

# Puntatori

Credits Prof. Alessandro Campi

# Variabili "rivisitate"

- Finora una variabile è stata accessibile solo mediante un *nome* (identificatore):

**$x = a;$**

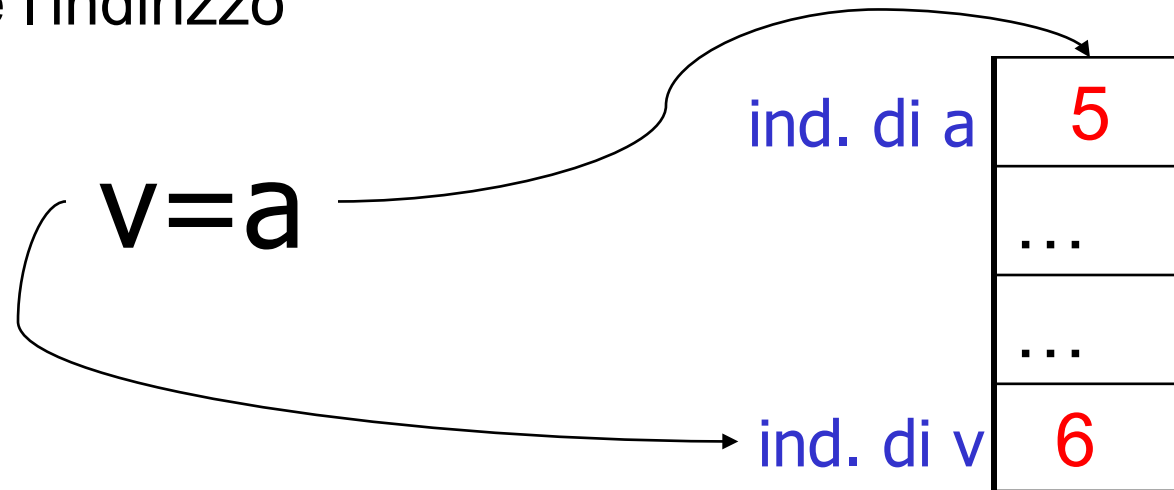
- Il valore contenuto nella cella identificata da **a** è memorizzato nella cella identificata da **x**
  - È possibile che le variabili occupino più celle
- Ma che cos'altro identifica una variabile?
  - Già più volte però si è parlato di *indirizzi* (assembler, scanf, array, ...)

# Indirizzi e valori

- Ogni variabile ha, tra gli elementi che la caratterizzano:
  - **Indirizzo**: è l'indirizzo della locazione di memoria associata alla variabile (ind. della prima cella)
  - **Valore**: è il valore contenuto nella locazione di memoria associata alla variabile
- L'indirizzo è immutabile, il valore muta durante l'esecuzione del programma
  - La variabile, cioè, **non si sposta**

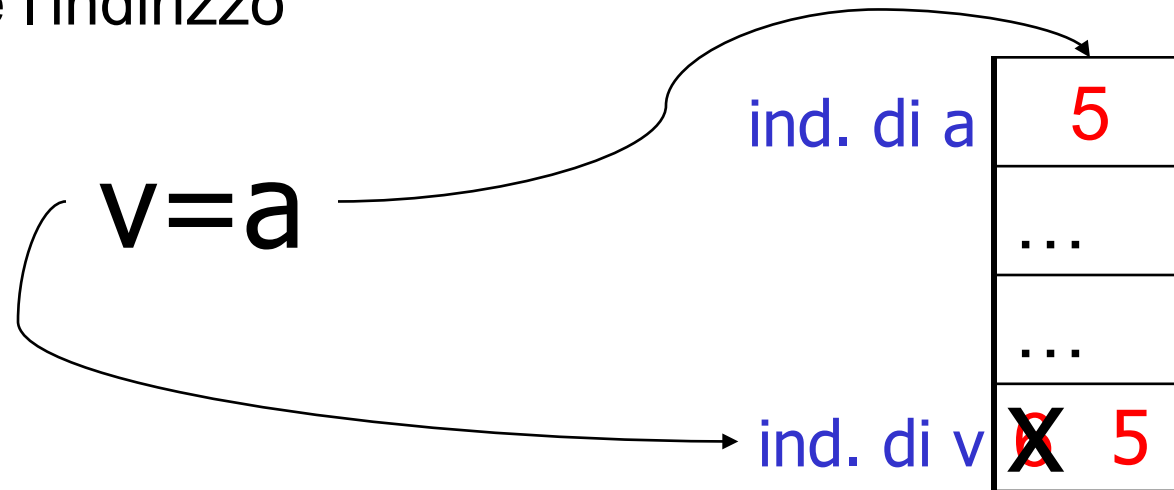
# Indirizzi e valori: gli assegnamenti

- Quando una variabile è a sinistra di un assegnamento, **si usa** il suo **indirizzo** per **modificare** il suo **valore**
- Quando è a destra, **si usa** il **valore**
  - a cui accediamo tramite l'indirizzo



# Indirizzi e valori: gli assegnamenti

- Quando una variabile è a sinistra di un assegnamento, **si usa** il suo **indirizzo** per **modificare** il suo **valore**
- Quando è a destra, **si usa** il **valore**
  - a cui accediamo tramite l'indirizzo



Gli identificatori servono "a noi" per distinguere agevolmente le variabili, ma per accedere alla RAM l'esecutore usa (ovviamente) gli indirizzi

# Indirizzi

- In alcuni linguaggi di programmazione non è possibile (per il programmatore) conoscere gli indirizzi delle variabili
  - Esempio: **Java**
- In C è possibile conoscere l'indirizzo delle locazioni di memoria associate alle variabili, mediante l'operatore **&**

# L'operatore &

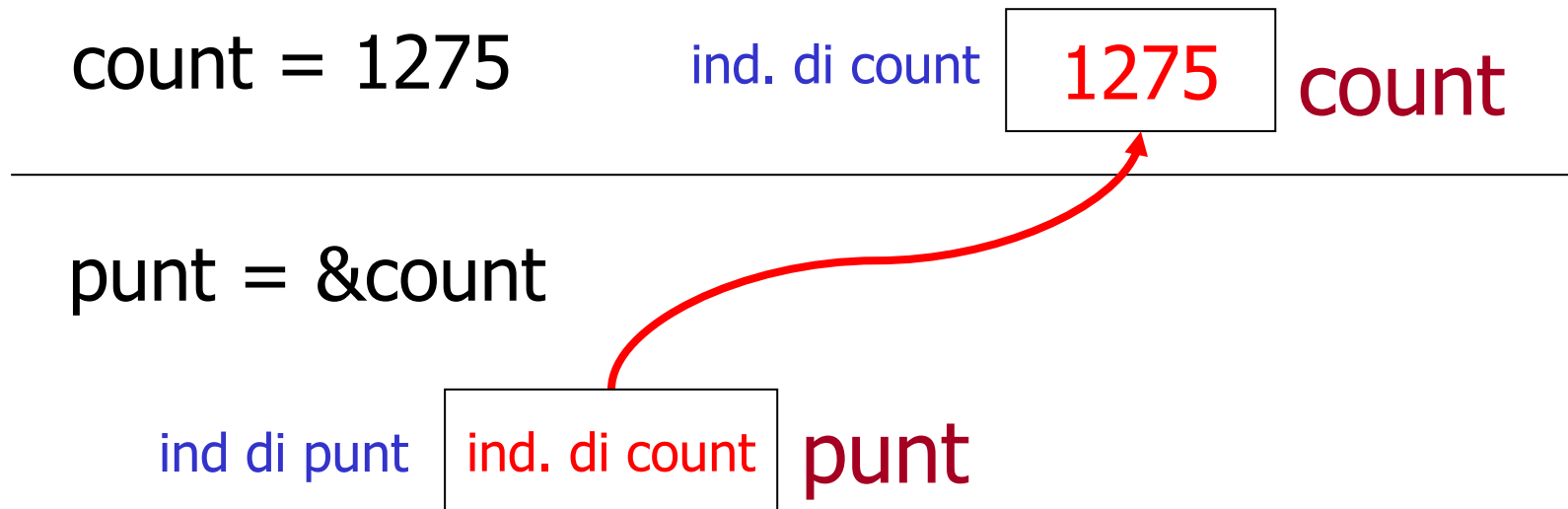
```
int main() {  
    int x = 3;  
    printf ( "indirizzo di x : %p \n", &x );  
    printf ( "valore di x : %d \n", x );  
}
```

