

# Linguaggio C: i puntatori

# Il puntatore

E' un tipo di dato scalare, che consente di rappresentare gli **indirizzi** delle variabili allocate in memoria.

## Dominio:

Il dominio di una variabile di tipo puntatore è un insieme di indirizzi: il valore di una variabile P di tipo puntatore può essere l'**indirizzo** di un'altra variabile (variabile **puntata** ).



# Il puntatore in C

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

## Definizione di una variabile puntatore:

```
<TipoElementoPuntato> *<NomePuntatore>;
```

dove:

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

## Ad esempio:

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

# Il puntatore in C

## Operatori:

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

## Ad esempio:

```
int *p1, *p2;
```

```
int A;
```

```
p1 = &A;
```

```
*p1 = 127;
```

```
p2 = p1;
```

```
p1 = NULL; /* NULL e' la costante che denota il  
puntatore nullo */
```



# Il puntatore in C

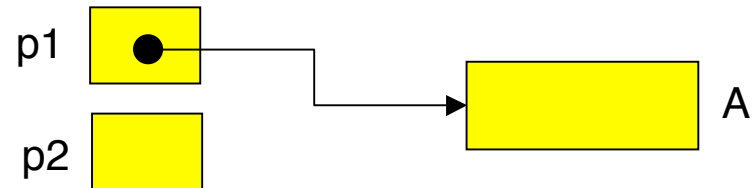
## Operatori:

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

## Ad esempio:

```
int *p1, *p2;
```

```
➔ int A;  
  p1 = &A;  
  *p1 = 127;  
  p2 = p1;  
  p1 = NULL;
```



# Il puntatore in C

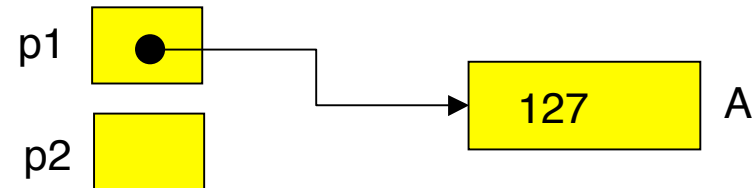
## Operatori:

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

## Ad esempio:

```
int *p1, *p2;  
int A;
```

```
➔ p1 = &A;  
  *p1 = 127;  
  p2 = p1;  
  p1 = NULL;
```



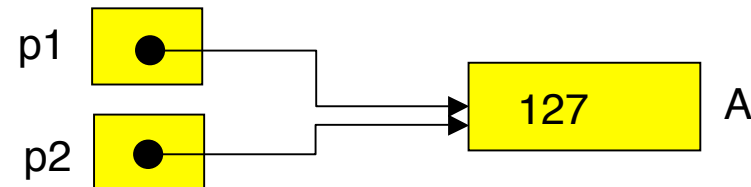
# Il puntatore in C

## Operatori:

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

## Ad esempio:

```
int *p1, *p2;  
int A;  
p1 = &A;  
➔ *p1 = 127;  
p2 = p1;  
p1 = NULL;
```



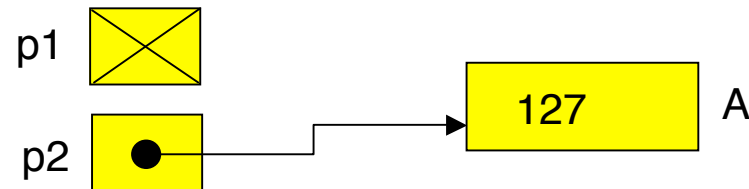
# Il puntatore in C

## Operatori:

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

## Ad esempio:

```
int *p1, *p2;  
int A;  
p1 = &A;  
*p1 = 127;  
➡ p2 = p1;  
p1 = NULL;
```





# Il puntatore in C: \* e &

## Operatore Indirizzo &:

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

## Operatore Dereferencing \*:

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

## Ad esempio:

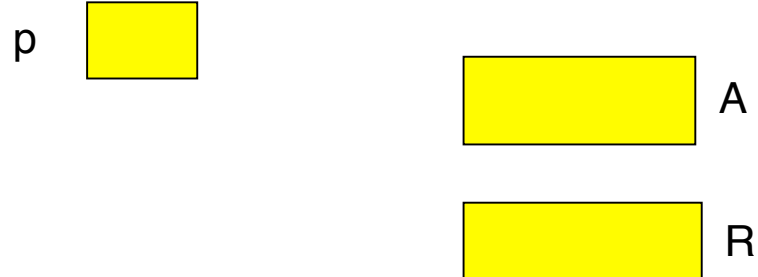
```
float *p;
```

```
float R, A;
```

```
p=&A; /* *p è un alias di A*/
```

```
R=2;
```

```
*p=3.14*R; /* A è modificato */
```



# Il puntatore in C: \* e &

## Operatore Indirizzo &:

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

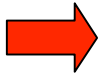
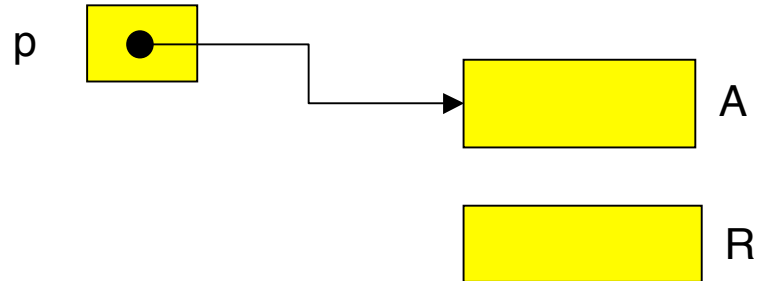
## Operatore Dereferencing \*:

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

## Ad esempio:

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

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



# Il puntatore in C: \* e &

## Operatore Indirizzo &:

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

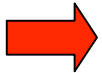
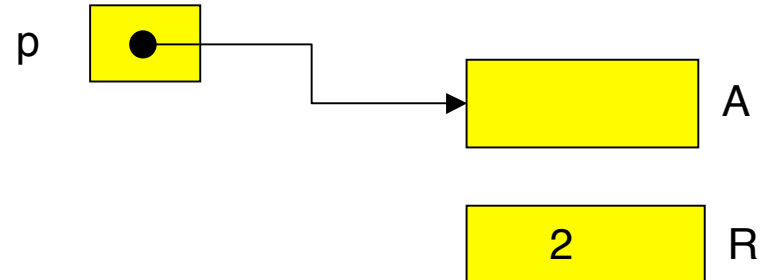
## Operatore Dereferencing \*:

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

## Ad esempio:

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

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



# Il puntatore in C: \* e &

## Operatore Indirizzo &:

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

## Operatore Dereferencing \*:

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

## Ad esempio:

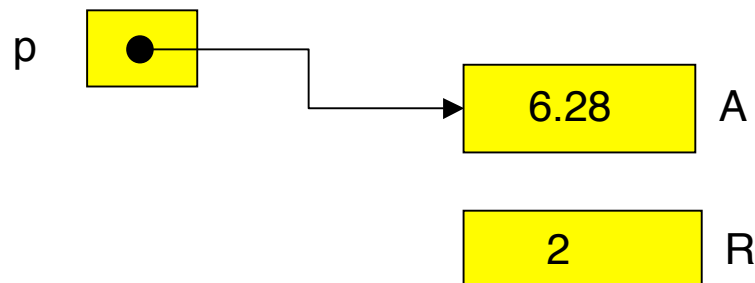
```
float *p;
```

```
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

Il costruttore di tipo "\*" puo` essere anche usato per dichiarare tipi non primitivi basati sul puntatore.

## 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;  
*p=0.56;
```

## Puntatori: controlli di tipo

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{int campo1; float campo2;}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 e' possibile classificare le variabili in base al loro tempo di vita.

