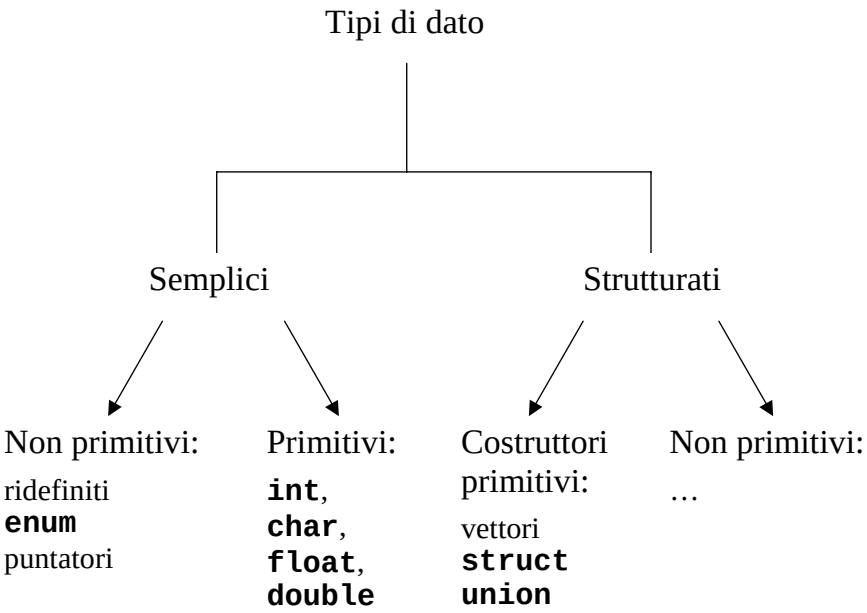
Tipi di dato strutturati



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

Vettori

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

Caratteristiche del vettore:

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

0	X
1	Y
2	Z
…	…
n-1	W

Definizione di vettori in C

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

Sintassi:

```
<def-vettore> :=
      <id-tipo> <id-variabile> [<dimensione>];
```

dove:

- **<id-tipo>** è l'identificatore di tipo degli elementi componenti;
- **<dimensione>** rappresenta il numero degli elementi componenti (è una **costante** intera);
- **<id-variabile>** è l'identificatore della variabile strutturata così definita.

Esempio:

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

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

```
int N;
char V[N]; /* ---> è sbagliato!!! */
```


Indici nei vettori

- È possibile fare riferimento ad una singola componente di un vettore **`V`** specificando l'**indice `i`** corrispondente:

```
V[i]
```

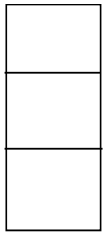
- Dal momento che deve esistere un ordinamento tra gli indici, l'indice deve essere di tipo **integral type** (cioè **`int`**, **`char`** o **`enum`**).

- Se *N* è la dimensione, il **dominio degli indici** è:

[0, *N*-1]

Esempio:

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



Operatori sui vettori

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 è possibile** l'assegnamento diretto tra vettori:

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

☞Non è 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.

Esempio:

```
int A[100];
```

☞Agli elementi di **A** è possibile applicare tutti gli operatori definiti per gli interi.

Quindi:

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

Vettori: tipo dell'indice

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

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.

Riassunto sui vettori

Variabili di tipo vettore:

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

Vettore come costruttore di tipo:

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

Vincoli

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

Utilizzo

- Il vettore è una sequenza di dimensione fissata <dim> di componenti dello stesso tipo <tipo_componente>.
- La singola componente **`i`**-esima di un vettore **`V`** è individuata dall'indice **`i`**-esimo, secondo la notazione **`V[i]`**.
- L'intervallo di variazione degli indici è:
  [0, <dim>-1]
- È possibile operare sui singoli elementi secondo le modalità previste dal tipo <tipo_componente>.

Inizializzazione di un vettore

Mediante un ciclo:

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

Esempio:

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

☞La direttiva #**`define`** rende il programma più facilmente modificabile.

