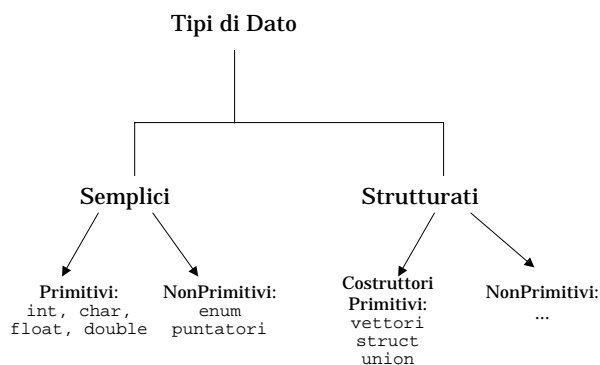


## Tipi di Dato Strutturati



### Tipi di dato strutturati:

I dati strutturati (o strutture di dati) sono ottenuti mediante composizione di altri dati (di tipo semplice, oppure strutturato).

### Tipi strutturati in C:

- **vettori** (o array)
- **record** (struct)
- **record varianti** (union)

## Vettori

Un vettore e' un insieme **ordinato** di elementi **tutti dello stesso tipo**.

### Caratteristiche del vettore:

- **omogeneita`**
- **ordinamento** ottenuto mediante dei valori interi (**indici**) che consentono di accedere ad ogni elemento della struttura.

0	X
1	Y
2	Z
..	..
..	..
n-1	..

## Definizione di Vettori in C

Nel linguaggio C per definire vettori, si usa il costruttore di tipo [ ].

### Sintassi:

```
<id-tipo> <id-variabile> [<dimensione>];
```

dove:

- **<id-tipo>** e' l'identificatore di tipo degli elementi componenti
- **<dimensione>** rappresenta il numero degli elementi componenti (e' una **costante** intera)
- **<id-variabile>** e' l'identificatore della variabile strutturata cosi` definita

### Esempio:

```
int V[10]; /* vettore di dieci elementi interi */
```

☞ La **dimensione** (numero di elementi del vettore) deve essere una costante intera, nota al momento della dichiarazione.

```
int N;  
char V[N]; ---> e' sbagliato!!!
```

## Vettori

### Indici:

- e' possibile riferire una singola componente specificando l'**indice** i corrispondente

V[i]

- L'indice deve essere di tipo **integral type** (cioe', int, char o enum).
- se N e' la dimensione, il **dominio degli indici** e' **[0,N-1]**

### Ad esempio:

```
int A[3];
```

0	
1	
2	

## Vettori

### Operatori:

In C **non esistono operatori specifici per i vettori**; per operare sui vettori è necessario operare singolarmente sugli elementi componenti (coerentemente con il tipo ad essi associato).

☞ **non e' possibile** l'assegnamento diretto tra vettori:

```
int V[10], W[10];
...
V=W; /* e` scorretto! */
```

☞ Non e' possibile leggere (o scrivere) un intero vettore (a parte, come vedremo, le *stringhe*); occorre leggere/scrivere le sue componenti; ad esempio, per leggere un vettore:

```
int V[100];
int i;

for(i=0; i<100; i++)
{
    printf("valore %d-simo? ", i);
    scanf("%d", &V[i]);
}
```

### Gestione degli elementi di un vettore:

Le singole componenti di un vettore possono essere manipolate coerentemente con il tipo ad esse associato.

**Ad esempio:** `int A[100];`

☞ agli elementi di A e' possibile applicare tutti gli operatori definiti per gli interi.

**Quindi:**

```
A[i] =n%i;
A[10]=A[0]+7;
scanf("%d", &A[i]);
...
```

## Vettori

☞ L'indice deve essere di tipo *integral type* (int, char o enum). Ad esempio:

```
#include <stdio.h>
typedef enum{blu, giallo, rosso} indice;

main()
{indice k=blu;
 int A[3];

 for(k=blu; k<=rosso; k++)
 {
    printf("Dammi elemento %d: ", k);
    scanf("%d", &A[k]);
 }
 printf("Valore %d:%d\n",blu,A[blu]);
 printf("Valore%d:%d\n",giallo,A[giallo]);
 printf("Valore %d:%d\n",rosso,A[rosso]);
}
```

## Vettore come costruttore di tipo

In C e' possibile utilizzare il costruttore [ ] per introdurre tipi di dato non primitivi che rappresentano particolari vettori.

### Sintassi della dichiarazione:

```
typedef <tipo-componente> <tipo-vettore> [<dim>]
```

dove:

- <tipo-componente> e' l'identificatore di tipo di ogni singola componente
- <tipo-vettore> e' l'identificatore che si attribuisce al nuovo tipo
- <dim> e' il numero di elementi che costituiscono il vettore (deve essere una costante).

### Ad esempio:

```
typedef int Vettori[30];

Vettori V1,V2;
```

☞ V1 e V2 sono variabili di tipo Vettori; ognuno rappresenta un vettore di 30 elementi interi.

☞ V1 e V2 possono essere utilizzati come vettori di interi.

## Vettori

### Riassumendo:

#### Variabili di tipo vettore:

```
<tipo-componente> <nome>[<dim>;
```

#### Vettore come costruttore di tipo:

```
typedef <tipo-componente> <tipo-vettore> [<dim>;
```

#### Vincoli:

- <dim> e' una **costante intera**.
- <tipo-componente> e' un **qualsiasi** tipo, semplice o strutturato.

#### Uso:

- il vettore e' una sequenza di dimensione fissata <dim> di componenti dello stesso tipo <tipo\_componente>.
- la singola componente i-esima di un vettore V e' individuata dall'indice i-esimo, secondo la notazione V[i].
  - ☞ L'intervallo di variazione degli indici e' [0,..<dim>-1]
- sui singoli elementi e' possibile operare secondo le modalita' previste dal tipo <tipo\_componente>.

## Inizializzazione di un vettore

### Mediante un ciclo:

Per attribuire un valore iniziale agli elementi di un vettore, si puo' attuare con una sequenza di assegnamenti alle N componenti del vettore.

#### Ad Esempio:

```
#define N 30
typedef int vettore [N];
vettore v;
int i;
...
for(i=0; i<N;i++)
    v[i]=0;
```

☞ La #define rende il programma piu' facilmente modificabile.

### Inizializzazione in fase di definizione:

In alternativa, e' possibile inizializzare un vettore in fase di definizione.