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.
- ➔ tutte le variabili viste finora rientrano nella categoria delle *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*, 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 tra l'allocazione e la deallocazione (che sono impartite esplicitamente dal programmatore).



# Variabili Dinamiche in C

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

- **Allocazione:** `malloc`
- **Deallocazione:** `free`

`malloc` e `free` sono definite a livello di sistema operativo, mediante la libreria standard `<stdlib.h>` (da includere nei programmi che le usano).

# Variabili Dinamiche in C

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

- **Allocazione:** `malloc`
- **Deallocazione:** `free`

`malloc` e `free` sono definite a livello di **sistema operativo**, mediante la libreria standard `<stdlib.h>` (da includere nei programmi che le usano).

# Variabili Dinamiche

## Allocazione di variabili dinamiche:

La memoria dinamica viene allocata con la funzione standard `malloc`. La sintassi da usare e`:

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

dove:

- `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 l'indirizzo della variabile creata.

# Variabili Dinamiche

## Esempio:

```
#include <stdlib.h>
```

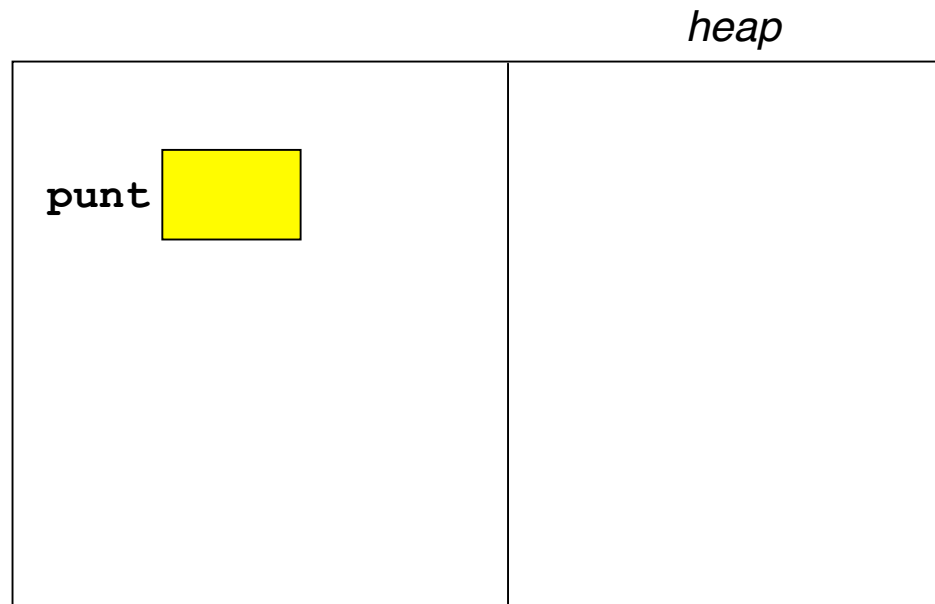
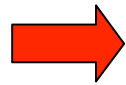
```
typedef int *tp;
```

```
tp punt;
```

```
...
```

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

```
*punt=12;
```



# Variabili Dinamiche

## Esempio:

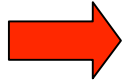
```
#include <stdlib.h>
```

```
typedef int *tp;
```

