Quindi con i puntatori possiamo considerare tre possibili valori:

pointer	contenuto o valore della variabile pointer (indirizzo della locazione di memoria a cui punta)
&pointer	indirizzo fisico della locazione di memoria del puntatore
*pointer	contenuto della locazione di memoria a cui punta

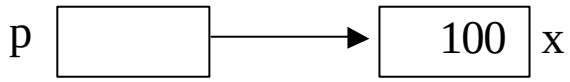
NB. Quando un puntatore viene dichiarato non punta a nulla, o meglio poichè il contenuto di una cella di memoria è casuale fino a che non viene inizializzata ad un valore noto, il puntatore punta ad una locazione di memoria casuale, che potrebbe non essere accedibile dal processo. Cosi':

```
int *ip;
*ip=100;
```

scrive il valore 100 in una locazione qualsiasi: Grave errore.
L'utilizzo corretto e' il seguente; prima di scrivere un valore nella locazione di memoria puntata dal puntatore ci assicuriamo che tale locazione di memoria appartenga al nostro programma. Ciò e possibile in due modi:

1) il primo modo consiste nell'assegnare al puntatore l'indirizzo di una variabile del nostro programma, quindi scriveremo il valore nella variabile puntata.

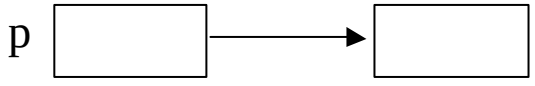
```
int *ip;
int x;
ip=&x;
*ip=100;        scrivo 100 in x
```



2) il secondo modo consiste nel farci riservare dal sistema operativo una porzione di memoria, salvare l'indirizzo di questa porzione di memoria nel nostro puntatore (ovvero far puntare il puntatore a quell'area di memoria) in modo che i successivi riferimenti alla locazione di memoria puntata dal puntatore lavorino su questa area di memoria che ci è stata riservata. Esiste una funzione di libreria standard malloc(), che permette l'allocazione dinamica della memoria; e' definita come:

void *malloc (int number_of_bytes).

Ad es.: int *p; p= (int *) malloc(sizeof(int)); assegna a p spazio per un int.



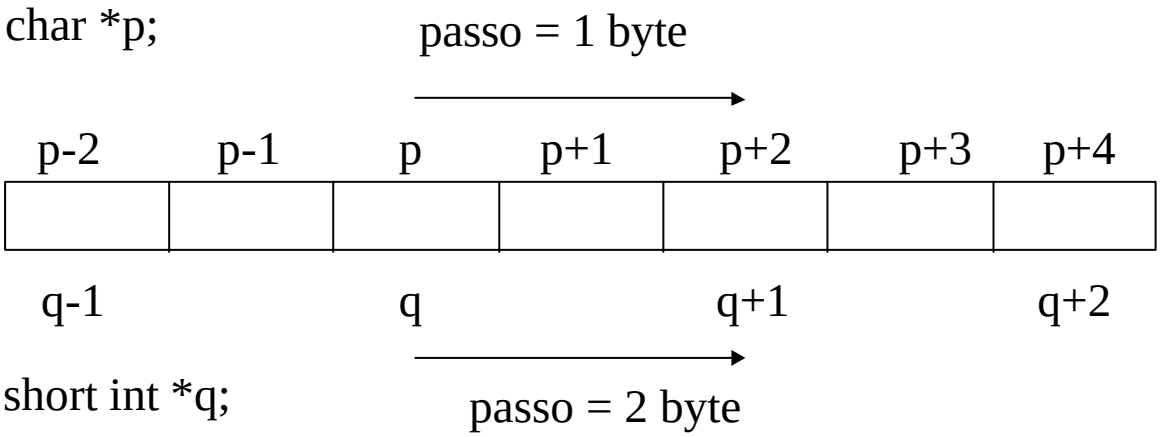
Aritmetica degli indirizzi

Si possono fare operazioni aritmetiche intere con i puntatori, ottenendo come risultato di far avanzare o riportare indietro il puntatore nella memoria, cioè di farlo puntare ad una locazione di memoria diversa.
Ovvero con i puntatori è possibile utilizzare due operatori aritmetici + e - , ed ovviamente anche ++ e --.