In fase di definizione:

In alternativa, è 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 è possibile:

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

☞La dimensione è determinata sulla base dell'inizializzazione.

Esempio 1:

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", &A[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 2:

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("Dammi 10 caratteri: ");
 for(i=0; (i<10); i++) {
  scanf("%c", &c);
 /* verifica unicità */
  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 vettori multidimensionali (matrici):

<vett-multid> :=
<id-tipo> <id-variable> [dim$_1$][dim$_2$]…[dim$_n$]

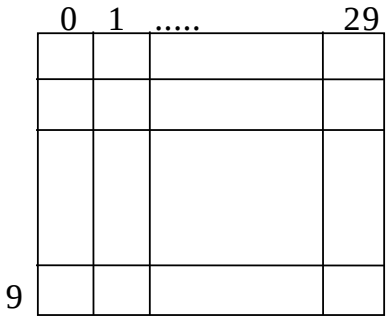
Significato:

- **<id-variabile>** è il nome di una variabile di tipo vettore di dim$_1$ componenti, ognuna delle quali è un vettore di dim$_2$ componenti, ognuna delle quali è un vettore di …, ognuna delle quali è un vettore di dim$_n$ componenti di tipo <id-tipo>.

Esempio:

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

è un vettore di 20 elementi, ognuno dei quali è un vettore di 30 elementi, ognuno dei quali è un **`float`**:



`M` è una matrice 20x30 di reali.

☞**`M[0]`** è un vettore di 30 reali (la prima componente di **`M`**).

☞**`M[1][29]`** è un **`float`** (l'ultimo elemento del secondo vettore componente di **`M`**).

Dichiarazione di tipi vettori multi-dimensionali:

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

Esempio 1:

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

Esempio 2:

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


Inizializzazione di matrici

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

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 $N \times N$ a valori interi:

$$C[i,j] = \sum_{(k=1\ldots N)} A[i][k]*B[k][j]$$

```
#include <stdio.h>
#define N 2
main() {
 typedef int Matrici[N][N];
 int Somma, 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.

☞È 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 è il **Metodo dei Massimi successivi:**

Dato un vettore: **`int V[dim]`**;

1. eleggi 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, quanti;

/* lettura dei dati */

/*ordinamento */

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

 /*ciclo di scansione del vettore
  i-esimo (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* è un vettore di caratteri, manipolato e gestito secondo una *convenzione*:

Ogni stringa è terminata dal ***carattere nullo*** **'\0'** (valore decimale zero).

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

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' è 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 massimo 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 del file **`string.h`**).

Lunghezza di una stringa

Non bisogna confondere la lunghezza di una stringa (il numero di caratteri che la compongono) con la sua lunghezza massima (ovvero la dimensione del vettore destinato a contenerla).

Nell'esempio precedente:

- la lunghezza massima di **`str`** è di 80 caratteri (definizione **`char str[81];`**);
- la lunghezza effettiva di **`str`** dipende da ciò che è stato inserito dall'utente.

☞Ovviamente, occorre sempre che sia:

lunghezza effettiva <= lunghezza massima

Quindi, il vettore di caratteri va dimensionato in maniera sufficiente a contenere la stringa più lunga possibile.

Cosa succede se, nell'esempio precedente, l'utente inserisce una stringa lunga più di 80 caratteri?

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 del file **`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 (ovvero che contiene le *dichiarazioni* di alcune funzioni) <**`string.h`**>:

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

Tra tutte, le funzioni più comunemente utilizzate sono:

- **`strlen()`**
- **`strcmp()`**
- **`strcat()`**
- **`strcpy()`**

Funzioni della 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.

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`**, restituendo:

- **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 **`str1`** segue **`str2`** (in ordine lessicografico).

Esempio:

```
char S1[10]="bologna";
char S2[10]="napoli";
int k;
...
k=strcmp(S1, S2); /* k<0 */
k=strcmp(S1, S1); /* k=0 */
k=strcmp(S2, S1); /* k>0 */
```

Concatenamento di stringhe

