---

istruzione typedef

Il C permette di definire esplicitamente nomi nuovi per i tipi di dati, tramite la parola chiave **typedef**. L'uso di typedef consente di rendere il codice più leggibile.

Il formato dell'istruzione typedef è il seguente:

typedef tipo nuovo_nome_tipo ;

in questo modo assegnamo al tipo **tipo** il nuovo nome **nuovo_nome_tipo**. Da questo momento in avanti potremo riferirci al tipo di dato **tipo** sia con il nome **tipo** sia con il nome **nuovo_nome_tipo**.

Es:

```
typedef   int   intero; /* ora posso usare intero come tipo invece di int */
intero i;               /* definisce una variabile i di tipo int.          */
```

Tipi di Dati Complessi

Il C mette a disposizione i seguenti dati di tipo complesso:

- **array monodimensionali**, in parte già visti,
- **strutture**, dette **struct**,
- **unioni**, dette **union**,
- **puntatori**.
- **array multidimensionali**,

Ci sarebbe da discutere sul fatto che i puntatori siano un dato di tipo complesso, in quanto in realtà gli indirizzi rappresentano uno dei dati di base per la programmazione a basso livello (assembler) per tutte le moderne architetture dei calcolatori. Col termine "dato complesso" intendiamo perciò indicare dati il cui uso richiede attenzione.

Strutture (struct)

Le strutture in C sono simili ai records in Pascal. Ad esempio vediamo come è definita un tipo di dato struct chiamato **luogo** che vuole rappresentare un punto geografico individuato da tre coordinate intere x, y, z ed un nome rappresentato da una stringa di 30 caratteri:
Vediamo come si definisce il tipo di dato "luogo":

```
struct luogo {
        char nome[30];
        int  x;
        int  y;
        int  z;
};
```

Una volta definito il tipo di dato, possiamo dichiarare una variabile di quel tipo, premettendo però la keyword **struct**:

```
struct luogo monte_bianco;
```

Una sintassi diversa permette di dichiarare in una sola istruzione il tipo di dato struct **luogo** ed alcune variabili di quel tipo.

```
struct luogo  {
        char nome[30];
        int  x;
        int  y;
        int  z;
}  monte_bianco , monte_rosa;
```

In questo caso, qualora solo queste variabili debbano essere dichiarate, ovvero qualora non ci sia altrove necessità di conoscere il tipo di dato "**luogo**", l'identificatore "**luogo** " può essere omesso, e si avrà:.

```
struct  {
        char nome[30];
        int  x;
        int  y;
        int  z;
}  monte_bianco , monte_rosa;
```

Anche con i dati di tipo struct possiamo ricorrere alla typedef per definire nuovi nomi per i tipi di dato. Ad es:

```
typedef struct  {
        char nome[30];
        int    x;
        int    y;
        int    z;
}   LUOGO;
```

In questo modo abbiamo definito un nome che può essere utilizzato per dichiarare delle variabili di tipo LUOGO in tutto e per tutto uguale al precedente tipo di dato **luogo**, che però in questa dichiarazione non compare.
Ora potremo dichiarare variabili omettendo la keyword **struct**.

```
LUOGO milano.
```

E' possibile comunque mantenere i due identificatori:

```
typedef struct luogo  {
        char nome[30];
        int    x;
        int    y;
        int    z;
}   LUOGO;
```

e definire due variabili di tipo uguale pure se con nomi diversi, ed il compilatore li tratta come dati di un solo tipo, permettendo ad es. gli assegnamenti.

```
struct luogo     Cesena;
LUOGO            Cesenatico.
Cesena = Cesenatico;
```

Accesso ai campi delle struct

Per accedere ai membri (o campi) di una struttura struct, il C fornisce l'operatore punto "**.**" .
Ad esempio:

```
Cesena.nome  accede alla stringa nome del dato Cesena di tipo luogo.
```

Inizializzazione delle struct

Una struttura struct puo' essere pre-inizializzata al momento della dichiarazione, mettendo le costanti che inizializzano uno per uno tutti i campi della struct tra parentesi graffe.

```
struct luogo rimini = { "RIMINI" , 100, 139 ,  10};
```

E possibile anche annidare le struct ed inizializzarle assieme, come nell'esempio qui di seguito:

```
#include <stdio.h>
void main() {
    struct luogo {
        char nome[30];
        int     x;
        int     y;
        int     z;
    };

    struct coppia_di_luoghi  {
        struct luogo    luogo1;
        struct luogo    luogo2;
    };

    struct coppia_di_luoghi  Rimini_Milano = {
        { "RIMINI" , 100, 139 ,  10} ,
        { "MILANO" , 80, 170 , 50 }
    };

    printf("luogo1=%s\n"  ,  Rimini_Milano.luogo1.nome);
}
```