Il risultato numerico di un'operazione aritmetica su un puntatore è diverso a seconda del tipo di puntatore, o meglio a seconda delle dimensioni del tipo di dato a cui il puntatore punta. Questo perchè **il compilatore interpreta diversamente la stessa istruzione p++ a seconda del tipo di dato**, in modo da ottenere il comportamento seguente:

- **Sommare un'unità ad un puntatore significa spostare in avanti in memoria il puntatore di un numero di byte corrispondenti alle dimensioni del dato puntato dal puntatore.**

Ovvero se p è un puntatore di tipo puntatore a char, char *p; poichè il char ha dimensione 1, l'istruzione p++ aumenta effettivamente di un'unita il valore del puntatore p, che punterà al successivo byte.
Invece se p è un puntatore di tipo puntatore a short int, short int *p; poichè lo short int ha dimensione 2 byte, l'istruzione p++ aumenterà effettivamente di 2 il valore del puntatore p, che punterà allo short int successivo a quello attuale.



In definitiva, ogni volta che un puntatore viene incrementato passa a puntare alla variabile successiva che appartiene al suo tipo base, mentre un decremento lo fa puntare alla variabile precedente. Quindi **incrementi e decrementi di puntatori a char fanno avanzare o indietreggiare i puntatori a passi di un byte**, mentre incrementi e decrementi di puntatori a dati di dimensione K fanno avanzare o indietreggiare i puntatori a passi di K bytes.

Il caso dei **puntatori a void** *void *p* **viene trattato come il caso dei puntatori a char**, cioè incrementato o decrementato a passi di un byte.

Sono possibili non solo operazioni di incremento e decremento (++ e --) ma anche somma e sottrazione di dati interi (char, int, long int) che comunque vengono effettuati sempre secondo le modalità di incremento decremento a passi di dimensioni pari alla dimensione del dato puntato dal puntatore.
es:

```
long int *p;
.....
p = p +9;    oppure    p +=9;
```

queste istruzioni fanno avanzare il puntatore p di 9*sizeof(long int)=9*4=36 byte.

```
long int *p;
char i;
i = 9;
.....
p = p +i;    oppure    p +=i;
```

Anche queste istruzioni fanno avanzare il puntatore p di i*sizeof(long int)=9*4=36 byte.

Non sono consentite altre operazioni oltre all'addizione e sottrazione di interi. **Non è possibile sommare sottrarre moltiplicare o dividere tra loro dei puntatori.**
Ad es, int *ptr, *ptr1; ptr = ptr + ptr1; da' errore in compilazione di tipo "invalid operands to binary".

Puntatori e Array monodimensionali.

Riassumendo, i puntatori sono delle variabili che contengono un indirizzo in memoria, e si dice che il puntatore "punta a quell'indirizzo". Il puntatore può essere di tipo "puntatore ad un tipo" oppure di tipo generico "puntatore a void". Il puntatore consente di accedere alla memoria a cui punta mediante l'operatore [].

Se p è un puntatore ad un certo *tipo* (*tipo* *p;) e contiene un certo valore addr, ovvero punta ad un certo indirizzo addr, l'espressione p[k] accede all'area di memoria che parte dal byte addr+k*sizeof(*tipo*) ed ha dimensione sizeof(*tipo*), trattandola come se fosse una variabile di tipo *tipo*.
Nel caso di un puntatore a void, viene considerata 1 la dimensione del dato puntato, cioè il puntatore punta ad un byte.

Anche per i vettori (gli array monodimensionali di dati di tipo *tipo*) l'accesso ai dati avviene secondo queste modalità vet[k], perchè in C, il nome di un array è TRATTATO dal compilatore come un puntatore *COSTANTE* alla prima locazione di memoria dell'array stesso . *(costante significa che non posso assegnare qualcosa al nome del vettore ma solo alle sue posizioni)*

A differenza dei vettori però, l'area di memoria a cui il puntatore punta non viene allocata staticamente come per i vettori a partire dalla loro definizione (che contiene le dimensioni del vettore stesso) , ma può essere allocata dinamicamente mediante alcune funzioni che richiedono al sistema operativo di riservare memoria e restituire un indirizzo a quell'area di memoria allocata, oppure può non essere allocata affatto.