**Output del programma:**

```
indirizzo di x : 0xbffff984  
valore di x : 3
```

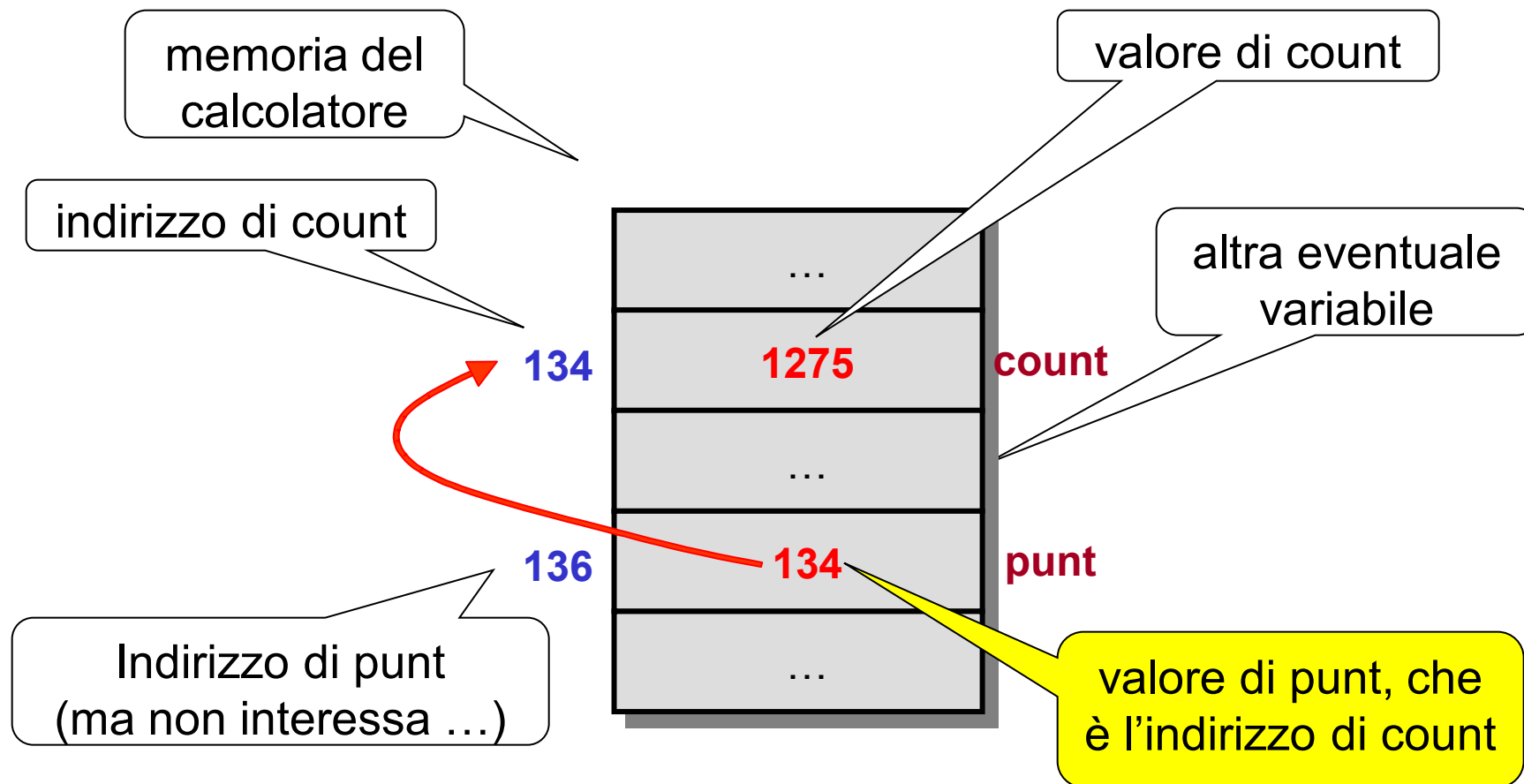
# I puntatori

- Ci sono delle variabili, le variabili ***puntatore***, che possono contenere l'indirizzo di un'altra variabile





# O anche ...



# Dichiarare i puntatori

Tipo \* nomePunt;

- TipoRef è un puntatore a dati di tipo Tipo
  - int \* intRef;
  - float \* floatRef;
- **ATTENZIONE:** *puntatori a dati di tipo diverso sono variabili di tipo diverso*
- Suggerimento: usare "Ref" (o Punt) in coda al nome per denotare i puntatori

# Il tipo puntatore

```
typedef Tipo * TipoRef;
```

– TipoRef è un puntatore a dati di tipo Tipo

- `typedef int *IntRef;`
- `IntRef myRef, yourRef;`
- `int * herRef;`

– **ATTENZIONE:** *tipi puntatore a dati di tipo diverso sono tipi diverso*

# Il modello

Quando abbiamo **myRef = &x** lo rappresentiamo così:



Diciamo che  
myRef *punta a* x

come possiamo accedere al valore  
di questa variabile usando myRef?

**dereferenziazione: \*myRef**

Operazione che permette di accedere al  
contenuto della cella puntata da myRef

# Dereferenziazione

- L'operatore unario `*` è detto di *dereferenziazione*
- Permette di estrarre il **valore** della variabile puntata dal puntatore che è argomento dell'operatore

```
typedef int * punt_a_int;
```

si legge anche dicendo che dereferenziano un `punt_a_int` si ottiene un `int`

```
int x = 3;
```

```
punt_a_int p = &x; /* inizializzazione di p */
```

```
printf("il valore di x e' %d\n", *p);
```

Equivalente a

```
int x = 3;
```

```
int * p = &x; /* inizializzazione di p */
```

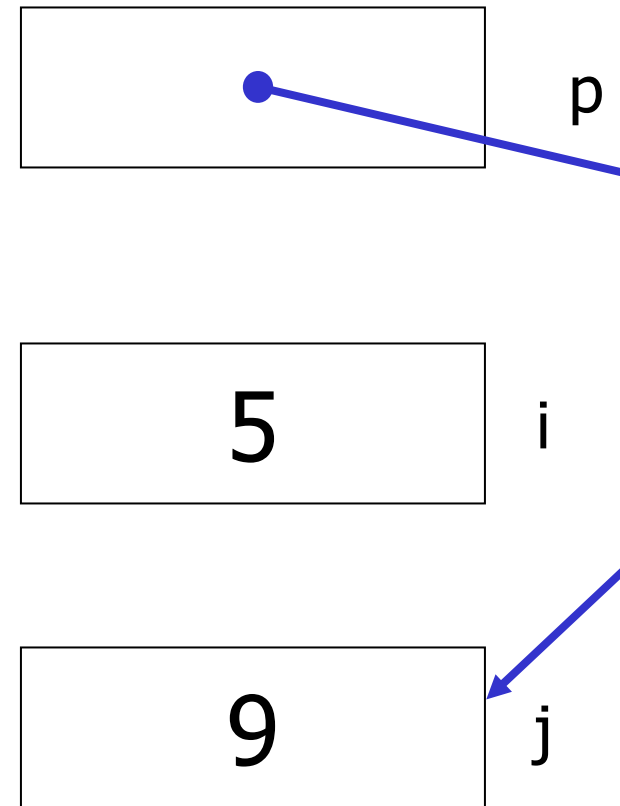
```
printf("il valore di x e' %d\n", *p);
```

# Dereferenziamento

```
char x = 'a';  
char * p = &x; /* inizializzazione di p */  
printf("il valore di x e' %c\n", *p);
```

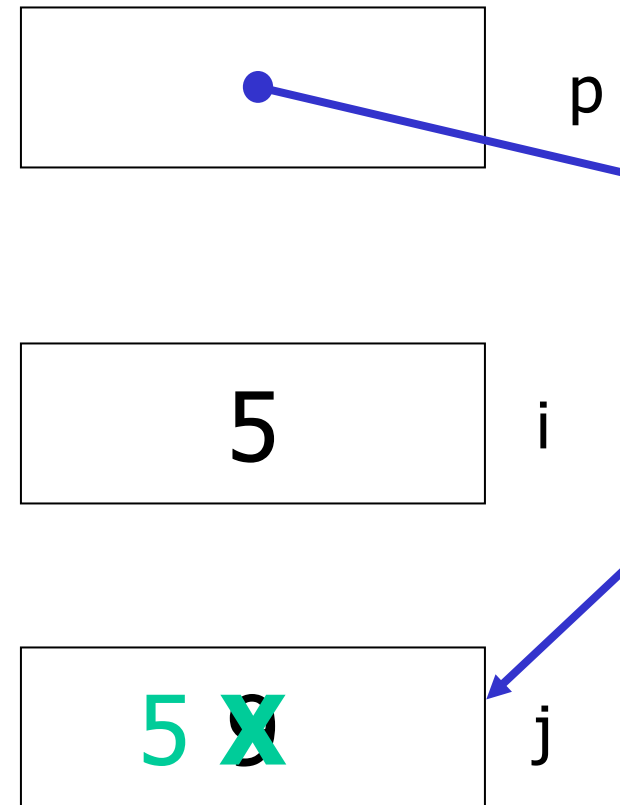
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;
```



# Esercizio: simulazione di esecuzione

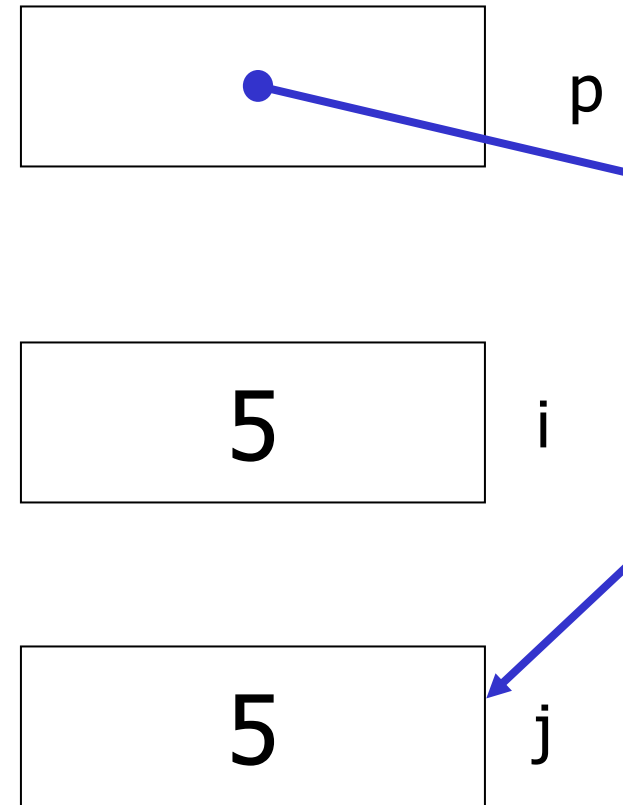
```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;
```





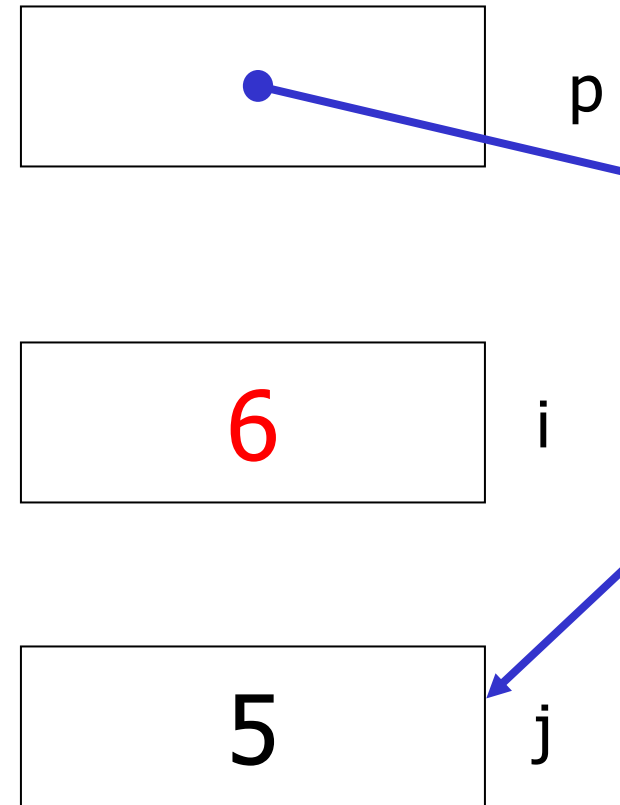
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;
```