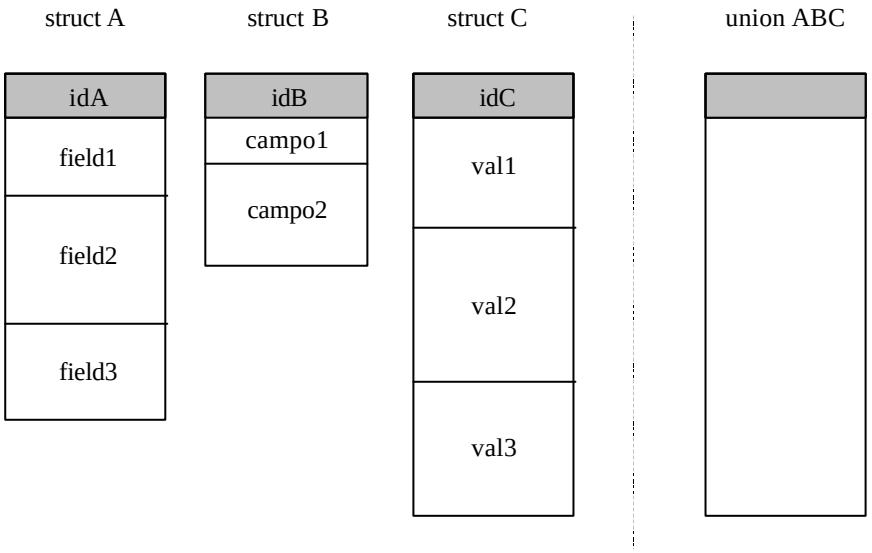
questo programma ha come effetto di stampare la stringa "RIMINI"

Unioni (union)

Un'union e' una variabile equivalente al record con varianti del Pascal, che puo' tenere (in momenti diversi) oggetti di diversa dimensione e tipo, che sono quindi posizionati sulla stessa area di memoria, cioè sovrapposti.
Spesso i diversi oggetti sono delle struct, ed hanno tutte un campo, eventualmente con un nome diverso da struct a struct, ma collocato nella stessa posizione in memoria, in cui viene mantenuta una costante che identifica quale struttura è contenuta nell'union in quel momento.
In questo modo, ad es, si può passare dati diversi ad una stessa funzione, e la funzione leggendo questo campo di identificazione della struttura, capisce che genere di struttura è contenuta in quel momento nella union, e può accedere correttamente ai dati della union. Es:

```
struct A { int idA; char field1[10]; char field2[20]; char field3[10] };
struct B { int idB; char campo1[8]; char campo2[10]; };
struct C { int idC; char val1[12]; char val2[25]; char val3[20] };
```

```
union ABC   {
      struct A    a;
      struct B    b;
      struct C    b;
} abc =

  { IS_A,  "field1" ,
   "field2" , "field3"
  };
```

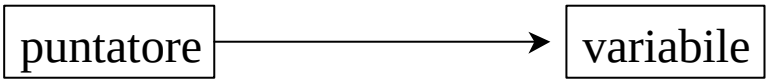


Il compilatore alloca area sufficente a contenere la più grande delle strutture contenute nella union. I campi sono acceduti col nome della struttura contenuta, seguita dal punto come nelle struct e dal nome del campo.
es: `if( abc.a.idA == IS_B ) printf("%s", abc.b.campo2);`

Puntatori

I puntatori sono il segreto della potenza e la flessibilita' del C, perchè:
- sono l'unico modo per effettuare alcune operazioni;
- servono a produrre codici sorgenti compatti ed efficienti, anche se a volte difficili da leggere.

- In compenso, la maggior parte degli errori che i programmatori commettono in linguaggio C sono legati ai puntatori. (sigh)
- Il C utilizza molto i puntatori in maniera esplicita con:
  - vettori;
  - strutture;
  - funzioni.
- In C ogni variabile è caratterizzata da due valori:
  - un indirizzo della locazione di memoria in cui sta la variabile,
  - ed il valore contenuto in quella locazione di memoria, che è il valore della variabile.
- Un puntatore e' un tipo di dato, è una variabile che contiene l'indirizzo in memoria di un'altra variabile, cioè un numero che indica in quale cella di memoria comincia la variabile puntata.