Comunque la differenza principale tra puntatori e vettori e' che **i puntatori sono variabili che contengono un indirizzo, e questo contenuto (indirizzo) può essere cambiato per puntare ad un'area di memoria diversa**.
Invece **per i vettori**, anche se il loro nome è trattato come un puntatore *(ma costante)* all'inizio del vettore stesso**, non esiste una variabile (una cella di memoria) in cui è mantenuto l'indirizzo della prima locazione del vettore, e quindi questo indirizzo non può essere modificato**, è costante.
Ad es: il compilatore da' errore qui: int vet[100]; vet++; vet =10;

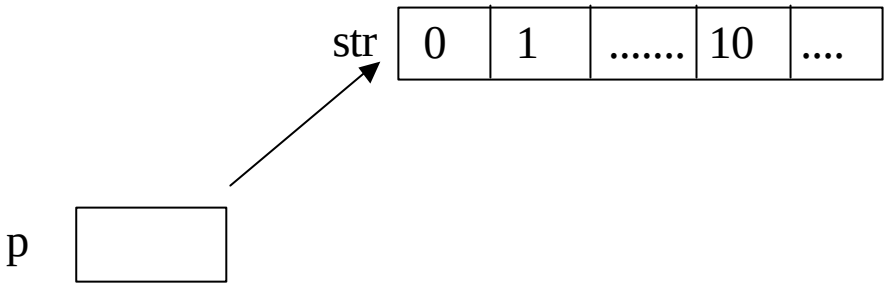
A partire da queste considerazioni sottolineamo che:

definite due variabili, un vettore int vet[100] ed un puntatore int *p

- mentre **è possibile fare riferimento all'indirizzo di un puntatore &p** ottenendo un indirizzo che punta all'indirizzo p, cioè che indica a che indirizzo sta la variabile p,
- nel caso dei vettori **l'indirizzo del vettore &vet è l'indirizzo che punta all'inizio del vettore stesso**, ovvero è l'indirizzo del primo elemento del vettore stesso. Ovvero le seguenti espressioni indicano tutte la stessa cosa, l'inizio del vettore: **vet , &(vet[0]) , &vet**

char str[100];
char *p;

p = str;



- **con questo assegnamento viene assegnato al puntatore p l'indirizzo della prima locazione di memoria del vettore. Da questo momento in avanti potremo accedere ai dati del vettore sia tramite str, sia tramite p esattamente negli stessi modi,**
  - o tramite l'indicizzazione tra parentesi quadre,
  - o tramite l'aritmetica dei puntatori.

str[10] *(str+10) p[10] *(p+10) sono tutti modi uguali per accedere alla 11-esima posizione del vettore str puntato anche da p.

es:

```
str[10] = 'A';
printf( "str[10]=%c\n",  str[10] );
printf( "str[10]=%c\n",  p[10] );
printf( " str[10]=%c\n",  *(p+10)   );
```

le printf stampano tutte il carattere A.

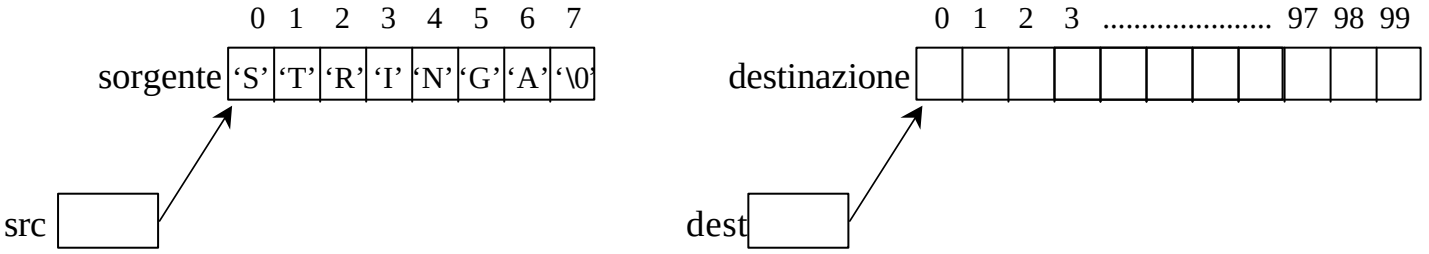
Un esempio classico di uso di puntatori: copia di una stringa in un'altra.

vediamo un esempio di applicazione dei puntatori, la copia di una stringa, confrontandola con un'implementazione che non usa puntatori.