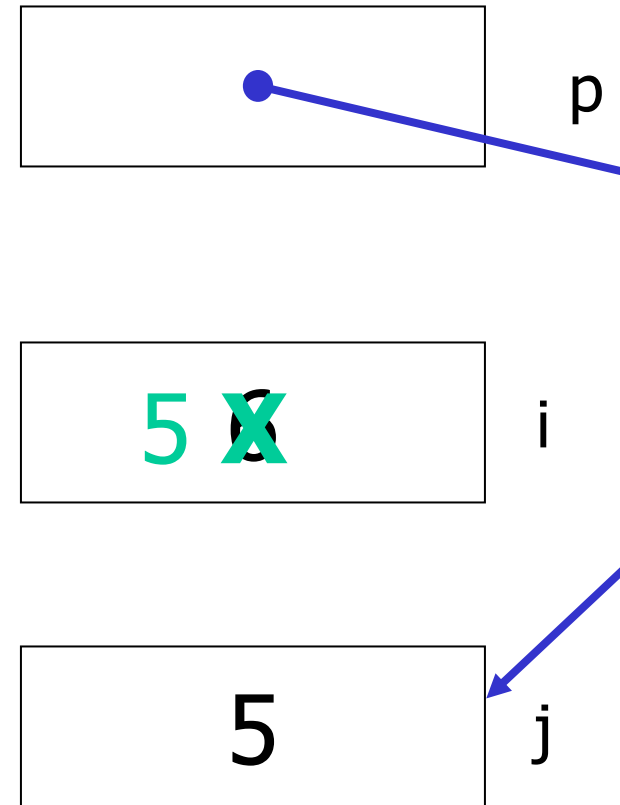
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;
```



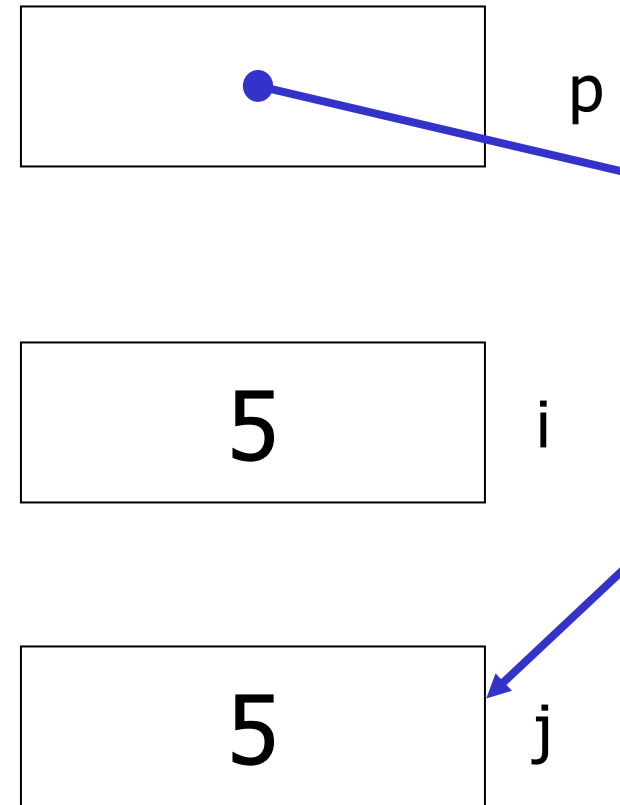
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;
```



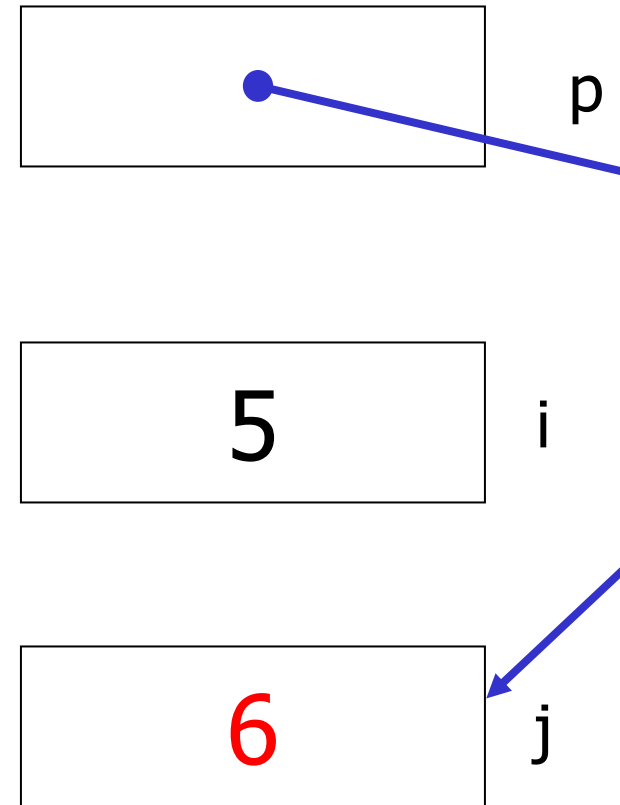
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;
```



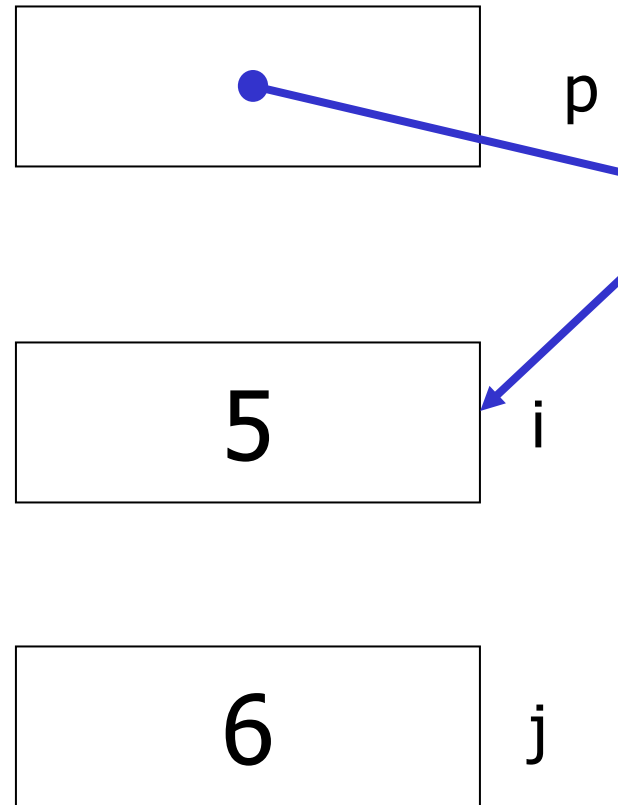
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;  
(*p)++;
```