```
char *strcat(char str1[], char str2[]);
```

Concatena le 2 stringhe date **`str1`** e **`str2`**. Il risultato del concatenamento è in **`str1`** (viene anche restituito come puntatore a carattere, v. vettori e puntatori).

Esempio:

```
char S1[10]="reggio ";
char S2[10]="emilia";
strcat(S1, S2); /* S1="reggioemilia" */
```

Copia di stringhe

```
char *strcpy(char str1[], char str2[]);
```

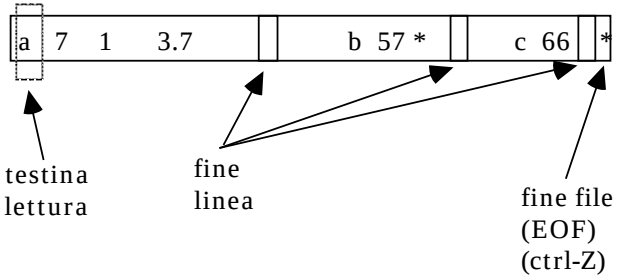
Copia la stringa **`str2`** in **`str1`** (restituisce **`str1`** come puntatore a carattere, v. vettori e puntatori).

Esempio:

```
char S1[10]="Giuseppe";
char S2[10];
...
strcpy(S1,S2); /* S1="Giuseppe" */
```


Input/output a caratteri

I dispositivi di ingresso ed uscita sono visti come file di caratteri (file ***testo***) terminati da un carattere speciale di ***end-of-file*** (EOF) ed organizzati eventualmente su più linee (separate dal carattere ***newline*** **`'\n'`**).



Esistono funzioni (libreria standard) per leggere e scrivere **singoli caratteri:**

- **`getchar()`**: restituisce il prossimo carattere disponibile in ingresso;
- **`putchar()`**: stampa un carattere su output.

Utilizzo

<variabile> **`= getchar()`**

`putchar(`<variabile>**`)`**

dove <variabile> è di tipo intero (o carattere).

putchar()/getchar()

getchar()

```
int getchar(void);
```

- non richiede argomenti e restituisce come risultato il carattere letto **convertito in `int`** (o **`EOF`** in caso di end-of-file o errore).

putchar()

```
int putchar(char c);
```

- richiede un argomento (il carattere da scrivere), e restituisce come risultato il carattere scritto (o **`EOF`** in caso di errore).

Esempio:

Programma che copia da input (la tastiera) su output (il video):

```
#include <stdio.h>

main()
{
 char c;
 c=getchar();
 while(c != EOF) {
  putchar(c);
  c=getchar();
 }
}
```

☞La funzione **`getchar` comincia** a restituire caratteri solo quando è stato battuto il tasto di invio (***input "bufferizzato"***).

Versione più compatta:

```
#include <stdio.h>
main()
{
 char c;
 while((c=getchar()) != EOF)
  putchar(c);
}
```

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

☞È possibile soltanto leggere stringhe che non contengono spazi bianchi;

Esempio:

Se l'input è:

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 funzione **`scanf`** non è 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()`**

La funzione `gets()`

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

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

- Assegna alla stringa **str** i caratteri letti da input fino al primo fine linea.
- Il carattere **'\n'** viene sostituito (nella stringa di destinazione **str**) da **'\0'**.
- La stringa **str** viene anche restituita come puntatore a carattere (v. vettori e puntatori).

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,"
```


La funzione `puts()`

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

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

- In caso di errore restituisce **EOF**.

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);`** è **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 tipo record

Esempio:

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

- Nome,
- Cognome,
- Reddito,
- Aliquota.

Con gli strumenti visti finora, per ogni contribuente è 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.

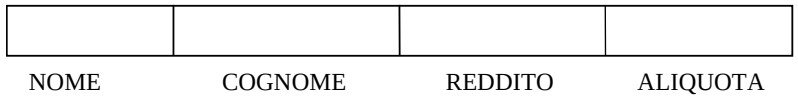
☞È necessario un costrutto che consenta l'aggregazione dei 4 attributi nell'astrazione "contribuente": il **record**.