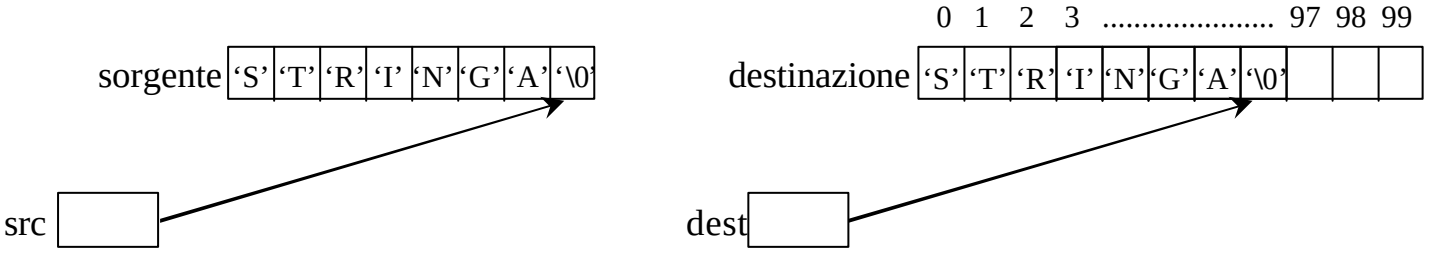
```
char sorgente[]="STRINGA";
char destinazione[100];
int i=0;

for (i=0; ; i++){
     destinazione[i]=sorgente[i];
     if(  ! destinazione[i] )
              break;
}
```

```
char sorgente[]="STRINGA";
char destinazione[100];
char *src, dest;

src=sorgente;
dest=destinazione;
while( *(dest++) = *(src++) )   ;
```

situazione all'inizio del loop while



situazione alla fine del loop while



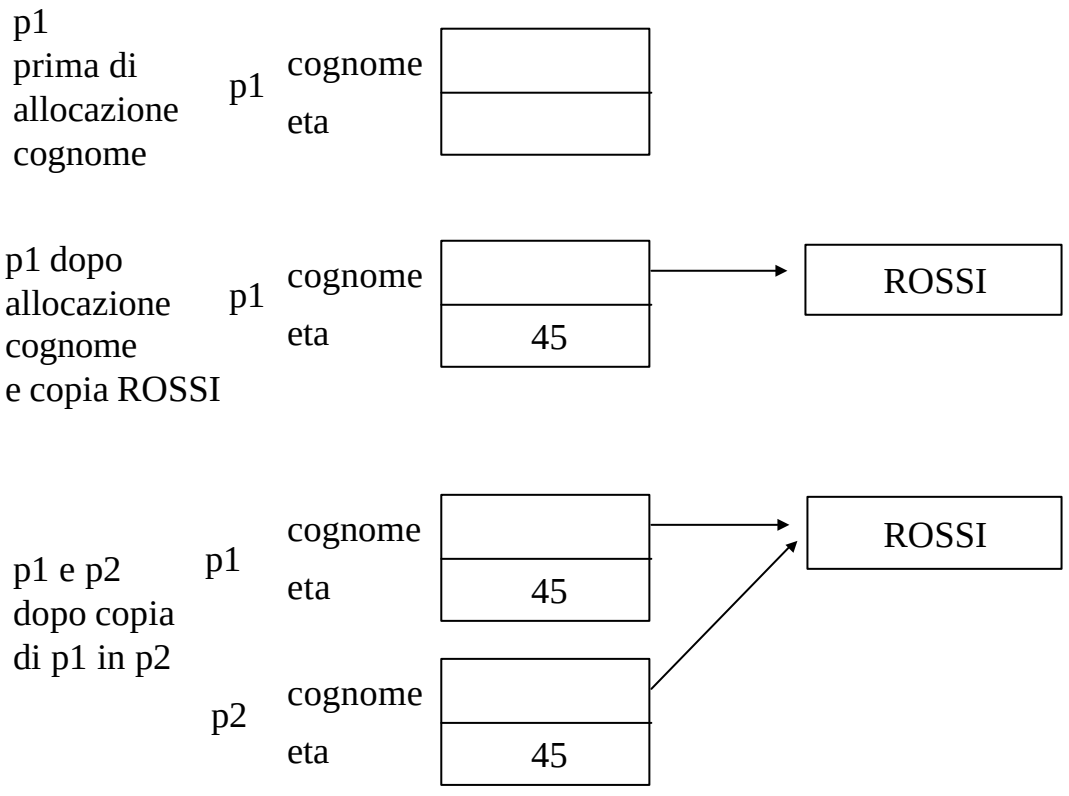
Un errore classico con Puntatori dentro le struct.

```
typedef struct {
        char *cognome;
        int     eta;
        }   persona;
```