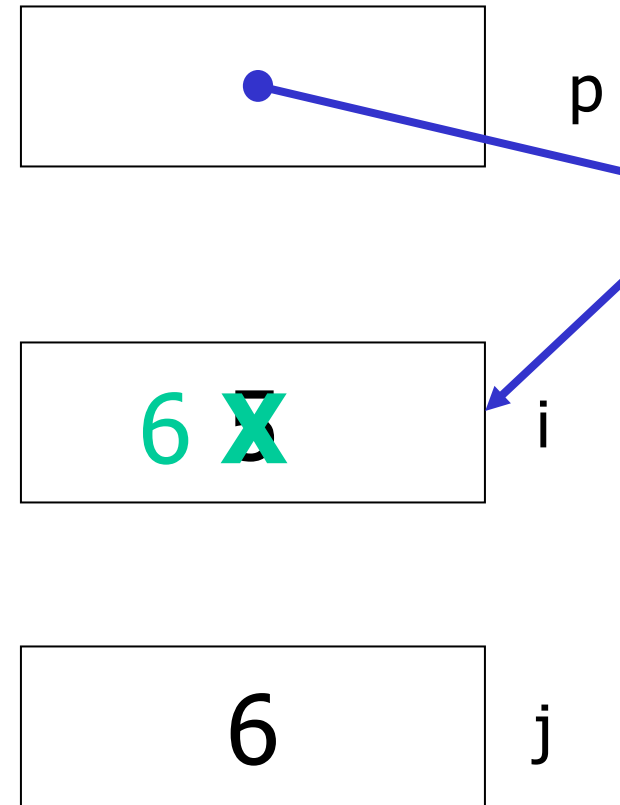
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;  
(*p)++;  
p = &i;
```



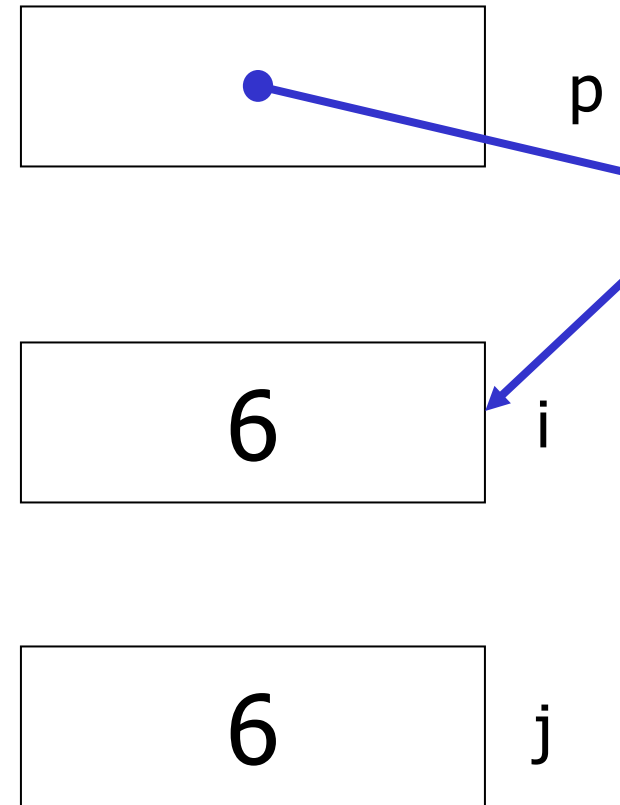
# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;  
(*p)++;  
p = &i;  
*p = j;
```



# Esercizio: simulazione di esecuzione

```
typedef int * Punt;  
Punt p;  
int i = 5, j = 9;  
p = &j;  
*p = i;  
++i;  
i = *p;  
(*p)++;  
p = &i;  
*p = j;
```





# Puntatori: riassunto

- `int *` è la dichiarazione di un puntatore `p` a un intero  
`int i, *p;`
- Con `&i` si denota l'indirizzo della variabile `i`
- `&` è l'operatore che restituisce l'indirizzo di una variabile  
`p = &i;` (operatore di **referenziamento**)
- L'operatore opposto è `*`, che restituisce il valore puntato  
`i = *p;` (operatore di **dereferenziamento**)
- **Attenzione:** non si confondano i molteplici usi dell'asterisco (moltiplicazione, dichiarazione di puntatore, e dereferenziamento)

# Esercizio

Data a seguente dichiarazione

```
int *p, *q;
```

Spiegare la differenza tra

```
p = q;
```

e

```
*p = *q;
```

- Nel primo caso si impone che il puntatore **p** punti alla stessa variabile a cui punta **q**
- Nel secondo caso si assegna il **valore** della variabile puntata da **q** al **valore** della variabile puntata da **p**

# Inizializzazione dei puntatori

Data a seguente dichiarazione

```
int *p;  
*p = 7;
```

- Tipicamente solleva un segmentation fault.
- La causa è che il contenuto della cella `p` potrebbe, una volta letto come un indirizzo, dar luogo a:
  - Un indirizzo in memoria non accessibile
  - Un indirizzo non esistente (e.g. memoria troppo piccola)

# Inizializzazione dei puntatori

Data a seguente dichiarazione

```
int *p;  
*p = 7;
```

- Tipicamente solleva un segmentation fault.
- La causa è che il contenuto della cella `p` potrebbe, una volta letto come un indirizzo, dar luogo a:
  - Un indirizzo in memoria non accessibile
  - Un indirizzo non esistente (e.g. memoria troppo piccola)
- Soluzione, inizializzare i puntatori sempre con una variabile dichiarata di appoggio:

```
int *p, i; // anche se i non si usa dirett.  
p = &i; // p punta ad una cella ammiss.  
*p = 7;
```

# Inizializzazione dei puntatori

E' possibile inizializzare un puntatore con la prima cella di un array

```
int *p, a[10];
```

```
p = a; // CORRETTO, copia in p &a[0]
```

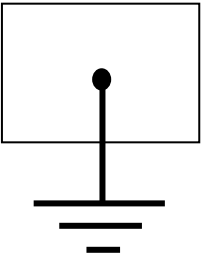
# Il valore NULL

- NULL: costante simbolica che rappresenta un valore speciale che può essere assegnato a un puntatore
- Significa che la variabile non punta a niente
  - È un errore dereferenziare la variabile che punta a NULL

```
int *p = NULL, a;
```

**myRef = NULL;**

**myRef**



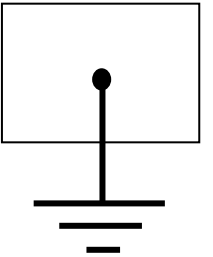
# Il valore NULL

- NULL: costante simbolica che rappresenta un valore speciale che può essere assegnato a un puntatore
- Significa che la variabile non punta a niente
  - È un errore dereferenziare la variabile che punta a NULL

```
int *p = NULL, a;  
*p = 7; // ERRORE!
```

**myRef = NULL;**

**myRef**



```
"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2023_Informatica_A_Boracchi\Lez10_codes\provaPuntatori_noArray.exe"
```

```
inserisci p: 23
```

```
Process returned -1073741819 (0xC0000005)   execution time : 3.606 s  
Press any key to continue.
```

# Il valore NULL

- NULL: costante simbolica che rappresenta un valore speciale che può essere assegnato a un puntatore
- Significa che la variabile non punta a niente

– È un errore dereferenziare la variabile che punta a NULL

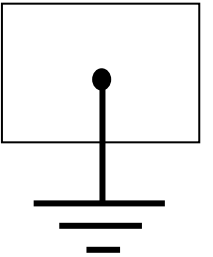
```
int *p = NULL, a;
```

```
*p = 7; // ERRORE!
```

```
p = &a; *p = 7; // CORRETTO!
```

**myRef = NULL;**

**myRef**



- Standard ANSI: impone che NULL rappresenti il valore 0
  - Test di nullità di un puntatore: if ( p == NULL ) oppure if ( !p )
  - Test di non nullità: if ( p != NULL ) oppure if (p)
- Utilizziamo il simbolo della "messa a terra"



# Inizializzazione dei puntatori

- Il valore iniziale di un puntatore **dovrebbe essere la costante speciale NULL**
- NULL significa che **non ci si riferisce ad alcuna cella di memoria**
- Dereferenziando NULL si ha un **errore in esecuzione**
- *Come al solito, non bisogna fare **MAI** affidamento sulle inizializzazioni implicite delle variabili che il C potrebbe fare*
  - *Alcune implementazioni inizializzano a NULL*

# Puntatori e tipo delle variabili puntate

- Il compilatore segnala l'uso di puntatori a dati di tipo diverso da quello a cui dovrebbero puntare
  - In forma di warning: sono errori *potenziali*
- I tipi “puntatore a tipo x” e “puntatore a tipo y” sono tutti diversi tra loro
- Il tipo **void \*** però è compatibile con tutti i puntatori ad altri tipi

# Puntatori e Array

# Inizializzazione dei puntatori

E' possibile inizializzare un puntatore con la prima cella di un array

```
int *p, a[10];
```

```
p = a; // CORRETTO, copia in p &a[0]
```

```
a = p; // SBAGLIATO! La parte a dx  
dell'uguale non è modificabile, è un  
indirizzo costante!
```

## Esempio: assegna a due puntatori l'indirizzo degli elementi con valore minimo e massimo in un array

```
#define LUNGHEZZAARRAY 100
int main () {
    int    i,    ArrayDiInt[LUNGHEZZAARRAY];
    int    *PuntaAMinore, *PuntaAMaggiore;
    . . . .
    PuntaAMinore = &ArrayDiInt[0];
    i = 1;
    while ( i < LUNGHEZZAARRAY ) {
        if ( ArrayDiInt[i] < *PuntaAMinore )
            PuntaAMinore = &ArrayDiInt[i];
        i = i + 1;
    }
    PuntaAMaggiore = &ArrayDiInt[0]; i = 1;
    while ( i < LUNGHEZZAARRAY ) {
        if ( ArrayDiInt[i] > *PuntaAMaggiore )
            PuntaAMaggiore = &ArrayDiInt[i];
        i = i + 1;
    }
    return 0;
}
```

# Aritmetica dei puntatori

- Il C permette operazioni di somma e sottrazione tra puntatori
- Per esempio:

```
int *p, a[10];
```

```
p = a;
```

```
p = p + 3; // salta di tre indirizzi. Il  
salto è definito dalla dimensione  
del tipo puntato
```

# Array e puntatori

- In C esiste una parentela stretta tra array e puntatori
- Il nome di un array (p. es. `v`) è una *costante* (simbolica) di tipo puntatore (al tipo componente l'array), di valore “indirizzo della prima cella allocata per l'array”
- Esempio:  

```
int v[3];
```

definisce `v` come `int const *`, cioè come un **puntatore costante** a una variabile intera
- Perciò `v[i]` è equivalente a `*(v + i)`
  - Calcolo dello **spiazzamento** nel vettore grazie all'aritmetica dei puntatori
- `v` è come `&v[0]`
- `v+3` è come `&v[3]`

# Ecco finalmente svelato uno dei misteri della scanf!

- Perché ci vuole **&** per memorizzare un valore in una variabile generica, ma non in una stringa?
  - La funzione **scanf()** riceve come parametri gli **indirizzi** delle variabili in cui scrivere i valori letti da terminale
  - Gli identificatori delle variabili “normali” rappresentano la variabile, e per estrarne l’indirizzo occorre l’operatore **&**.
  - Gli identificatori degli array rappresentano già di per sé i puntatori ai primi elementi, quindi nel caso delle stringhe (che sono array) non occorre **&**.
  - Se devo acquisire un puntatore ad un tipo base posso usare la scanf



# Ecco finalmente svelato uno dei misteri della scanf!

Per acquisire il valore di una cella puntata dal puntatore ad un tipo base, posso usare la scanf e non devo usare &

```
int *p, i;
```

```
p = &i;
```

```
scanf("%d", p);
```



p contiene un indirizzo, quindi la sintassi è corretta. Il risultato delle due scanf è identico!