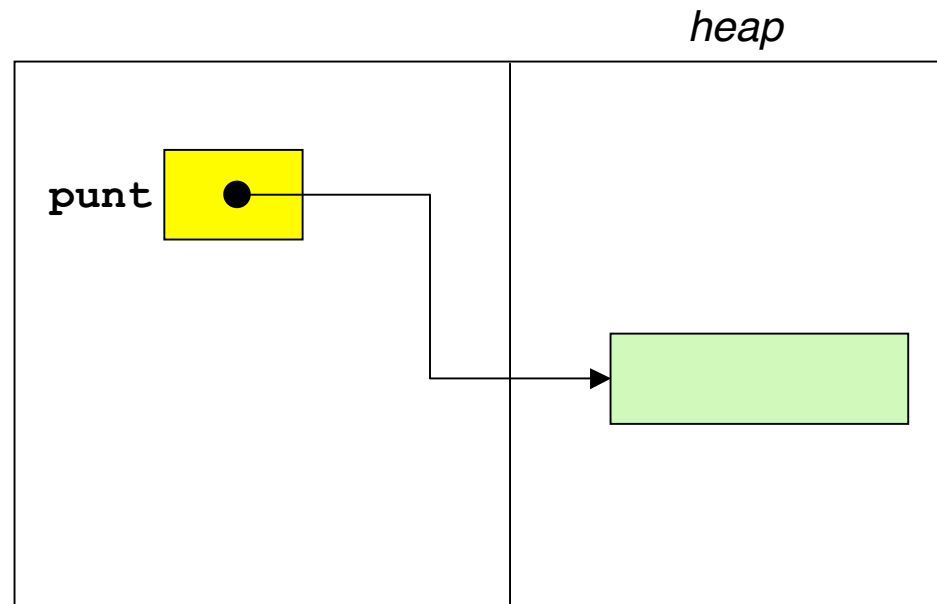
```
tp punt;
```

```
...
```

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



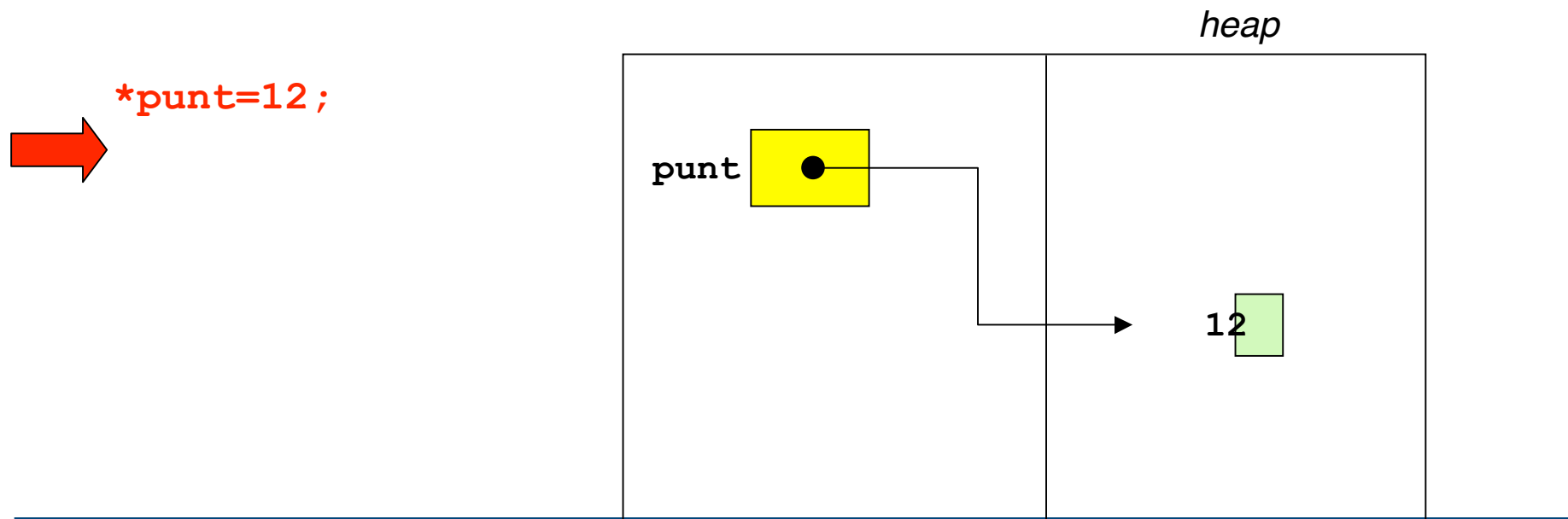
```
*punt=12;
```



# Variabili Dinamiche

## Esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int)) ;
```



# Variabili dinamiche

## Deallocazione:

Si rilascia la memoria allocata dinamicamente con:

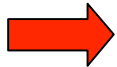
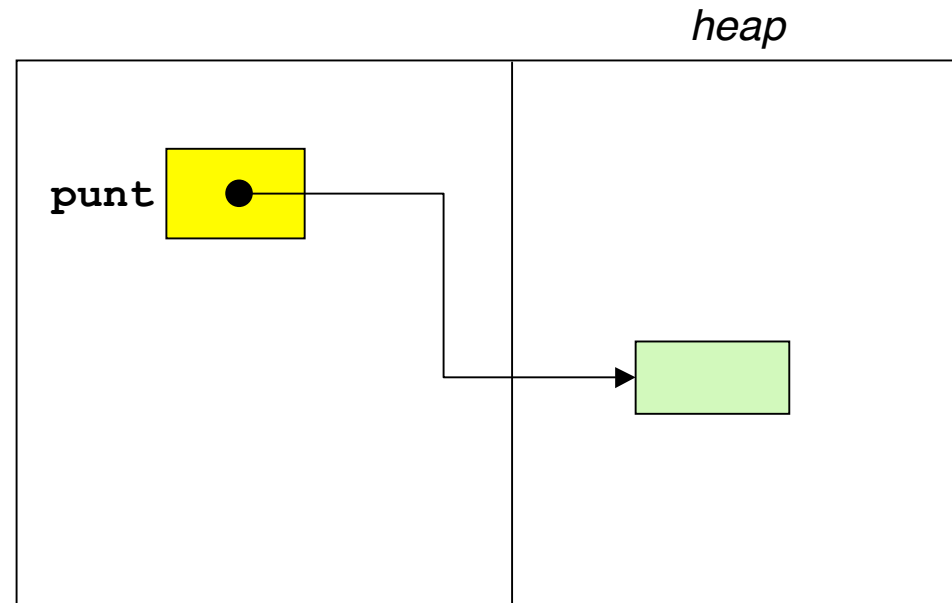
```
free (punt) ;
```

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

→ Dopo questa operazione, la cella di memoria occupata da `*punt` viene liberata: *\*punt non esiste piu`*.

### Esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int));
*punt=12;
...<uso di punt>...
free (punt) ;
```



# Variabili dinamiche

## Deallocazione:

Si rilascia la memoria allocata dinamicamente con:

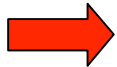
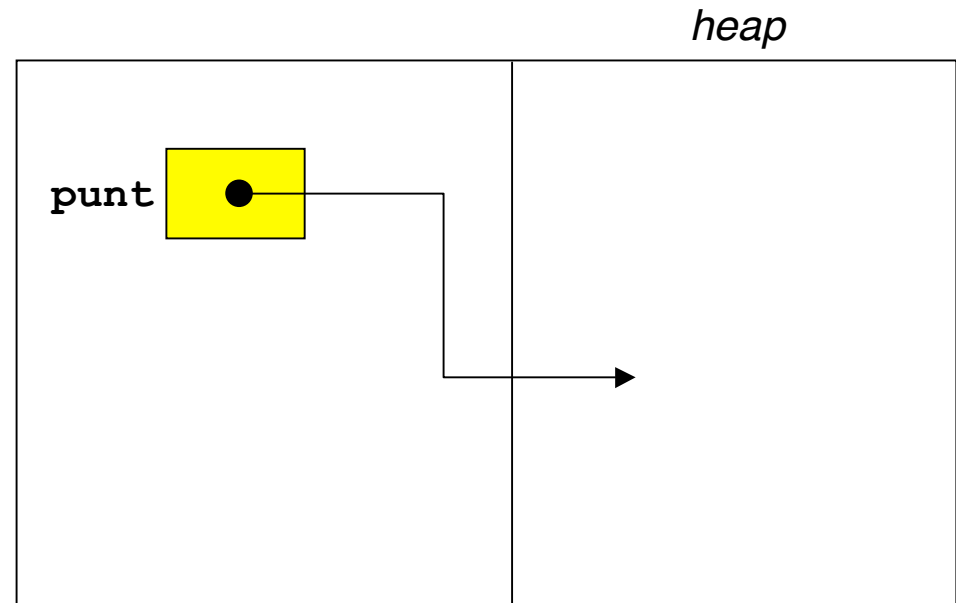
```
free (punt) ;
```

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

→ Dopo questa operazione, la cella di memoria occupata da `*punt` viene liberata: *\*punt non esiste piu`*.