Tipi strutturati: il record

Un record è un **insieme finito** di elementi **non omogenei**:

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

Esempio:



Formalmente:

Il record è un tipo strutturato il cui dominio si ottiene mediante **prodotto cartesiano**:
Dati n insiemi, A_{c1}, A_{c2}, ..., A_{cn}, il prodotto cartesiano tra essi

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

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

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

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

Ad esempio: Il tipo "numero complesso" è definito attraverso il prodotto cartesiano: $\Re \times \Re$

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:

```
<definizione-record> :=
    struct {
    <elenco-definizioni-campi>
    } <id-variabile>;
```

dove:

- **<elenco-definizioni-campi> è** l'insieme delle definizioni dei campi componenti, costruito usando le stesse regole sintattiche della definizione di variabili:

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

- **<id-variabile> è** l'identificatore della variabile di tipo record così 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** (**=**): è possibile l'**assegnamento** diretto tra record di tipo equivalente.
- Operatori di **uguaglianza** e **disuguaglianza** relazionale (**==**, **!=**).

Accesso ai campi

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

- 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 poi essere manipolati con gli operatori previsti per il tipo ad essi associato.

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

È possibile inizializzare i record in fase di definizione.

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`** può essere utilizzato per dichiarare tipi non primitivi basati sul record:

Dichiarazione di tipo strutturato record:

```
typedef struct {
     <elenco-dichiarazioni-campi> } <id-tipo>
```

dove:

- **<elenco-definizioni-campi>** è l'insieme delle definizioni dei campi componenti;
- **<id_tipo>** è l'identificatore del nuovo tipo.

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**).

Riassunto sui record

Sintassi:

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

Vincoli:

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

Utilizzo:

- La struttura è una collezione di un numero fissato di elementi di vario tipo (<tipo_campo_i>).
- Il singolo campo <nome_campo_i> di un record **`R`** è individuato mediante la notazione: **`R.`**<nome_campo_i>.
- Se due strutture di dati di tipo **`struct`** hanno lo stesso tipo, allora è 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**).

Esempio:

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

☞**`archivio`** è un vettore di 1000 elementi, ciascuno dei quali è 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 portabilità occorre **usare sempre equivalenza nominale**.

Esempio:

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

coordinate A, B;

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

Equivalenza strutturale:

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

```
A=B; /* lecita */

A=C; /* lecita */
```

Equivalenza nominale:

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 si memorizzano in P */
 printf("ascissa? ");
 scanf("%f", &P.x);
 printf("ordinata? ");
 scanf("%f", &P.y);
```

```
/* lettura dell'operazione richiesta:
 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("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 composta da 20 studenti:

- **nome**
- **età**
- **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);
 }
}
```


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:

<definizione-var-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.

Esempio:

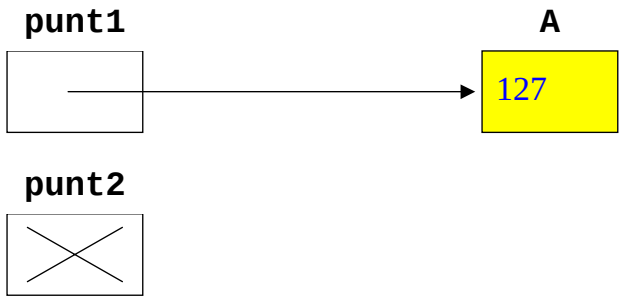
```
int *P; /*P è un puntatore a intero */
```

Operatori per il tipo puntatore

- ***Assegnamento***: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante **`NULL`**, per indicare l'indirizzo nullo (non valido).
- Operatore di ***de-referenziazione*** **`*`**: 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 è allocata la variabile.
- Operatori ***aritmetici*** (v. vettori & puntatori).
- Operatori ***relazionali***: **`>`**, **`<`**, **`==`**, **`!=`**.

Esempio:

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



Operatore indirizzo &

- **`&`** si applica solo ad ***oggetti che esistono in memoria*** (quindi, già ***definiti***).
- **`&` non è applicabile ad espressioni** (perché?).

Operatore de-referenziazione *

- 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>;
```

dove:

- <TipoElementoPuntato> è il tipo della variabile puntata;
- <NomePuntatore> è il nome del tipo dichiarato.