# Riassumendo: array e puntatori

- Con la seguente dichiarazione:

```
int a[5], i, * p;
```

`a[i]` equivale a `*(a + i)`

`p = a` equivale a `p = &a[0];`

`p = a + 1` equivale a `p = &a[1];`

`a = p;` è un ERRORE

`a = a + 1;` è un ERRORE

- **Cioè occorre ricordare che `a` è un array e che sebbene contenga un indirizzo, questo non può variare.**

# Ancora sull'aritmetica dei puntatori

- Se  $p$  e  $q$  puntano a due diversi elementi di uno stesso array, la differenza:

$$p - q$$

dà la distanza **nell'array** tra gli elementi puntati

- Tipicamente **non** coincide con la differenza “aritmetica” tra i valori numerici dei due puntatori
  - È una distanza espressa in “numero di elementi”
- Sono ammissibili le seguenti operazioni tra puntatori:

$$+, ++, -, --, ==, =$$

```

#include<stdio.h>
#define L 5

int main(){
    int vet[L];
    int *p, i;

    // inicializzo a -1 il vettore
    for(i = 0; i < L; i++)
        vet[i] = -1;
    p = vet; //copio nel puntatore l'indirizzo di vet[0], uguale a p = &vet[0]

    printf("\n valori di vet: [");
    for(i = 0; i < L; i++)
        printf("%d ", vet[i]);
    printf("]");

    for(i = 0; i < L; i++)
    {
        *(p + i) = 2*i; // così scrivo nella cella di indirizzo &vet[i];
        // non modifico p, modifico solo gli elementi di vet
        printf("\ni = %d, vet[%d] = %d, *p=%d, (p=%p)", i, i, vet[i], *p, p); //
        // stampo l'indirizzo e vedo che sono in sequenza scritta in esadecimale
    }
    printf("\n valori di vet: [");
    for(i = 0; i < L; i++)
        printf("%d ", vet[i]);
    printf("]");
    return 0;
}

```

# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = 0, *p=0, (p=0060FEF4)
i = 1, vet[1] = 2, *p=0, (p=0060FEF4)
i = 2, vet[2] = 4, *p=0, (p=0060FEF4)
i = 3, vet[3] = 6, *p=0, (p=0060FEF4)
i = 4, vet[4] = 8, *p=0, (p=0060FEF4)
valori di vet: [0 2 4 6 8 ]
Process returned 0 (0x0)    execution time : 0.096 s
Press any key to continue.
```

Di fatto p non cambia mai, l'indirizzo è costante (scritto in esadecimale) e punta sempre al vet[0] che vale 0

```

#include<stdio.h>
#define L 5

int main(){
    int vet[L];
    int *p, i;

    // inizializzo a -1 il vettore
    for(i = 0; i < L; i++)
        vet[i] = -1;
    p = vet; //copio nel puntatore l'indirizzo di vet[0], uguale a p = &vet[0]

    //[...]
    for(i = 0; i < L; i++)
    {
        *(p++) = 2*i; // equivale a *p = 2*i; p++;
        // 1) scrivo nella cella puntata da p il valore 2*i
        // 2) passo all'indirizzo successivo con p
        // la prima volta scrive in vet[0] il valore 0
        // [...]
        // l'ultima volta scrive in vet[9] il valore 18
        // ora p punta alla prox cella (è fuori dal range di vet)

        printf("\ni = %d, vet[%d] = %d, *p=%d, (p=%p)", i, i, vet[i], *p, p); }

    printf("\n valori di vet: [");
    for(i = 0; i < L; i++)
        printf("%d ", vet[i]);
    printf("]");
    return 0;
}

```

# Output Esecuzione

$* (p++) = 2*i;$  (prima era  $* (p + i) = 2*i$ )

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = 0, *p=-1, (p=0060FEF8)
i = 1, vet[1] = 2, *p=-1, (p=0060FEFC)
i = 2, vet[2] = 4, *p=-1, (p=0060FF00)
i = 3, vet[3] = 6, *p=-1, (p=0060FF04)
i = 4, vet[4] = 8, *p=4, (p=0060FF08)
valori di vet: [0 2 4 6 8 ]
Process returned 0 (0x0)    execution time : 0.114 s
Press any key to continue.
```

- L'indirizzo di p cambia ad ogni iterazione
- Gli indirizzi aumentano di 4 parole (si guardi ultima cifra)

# Output Esecuzione

$* (p++) = 2*i;$  (prima era  $*(p + i) = 2*i$ )

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = 0, *p=-1, (p=0060FEF8)
i = 1, vet[1] = 2, *p=-1, (p=0060FEFC)
i = 2, vet[2] = 4, *p=-1, (p=0060FF00)
i = 3, vet[3] = 6, *p=-1, (p=0060FF04)
i = 4, vet[4] = 8, *p=4, (p=0060FF08)
valori di vet: [0 2 4 6 8 ]
Process returned 0 (0x0)    execution time : 0.114 s
Press any key to continue.
```

- L'indirizzo di p cambia ad ogni iterazione
- Gli indirizzi aumentano di 4 parole (si guardi ultima cifra)
- il valore puntato da p fa riferimento «alla prima cella successiva», visto che c'è il post-incremento p++
- Come mai \*p = 4 nell'ultima iterazione?



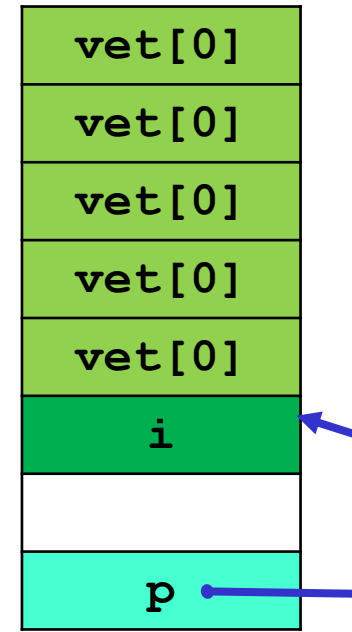
# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

$* (p++) = 2*i;$

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = 0, *p=-1, (p=0060FEF8)
i = 1, vet[1] = 2, *p=-1, (p=0060FEFC)
i = 2, vet[2] = 4, *p=-1, (p=0060FF00)
i = 3, vet[3] = 6, *p=-1, (p=0060FF04)
i = 4, vet[4] = 8, *p=4, (p=0060FF08)
valori di vet: [0 2 4 6 8 ]
```

Process returned 0 (0x0) execution time : 0.114 s  
Press any key to continue.



- il valore puntato da **p** fa riferimento «alla prima cella successiva», visto che c'è il post-incremento **p++**
  - quindi **p** esce dallo spazio riservato a **vet** e raggiunge lo spazio riservato a **i**

# Output Esecuzione

```
* (p++) = 2*i;
```

"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018\_Informatica\_A\Lez10\puntatori\_e\_array.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]
```

```
&i=0060FF08
```

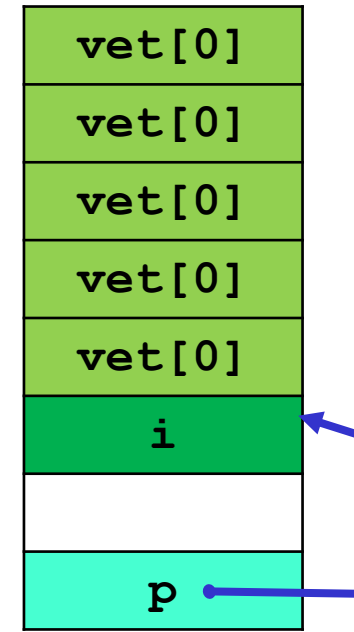
```
i = 0, vet[0] = 0, *p=-1, (p=0060FEF8)  
i = 1, vet[1] = 2, *p=-1, (p=0060FEFC)  
i = 2, vet[2] = 4, *p=-1, (p=0060FF00)  
i = 3, vet[3] = 6, *p=-1, (p=0060FF04)  
i = 4, vet[4] = 8, *p=4, (p=0060FF08)
```

```
p punta a i dopo il ciclo
```

```
valori di vet: [0 2 4 6 8 5 6356744 2396160 4199040 6356884 ]
```

```
Process returned 0 (0x0) execution time : 0.091 s
```

```
Press any key to continue.
```



Per verificare se davvero **p** punta ad **i**, posso stampare l'indirizzo di **i**