### Esempio:

```
#include <stdlib.h>
typedef int *tp;
tp punt;
...
punt=(tp )malloc(sizeof(int));
*punt=12;
...<uso di punt>...
free (punt) ;
```

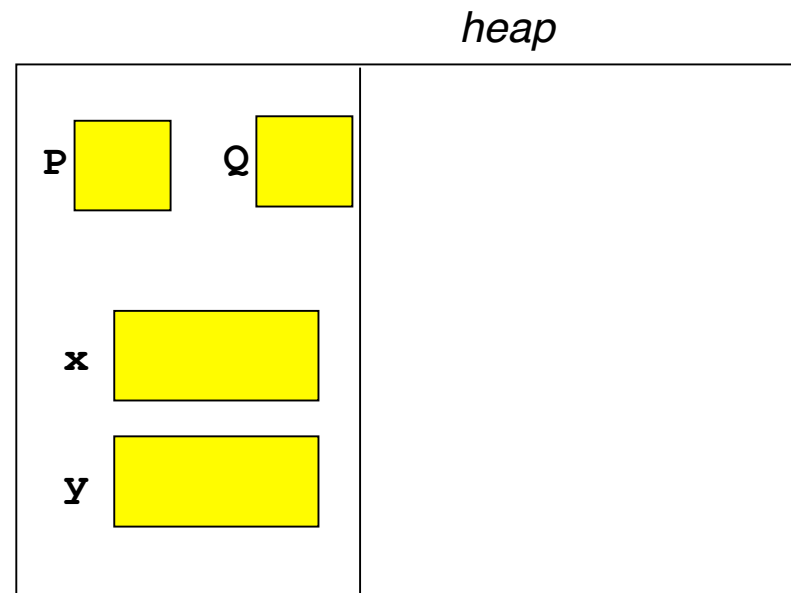




## Puntatori e variabili dinamiche: esempio

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

```
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 = &x;
}
```



# Puntatori e variabili dinamiche: esempio

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

```
main()
```

```
{ int *P, *Q, x, y;
```

```
  x=5;
```

```
  y=14;
```

```
  → P=(int *)malloc(sizeof(int));
```

```
  Q=(int *)malloc(sizeof(int));
```

```
  *P = 25;
```

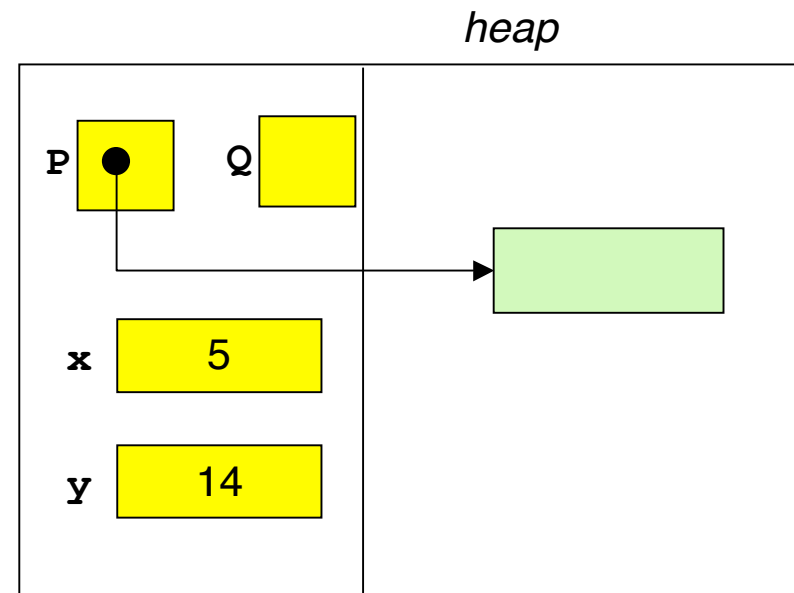
```
  *Q = 30;
```

```
  *P = x;
```

```
  y = *Q;
```

```
  P = &x;
```

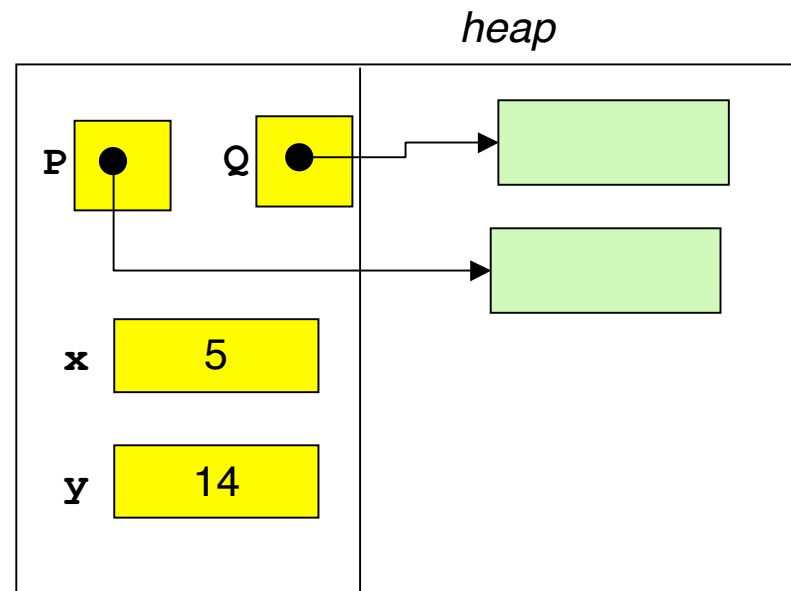
```
}
```



## Puntatori e variabili dinamiche: esempio

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

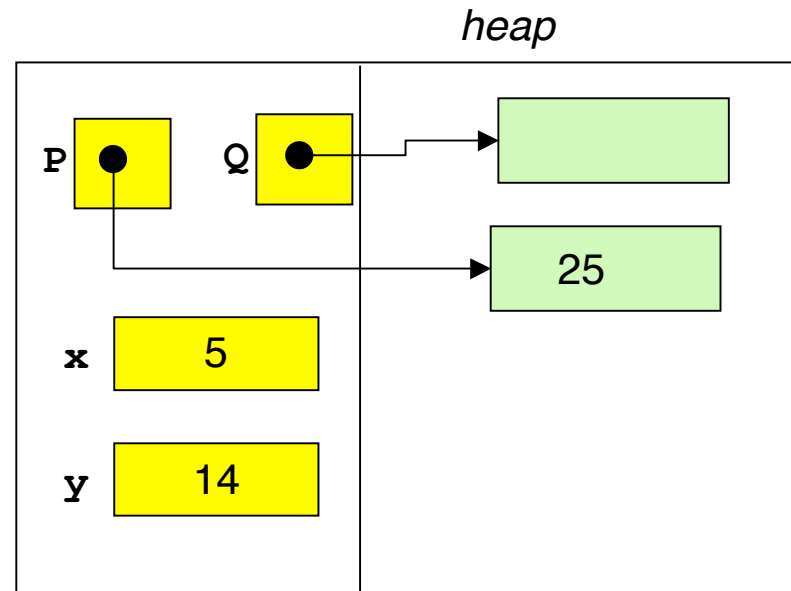
```
main()  
{ int *P, *Q, x, y;  
  x=5;  
  y=14;  
  P=(int *)malloc(sizeof(int));  
  → Q=(int *)malloc(sizeof(int));  
  *P = 25;  
  *Q = 30;  
  *P = x;  
  y = *Q;  
  P = &x;  
}
```