#### Esempio:

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
/* v[0] = 1; v[1]=2;...v[9]=10; */
```

### Addirittura e' possibile fare:

```
int v[]={1,2,3,4,5,6,7,8,9,10};
```

☞ La dimensione e' determinata sulla base dell'inizializzazione.

### Esempio:

**Somma di due vettori:** si realizzi un programma che, dati da standard input gli elementi di due vettori A e B, entrambi di 10 interi, calcoli e stampi gli elementi del vettore C (ancora di 10 interi), ottenuto come la somma di A e B.

```
#include <stdio.h>
typedef int vettint[10];

main()
{
    vettint A,B,C;
    int i;

    /* lettura dei dati */
    for(i=0; i<10; i++)
    { printf("valore di A[%d] ?, i);
      scanf("%d", &A[i]);
    }
    for(i=0; i<10; i++)
    { printf("valore di B[%d] ?, i);
      scanf("%d", &B[i]);
    }

    /* calcolo del risultato */
    for(i=0; i<10; i++)
        C[i]=A[i]+B[i];

    /* stampa del risultato */
    for(i=0; i<10; i++)
        printf("C[%d]=%d\n",i, C[i]);
}
```

### Esempio:

Leggere da input alcuni caratteri alfabetici maiuscoli (si suppongano, al massimo, 10) e riscriverli in uscita evitando di ripetere caratteri già stampati.

### Soluzione:

```
while <ci sono caratteri da leggere>
{
    <leggi carattere>;
    if <non già memorizzato>
        <memorizzalo in una struttura dati>;
};
while <ci sono elementi della struttura dati>
    <stampa elemento>;
```

Occorre una struttura dati in cui memorizzare (senza ripetizioni) gli elementi letti in ingresso.

```
char A[10];
```

### Codifica:

```
#include <stdio.h>

main()
{
    char A[10], c;
    int i, j, inseriti, trovato;

    inseriti=0;
    printf("\n Dammi 10 caratteri: ");
    for (i=0; (i<10); i++)
    {
        scanf("%c", &c);
        /* verifica unicita': */
        trovato=0;
        for(j=0;(j<inseriti)&&!trovato;j++)
            if (c==A[j])
                trovato=1;
        if (!trovato)
        {
            A[inseriti]=c;
            inseriti++;
        }
    }
    printf("Inseriti %d caratteri \n",
           inseriti);
    for (i=0; i<inseriti; i++)
        printf("%c\n", A[i]);
}
```

## Vettori multi-dimensionali

Non vi sono vincoli sul tipo degli elementi di un vettore:

☞ Gli elementi di un vettore possono essere a loro volta di tipo vettore: in questo caso si parla di vettori multidimensionali (o *matrici*)

### Definizione di vett. multidimensionali (matrici):

```
<id-tipo> <id-variable> [dim1] [dim2] ... [dimn]
```

### Significato:

<id-variabile> e' il nome di una variabile di tipo vettore di dim<sub>1</sub> componenti, ognuna delle quali e' un vettore di dim<sub>2</sub> componenti, ognuna delle quali e' un vettore di ..., ognuna delle quali e' un vettore di dim<sub>n</sub> componenti di tipo <id-tipo>.

### Ad esempio:

```
float M[20][30];
```

e' un vettore di 20 elementi, ognuno dei quali e' un vettore di 30 elementi, ognuno dei quali e' un float:

	0	1	.....	29
0				
1				
⋮				
⋮				
⋮				
19				

M e' una matrice 20x30 di reali.

☞ M[0] e' un vettore di 30 reali (la prima componente di M).

• M[1][29] e' un reale (l'ultimo elemento del secondo vettore componente di M).

## Dichiarazione di tipi vettori multi-dimensionali:

```
typedef <id-tipo> <id-tipo-vettore> [dim1] [dim2] ... [dimn]
```

### Esempi:

```
typedef float MatReali [20] [30];
MatReali Mat;
/*Mat e` un vettore di venti elementi,
ognuno dei quali e` un vettore di
trenta reali; quindi, matrice di 20x30 di
reali*/
```

### Altro esempio:

```
typedef float VetReali[30];
typedef VetReali MatReali[20];
MatReali Mat;
```

## Inizializzazione di matrici

Anche nel caso di vettori multi-dimensionali l'inizializzazione si puo` effettuare in fase di definizione, tenendo conto che, in questo caso, gli elementi sono a loro volta vettori:

### Ad esempio:

```
int matrix[4][4] = {{1,0,0,0},
                   {0,1,0,0},
                   {0,0,1,0},
                   {0,0,0,1}};
```

La memorizzazione avviene "per righe":

matrix	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

Si puo' anche omettere la dimensione:

```
int matrix[][4]={{1,0,0,0},
                 {0,1,0,0},
                 {0,0,1,0},
                 {0,0,0,1}};
```

### Esempio: lettura e stampa di matrici.

```
#include <stdio.h>
#define R 10
#define C 25

typedef float matrice[R][C];
main()
{ matrice M;
  int i, j;
  /* lettura: */
  for(i=0; i<R; i++)
    for(j=0; j<C; j++)
      { printf("M[%d][%d]? ", i, j);
        scanf("%f", &M[i][j]);
      }
  /*stampa: */
  for(i=0; i<R; i++)
    { for(j=0; j<C; j++)
      printf("%f\t", M[i][j]);
      printf("\n");
    }
}
```

### ✓ Esercizio:

Programma che esegue il prodotto (righe × colonne) di matrici quadrate N×N a valori interi:

```
C[i,j] = ∑(k=1..N) A[i][k]*B[k][j]

#include <stdio.h>
#define N 2
main()
{
  typedef int Matrici[N][N];
  int Somma;
  int i,j,k;
  Matrici A,B,C;
  /* inizializzazione di A e B */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      scanf("%d",&A[i][j]);
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      scanf("%d",&B[i][j]);
  /* prodotto matriciale */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      {Somma=0;
        for (k=0; k<N; k++)
          Somma=Somma+A[i][k]*B[k][j];
        C[i][j]=Somma;
      }
  /* stampa */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      printf("%d",C[i][j]);
}
```

### Esercizio:

Dati n valori interi forniti in ordine qualunque, stampare in uscita l'elenco dei valori dati in ordine crescente.

☞ E' necessario mantenere in memoria tutti i valori dati per poter effettuare i confronti necessari.

➔ Utilizziamo i vettori

### Ordinamento di un vettore:

Esistono vari procedimenti risolutivi (v. algoritmi di ordinamento); uno di questi e' il

#### Metodo dei Massimi successivi:

Dato un vettore: int V[dim];

1. scegli un elemento come massimo temporaneo (V[max])
2. confronta il valore di V[max] con tutti gli altri elementi del vettore (V[i]):  
se V[i]>V[max], max=i
3. quando hai finito i confronti (se max!=dim-1) scambia V[max] con V[dim-1] il massimo ottenuto dalla scansione va in ultima posizione.
4. riduci il vettore di un elemento (dim=dim-1) e, se dim>1, torna a 1.

### Codifica:

- Primo livello di specifica:

```
#include <stdio.h>
#define dim 10

main()
{
  int V[dim], i,j, max, tmp, quanti;

  /* lettura dei dati */

  /*ordinamento */

  for(i=0; i<dim; i++)
  {quanti=dim-i;

  /*ciclo di scansione
  del vettore i-simo
  (di dimensione=quanti) */
  }
}

/*stampa del vettore V ordinato */
}
```

### Codifica:

```
#include <stdio.h>
#define dim 10

main()
{
  int V[dim], i,j, max, tmp, quanti;

  /* lettura dei dati */
  for (i=0; i<dim; i++)
  { printf("valore n. %d: ",i);
    scanf("%d", &V[i]);
  }

  /*ordinamento */

  for(i=0; i<dim; i++)
  {quanti=dim-i;
  max=quanti-1;
  for( j=0; j<quanti; j++)
  { if (V[j]>V[max])
    max=j;
  }
  if (max<quanti-1)
  { /*scambio */
    tmp=V[quanti-1];
    V[quanti-1]=V[max];
    V[max]=tmp;
  }
}
/*stampa */
for(i=0; i<dim; i++)
  printf("Valore di V[%d]=%d\n", i, V[i]);
}
```

## Vettori di Caratteri: le Stringhe

Una *stringa* e' un vettore di caratteri, manipolato e gestito la *convenzione*:

ogni stringa e' terminata dal **carattere nullo** '\0' (valore decimale zero).

☞ E' responsabilita' del programmatore gestire tale struttura in modo consistente con il concetto di stringa (ad esempio, garantendo la presenza del terminatore '\0').

### Ad esempio:

```
char A[10]={'b','o','l','o','g','n','a','\0'};
```

'b'	'o'	'l'	'o'	'g'	'n'	'a'	'\0'	?	?
-----	-----	-----	-----	-----	-----	-----	------	---	---

oppure:

```
char A[10]="bologna"; /* il terminatore '\0' e'
aggiunto automaticamente */
```

### Esempio:

Programma che calcola la lunghezza (cioè, il numero di caratteri significativi) di una stringa.

```
#include <stdio.h>

/* lunghezza di una stringa */
main()
{
    char  str[81];
    /* str ha al max. 80 caratteri*/
    int  i;

    printf("\nImmettere una stringa:");
    scanf("%s",&str); /* %s formato
                        stringa */

    for (i=0; str[i]!='\0'; i++);

    printf("\nLunghezza: \t %d\n",i);
}
```

Vengono acquisiti i caratteri in ingresso fino al primo carattere di spaziatura (bianco, newline, tabulazione, salti pagina).

☞ la lunghezza di una stringa si può anche calcolare utilizzando la funzione standard di libreria **strlen()** (previa inclusione di `string.h`)

### Esempio:

Programma che concatena due stringhe date.

```
#include <stdio.h>

/* concatenamento di due stringhe */

main()
{
    char  s1[81], s2[81];
    int  l,i;

    printf("\nPrima stringa: \t");
    scanf("%s",&s1);
    printf("\nSeconda stringa: \t");
    scanf("%s",&s2);

    for (l=0; s1[l]!='\0'; l++);

    for (i=0; s2[i]!='\0' && i+l<79; i++)
        s1[i+l]=s2[i];

    s1[i+l]='\0'; /* fine stringa */

    printf("\nStringa:\t%s\n",s1);
}
```

☞ il concatenamento di due stringhe si può anche ottenere utilizzando la funzione standard di libreria **strcat()** (previa inclusione di `string.h`)

## Libreria standard sulle stringhe

Il C fornisce una libreria standard di funzioni per la gestione di stringhe.

Per poterla utilizzare è necessario includere il file header `<string.h>`:

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

Tra tutte, le funzioni più comunemente utilizzate sono:

- **strlen**
- **strcmp**
- **strcat**
- **strcpy**

## Libreria standard sulle stringhe

### • Lunghezza di una stringa:

**int strlen(char str[]);**

restituisce la lunghezza (cioè, il numero di caratteri significativi) della stringa `str` specificata come argomento.

### Ad esempio:

```
char S[10]="bologna";
int k;
...
k=strlen(S); /* k vale 7*/
```

### • Confronto tra stringhe:

**int strcmp(char str1[], char str2[])**

esegue il confronto tra le due stringhe date `str1` e `str2`. Restituisce:

- **0** se le due stringhe sono identiche;
- un **valore negativo** (ad esempio, -1), se `str1` precede `str2` (in ordine lessicografico);
- un **valore positivo** (ad esempio, +1), se `str2` precede `str1` (in ordine lessicografico).



### Esempio:

Ricopiatura dell'input sull'output convertendo minuscole in maiuscole.

```
#include <stdio.h>
#define scostamento 'a'-'A'

main()
{
  char c;
  while ((c = getchar()) != EOF)
    if (c>='a' && c<='z')
      putchar(c-(scostamento));
    else putchar(c);
}
```

⇒ **putchar(c-(scostamento))** viene espanso in:  
⇒ putchar(c-'a'-'A')

## Input/Output a linee di caratteri

La scanf (con formato %s) prevede come separatore anche il *blank* (spazio bianco):

⇒ e' possibile soltanto leggere stringhe che non contengono bianchi;

### Ad esempio, se l'input e':

```
Nel mezzo del cammin di nostra vita,
mi ritrovai in una selva oscura...
...
```

### Leggendo con:

```
char s1[80], s2[80];
scanf("%s", &s1); /* s1 vale "Nel" */
scanf("%s", &s2); /* s2 vale "mezzo" */
```

⇒ la scanf non e' adatta a leggere intere **linee** (che possono contenere spazi bianchi, caratteri di tabulazione, etc.).

### Per questo motivo, in C esistono funzioni specifiche per fare I/O di linee:

- gets
- puts

## Input/Output a linee di caratteri

### gets:

e' una funzione standard, che legge una intera riga da input, fino al primo carattere di fine linea ('\n', *newline*) e l'assegna ad una stringa.

```
gets(char str[]);
```

- assegna alla stringa str i caratteri letti
- Il carattere '\n' viene sostituito (nella stringa di destinazione str) da '\0'.

### Ad esempio, dato l'input:

```
Nel mezzo del cammin di nostra vita,
mi ritrovai in una selva oscura...
...
```

```
char S[80];
gets(S);
```

⇒ S vale:

"Nel mezzo del cammin di nostra vita,"

## Input/Output a linee di caratteri

### puts:

E' una funzione standard che scrive una stringa sull'output aggiungendo un carattere di fine linea ('\n', *newline*).