```
printf("\n &i = %p", &i);
```

Oppure usare l'aritmetica dei puntatori

```
if (p == &i)
```

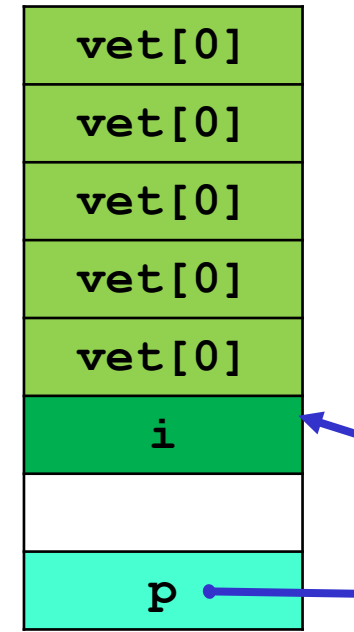
```
printf("\n p punta a i dopo il ciclo");
```

# Output Esecuzione

```
* (p++) = 2*i;
```

E come mai non c'è segmentation fault?

- Accedo solo in lettura (nella printf) ad una cella che è dimensionata correttamente per contenere un intero

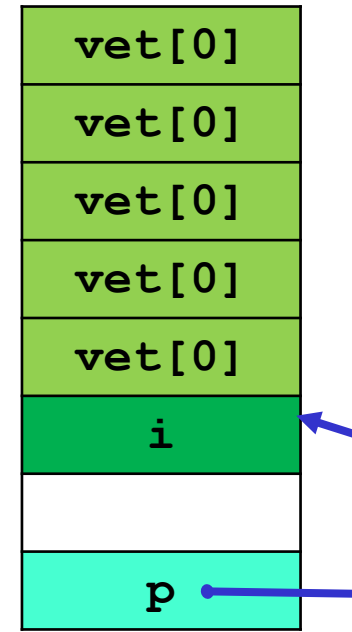


# Output Esecuzione

`*(p++) = 2*i;`

È sempre così?

- No, chi dispone le variabili nella memoria è il compilatore, non sappiamo come questo avviene. Ci limitiamo ad osservare cosa sta avvenendo. Aggiungere una variabile, cambiare l'ordine con cui vengono dichiarate potrebbe cambiare tutto



```

#include<stdio.h>
#define L 5

int main(){
    int vet[L];
    int *p, i;

    // inizializzo a -1 il vettore
    for(i = 0; i < L; i++)
        vet[i] = -1;
    p = vet; //copio nel puntatore l'indirizzo di vet[0], uguale a p = &vet[0]

    //[...]
    for(i = 0; i < L; i++)
    {
        *(++p) = 2*i; // equivale a p++; *p = i;
        // 1) passo all'indirizzo successivo con p
        // 2) scrivo nella cella puntata da p il valore 2*i
        // salta vet[0]
        // la prima volta scrive in vet[1] il valore 0
        // [...]
        // la penultima volta scrive in vet[9] il valore 16
        // l'ultima volta scrive in vet[10] il valore 18
        // ora p punta alla prox cella (è fuori dal range di vet)
        printf("\ni = %d, vet[%d] = %d, *p=%d, (p=%p)", i, i, vet[i], *p, p); }

    printf("\n valori di vet: [");
    for(i = 0; i < L; i++)
        printf("%d ", vet[i]);
    printf("]");
    return 0;
}

```

# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
* (++p) = 2*i;
```

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 ]  
Process returned 0 (0x0)    execution time : 0.151 s  
Press any key to continue.
```

- p cambia ad ogni iterazione
- Con il pre-incremento, nella prima iterazione modifico la cella seguente a vet[0]. Quindi vet[0] rimane -1
- Di fatto, nell'iterazione i-sima, p modifica vet[i+1]

# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
* (++p) = 2*i;
```

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 ]
```

```
Process returned 0 (0x0)    execution time : 0.151 s  
Press any key to continue.
```

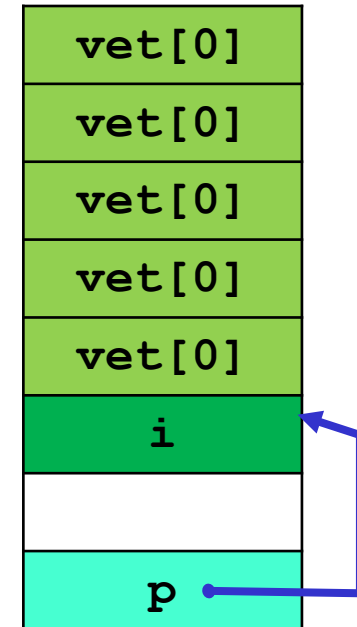
- Come mai `i = 8` nell'ultima iterazione?

# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

$* (++p) = 2*i;$

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)
i = 2, vet[2] = 2, *p=4, (p=0060FF00)
i = 3, vet[3] = 4, *p=6, (p=0060FF04)
i = 8, vet[8] = 2, *p=8, (p=0060FF08)
valori di vet: [-1 0 2 4 6 ]
Process returned 0 (0x0)    execution time : 0.151 s
Press any key to continue.
```



- Come mai  $i = 8$  nell'ultima iterazione?
  - Al termine del ciclo,  $p$  punta alla prima cella libera fuori dall'array. In questo caso troviamo la cella di  $i$
  - A differenza dell'esempio precedente, ora prima si incrementa  $p$  e poi si modifica il valore della cella puntata. Quindi ora si modifica  $i$

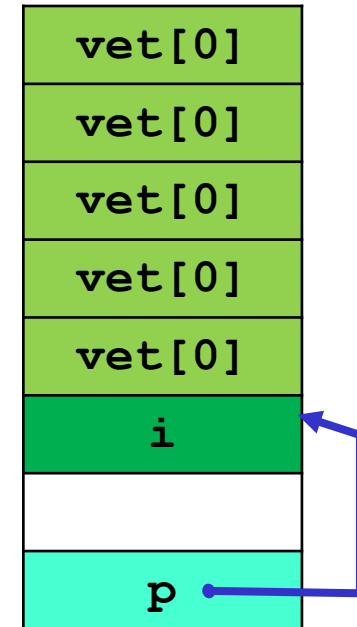


# Output Esecuzione

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

$* (++p) = 2*i;$

```
valori di vet: [-1 -1 -1 -1 -1 ]
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)
i = 2, vet[2] = 2, *p=4, (p=0060FF00)
i = 3, vet[3] = 4, *p=6, (p=0060FF04)
i = 8, vet[8] = 2, *p=8, (p=0060FF08)
valori di vet: [-1 0 2 4 6 ]
Process returned 0 (0x0)    execution time : 0.151 s
Press any key to continue.
```



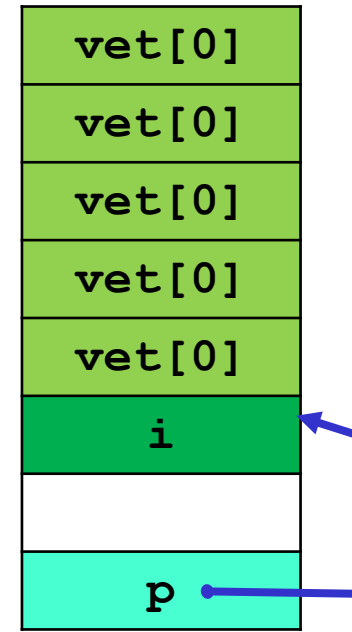
- Come mai  $i = 8$  nell'ultima iterazione?
  - $vet[8] = 2$  si ha perché era così inizializzata così la memoria: non viene modificata quella cella
  - $p$  modifica comunque  $vet[4]$  e ci scrive 6, semplicemente la stampa salta

# Output Esecuzione

`*(p++) = 2*i;`

E come mai non c'è segmentation fault?

- Anche se accedo in scrittura, e scrivo in `vet[5]` il valore 8, l'operazione viene in questo caso permessa perché il processore non riscontra inconsistenze e non si esce dallo spazio di memoria riservato per il programma.
- Questo ribadisce quanto sia importante controllare gli accessi alla memoria, perché questi potrebbero dar luogo a comportamenti difficilmente controllabili e che non portano a errori a runtime (ed es nel caso in cui scrivessi in `i` un valore inferiore a `L`)



# Se stampassi oltre...

```
printf("\n valori di vet: [");  
for(i = 0; i < L + 5; i++)  
    printf("%d ", vet[i]);  
printf("]");
```

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 5 6356744 49 2 6356884 ]  
Process returned 0 (0x0)    execution time : 0.111 s  
Press any key to continue.
```

Ritroviamo il valore 2 in `vet[8]` come ci si aspettava (sapevamo che era lì)

# Se stampassi oltre...

```
printf("\n valori di vet: [");  
for(i = 0; i < L + 5; i++)  
    printf("%d ", vet[i]);  
printf("]");
```

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 ] 5 6356744 49 2 6356884 ]  
Process returned 0 (0x0)   execution time : 0.111 s  
Press any key to continue.
```

Nelle prime 5 posizioni abbiamo il vettore  
come l'abbiamo riempito prima (inclusa la posizione 5)

# Se stampassi oltre...

```
printf("\n valori di vet: [");  
for(i = 0; i < L + 5; i++)  
    printf("%d ", vet[i]);  
printf("]");
```

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 5 6356744 49 2 6356884 ]  
Process returned 0 (0x0)    execution time : 0.111 s  
Press any key to continue.
```

Come mai abbiamo 5 invece di 8?

- Perché quella è la cella riservata ad `i` ed il ciclo sopra ne sta sovrascrivendo il valore (usa sempre `i!!!`)
- **Quella cella rimane comunque "della variabile `i`"**

# Se stampassi oltre...