# Puntatori e variabili dinamiche: esempio

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

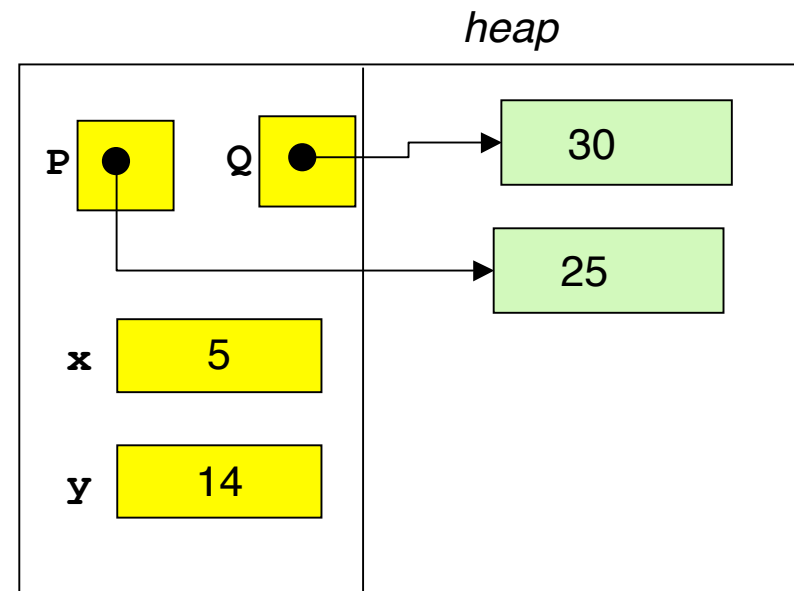
```
main()
{ int *P, *Q, x, y;
  x=5;
  y=14;
  P=(int *)malloc(sizeof(int));
  Q=(int *)malloc(sizeof(int));
  → *P = 25;
    *Q = 30;
    *P = x;
    y = *Q;
    P = &x;
}
```



# Puntatori e variabili dinamiche: esempio

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

```
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 = &x;  
}
```



# Puntatori e variabili dinamiche: esempio

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

```
main()
```

```
{ int *P, *Q, x, y;
```

```
    x=5;
```

```
    y=14;
```

```
    P=(int *)malloc(sizeof(int));
```

```
    Q=(int *)malloc(sizeof(int));
```

```
    *P = 25;
```

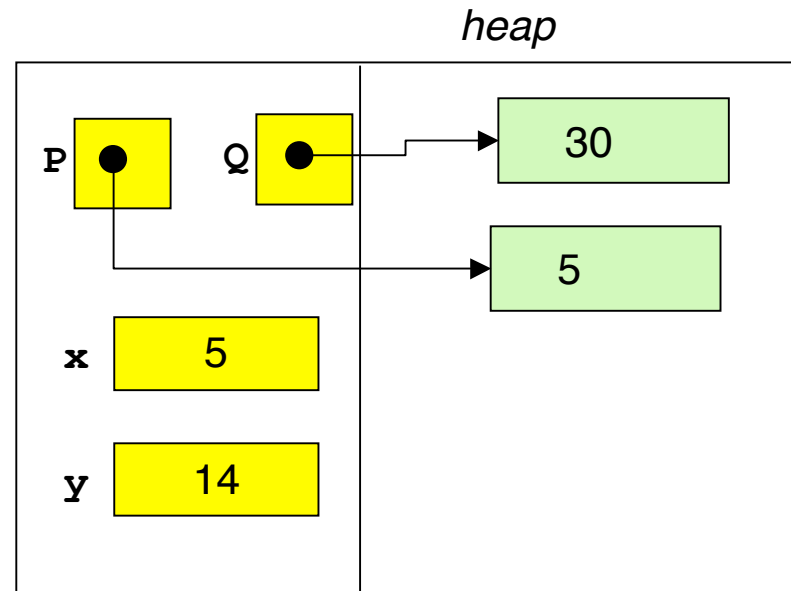
```
    *Q = 30;
```

```
    *P = x;
```

```
    y = *Q;
```

```
    P = &x;
```

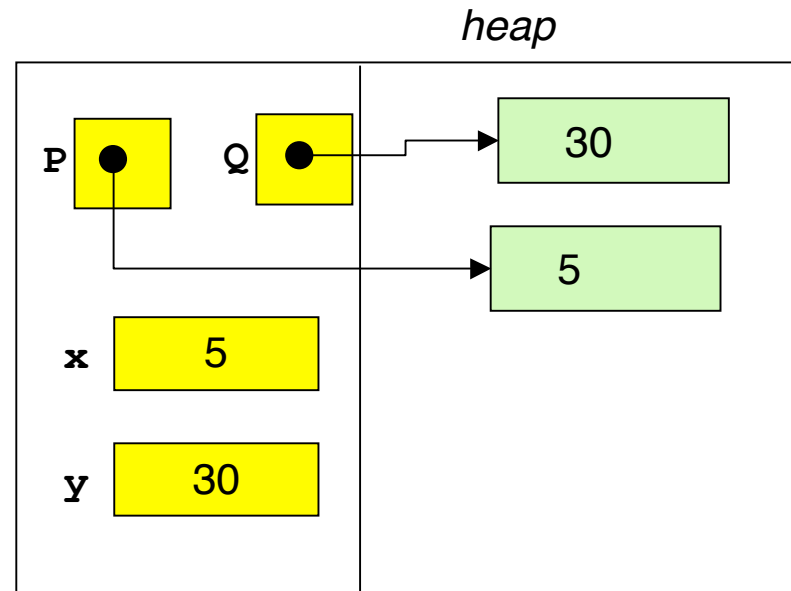
```
}
```



# Puntatori e variabili dinamiche: esempio

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

```
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 = &x;
}
```



# Puntatori e variabili dinamiche: esempio

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

```
main()
```

```
{ int *P, *Q, x, y;
```

```
  x=5;
```

```
  y=14;
```

```
  P=(int *)malloc(sizeof(int));
```

```
  Q=(int *)malloc(sizeof(int));
```

```
  *P = 25;
```