- Si possono avere puntatori a qualsiasi tipo di variabile.
- La dichiarazione di un puntatore include il tipo dell'oggetto a cui il puntatore punta; questo serve al compilatore che deve accedere alla locazione di memoria puntata dal puntatore per sapere cosa troverà, in particolare per sapere le dimensioni della variabile puntata dal puntatore.
- Per dichiarare un puntatore ***p*** ad una variabile di tipo ***tipo***, l'istruzione e':

tipo *p ;

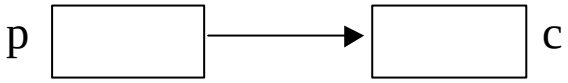
L' operatore & fornisce l'indirizzo di una variabile, perciò l' istruzione
p = & c scrive nella variabile p l'indirizzo della variabile c, ovvero**:**

tipo c, *p; dichiaro una var c di tipo *tipo* ed un puntatore p a *tipo*
p = & c ; assegno a p l'indirizzo di c

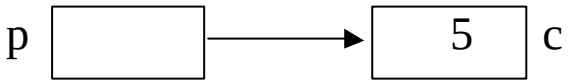
L'operatore * viene detto operatore di indirezione o deriferimento, ed è l'equivalente dell'operatore ^ del Pascal. Quando una variabile di tipo puntatore è preceduta dall'operatore * , indica che stiamo accedendo all'oggetto puntato dal puntatore.
Quindi con *p indichiamo la variabile puntata dal puntatore.

int c, *p; dichiaro una var c di tipo int ed un puntatore p a int

p = &c ; assegno a p l'indirizzo di c



c = 5; assegno a c il valore 5



printf("%d\n", ***p**); stampo il valore puntato dal puntatore p
viene stampato 5

Consideriamo gli effetti delle seguenti istruzioni:

```
int *pointer;  /* dichiara pointer come un puntatore a int */
int x=1,y=2;
(1) pointer= &x;  /* assegna a pointer l'indirizzo di x */
(2) y=*pointer;   /* assegna a y il contenuto dell'int puntato da pointer, x */
(3) x=pointer     /* assegna ad x l'indirizzo contenuto in pointer, serve cast
                     perchè pointer a int è diverso da int */
(4) *pointer=3;   /* assegna alla variabile puntata da pointer il valore 3 */
```

Vediamo cosa succede in memoria. Supponiamo che la variabile x si trovi nella locazione di memoria 100, y nella 200 e pointer nella 1000.
L'istruzione (1) fa si che pointer punti alla locazione di memoria 100 (quella di x), cioè che pointer contenga il valore 100.
La (2) fa si che y assuma valore 1 (il valore di x).
La (3) fa si che x assuma valore 100 (cioe' il valore di pointer).
La (4) fa si che il valore del contenuto di pointer sia 3 (quindi x=3).

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

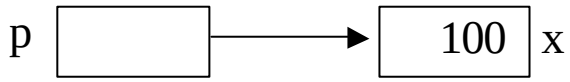
NB. Quando un puntatore viene dichiarato non punta a nulla, o meglio poichè il contenuto di una cella di memoria è casuale fino a che non viene inizializzata ad un valore noto, il puntatore punta ad una locazione di memoria casuale, che potrebbe non essere accedibile dal processo. Cosi':

```
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.

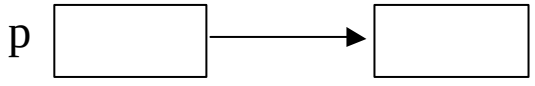
```
int *ip;
int x;
ip=&x;
*ip=100;        scrivo 100 in x
```



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.



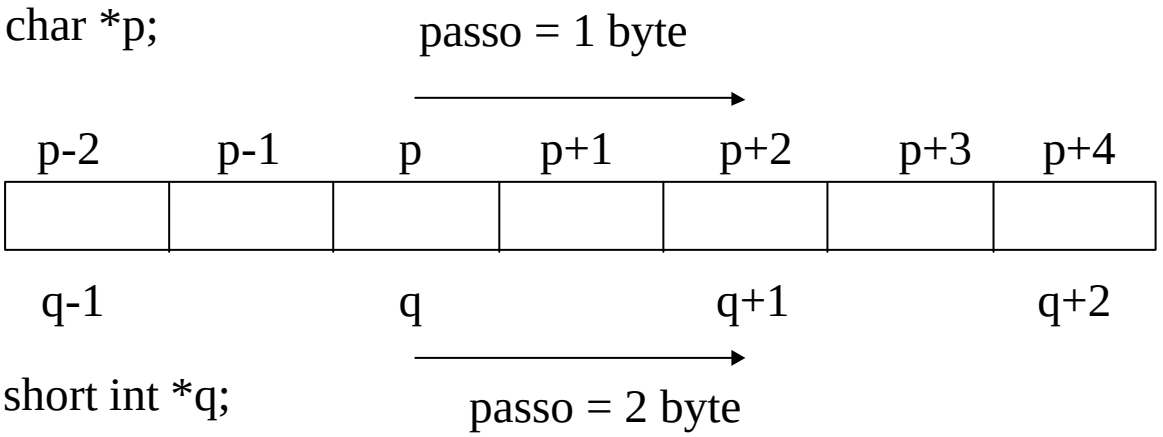
Aritmetica degli indirizzi

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".

Confronto tra Puntatori.

Si possono effettuare confronti tra puntatori, per verificare ad es. quale tra due puntatori punta ad una locazione di memoria precedente, oppure se due puntatori puntanio ad una stessa locazione di memoria.
Ad es. sono valide istruzioni del tipo:

```
int x, y;
int *p, *q;
p=&x;
q=&y;
if(p<q) printf("minore");
```

Le funzioni di Allocazione Dinamica della memoria.

I programmi C suddividono la memoria in 4 aree distinte: codice, dati globali, stack e heap.
Lo heap è un'area di memoria libera che viene gestita da funzioni di allocazione dinamica del C quali la malloc() e la free().
malloc() alloca la memoria (chiede al s.o. di riservare una area di memoria) e restituisce un puntatore a void che punta all'inizio di quest'area di memoria allocata. Se non è disponibile sufficiente memoria restituisce NULL, (l'equivalente a NIL del Pascal) ad indicare che l'allocazione non è stata effettuata. NULL equivale al valore zero, quindi è possibile testare il risultato della malloc() in questo modo:

```
int *p;
p = (int*) malloc( 100*sizeof(int) );  /* chiede di allocare 100 interi */
if( !p )  {
     printf("errore: allocazione impossibile");
    exit(1);
}  else {
     ....  uso la memoria
    free(p);   /* rilascia la memoria allocata v*/
}
```

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;

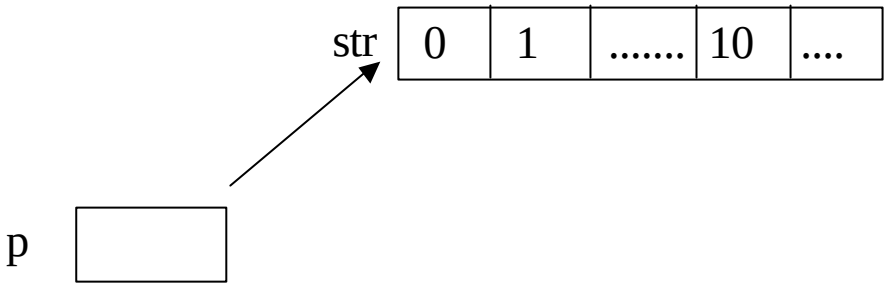
A partire da queste considerazioni sottolineamo che:

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.

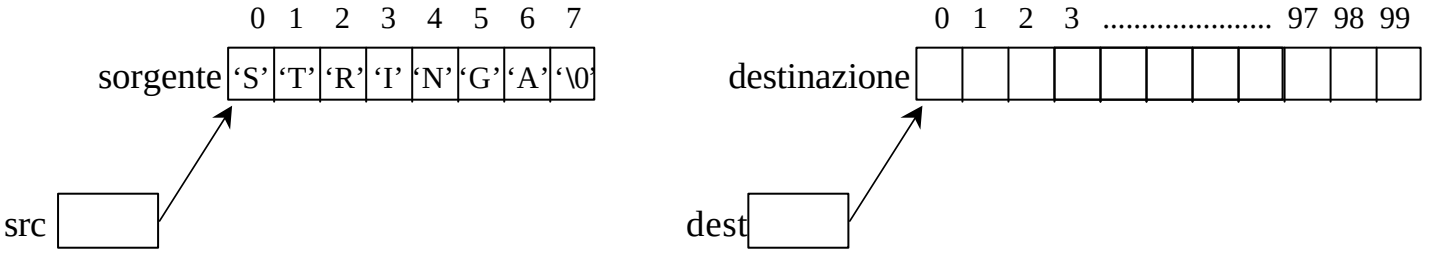
Un esempio classico di uso di puntatori: copia di una stringa in un'altra.

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

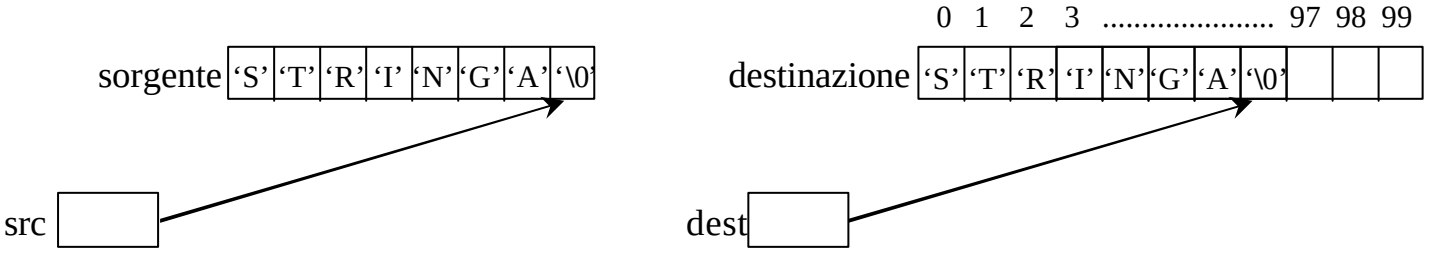
```
char sorgente[]="STRINGA";
char destinazione[100];
int i=0;