Esempio:

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


Puntatori

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

☞Il compilatore può 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 è possibile classificare le variabili in base al loro tempo di vita. Si possono individuare due categorie:

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

Variabili automatiche

- L'allocazione e la de-allocazione è effettuata automaticamente dal sistema (senza l'intervento del programmatore).
- Possiedono un nome, attraverso il quale ci si può riferire ad esse.
- Il programmatore non ha la possibilità di influire sul loro tempo di vita (che è stabilito dalla loro definizione).

Variabili dinamiche

- Devono essere allocate e de-allocate esplicitamente dal programmatore.
- L'area di memoria in cui vengono allocate si chiama ***heap***.
- Non hanno un identificatore associato ad esse, ma ci si può riferire ad esse soltanto attraverso il loro indirizzo (mediante i puntatori).
- Il loro tempo di vita è l'intervallo di tempo che intercorre l'allocazione e la de-allocazione (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** e **de-allocazione** 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`**>.

`malloc()`:

```
void *malloc(size_t size);
```

- Alloca **`size`** byte sullo ***heap*** e ne restituisce l'indirizzo (come puntatore generico).
- Occorre sapere qual è l'occupazione della variabile dinamica che si vuole allocare.

`free()`:

```
void free(void *ptr);
```

- De-alloca il puntatore generico **`ptr`** liberando lo spazio occupato sullo ***heap***.

`sizeof()`:

- Operatore che, dato un tipo o una variabile, restituisce il numero di byte necessari per la sua memorizzazione.

Variabili dinamiche: allocazione

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

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

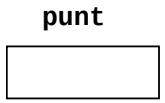
- **`tipodato`** è il tipo della variabile puntata;
- **`punt`** è una variabile di tipo **`tipodato *`**;
- Dal momento che il tipo restituito dalla funzione **`malloc`** è **`(void *)`**, è necessario convertirlo esplicitamente (*casting*).

Significato

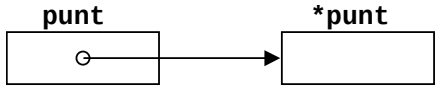
La funzione **`malloc`** provoca la creazione di una variabile dinamica nell'***heap*** e restituisce come valore l'indirizzo della variabile creata.

Esempio:

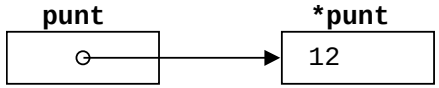
```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
```




```
punt=(tp )malloc(sizeof(int));
```




```
*punt=12
```

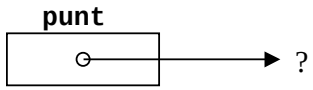


Variabili dinamiche: de-allocazione

Si rilascia la memoria allocata dinamicamente con l'istruzione:

```
free(punt);
```

dove **punt** è l'indirizzo della variabile da de-allocare.



Dopo questa operazione, la cella di memoria occupata da ***punt** viene de-allocata: ***punt** non esiste più.

Esempio:

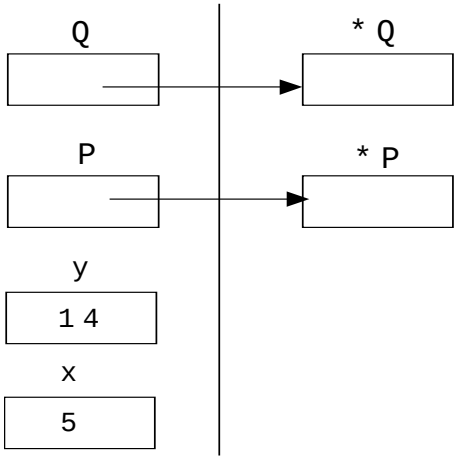
```
main()
{
 char A, *p;

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


Esempio:

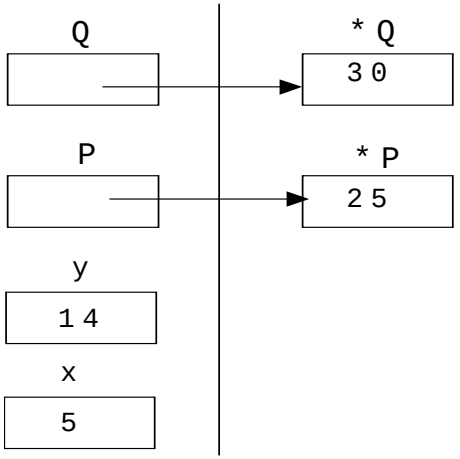
```
#include <stdlib.h>
main()
{
 int *p;
  /* definizione del puntatore p ad
     intero; il contenuto di p non è
     ancora definito */

 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:

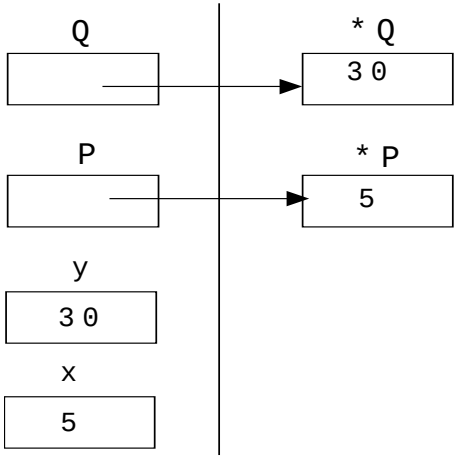
```
main()
{
int *P, *Q, x, y;

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



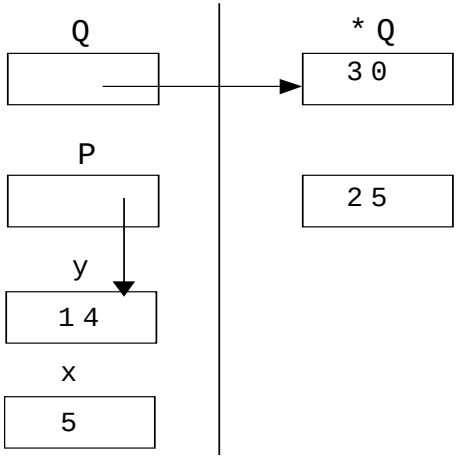

```
*P = 25;
*Q = 30;
```




```
*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!

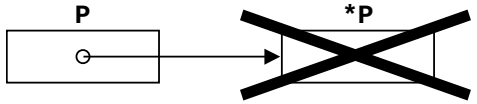
Problemi legati all'uso dei puntatori

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! */
```



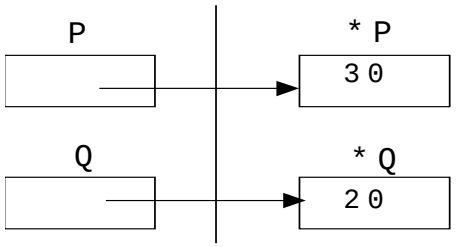
Problemi legati all'uso dei puntatori

Aree inutilizzabili

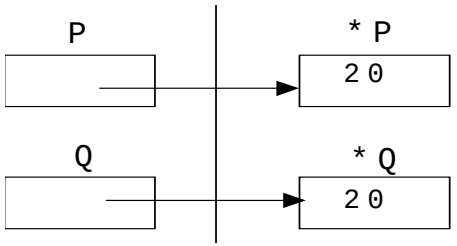
- Possibilità di perdere il riferimento ad aree di memoria allocate al programma (non più riusabili).

Esempio:

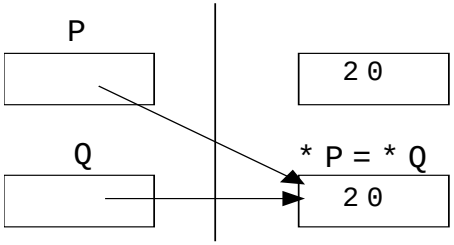
```
int *P,*Q;
P = (int *) malloc ( sizeof (int));
Q = (int *) malloc ( sizeof (int));
*P = 30;
*Q = 20;
```