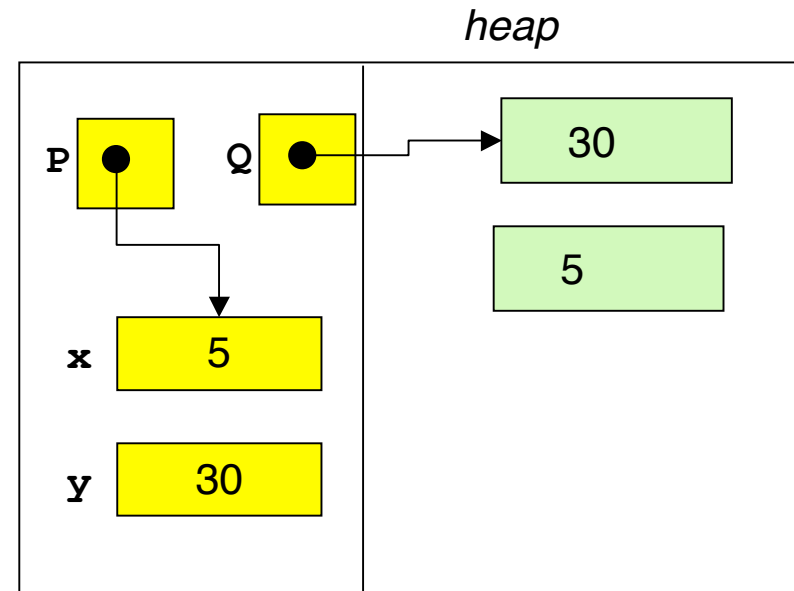
```
  *Q = 30;
```

```
  *P = x;
```

```
  y = *Q;
```

```
  → P = &x;
```

```
}
```



- 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

## 1. Aree inutilizzabili:

Possibilità di perdere l'indirizzo di aree di memoria allocate al programma che quindi non sono più accessibili. (v. esempio precedente).

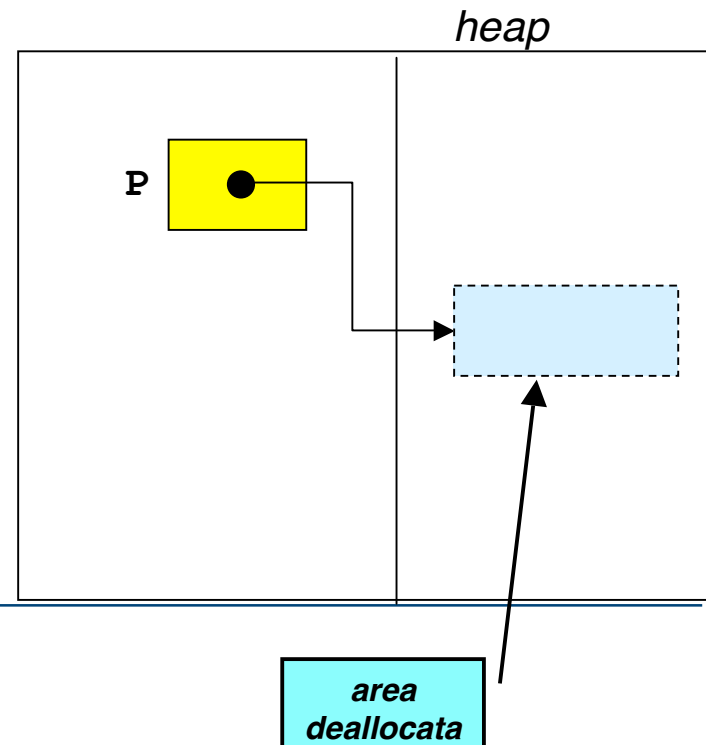
## 2. Riferimenti pendenti (dangling references):

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

## 3. Aliasing:

Possibilita` di riferire la stessa variabile con puntatori diversi.

Ad esempio:

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

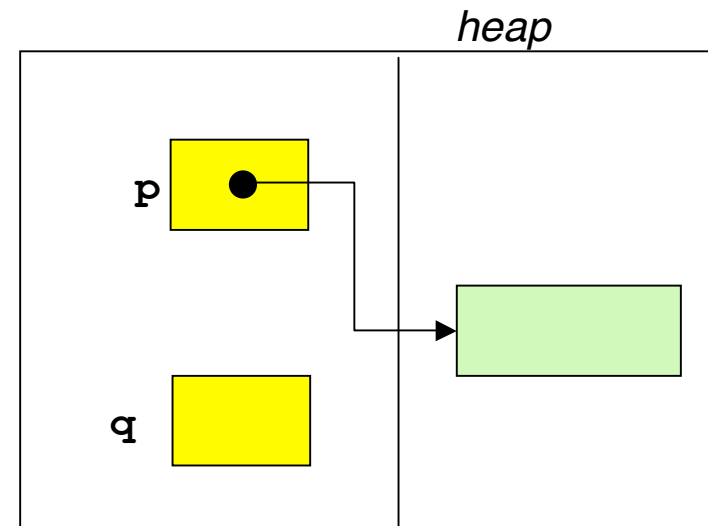
```
p=(int *)malloc(sizeof(int));
```

➔ 

```
*p=3;
```

```
q=p; /*p e q puntano alla stessa  
      variabile */
```

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



# Problemi legati all'uso dei Puntatori

## 3. Aliasing:

Possibilita` di riferire la stessa variabile con puntatori diversi.

**Ad esempio:**

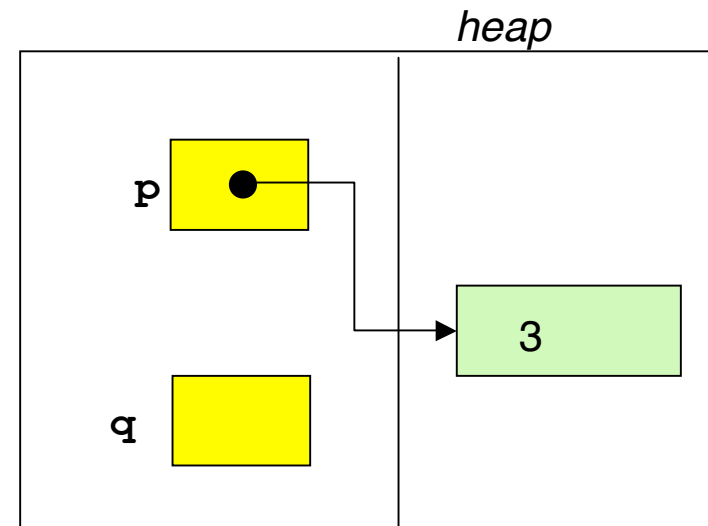
```
int *p, *q;
```

```
p=(int *)malloc(sizeof(int));
```

```
*p=3;
```