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



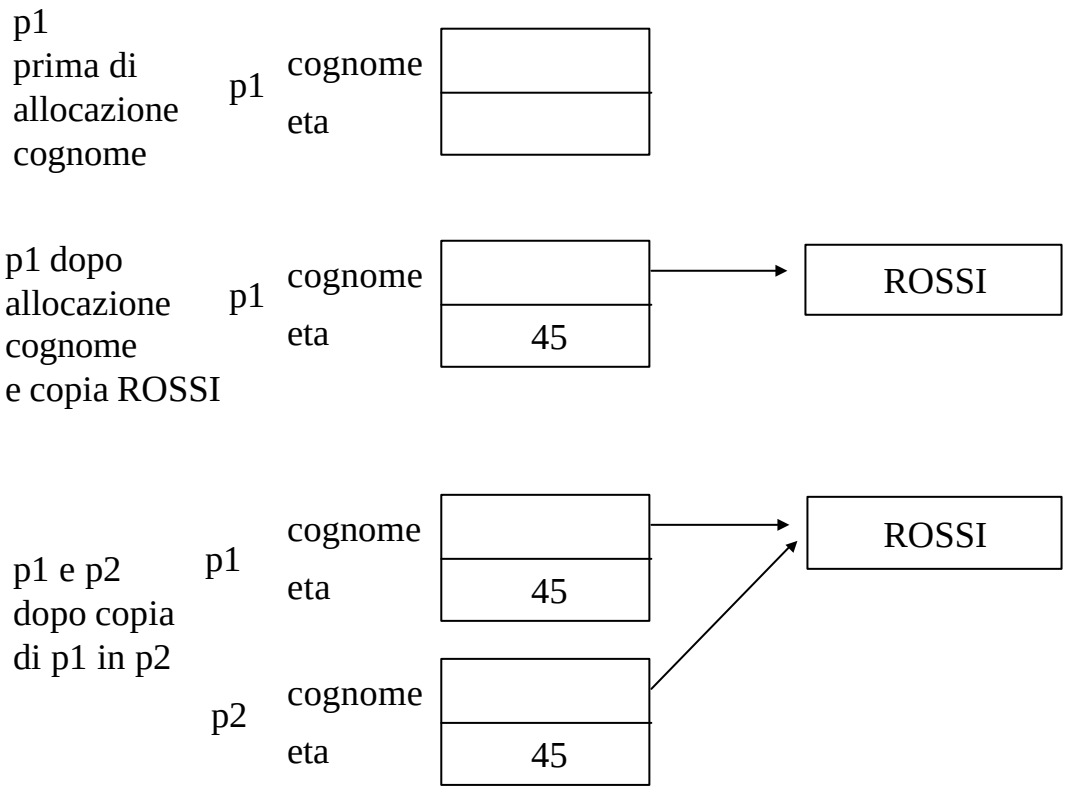
Un errore classico con Puntatori dentro le struct.

```
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**!!



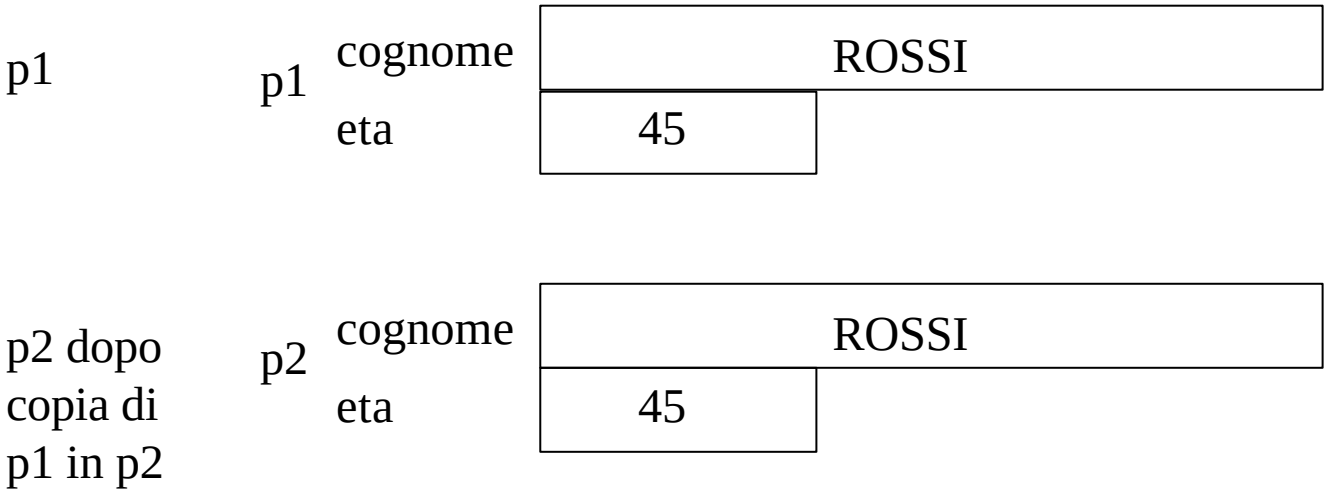