```
puts(char str[]);
```

- in caso di errore restituisce EOF

### Ad esempio:

```
..
char S1[80]="Dante Alighieri";
char S2[80]="La Divina Commedia";
puts(S1);
puts(S2);
..
```

hanno come effetto sullo standard output:

```
Dante Alighieri
La Divina Commedia
```

⇒ **puts(S)** e' equivalente a **printf("%s\n", S);**

### Esempi:

- Ricopia l'input nell'output (a linee):

```
#include <stdio.h>
main()
{
char s[81];

while (gets(s))
puts(s);
}
```

- Uso di puts e putchar:

```
putchar('A');
putchar('B');
puts("C");
putchar('D');
```

### Effetto:

```
ABC
D
```

## Il Record

### Esempio:

Si vuole realizzare l'astrazione "contribuente", caratterizzata dai seguenti attributi

- Nome,
- Cognome,
- Reddito,
- Aliquota

Con gli strumenti visti finora, per ogni contribuente e' necessario introdurre 4 variabili:

```
char Nome[20], Cognome[20];
int Reddito, Aliquota;
```

soluzione scomoda e non "astratta": le quattro variabili sono indipendenti tra di loro.

- E' necessario un costrutto che consenta l'aggregazione dei 4 attributi nell'astrazione "contribuente": il **record**.

## Tipi strutturati: il Record

Un record e' un **insieme finito** di elementi **non omogeneo**:

- il numero degli elementi e' rigidamente fissato a priori.
- gli elementi possono essere di tipo diverso.
- il tipo di ciascun elemento componente (**campo**) e' prefissato.

### Ad esempio:



### Formalmente:

Il record e' un tipo strutturato il cui dominio si ottiene mediante **prodotto cartesiano**: dati n insiemi,  $A_{c1}, A_{c2}, \dots, A_{cn}$ , il prodotto cartesiano tra essi:

$$A_{c1} \times A_{c2} \times \dots \times A_{cn}$$

consente di definire un tipo di dato strutturato (il record) i cui elementi sono n-ple ordinate:

$$(a_{c1}, a_{c2}, \dots, a_{cn})$$

dove  $a_{ci} \in A_{ci}$ .

**Ad esempio:** Il numero complesso e' definito attraverso il prodotto cartesiano:  $\mathbb{R} \times \mathbb{R}$

## Il Record in C: struct

Collezioni con un numero finito di campi (anche disomogenei) sono realizzabili in C mediante il costruttore di tipo strutturato **struct**.

### Definizione di variabile di tipo record:

```
struct { <lista definizioni campi> } <id-variabile>;
```

dove:

- **<lista definizioni campi>** e' l'insieme delle definizioni dei campi componenti, costruita usando stesse regole sintattiche della definizione di variabili:

```
<tipo1> <campo1>;
<tipo2> <campo2>;
...
<tipoN> <campoN>;
```

- **<id-variabile>** e' l'identificatore della variabile di tipo record cosi' definita.

### Esempio:

```
struct { char Nome[20];
char Cognome[20];
int Reddito;
int Aliquota;
} contribuente;
```

Nome, Cognome, Reddito ed Aliquota sono i campi della variabile contribuente (di tipo record, o struct).

## Il tipo struct

### Operatori:

Gli unici operatori previsti per dati di tipo struct sono:

- l'operatore di **assegnamento** (=): e' possibile l'**assegnamento** diretto tra record di tipo equivalente.

### Accesso ai campi:

E' possibile accedere (e manipolare) i singoli campi di un record.

1. Per accedere ai campi di un record, in C si usa la notazione *postfissa* :

**<id-variabile>.<componente>**

indica il valore del campo **<componente>** della variabile **<id-variabile>** .

- I singoli campi possono essere manipolati con gli operatori previsti per il tipo ad essi associato.

### Ad esempio:

```
struct { char Nome[20];
        char Cognome[20];
        int   Reddito;
        int   Aliquota;
    }contribuente;

contribuente.Reddito=2000+1500;
strcpy(contribuente.Nome,"Mario");
strcpy(contribuente.Cognome,"Rossi");
contribuente.Aliquota=40;
```

## Inizializzazione di record:

E' possibile inizializzare i record in fase di definizione.

### Ad esempio:

```
struct { char Nome[20];
        char Cognome[20];
        int   Reddito;
        int   Aliquota;
    }p=("Mario","Rossi",17000,10);
```

## Il costruttore di tipo struct

Il costruttore struct puo' essere utilizzato per dichiarare tipi non primitivi basati sul record:

### Dichiarazione di tipo strutturato record:

**typedef struct{<lista dichiarazioni campi>} <id-tipo>;**

dove:

- **<lista definizioni campi>** e' l'insieme delle definizioni dei campi componenti;
- **<id\_tipo>** e' l'identificatore del nuovo tipo.

### Ad esempio:

```
typedef struct { int anno;
                int mese;
                int giorno;
            }tipodata;

tipodata data;
unsigned int anno=1999;

data.anno=anno;
data.mese=1;
data.giorno=6;
```

- ☞ Gli identificatori di campo di un record devono essere distinti tra loro, ma non necessariamente diversi da altri identificatori (ad es., anno).

## I Record in C

### Riassumendo:

#### Sintassi:

```
[typedef] struct {  
    <tipo_1> <nome_campo_1>;  
    <tipo_2> <nome_campo_2>;  
    ...  
    <tipo_N> <nome_campo_N>;  
} <nome>;
```

#### Vincoli:

- <nome\_campo\_i> e' un identificatore stabilito che individua il campo i-esimo;
- <tipo\_i> e' un **qualsiasi** tipo, semplice o strutturato.
- <nome> e' l'identificatore della struttura (o del tipo, se si usa **typedef**)

#### Uso:

- la struttura e' una collezione di un numero fissato di elementi di vario tipo (<tipo\_campo\_i>);
- il singolo campo <nome\_campo\_i> di un record R e' individuato mediante la notazione: R.<nome\_campo\_i>;
- se due strutture di dati di tipo **struct** hanno lo stesso tipo, allora e' possibile l'assegnamento diretto.

## Vettori e record

Non ci sono vincoli riguardo al tipo degli elementi di un vettore: si possono realizzare anche **vettori di record (tabelle)**.

#### Ad esempio:

```
typedef struct { char Nome[20];  
                char Cognome[20];  
                int Reddito;  
                int Aliquota;  
                } Contribuente;  
contribuente archivio[1000];
```

☞ archivio e' un vettore di 1000 elementi, ciascuno dei quali e' di tipo **contribuente** vettore di record (struttura **tabellare**).

	Nome	Cognome	Reddito	Aliquota
0				
1				
2				
...				
999				

## Vettori e Record

Allo stesso modo, si possono fare record di record e record di vettori; ad esempio:

```
typedef struct{ int giorno;  
               int mese;  
               int anno;  
               }data  
  
typedef struct{ char nome[20];  
               char cognome[40];  
               data data_nasc;  
               } persona;  
  
persona P;  
...  
P.data_nasc.giorno=25;  
P.data_nasc.mese=3;  
P.data_nasc.anno=1992;  
...
```

## Equivalenza di tipo

**Strutturale** Variabili con la **stessa struttura interna** sono considerate **equivalenti** (anche se non hanno lo stesso identificatore di tipo).

**Nominale** Sono equivalenti solo variabili che fanno riferimento allo **stesso identificatore di tipo**.

☞ In **C** non si specifica quale tipo di equivalenza venga adottato: esistono realizzazioni del linguaggio che adottano equivalenza strutturale, altre equivalenza nominale.

➔ Per garantire la portabilita' **usare sempre equivalenza nominale**.

### Equivalenza strutturale:

```
typedef struct {float x;
               float y;
               }coordinate;

coordinate A, C;

struct {float x;
       float y;
       }B;
```

### A, C e B hanno lo stesso tipo:

A=B;      *lecita*

A=C;      *lecita*

### Equivalenza nominale:

```
typedef struct {float x;
               float y;
               }coordinate;

coordinate A, B;
struct {float x;
       float y;
       }C;
```

A, B hanno lo stesso tipo, C viene considerato di tipo diverso:

A=B;      *lecita*

A=C;      *non lecita*

### Esercizio:

Realizzare un programma che, lette da input le coordinate di un punto P del piano, sia in grado di applicare a P alcune trasformazioni geometriche (traslazione, e proiezioni sui due assi).

```
#include <stdio.h>

main()
{
  typedef struct{float x,y;}punto;

  punto P;
  unsigned int op;
  float Dx, Dy;

  /* si leggono le coordinate da input i
  dati e le si memorizza in P */

  printf("ascissa? ");
  scanf("%f",&P.x);
  printf("ordinata? ");
  scanf("%f",&P.y);

  /* lettura dell'operazione richiesta:
  0: termina
  1: proietta sull'asse x
  2: proietta sull'asse y
  3: trasla di Dx, Dy */
  printf("%s\n","operazione(0,1,2,3)?");
  scanf("%d",&op);
```

```
switch (op)
{case 1: P.y= 0;break;
 case 2: P.x= 0; break;
 case 3:printf("%s","Traslazione?");
        scanf("%f%f",&Dx,&Dy);
        P.x=P.x+Dx;
        P.y=P.y+Dy;
        break;
 default: printf("errore!");
 }
printf("%s\n","nuove coordinate ");
printf("%f\t%f\n",P.x,P.y);
}
```

### ✍ Esercizio:

Scrivere un programma che acquisisca i dati relativi agli studenti di una classe:

- **nome**
- **eta**
- **voti**: rappresenta i voti dello studente in 3 materie (italiano, matematica, inglese);

il programma deve successivamente calcolare e stampare, per ogni studente, la media dei voti ottenuti nelle 3 materie.

```
#include <stdio.h>
typedef enum {ita, mat, ing}materie;

typedef struct { char nome[30];
                int eta;
                int voto[3];
            } studente;

main()
{ studente classe[20];
  float m;
  int i;
  materie j;
  /* lettura dati */
  for(i=0;i<20; i++)
  { scanf("%s%d", &classe[i].nome,
          classe[i].eta);
    for(j=ita; j<=ing; j++)
      scanf("%d", &classe[i].voto[j]);
  }
```

```
/* stampa delle medie */
for(i=0;i<20; i++)
{ for(m=0, j=ita; j<=ing; j++)
  m+=classe[i].voto[j]);
  printf("media di %s: %d\n",
        classe[i].nome, m);
}
```

## Unione discriminata

L'**unione discriminata** (o record variante) e' un tipo di dato per il quale sono possibili definizioni differenti ed in alternativa tra loro: ogni alternativa viene detta **variante**.

### Ad esempio:

una figura geometrica piana può essere rappresentata in modo diverso, a seconda del tipo. Per esempio:

- un **triangolo** è rappresentato dalla terna dei suoi lati
- un **quadrato** è rappresentato dal lato;
- un **rettangolo** è rappresentato dalla coppia dei suoi lati;
- una **circonferenza** è rappresentata dal suo raggio.

Mediante l'unione discriminata, si può realizzare un tipo di dato dotato di quattro rappresentazioni alternative (una per il triangolo, una per il quadrato, una per il rettangolo ed una per la circonferenza) che, in generale, rappresenta una figura geometrica.

☞ In C, l'unione discriminata di tipi strutturati viene realizzata attraverso il costruttore di tipo **union**.

## Unione discriminata in C: union

In C è disponibile il costruttore union per introdurre dati e tipi basati sull'unione discriminata.

### Definizione di variabili union:

```
union { <definizione 1>;
        <definizione 2>;
        ...
        <definizione N>;
    }<nome>;
```

dove:

- <nome> è l'identificatore della nuova variabile;
- <definizione 1>, <definizione 2>, .. <definizione N> sono le definizioni alternative (*varianti*) per la variabile <nome>.

### Ad esempio:

```
union{ float raggio_circ;
      float lati Rettangolo[2];
      float lati triangolo[3];
      float lato_quadrato;
}oggetto;
```

- La variabile oggetto ha quattro rappresentazioni alternative: a seconda di quale figura geometrica si vuole memorizzare in oggetto, verrà scelta una particolare variante.
- le varianti previste si escludono mutuamente, ma tutti i componenti condividono la stessa memoria: l'estensione dell'area allocata per un dato di tipo union, e' pari all'area richiesta dalla struttura che occupa piu' memoria (nell'esempio, lo spazio necessario per 3 float).

## Union

### Operatori

Gli unici operatori previsti per le union sono l'assegnamento e gli operatori di uguaglianza/disuguaglianza relazionale.

### Accesso a record varianti:

per scegliere una tra le rappresentazioni alternative si usa la notazione postfissa:

**<nome>.<definizione\_i>;**

- in base alla variante scelta (tra le N possibili alternative), si puo' operare sul dato con gli strumenti previsti dal tipo associato ad esso.

### Ad esempio:

```
union{ float raggio_circ;
      float lati Rettangolo[2];
      float lati triangolo[3];
      float lato_quadrato;
}oggetto;
float AREA;
/*se oggetto rappresenta un rettangolo:*/
scanf("%f", &oggetto.lati_Rettangolo[0]);
scanf("%f", &oggetto.lati_Rettangolo[1]);
AREA= oggetto.lati_Rettangolo[0]*
      oggetto.lati_Rettangolo[1];
...
```

## Il costruttore di tipo union

Mediante il costruttore union si possono dichiarare tipi non primitivi.

### Dichiarazione di tipo non primitivi:

```
typedef union { <definizione 1>;
              <definizione 2>;
              ...
              <definizione N>;
} <nometipo>;
```

dove:

- <nometipo> è l'identificatore del nuovo tipo;
- <definizione 1>, <definizione 2>, .. <definizione N> sono le definizioni alternative (*varianti*) per la variabile <nome>.

### Ad esempio:

```
typedef union{ float raggio_circ;
              float lati Rettangolo[2];
              float lati triangolo[3];
              float lato_quadrato;
}figura;
figura O1, O2;
...
O1.raggio_circ=12.5;
O2.lato_quadrato=5.25;
...
```

## Record e Union

Non ci sono vincoli riguardo al tipo dei campi di un record: è possibile realizzare uno o più campi di un record mediante union.

☞ In questo modo si possono realizzare strutture che hanno:

- una **parte fissa**
- una **parte variante**

### Esempio:

Si vuole realizzare un tipo di dato per rappresentare i libri gestiti da una biblioteca. In particolare, per ogni libro si vogliono memorizzare le seguenti informazioni:

- Codice
- Titolo
- Autore
- Editore
- Anno

Inoltre per ogni libro:

- può essere necessario registrare la sua **collocazione** all'interno della biblioteca (se è disponibile);
- oppure può essere necessario memorizzare i **dati del cliente** al quale è imprestatato. (se è in prestito).

☞ Notiamo che il libro è caratterizzato da una parte fissa (codice, autore, titolo, etc.) e da una parte variante (cliente o collocazione).

```
typedef struct { char Cognome[30];
                char Indirizzo[40];
                char N_tel[15];
                int giorno;
                int mese;
                int anno;
            }cliente;

typedef struct { int stanza;
                int scaffale;
            }collocazione;

typedef struct
{ char Codice[6];
  char Titolo[30];
  char Autore[30];
  char Editore[20];
  int Anno;
  union { cliente Cli;
          collocazione Col;
        }V; /* p. variante */
} libro;

libro L1, L2;
/* L1 è disponibile: */
L1.V.Col.stanza=3;
L1.V.Col.scaffale=7;
/* L2 è in prestito: */
scanf("%s", &L2.V.Cli.Cognome);
gets(L2.V.Cli.Indirizzo);
...
```

## Union

### Esempio:

```
...
libro L;
...
/* L è disponibile: */
L.V.Col.stanza=3;
L.V.Col.scaffale=7;
printf("%s", L.V.Cli.Cognome); /* ?? */
...
```

☞ Non c'è alcun controllo sulla rappresentazione adottata: anche se L rappresenta un libro disponibile, si può accedere ad L.V.Cli.Cognome !!

### Problema:

Come riconoscere il tipo di rappresentazione valida (ad un certo istante) per un dato di tipo union?

☞ Per favorire una gestione corretta delle union, può essere utile introdurre un campo aggiuntivo (**tag**) in base al cui valore si può dedurre il tipo di rappresentazione adottata.

### Esempio:

```
typedef struct
{ char Codice[6];
  char Titolo[30];
  char Autore[30];
  char Editore[20];
  int Anno;
  int disponibile; /* tag */
  union { cliente Cli;
          collocazione Col;
        }V; /* p. variante */
} libro;

libro L;
...
if (L.disponibile) /* L è disponibile: */
{ L.V.Col.stanza=3;
  L.V.Col.scaffale=7;
}
else /* L è in prestito: */
{ printf("%s", L.V.Cli.Cognome);
  printf("%s", L.V.Cli.Indirizzo);
}
...
```

☞ In questo modo si mantiene **esplicitamente** la **consistenza** della struttura dati.

## Il puntatore

È un tipo scalare, che consente di rappresentare gli **indirizzi** delle variabili allocate in memoria. Il dominio di una variabile di tipo puntatore è un insieme di indirizzi:

☞ Il valore di una variabile di tipo puntatore può essere l'indirizzo di un'altra variabile (variabile *puntata*).

In C i puntatori si definiscono mediante il costruttore \*.

### Definizione di una variabile puntatore:

**<TipoElementoPuntato> \*<NomePuntatore>;**

dove:

- <TipoElementoPuntato> è il tipo della variabile puntata
- <NomePuntatore> è il nome della variabile di tipo puntatore
- il simbolo \* è il costruttore del tipo puntatore.

### Ad esempio:

```
int *P; /*P è un puntatore a intero */
```

## Il puntatore

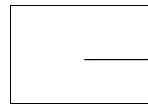
### Operatori:

- Assegnamento: e' possibile l'assegnamento tra puntatori (dello stesso tipo). E' disponibile la costante NULL, per indicare l'indirizzo nullo.
- operatore di **dereferencing** \*: è un operatore unario. Si applica a un puntatore e restituisce il valore contenuto nella cella puntata => serve per accedere alla variabile puntata.
- Operatore **indirizzo**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale e' allocata la variabile.
- operatori **aritmetici** (vedi *vettori & puntatori*).
- Operatori relazionali: >, <, ==, !=

### Ad esempio:

```
int *punt1, *punt2;
int A;
punt1=&A;
*punt1=127;
punt2=punt1;
punt1=NULL;
```

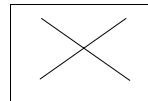
punt2



A

127

punt1



### Operatore Indirizzo &:

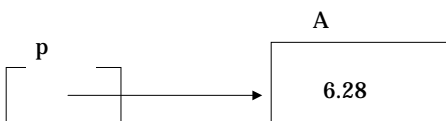
- ☞ & si applica solo ad **oggetti che esistono in memoria** (quindi, già definiti).
- ☞ & non e' applicabile ad espressioni.

### Operatore Dereferencing \*:

- ☞ consente di accedere ad una variabile specificando il suo indirizzo
- ☞ l'indirizzo rappresenta un modo alternativo (alias) al nome per accedere e manipolare la variabile:

```
float *p;
float R, A;

p=&A; /* *p è un alias di A*/
R=2;
*p=3.14*R; /* A è modificato */
```



## Puntatore come costruttore di tipo

### Dichiarazione di un tipo puntatore:

```
typedef <TipoElementoPuntato> *<NomeTipo>;
```

- <TipoElementoPuntato> e' il tipo della variabile puntata
- <NomePuntatore> e' il nome del tipo dichiarato.

### Ad esempio:

```
typedef float *tpf;
tpf p;
float f;
p=&f;
...
```

## Puntatori

Nella definizione di un puntatore e' necessario indicare il tipo della variabile puntata.

→ il compilatore puo' effettuare controlli statici sull'uso dei puntatori.

### Esempio:

```
typedef struct{...}record;
```

```
int *p, A;  
record *q, X;
```

```
p=&A;  
q=p; /*warning!*/  
q=&X;  
*p=*q; /* errore! */
```

☞ Viene segnalato dal compilatore (**warning**) il tentativo di utilizzo congiunto di puntatori a tipi differenti.

## Variabili Dinamiche

In C si possono definire e' possibile classificare le variabili in base al loro tempo di vita; e' possibile individuare due categorie:

- variabili **automatiche**
- variabili **dinamiche**

### Variabili automatiche:

- L'allocazione e la deallocazione di variabili automatiche e' effettuata automaticamente dal sistema (senza l'intervento del programmatore).
- Ogni variabile automatica ha un nome, attraverso il quale la si puo' riferire.
- Il programmatore non ha la possibilita` di influire sul tempo di vita di variabili automatiche.

## Variabili Dinamiche

### Variabili dinamiche:

- Le variabili **dinamiche** devono essere allocate e deallocate esplicitamente dal programmatore.
- L'area di memoria in cui vengono allocate le variabili dinamiche si chiama **heap**.
- Le variabili dinamiche non hanno un identificatore associato ad esse, ma possono essere riferite soltanto attraverso il loro indirizzo (mediante i puntatori).
- Il tempo di vita delle variabili dinamiche e' l'intervallo di tempo che intercorre l'allocazione e la deallocazione (che sono stabilite dal programmatore).

☞ tutte le variabili viste finora rientrano nella categoria delle **variabili automatiche**.

## Variabili Dinamiche

☞ Il C prevede funzioni standard di **allocazione deallocazione** per variabili dinamiche:

- malloc
- free

Non sono definite a livello di linguaggio di programmazione, ma a **livello di sistema operativo**, mediante la libreria standard **<stdlib.h>**.

## Variabili Dinamiche

### Allocazione di variabili dinamiche:

La memoria dinamica viene allocata con la funzione standard **malloc**:

```
punt = (tipodato *) malloc ( sizeof (tipodato));
```

- **tipodato** e' il tipo della variabile puntata
- **punt** e' una variabile di tipo **tipodato \***
- **sizeof()** e' una funzione standard che calcola il numero di bytes che occupa il dato specificato come argomento
- e' necessario convertire esplicitamente il tipo del valore ritornato (casting): (tipodato \*) malloc(..)

### Significato:

☞ La malloc provoca la creazione di una variabile dinamica nell'**heap** e restituisce come valore l'indirizzo della variabile creata.

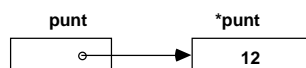
### Ad esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
```

```
      punt
      ┌───┐
      │   │
      └───┘
punt=(tp )malloc(sizeof(int));
```



```
*punt=12
```



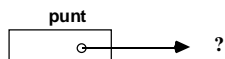
## Variabili dinamiche

### Deallocazione:

Si rilascia la memoria allocata dinamicamente con:

```
free (punt);
```

dove punt e' l'indirizzo della variabile da deallocare.



Dopo questa operazione, la cella di memoria occupata da \*punt viene deallocata: \*punt non esiste piu'.

### Esempio:

```
main()
{
  char A, *p;

  A='Z';
  p=(char *)malloc(sizeof(char));
  *p=A;
  ...
  <uso di *p>
  ...
  free(p);
}
```

### Esempio:

```
#include <stdlib.h>
main()
{
  int *p;
  /*definizione del puntatore p
  ad intero;il contenuto di p non è
  ancora definito */

  p = (int *) malloc(sizeof (int));
  /*definizione del contenuto di p:
  indirizzo di una cella di memoria
  allocata dinamicamente*/

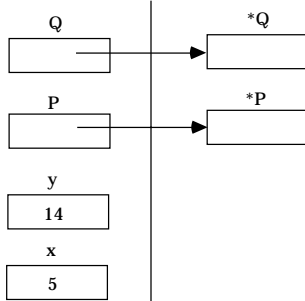
  *p = 55;
  /* assegnamento di un valore alla
  cella *p referenziata da p */

  free(p);
  /* deallocazione della cella
  referenziata da p; il contenuto
  di p non è più definito */
}
```

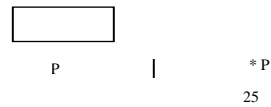
**Esempio:**

```
main()
{
  int *P, *Q, x, y;

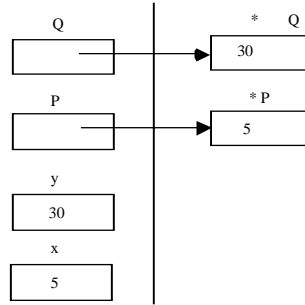
  x=5;
  y=14;
  P=(int *)malloc(sizeof(int));
  Q=(int *)malloc(sizeof(int));
}
```



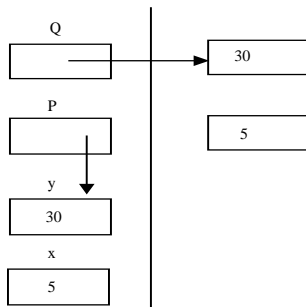
```
*P = 25;
*Q = 30;
```



```
*P = x;
y = *Q;
```



```
P = &y;
```



```
...
}
```

☞ l'ultimo assegnamento ha come effetto collaterale la perdita dell'indirizzo di una variabile dinamica (quella precedentemente referenziata da P) che rimane allocata ma non è più utilizzabile!

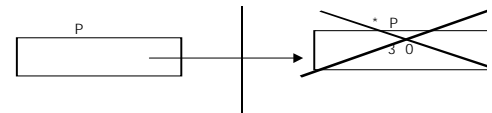
**Problemi legati all'uso dei Puntatori**

**Riferimenti pendenti (dangling references):**

Possibilità di fare riferimento ad aree di memoria non più allocate.