Si noti che cognome è un puntatore e non un vettore.

```
/persona p1, p2;

/* alloco spazio per il cognome di p1*/
p1.cognome = (char*) malloc( 50 * sizeof(char));
/* copio ROSSI nel cognome di p1 */
strcpy( p1.cognome ,  "ROSSI" );
p1.eta = 45;

/* copio la struttura p1 nella struttura p2 */
p2 = p1;
```

risultato: **disastro, le due variabili p1 e p2 hanno una parte in comune**!!



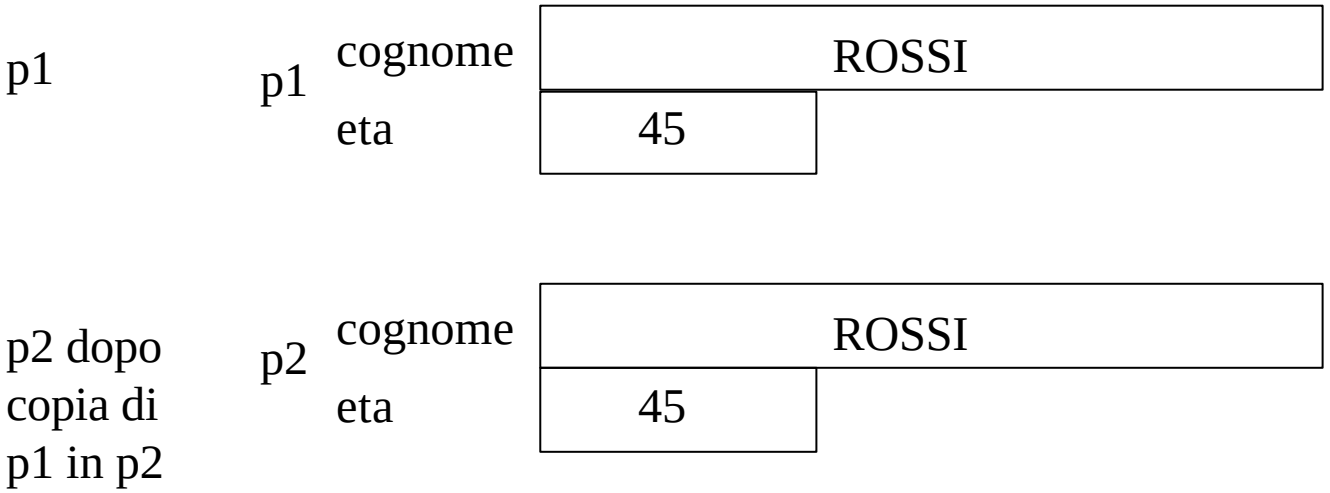
La situazione sarebbe stata corretta se io avessi definito la struct persona con un vettore già allocato invece che con un puntatore a char

```
typedef struct {
      char cognome[50];
      int    eta;
      }   persona;
```

Si noti che in questo modo cognome ha già spazio allocato.