La situazione sarebbe stata corretta se io avessi definito la struct persona con un vettore già allocato invece che con un puntatore a char

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

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

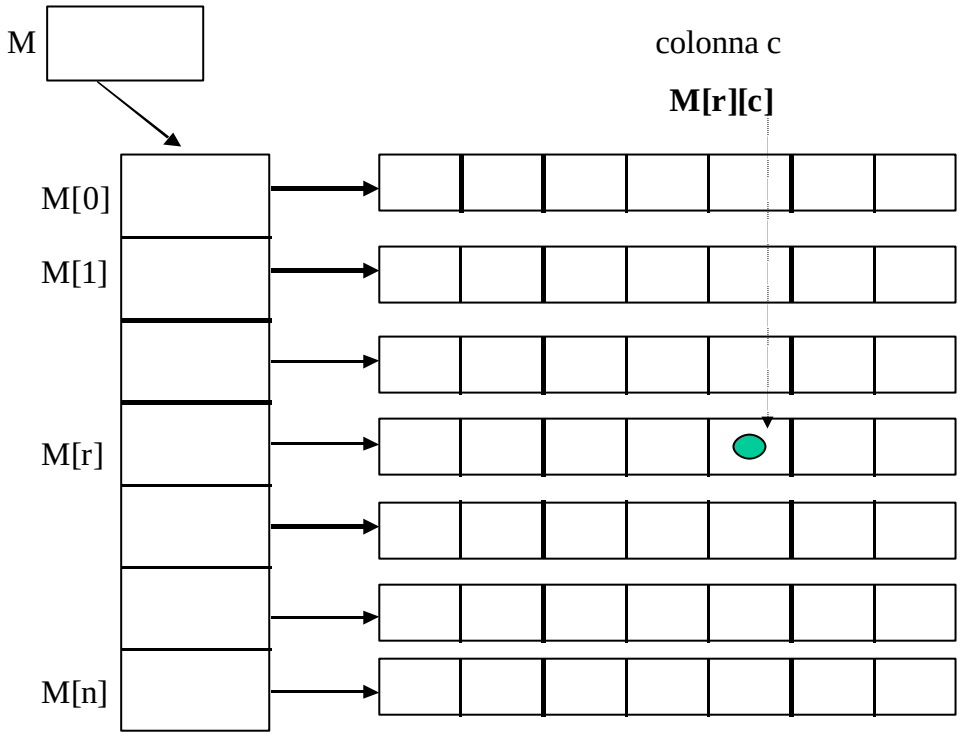
```
/persona p1, p2;