```
printf("\n valori di vet: [");  
for(i = 0; i < L + 5; i++)  
    printf("%d ", vet[i]);  
printf("]");
```

Se aggiungessi un'altra variabile *j* (da usare nel ciclo di stampa) cambierebbe la disposizione in memoria e potrei avere altre inconsistenze

```
valori di vet: [-1 0 2 4 6 5 6356744 49 2 6356884 ]  
Process returned 0 (0x0)    execution time : 0.111 s  
Press any key to continue.
```

Come mai abbiamo 5 invece di 8?

- Perché quella è la cella riservata ad *i* ed il ciclo sopra ne sta sovrascrivendo il valore (usa sempre *i!!!*)

# Se stampassi oltre...

```
printf("\n valori di vet: [");  
for(i = 0; i < L + 10000; i++)  
    printf("%d ", vet[i]);  
printf("]");
```

"C:\Users\Giacomo Boracchi\Desktop\puntatori.exe"

```
valori di vet: [-1 -1 -1 -1 -1 ]  
i = 0, vet[0] = -1, *p=0, (p=0060FEF8)  
i = 1, vet[1] = 0, *p=2, (p=0060FEFC)  
i = 2, vet[2] = 2, *p=4, (p=0060FF00)  
i = 3, vet[3] = 4, *p=6, (p=0060FF04)  
i = 8, vet[8] = 2, *p=8, (p=0060FF08)  
valori di vet: [-1 0 2 4 6 5 6356744 49 2 6356884 4198653 1 10292680  
10294952 4214784 6356816 -1 6356820 1994115616 -274007067 -2 199411482  
2 1994114848 10294952 0 1994093245 1 4177920 4199061 1 0 0 0 0 0 198  
2891140 4177920 1982891104 1585647057 6356956 1995911258 4177920 10889  
21062 0 0 4177920 0 0 0 0 1088921062 6356896 0 6356964 1995974016 9316  
55114 0 6356972 1995911210 -1 1996025011 0 0 4199040 4177920 0  
Process returned -1073741819 (0xc0000005)    execution time : 3.213 s  
Press any key to continue.
```

Anche se non ho segmentation fault a leggere in poche celle adiacenti al vettore, avrò certamente segmentation fault se esco dallo spazio di memoria dedicato al programma

# All in all...

Bisogna usare molta cautela con i puntatori

- Uscire dagli spazi di memoria dedicati ad una variabile potrebbe non dare un segmentation fault
- Bisogna quindi prestare molta attenzione durante lo sviluppo



# Puntatori e Vettori

```
#include<stdio.h>

int main()
{
    int i;
    int v[10]={0};

    for(i = 0; i < 10; i++)
        i[v]++;

    printf("i = %d", i);

    for(i = 0; i < 10; i++)
        printf("\nv[%d] = %d", i, v[i]);

    return 0;
}
```

# Puntatori e Vettori

```
#include<stdio.h>

int main()
{
    int i;
    int v[10]={0};

    for(i = 0; i < 10; i++)
        i[v]++;

    printf("i = %d", i);

    for(i = 0; i < 10; i++)
        printf("\nv[%d] = %d", i, v[i]);

    return 0;
}
```

Non da errore né a compile né a runtime

```
"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2021_Informatica_A_Boracchi\Es4\significatoDelleQuadre.exe"
i = 10
v[0] = 1
v[1] = 1
v[2] = 1
v[3] = 1
v[4] = 1
v[5] = 1
v[6] = 1
v[7] = 1
v[8] = 1
v[9] = 1
Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
-
```

# Puntatori e Vettori

```
#include<stdio.h>

int main()
{
    int i;
    int v[10]={0};

    for(i = 0; i < 10; i++)
        i[v]++;

    printf("i = %d", i);

    for(i = 0; i < 10; i++)
        printf("\nv[%d] = %d", i, v[i]);

    return 0;
}
```

equivale a  $*(v+i)++$

il + tra intero e puntatore:

- i) commuta,
- ii) viene interpretato come uno salto di indirizzo

Quindi  $i[v]++$  equivale a  $v[i]++$

# Struct e puntatori

```
typedef struct {  
    int    PrimoCampo;  
    char   SecondoCampo;  
} TipoDato;  
TipoDato t;  
TipoDato * p = &t;
```

*Sintassi per accedere  
ai campi di una struct  
tramite un puntatore **p***

```
p->PrimoCampo    = 12;  
(*p) . PrimoCampo = 12;
```

} *equivalenti*

# Come accedere ai campi di una struttura tramite puntatori

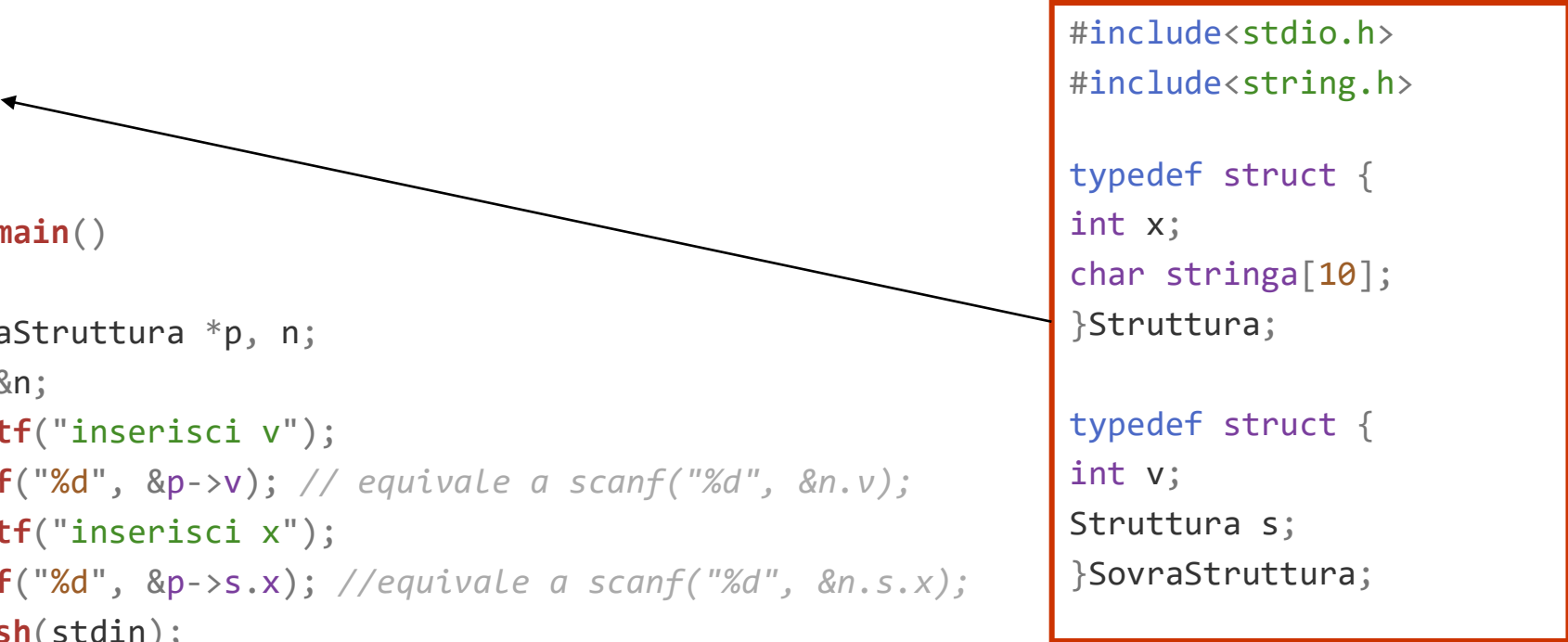
```
int main()
{
    SovraStruttura *p, n;
    p = &n;
    printf("inserisci v");
    scanf("%d", &p->v); // equivale a scanf("%d", &n.v);
    printf("inserisci x");
    scanf("%d", &p->s.x); //equivale a scanf("%d", &n.s.x);
    fflush(stdin);
    printf("inserisci stringa");
    scanf("%s", p->s.stringa); //scanf("%d", n.s.stringa);

    printf("\ncon punt:\np->v = %d, p->s.x = %d, p->s.stringa = %s", p->v, p->s.x, p->s.stringa);
    printf("\ncon var :\nn.v = %d, n.s.x = %d, n.s.stringa = %s", n.v, n.s.x, n.s.stringa);
    return 0 ;
}
```

```
#include<stdio.h>
#include<string.h>

typedef struct {
    int x;
    char stringa[10];
}Struttura;

typedef struct {
    int v;
    Struttura s;
}SovraStruttura;
```



# Esempi di dichiarazioni

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;
```

```
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;
```

# Esempi di dichiarazioni

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;
```

```
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;
```

```
Q1 = &z;  
P = &x;  
P = Q;  
*P = *Q;
```



# Doppi Puntatori

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;  
  
TipoDato *punt;  
TipoDato **doppioPunt; /* doppioPunt punta a un  
                        puntatore a TipoDato */  
  
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;
```

# Doppi Puntatori

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;
```

```
TipoDato *punt;  
TipoDato **doppioPunt; /* doppioPunt punta a un  
                        puntatore a TipoDato */
```

```
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;
```

```
punt = &y;  
doppioPunt = &P;  
y = **doppioPunt
```

# Doppi Puntatori

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;
```

```
TipoDato *punt;  
TipoDato **doppioPunt; /* doppioPunt punta a un  
                        puntatore a TipoDato */
```

```
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;  
  
*punt = x;  
P = *doppioPunt;  
z = *P1;  
punt = P;
```