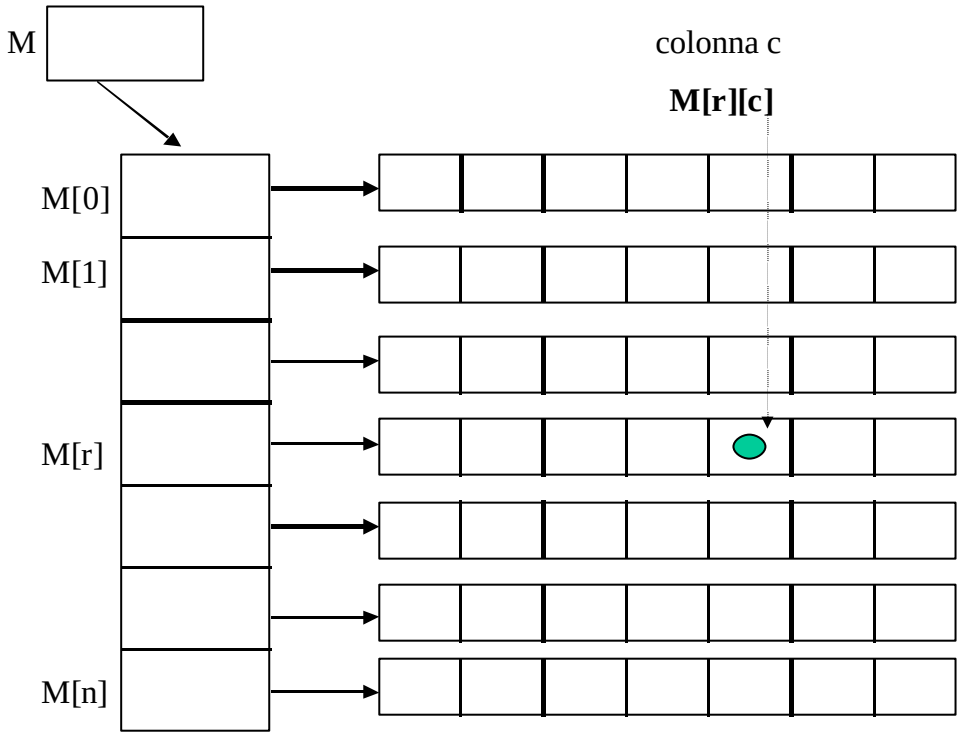
```
/persona p1, p2;

/* copio ROSSI nel cognome di p1 */
strcpy( p1.cognome ,  "ROSSI" );
p1.eta = 45;

/* copio la struttura pers1 nella struttura p2 */
p2 = p1;
```

risultato: OK !



Vettori di Puntatori.

Vogliamo costruire dinamicamente un dato M che sia un **array di n puntatori ad int**, per poi passare ad allocare, per ciascuno dei puntatori ad int dell'array, spazio sufficiente ad **m** int, ottenendo in definitiva un array di vettori di interi.



Se gli elementi dell'array dinamico M sono puntatori ad interi, il nostro puntatore M sarà un puntatore di puntatore ad int, cioè sarà un **int* *M**;
L'elemento intero che sta nella posizione c-esima dell'r-esimo vettore di interi verrà acceduta mediante un'espressione: **M[r][c]**

i = M[r][c]

come vedete questo modo è simile (fatta salva la differenza nell'uso delle parentesi quadre al posto delle parentesi tonde) al modo che si usa in pascal per accedere ad una matrice bidimensionale.

In effetti questa struttura dati è spesso usata come matrice in C, quando non sono note a priori le dimensioni della matrice da realizzare, e si preferisce non sovradimensionarla con un'allocazione statica.

```
/* allocazione della struttura dati */
n ed m sono state lette ad es da
int n,m    r,c;
int* *M;

/* valori ad n ed m assegnati run-time */
n  = .....
m = .....
if (  ! (M = malloc( n * sizeof(int*) ))     ) exit(1);
for ( r=0; r<n; r++)
    if (  ! (M[r] = malloc( m * sizeof(int) ))     ) exit(1);
    else
          for ( c=0; c<m; c++)
                M[r][c] = valore.....
```

Il vettore di puntatori qui visto viene utilizzato in un caso particolare, in cui tutti i vettori allocati hanno stessa dimensione. In generale invece è possibile per ogni puntatore allocare un vettore di dimensioni diverse.

Inizializzazione statica di un Vettore di Puntatori.

E' inoltre possibile costruire un dato M che sia un **array di n puntatori**, non solo dinamicamente, ma anche staticamente al momento della dichiarazione.
La seguente istruzione **crea ed inizializza** un array di puntatori a char, cioè un array di stringhe di lunghezza diversa (nomi è un vettore di puntatori a char).

```
char *nomi [] = {
        "paola",
        "marco",
        "giovanna"
};
```

Puntatori a Strutture.

E' possibile utilizzare puntatori che puntano a dati di tipo strutturato come le struct. La definizione di questo tipo di puntatore ricalca esattamente la definizione generale, cioè prevede di **definire il tipo di dato**, e **poi di definire il puntatore a quel tipo di dato**.

Es.

```
struct PUNTO {char nome[50]; int x; int y; int z; };  /* definiz. tipo */
struct PUNTO *ptrpunto;        /* dichiaraz. puntatore a var. strutturata */
struct PUNTO punto;            /* dichiaraz. variabile di tipo strutturato */

ptrpunto = &punto;         /* assegno l'indirizzo della var. punto */
```

E' necessario **per semplicita'** definire un operatore che permetta di accedere ai campi della struttura direttamente dal puntatore.