/* 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;

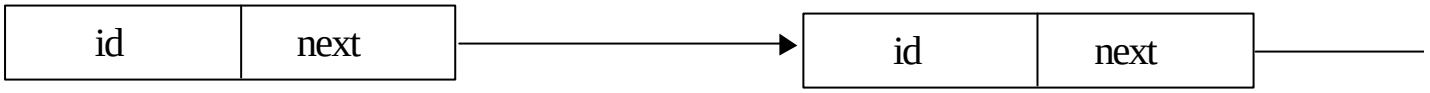
equivale all'istruzione:

(*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.

Funzioni.

- In C non esiste la distinzione che esiste in pascal tra funzioni e procedure, in C sono tutte funzioni, che possono restituire un qualche risultato oppure no, nel qual caso restituiscono void.
- Le chiamate ad ogni funzione in C si effettuano chiamando il nome della funzione seguita dalle parentesi tonde, aperta e chiusa, all'interno delle quali vengono passati i parametri necessari.
- Anche se la funzione non richiede nessun argomento, nella chiamata il suo nome deve essere seguito dalle parentesi tonde aperta e chiusa.
- Il main stesso è una funzione.

La forma generale della definizione di una funzione e':

```
tipo_restituito  nome_funzione (paramdef1, paramdef2, ...)
{
        dichiarazione variabili locali
        istruzioni
}
```

Dove:

- paramdef1, paramdef2, ecc. sono la definizione degli argomenti da passare alla funzione all'atto della chiamata (ad es. int i).
- tipo_restituito è il tipo del dato che viene restituito come risultato dalla funzione.
- Se manca la definizione del tipo di dato restituito dalla funzione ("tipo_restituito"), il C assume che il risultato della funzione e' di tipo int.
- La funzione restituisce il risultato mediante un'istruzione detta return(), che ha forma: **return espressione;** o **return (espressione);**. La return termina la funzione e restituisce il controllo al chiamante.
- Se la funzione non deve restituire nessun valore, si dichiara il tipo_restituito come **void**, e non si esegue la return o la si esegue senza passare alcun argomento (es.: **return;)**.

Es. funzione che somma due valori di tipo int e restituisce un int:

```
int somma(int a, int b)
{
    int sum;
    sum = a+b;
    return(sum);
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31; int risultato;
    risultato = somma(A,B);
    printf("somma= %d\n", risultato);
}
```

Es. funzione che non restituisce alcun valore:

```
void somma(int a, int b)
{
    int sum;
    sum = a+b;
    printf("somma= %d\n", sum);
    /* non serve la return */
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31;
    somma(A,B);
}
```

Modalità di passaggio degli Argomenti delle Funzioni: Dati Semplici

I dati di tipo semplice (char, int, long float, double e **puntatori**), che vengono passati come argomenti ad una funzione, **vengono passati per valore** (la modalità standard del pascal) e **non** per indirizzo (il modo var del pascal).
Ciò significa **che nel momento della esecuzione della funzione, il valore degli argomenti viene copiato sullo stack**, e **la funzione usa (ed eventualmente modifica) questa copia degli argomenti**.
In tal modo il dato originale rimane invariato dopo che la funzione ha restituito il controllo al chiamante.

Se invece **si vuole che un argomento di tipo semplice passato alla funzione conservi le modifiche apportate dalla funzione durante l'esecuzione della funzione, l'argomento deve essere passato per indirizzo**, cioè alla funzione deve essere passato l'indirizzo della variabile, in modo che la funzione (tramite l'indirizzo) modifica la variabile originale.
Es. funzione che somma due valori di tipo int e restituisce un int:

```
void azzera_variabile( int *ptr_a )
{
    *ptr_a = 0;
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23;
    azzera_variabile (  &A );   /* viene passato l'indirizzo */
}
```

Modalità di passaggio degli Argomenti delle Funzioni: Array

I dati di tipo array (vettori, matrici, array di dimensioni maggiori) char, int, long float, double e puntatori), che vengono passati come argomenti ad una funzione, **in C vengono passati per indirizzo, nel senso che "passando l'array per nome si passa l'indirizzo in cui comincia l'array".**

----- Vettori -----

Ciò significa che **quando, all'atto della chiamata, passiamo ad una funzione il nome di un vettore, passiamo l'indirizzo del primo elemento del vettore**, e non tutti i dati del vettore (i 100 interi dell' esempio)

```
void main(void)
{
    int vet[100];
    modifica_vet (vet , 100 );
}
```

di conseguenza la definizione della funzione dovrà contenere come argomento formale il puntatore al tipo di dati del vettore (un puntatore ad int nell'esempio)

```
void modifica_vet( int *v , int size  )
{
    v[0] = 137;     /* modifica conservata fuori dalla funzione */
    v = NULL;       /* annullo l'indirizzo in v, ma questa modifica
                       NON viene mantenuta fuori dalla funzione */
}
```

In tal modo se il vettore passato come argomento viene modificato dalla funzione, cioè **se la funzione modifica il contenuto degli elementi del vettore, queste modifiche sono permanenti, cioè restano nel vettore anche dopo che la funzione è terminata.**

----- Vettori , notazione alternativa ma equivalente-----

Quando l'argomento passato è un vettore, nella definizione della funzione l'argomento può essere indicato o come un puntatore oppure in un modo alternativo, come segue:

```
void modifica_vet( int v[]  , int size  )
{
      v[0] = 137;     /* modifica conservata fuori dalla funzione */
      v = NULL;       /* annullo l'indirizzo in v, ma questa modifica
                         NON viene mantenuta fuori dalla funzione */
}
```

Questa notazione **int v[]** vuole indicare che v è un vettore di interi di dimensione sconosciuta (non si sa di quanti interi è composto il vettore).

Comunque le due notazioni int *v e int v[] sono assolutamente equivalenti, entrambi i parametri vengono trattati come puntatori.

----- Matrici -----

Analogamente ai vettori, **quando, all'atto della chiamata, passiamo ad una funzione il nome di una matrice, passiamo l'indirizzo del primo elemento della matrice, quindi le modifiche sui dati della matrice vengono conservate dopo l'uscita dalla funzione.**

```
void main(void)
{    int mat[10][20];
      modifica_mat ( mat  );
}
void modifica_mat( int m[][20]  )
{    m[1][0] = 137;       /* modifica conservata fuori dalla funzione */
      m = NULL;           /* questa modifica  NON viene mantenuta  */
}
```

Analogamente ai vettori, la definizione della funzione dovrà contenere un puntatore ad array di 20 (= num. colonne) interi.