# Doppi Puntatori

```
typedef TipoDato *TipoPuntatore;  
typedef AltroTipoDato *AltroTipoPuntatore;
```

```
TipoDato *punt;  
TipoDato **doppioPunt; /* doppioPunt punta a un  
                        puntatore a TipoDato */
```

```
TipoPuntatore P, Q;  
AltroTipoPuntatore P1, Q1;  
TipoDato x, y;  
AltroTipoDato z, w;
```

**ERRORI (tipi non rispettati):**

```
P1 = P;
```

```
w = *P;
```

```
*doppioPunt = y;
```

```
punt = doppioPunt;
```

```
*P1 = *Q;
```

# Esempi di istruzioni

## *CORRETTE*

```
punt = &y;  
doppioPunt = &P;  
y = **doppioPunt  
Q1 = &z;  
P = &x;  
P = Q;  
*P = *Q;  
*punt = x;  
P = *doppioPunt;  
z = *P1;  
punt = P;
```

## *SCORRETTE*

```
P1 = P;  
w = *P;  
*doppioPunt = y;  
punt = doppioPunt;  
*P1 = *Q;
```

# Array multidimensionali

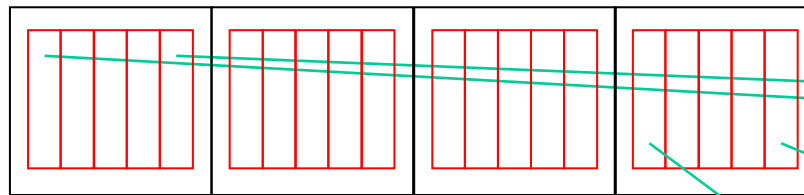
Dobbiamo pensare agli

- array 2D come array 1D i cui elementi sono array 1D
- array N-D come array 1D i cui elementi sono array (N-1)D

```
typedef int Vettore[5];
```

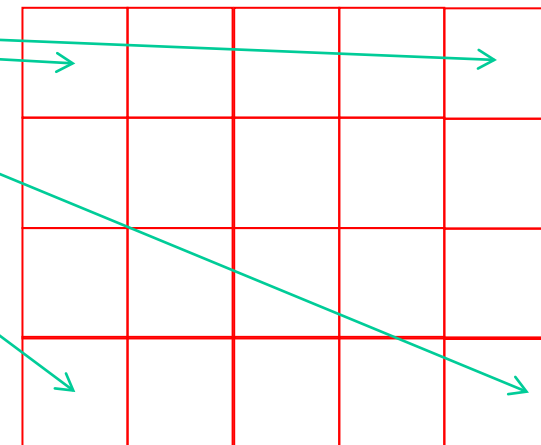


```
typedef Vettore Matrice4Per5[4];
```



```
Matrice4Per5 matrice1;
```

L'i-simo element del vettore  
contiene un puntatore al vettore  
contenente i valori dell'i-sima riga



# Matrici e Doppie puntatori

Il calcolo dello **spiazzamento nelle matrici** richiede di conoscere le dimensioni intermedie

- Tipo `m[R][C]`; /\*N.B.: R righe, C colonne\*/
- `m[i][j]` → accesso al j-esimo elemento della i-esima riga

Per come rappresento un vettore, risulta che la matrice viene salvata mediante un **doppio puntatore**

- $m[i][j] \equiv * ( * (m + i) + j )$
- $\&m[i][j] \approx m + C \cdot i + j$

Serve conoscere sia la dimensione del tipo sia il numero di colonne (`sizeof(Tipo)` e `C`; la "altezza" `R` non serve)

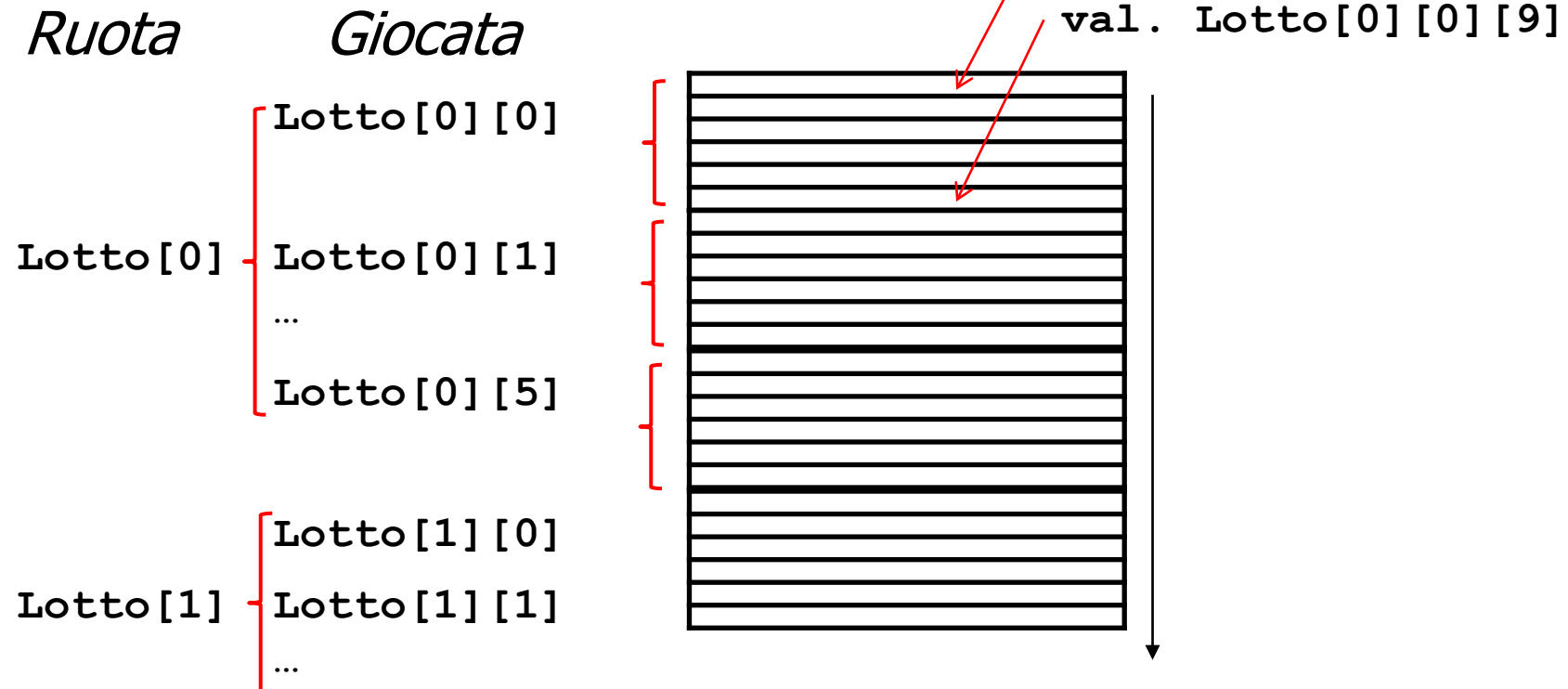
OSS: non serve conoscere le dimensioni effettive (cioè la parte piena dell'array) ma quelle reali (con cui è definita)

# Mappa di Memorizzazione di array 3D

Definire un tipo di dato atto a contenere i numeri estratti nelle ultime 10 giocate su tutte le 11 ruote (vengono estratti 5 numeri per giocata)

```
typedef int Lotto[11][5][10];
```

*Storico*





# Array multidimensionali

Sugli array multidimensionali utilizzeremo puntatori a puntatori...a puntatori

- Tipo `p[X][Y][Z]`
- `p[i][j][k] ≡ *((*(p+i)+j)+k)`
- `&p[i][j][k] ≈ p + Y·Z·i + Z·j + k`

serve conoscere dimensione del tipo, altezza e larghezza (`sizeof(Tipo)`, `Y` e `Z`; la "profondità" `X` non serve)

# Tipi e memoria occupata

- Le variabili occupano in memoria un numero di parole che dipende dal tipo
- Sono allocate in parole di memoria consecutive
- L'operatore `sizeof()` dà il numero di byte occupati da un tipo (o da una variabile):

```
double A[5], *p;
```

```
sizeof(A[2]) → 8
```

```
sizeof(A) → 40
```

```
sizeof(p) → 4
```

```

#include<stdio.h>
int main() {
    int **p, *q, i = 9;
    int vet[10], mat[10][10];
    char c = '0';
    double d = 0.0, *pd;

    p = &q;
    q = &i;
    printf("**p=%d, *q=%d, i=%d", **p, *q, i);

    printf("\nsizeof(c) = %d", sizeof(c));
    printf("\nsizeof(i) = %d", sizeof(i));
    printf("\nsizeof(d) = %d", sizeof(d));
    printf("\nsizeof(vet) = %d", sizeof(vet));
    printf("\nsizeof(mat) = %d", sizeof(mat));
    printf("\nsizeof(p) = %d", sizeof(p));
    printf("\nsizeof(q) = %d", sizeof(q));
    printf("\nsizeof(pd) = %d", sizeof(pd));
    return 0;
}

```

# L'operatore sizeof

"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018\_Informatica\_A\Lez11\puntatori\_doppi.exe"

```
i| **p = 9, *q = 9, i = 9
sizeof(c) = 1
sizeof(i) = 4
sizeof(d) = 8
sizeof(vet) = 40
sizeof(mat) = 400
sizeof(p) = 4
sizeof(q) = 4
sizeof(pd) = 4
Process returned 0 (0x0)    execution time : 0.105 s
Press any key to continue.
```

- Le dimensioni in memoria dipendono dal tipo della variabile
- Le dimensioni dei puntatori (e doppi puntatori) sono quelle della singola cella, indipendentemente dal loro tipo
- Le dimensioni dei vettori contengono tutte le celle allocate per l'array