➔ `q=p; /*p e q puntano alla stessa  
variabile */`

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



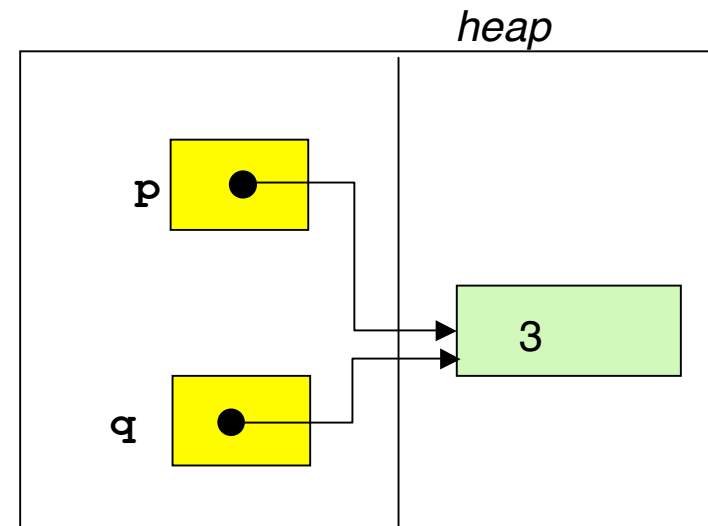
# Problemi legati all'uso dei Puntatori

## 3. Aliasing:

Possibilita` 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! */
```



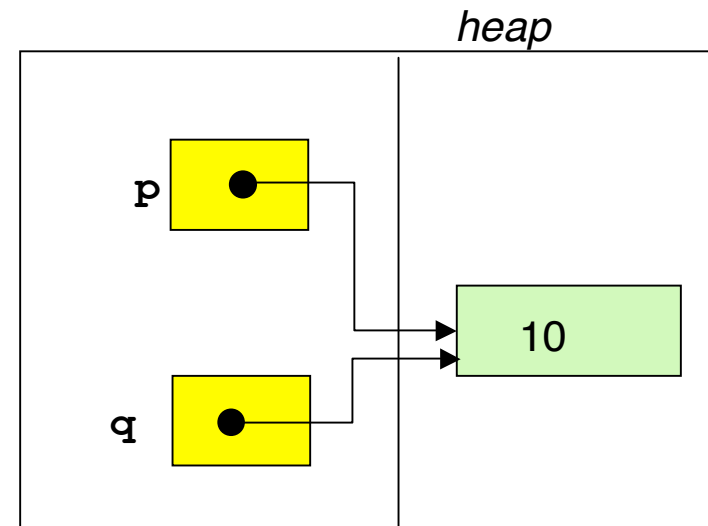
# Problemi legati all'uso dei Puntatori

## 3. Aliasing:

Possibilita` 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

Un puntatore puo' *puntare* a variabili di tipo qualunque (semplici o strutturate):

→ puo' *puntare* anche a un puntatore:

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

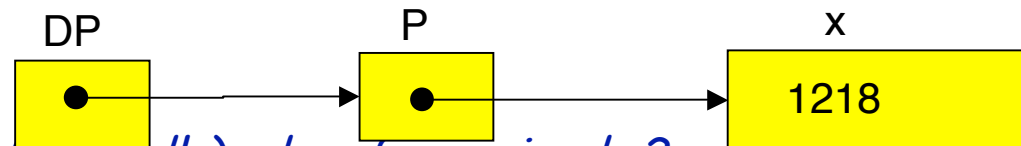
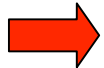
**Ad esempio:**

```
int x, *P, **DP;
```

```
P = &x;
```

```
Dp = &P;
```

```
**DP=1218;
```



→ DP e' un **doppio puntatore** (o *handle*): *dereferenziando 2 volte DP*, si accede alla variabile puntata dalla "catena" di riferimenti.

# Vettori & Puntatori

Nel linguaggio C, i vettori sono rappresentati mediante puntatori:

→ il **nome** di una variabile di tipo vettore denota l'**indirizzo del primo elemento** del vettore.

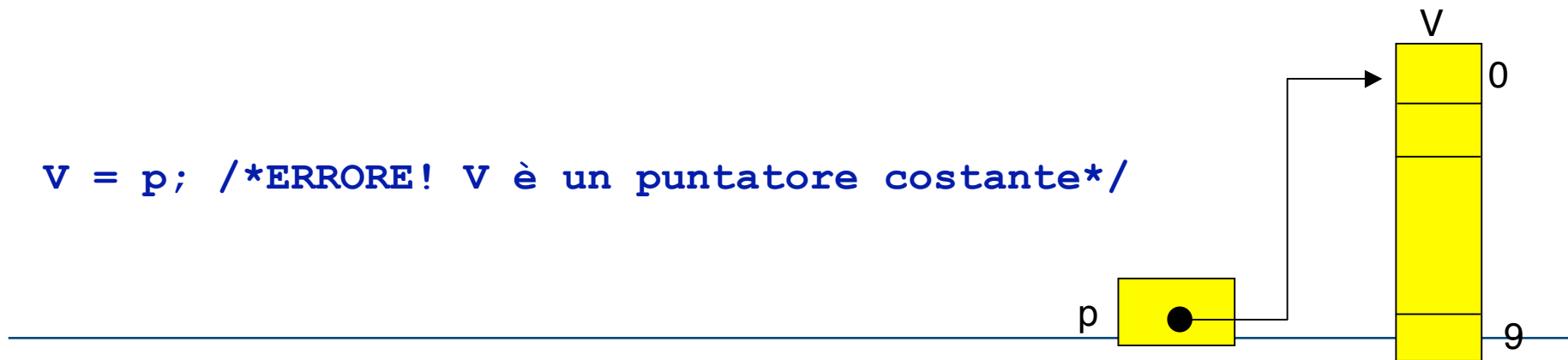
**Ad esempio:**

```
float V[10]; /*V è una costante di tipo puntatore:  
            V equivale a &V[0];  
            V e` un puntatore (costante!) a float  
            */
```