Notare: si passa la dimensione delle righe, per calcolare l'indice r*NC+c

Modalità di passaggio degli Argomenti delle Funzioni: Strutture

I dati di tipo **struct** possono venire passati alle funzioni sia per valore sia per puntatore, esplicitamente.

```
struct point {
        int         x
        int         y;
}

/* copia i valori di p1 nella struttura puntata da pp2 */
void copia_e_modifica_struct( struct point p1, struct point *pp2  )
{
        pp2->x  = p1.x  + 10;              /* modifiche permanenti */
        pp2->y  = p1.y   + 10;
}

void main(void)
{
        struct point p1 = {   14 , 27 };
        struct point p2;
        copia_e_modifica_struct( p1, & p2  );
}
```

OSS: Se la struttura da passare è molto grande, è bene passarla per puntatore, per non sovraccaricare lo stack, soprattutto quando le chiamate sono molto annidate.

Tipi di dato restituiti dalle Funzioni

Le funzioni possono restituire:

- dati di tipo semplice, come char, int, long, float, double, puntatori a void, o puntatori a qualche tipo di dato,
- ma anche strutture (struct) o puntatori a struct.

All'atto della chiamata, il valore restituito da una funzione:

```
double somma( double f, double g);
```

- può essere utilizzato come espressione booleana,

```
if ( somma(a,b) > 100.3 ) .....
```

- può essere utilizzato come membro di destra in un'istruzione di assegnamento,

```
f = somma(a,b);
```

- oppure può non essere considerato affatto.

```
somma(a,b);
```

Vediamo un esempio di restituzione di una struct.

```
struct point {   int    x;     int    y;     };

struct point crea_point( int x, int y  )
{
        struct point p;
        p.x = x;
        p.y = y;
        return p ;
}
void main(void)
{
        struct point p1;
        int x=21 , y = -10987;
        p1 = crea_point( x, y );
        printf ( "p1.x=%d p1.y=%d \n" , p1.x, p1.y );
}
```

Considerazioni sui Puntatori restituiti dalle Funzioni

Una funzione può restituire un puntatore ad un qualche tipo di dato, ma la correttezza nell'uso di questo puntatore si può fare, dipende da come lo spazio in memoria è allocato. Cioè:

esempio corretto: p è allocato dinamicamente quindi, anche se p è una variabile locale, lo spazio allocato sopravvive alla terminazione della funzione

```
int *alloca_vettore_1(  int size )
{
    int *p;
    p = malloc(size*sizeof(int));
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ...  usa v ....
}
```

---

esempio sbagliato: p è una variabile locale, lo spazio allocato non sopravvive alla terminazione della funzione

```
int *alloca_vettore_2(  int size )
{
    int p[1000];
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ... usa v .....
}
```

---

esempio corretto: vet_globale è una variabile globale, quindi sopravvive alla terminazione della funzione, ma cosìsia chiaro

```
int vet_globale[10000];

int *spazio_pre_allocato( void )
{
    return vet_globale ;
}
```

```
void main(void)
{
    int *v;
    v = spazio_pre_allocato();
    ...  usa v ....
}
```

Il Prototipo delle funzioni

Il prototipo o dichiarazione della funzione è un'istruzione che ripete l'intestazione della funzione, senza il codice.
es:

```
double somma( double a, double b) ;          dichiarazione

void main( void)
{
      double A=10 , B=29,  C;
      C = somma(A,B);                        chiamata
}

double somma( double a, double b)            definizione
{     return a+b ;     }
```

Il prototipo serve a due scopi:

1) **dare visibilità alla funzione quando il luogo in cui è chiamata precede il luogo in cui è definita** (se stiamo sullo stesso file), altrimenti il compilatore non sa cos'è il simbolo nome della funzione. L'esempio qui sopra illustra questo caso.
   Se il compilatore trova un simbolo nuovo seguito da parentesi tonde, lo tratta come una funzione che restituisce un valore intero.
2) **informare il compilatore su come deve trattare il dato restituito dalla funzione, e su come passare i dati agli argomenti della funzione.** Quindi **il prototipo e la definizione della funzione devono essere coerenti**, altrimenti il main (nell'esempio) **tratterebbe i dati scambiati in modo diverso da come la funzione somma si aspetta**, generando errori. **Il compilatore si accorge di un'inconsistenza tra prototipo e definizione della funzione solo se entrambe stanno su uno stesso modulo**. Se stanno su due moduli diversi il compilatore non se ne accorge e non ci avvisa.