In C esiste **l'operatore** -> che permette l'accesso ad un campo della struttura puntata.

Nell'esempio, ptrpunto-> x = 102; accede in scrittura al campo x della struttura di tipo PUNTO puntata da ptrpunto.

Quindi l'operatore -> equivale ad usare in maniera combinata l'operatore * facendolo precedere al nome del puntatore per accedere al dato strutturato, seguito dall'operatore **.** per accedere al campo di interesse.

Nell'esempio, l'istruzione

ptrpunto-> x = 102;

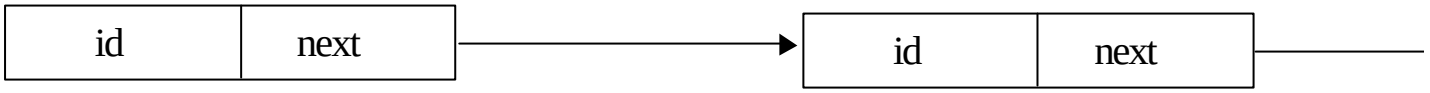
equivale all'istruzione:

(*ptrpunto).x = 102;

Puntatori in Strutture.

Ora che abbiamo introdotto i puntatori, vediamo perchè è stato reso possibile utilizzare per le struct quella sintassi particolare, con il nome della struct prima della sua descrizione.

Vogliamo definire una struttura dati che serva come nodo di una lista semplice, una struttura che contenga cioe' un intero (il dato della lista) ed un puntatore all'elemento successivo della lista.



Il problema è **come definire un puntatore ad un tipo di dato mentre ancora il tipo di dato non è stato definito**. Vediamo due esempi alternativi di come procedere, in un caso utilizzando la typedef, nell'altro no.

```
struct nodolista {
      int   id;
      struct nodolista *next;
      };

struct nodolista nodo;
```

```
typedef struct nodolista {
      int   id;
      struct nodolista *next;
      } NODOLISTA;

NODOLISTA nodo;
```

Array multidimensionali.

Un array multidimensionale è un array, i cui elementi sono a loro volta degli array, i cui elementi ecc.. fino ad esaurire le dimensioni volute.
Facciamo riferimento per semplicità al caso degli array bidimensionali, **le matrici**.
Le matrici in C sono locazioni di memoria contigue che contengono i dati memorizzati per righe (come in pascal, mentre il fortran memorizza per colonne).

Quindi la matrice, anche se viene pensata come un oggetto a due dimensioni del tipo ad es. (3 righe per 4 colonne), cioè come un vettore di righe, ciascuna delle quali è un vettore,

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	**r,c**	2,2	2,3

è in realtà un oggetto disposto linearmente in memoria, riga dopo riga, come un vettore

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	**r,c**	2,2	2,3

Indicizzando righe e colonne a partire da 0, se NR è il numero di righe, ed NC è il numero di colonne, allora l'elemento in posizione (**n,c**) della matrice sta fisicamente nella posizione **r*NC+c** del vettore.

La matrice (es. di int) viene quindi dichiarata come un vettore di NR elementi di tipo vettore di NC interi, ovvero come segue:

```
#define NR 3
#define NC 4
int matrice[NR][NC];
```

Nella dichiarazione la dimensione delle matrici deve essere una costante, perchè il compilatore per effettuare l'accesso all'elemento (r,c) della matrice ovvero all'elemento **r*NC+c** (a partire da dove inizia la matrice) deve conoscere la dimensione delle righe cioè il numero NC di colonne.

L'accesso al dato in posizione r,c della matrice viene effettuato mediante l'operatore [] già usato per i vettori.

Ad es. matrice[1][3]=7; scrive il valore 7 nell'elemento che sta nella seconda riga in quarta colonna.

Analogamente ai vettori, il nome della matrice rappresenta il puntatore al primo elemento della matrice, cioè alla prima riga.

Inizializzazione delle matrici

All'atto della dichiarazione è possibile inizializzare le matrici come di seguito indicato:

```
int matrice [2][3] =
    {
        { 1,2,3 } ,
        { 4,5,6 }
    };
```

L'inizializzazione viene fatta scrivendo tra parentesi graffe i valore di ciascun elemento, separati da virgole. Come si vede, essendo la matrice un vettore di righe, ciascuna riga viene inizializzata in modo analogo, scrivendo tra parentesi graffe i valori di ciascun elementoseparati da virgole.