```
*P = *Q;
```




```
P = Q;
```



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

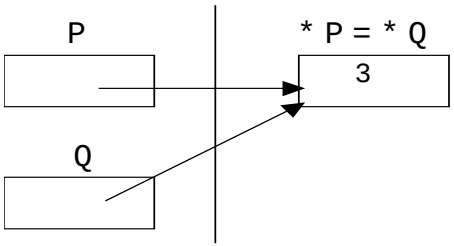
Problemi legati all'uso dei puntatori

Aliasing

- Possibilità di fare riferimento alla stessa variabile con puntatori diversi:

Esempio:

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




```
*q = 10; /* anche *p è cambiato! */
```

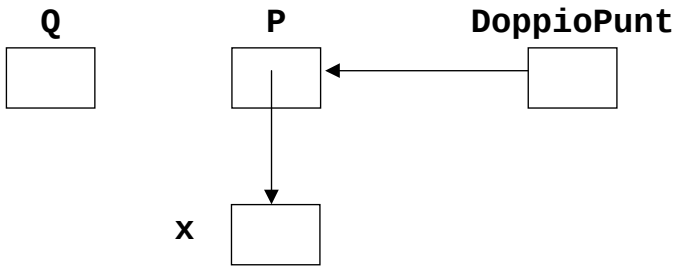

Puntatori a puntatori (*handle*)

Un puntatore può puntare a variabili di tipo qualunque (semplici o strutturate) ➠ può puntare anche a un puntatore:

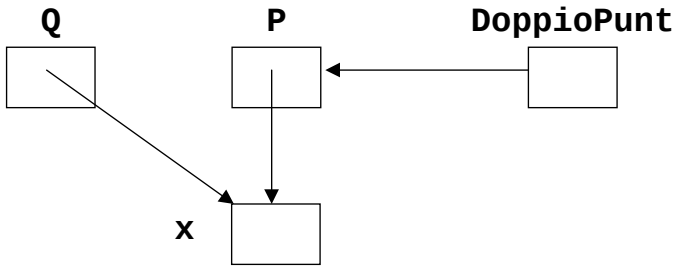
```
[typedef] TipoDato **TipoPunt;
```

Ad esempio:

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




```
DoppioPunt = &P;

Q = *DoppioPunt;
```



Vettori & Puntatori

- 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.

Esempi:

```
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**.

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 & Puntatori

- In C ogni riferimento ad un elemento di un vettore è espanso come un *puntatore de-referenziato*:

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

Esempio:

```
main()
{
char a[] = "0123456789"; /*a è 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`**.

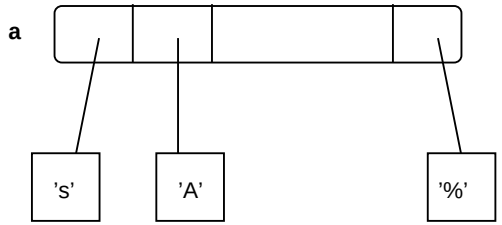
Vettori & Puntatori

☞**`[ ]`** 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

È possibile utilizzare i puntatori per accedere a variabili di tipo **`struct`**.

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:

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 con 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;

 scanf("%d", &N);
/* allocazione del vettore */
 p=(parola *)malloc(N*sizeof(parola));
/* lettura della sequenza */
 for(i=0; i<N; i++)
  gets(&p[i]);
 for(i=N-1; i>=0; i--) { /* stampa */
  j=19;
  do     j--;
  while(p[i][j]!='\0');
  for(j--;j>=0; j--)
   putchar(p[i][j]);
  printf("\n");
 }
 free(p); /* deallocazione */
}
```