```
float *p;
```

```
→ p=V; /* p punta a V[0] */  
*p=0.15; /* equivale a V[0]=0.15 */
```

```
V = p; /*ERRORE! V è un puntatore costante*/
```



# Vettori & Puntatori

Nel linguaggio C, i vettori sono rappresentati mediante puntatori:

il **nome** di una variabile di tipo vettore denota l'**indirizzo del primo elemento** del vettore.

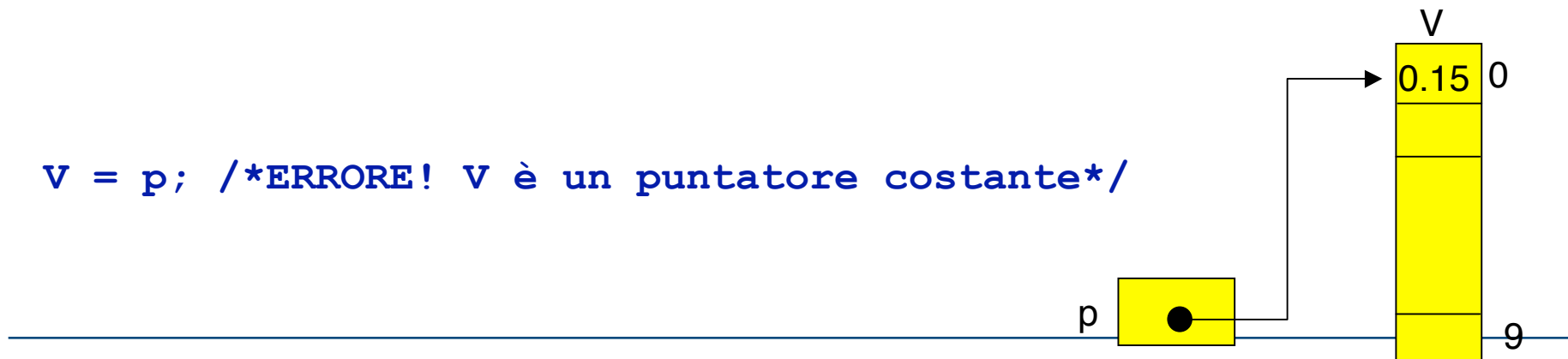
## Ad esempio:

```
float V[10]; /*V è una costante di tipo puntatore:
              V equivale a &V[0];
              V e` un puntatore (costante!) a float
              */
```

```
float *p;
p=V; /* p punta a V[0] */
```

➔ **\*p=0.15; /\* equivale a V[0]=0.15 \*/**

**V = p; /\*ERRORE! V è un puntatore costante\*/**





## Operatori *aritmetici* su puntatori a vettori

Nel linguaggio C, gli elementi di un vettore vengono allocati in memoria in **parole consecutive** (cioè, in celle fisicamente adiacenti), la cui dimensione dipende dal tipo dell'elemento.

→ Conoscendo l'indirizzo del primo elemento e la dimensione dell'elemento, è possibile calcolare l'indirizzo di qualunque elemento del vettore:

### Operatori *aritmetici* (somma e sottrazione) 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 puntato da  $V$ ;
- $(V-i)$  restituisce l'indirizzo dell'elemento spostato di  $i$  posizioni all'indietro rispetto a quello puntato da  $V$ ;
- $(V-W)$  restituisce l'intero che rappresenta il numero di elementi compresi tra  $V$  e  $W$ .

# Operatori *aritmetici* su puntatori a vettori

## Esempio:

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

```
main()
```

```
{ float V[8]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8};
```

```
    int k;
```

```
    float *p, *q;
```

```
    p=V+7;
```

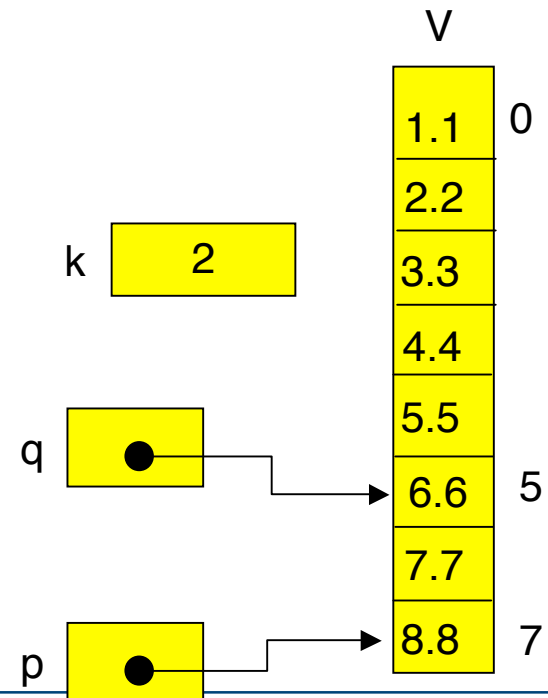
```
    q=p-2;
```

```
    k=p-q;
```

```
    printf("%f,\t%f,\t%d\n",*p, *q, k);
```

```
}
```

➔ **Stampa:** 8.8, 6.6, 2



# Vettori e Puntatori

Durante l'esecuzione di ogni programma C, ogni riferimento ad un elemento di un vettore è tradotto in un puntatore dereferenziato; per esempio:

V[0]	viene tradotto in	*(V)
V[1]	viene tradotto in	*(V + 1)
V[i]	viene tradotto in	*(V+i)
V[expr]	viene tradotto in	*(V + expr)

## Esempio:

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

```
main ()
{
    char a[ ] = "0123456789"; /* a e` un vettore di 10 char */
    int i = 5;
    printf("%c%c%c%c%c\n", a[i], a[5], i[a], 5[a], (i-1)[a]); /* !!! */
}
```

→ **Stampa:**            5 5 5 5 4

**NB:** Per il compilatore `a[i]` e `i[a]` sono lo stesso elemento, perché viene sempre eseguita la conversione: `a[i]` => `*(a+i)` (senza eseguire alcun controllo ne` su `a`, ne` su `i`).

# Complementi sui puntatori

## Vettori di puntatori:

Il costruttore [ ] ha precedenza rispetto al costruttore \*. Quindi:  
`char *a[10];` equivale a `char *(a[10]);`

→ a è un vettore di 10 puntatori a carattere.

**NB:** 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, tenendo conto che il punto della notazione postfissa ha la precedenza sull'operatore di dereferencing \*.

## Esempio:

```
typedef struct{ int Campo_1,Campo_2; } TipoDato;  
TipoDato S, *P;  
P = &S;  
(*P).Campo1=75; /* assegnamento della costante 75 al Campo1 della  
                 struct puntata da P* (e` necessario usare le  
                 parentesi) */
```

## Operatore ->:

L'operatore -> consente di accedere ad un campo di una struttura referenziata da un puntatore in modo più sintetico: `P->Campo1=75;`

## Esercizio

Si vuole realizzare un programma che, data da input una sequenza di  $N$  parole (ognuna, al massimo, di 20 caratteri), stampi in ordine **inverso** le parole date, ognuna "**ribaltata**" (cioè, stampando i caratteri in ordine inverso: dall'ultimo al primo). Si supponga che  $N$  non sia noto a priori, ma venga fornito da input. Utilizzare una struttura dinamica.

### Progetto dei dati.

Memorizziamo le parole in un vettore di  $N$  stringhe che verterà allocato dinamicamente.

```
typedef char parola[20]; /*tipo associato alla  
                           singola parola*/  
parola *p; /* puntatore per l'accesso al  
            vettore delle parole
```

# Soluzione

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

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

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

# Esercizio

```
/* ..Continuua: stampa */
for(i=N-1; i>=0; i--)
{
    j=19;
    do
        j--;
    while(p[i][j]!='\0');
    for(j--;j>=0; j--)
        printf("%c",p[i][j]);
    printf("\n");
}
free(p); /* deallocazione del vettore*/
}
```