**Ad esempio:**

```
int *P;
P = (int *) malloc(sizeof(int));
...
free(P);
*P = 100; /* Da non fare! */
```



## Problemi legati all'uso dei Puntatori

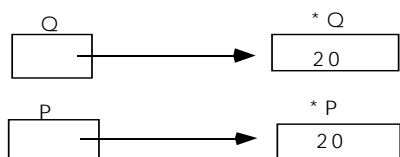
### Aree inutilizzabili:

Possibilità di perdere il riferimento ad aree di memoria allocate al programma (non più riusabili).

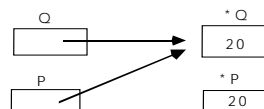
### Ad esempio:

```
int *P,*Q;
P = (int *) malloc ( sizeof ( int));
Q = (int *) malloc ( sizeof ( int));
*P = 30;    *Q = 20;
```

**\*P = \*Q;**



**P = Q;**



L'area che era puntata da P non è più raggiungibile, ma rimane allocata al programma!

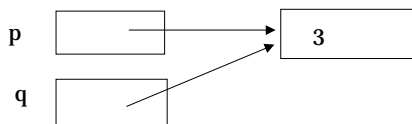
## Problemi legati all'uso dei Puntatori

### Aliasing:

Possibilità di riferire la stessa variabile con puntatori diversi:

### Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
variabile */
```



**\*q = 10; /\* anche \*p e' cambiato! \*/**

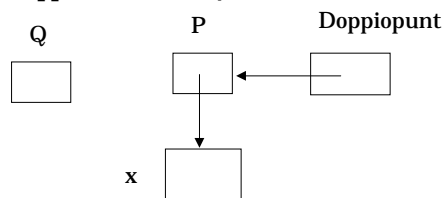
## Puntatori a puntatori (handle)

Un puntatore pu' puntare a variabili di tipo qualunque (semplici o strutturate); puo' puntare anche a un puntatore:

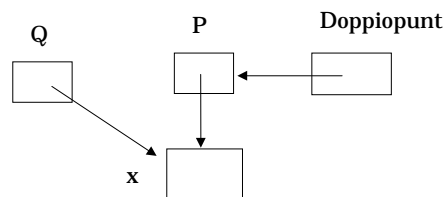
```
[typedef] TipoDato **TipoPunt;
```

### Ad esempio:

```
int x, *P, *Q, **DoppioPunt;
P = &x;
DoppioPunt = &P;
```



**Q = \*DoppioPunt;**



## Vettori & Puntatori

### Vettori:

- in C, i vettori vengono allocati in memoria in **parole consecutive** (cioè parole fisicamente adiacenti), la cui **dimensione** dipende dal tipo degli elementi del vettore.
- Il **nome** di una variabile di tipo vettore viene considerato dal C come l'**indirizzo** del primo elemento del vettore.

### Ad esempio:

```
int V[10];
```

☞ V è una **costante**:

- V equivale a `&V[0]`
- come tipo è un puntatore ad intero:

```
int *p, V[10];
p=V; /* p punta a V[0] */
V = p; /*NO! V è un puntatore costante*/
```

## Vettori & Puntatori

Il C consente di eseguire operazioni di somma e sottrazione sui puntatori (a vettori).

### Operatori aritmetici su puntatori a vettori:

Se V e W sono puntatori ad elementi di vettori ed i è un intero:

- (V+i) restituisce l'indirizzo dell'elemento spostato di i posizioni in avanti rispetto a quello indicato da i;
- (V-W): restituisce l'intero che rappresenta il numero di elementi compresi tra V e W.

### Ad esempio:

```
float V[100], *p, *q;
int k;
p=V+7; /* p punta a V[7] */
q=V+2; /* p punta a V[5] */
k=p-q; /* k vale 5 */
...
```

## Vettori e Puntatori

☞ In C, ogni riferimento ad un elemento di un vettore è espanso come un *puntatore dereferenziato*.

V[0]	equivale a	*(V)
V[1]	equivale a	*(V + 1)
V[i]	equivale a	*(V+i)
V[expr]	equivale a	*(V + expr)

### Ad esempio:

```
main ()
{
char a[] = "0123456789"; /*a e' un
                           vettore di
                           caratteri */

int i = 5;

printf("%c%c%c%c\n", a[i], a[5], i[a], 5[a]);
}
```

### Stampa:

5 5 5 5

☞ Per il compilatore V[i] e i[V] sono lo stesso elemento, perché viene sempre eseguita la conversione: **V[i] => \*(V+i)** senza eseguire alcun controllo né su V né su i.

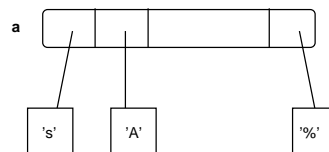
## Vettori & Puntatori

☞ [] ha precedenza rispetto a \*

### Quindi:

**char \*a[10];** => equivale a **char \*(a[10]);**

a è un **vettore di puntatori a carattere**.



☞ Per un puntatore ad un vettore di caratteri è necessario forzare la precedenza (con le parentesi)

**char (\* a) [10];**

## Puntatori a strutture:

E' possibile utilizzare i puntatori per accedere a variabili di tipo struct.

### Ad esempio:

```
typedef struct{ int Campo_1,Campo_2;
                } TipoDato;
```

```
TipoDato S, *P;
```

```
P = &S;
```

Il punto della notazione postfissa ha precedenza sull'operatore di dereferencing \*; per accedere alle componenti della struttura referenziata da P è necessario utilizzare le parentesi tonde:

```
(*P).Campo_1=75;
```

### Operatore ->:

L'operatore -> consente di accedere ad un campo di una struttura referenziata da un puntatore in modo più sintetico:

```
P->Campo_1=75;
```

## Esempio

Si vuole realizzare un programma che data da input una sequenza di N parole (di, al massimo, 20 caratteri ciascuna), una per riga, stampi in ordine inverso le parole date, ognuna "ribaltata" (cioè, stampando i caratteri in ordine inverso: dall'ultimo al primo).

Utilizzare una struttura dinamica.

```
#include <stdio.h>
#include <stdlib.h>
typedef char parola[20];
parola w, *p;

main()
{ parola w, *p;
  int i, j, N;

  printf("Quante parole? ");
  scanf("%d", &N);
  /* allocazione del vettore */
  p=(parola *)malloc(N*sizeof(parola));
  /* lettura della sequenza */
  for(i=0; i<N; i++)
    gets(&p[i]);
```

```
/* stampa */
for(i=N-1; i>=0; i--)
{ j=19;
  do
  j--;
  while(p[i][j]!='\0');
  for(j--;j>=0; j--)
    putchar(p[i][j]);
  printf("\n");
}
free(p); /* deallocazione */
}
```