ?

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[0]; /* equivalente */
```

Array Statici e Puntatori

```
int *pa, a[]= {9, 5, 3, 2, 5};
. . . . .
pa = a;  /* corretto */
pa = &a[0]; /* equivalente */
```

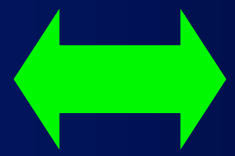


Se ***pa*** punta al primo elemento dell'array, ***pa+1*** punta a quello successivo: ***pa+n*** punta all'n-esimo elemento dopo il primo, indipendentemente dal tipo o dimensione delle variabili nell'array.

Array Statici e Puntatori

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

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*/
```

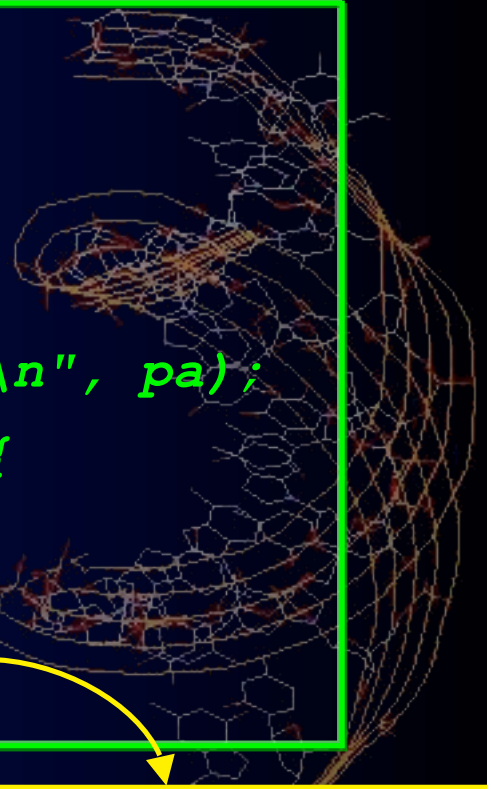
OK !

```
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

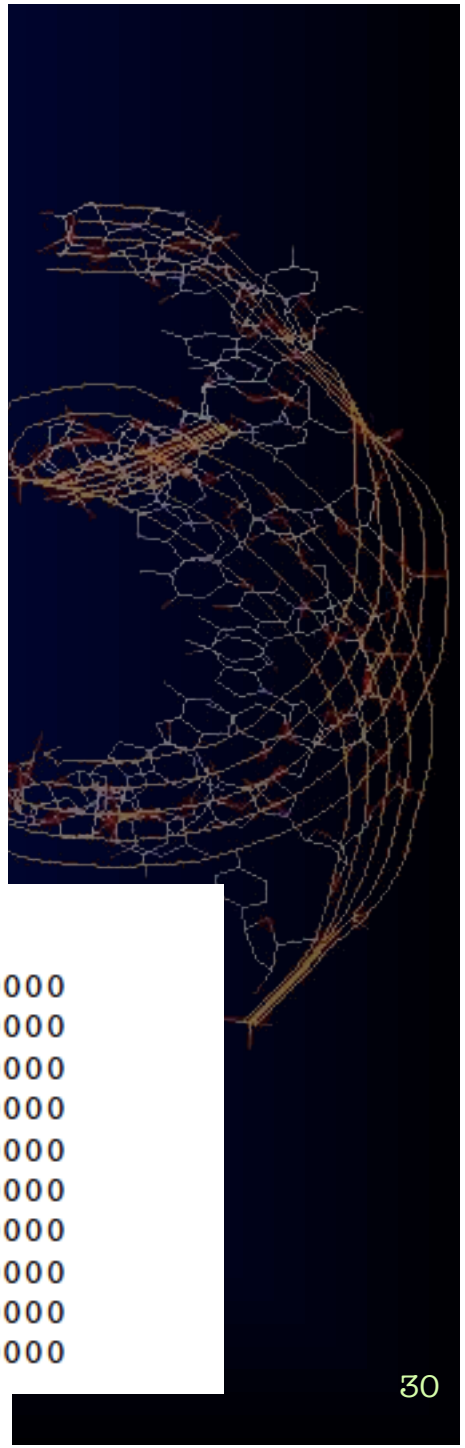
```
main(){
    int i, *pa, a[]= {8,12,56,3,9};

    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.

```
/* Demonstrates using pointer arithmetic to access */
/* array elements with pointer notation. */

#include <stdio.h>
#define MAX 10

/* Declare and initialize an integer array. */

int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };

/* Declare a pointer to int and an int variable. */

int *i_ptr, count;

/* Declare and initialize a float array. */

float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };

/* Declare a pointer to float. */

float *f_ptr;

int main()
{
    /*Initialize the pointers. */

    i_ptr = i_array;
    f_ptr = f_array;

    /* Print the array elements. */

    for (count = 0; count < MAX; count++)
        printf("%d\t%\n", *i_ptr++, *f_ptr++);

    return 0;
}
```



OUTPUT:

```
0       0.000000
1       0.100000
2       0.200000
3       0.300000
4       0.400000
5       0.500000
6       0.600000
7       0.700000
8       0.800000
9       0.900000
```