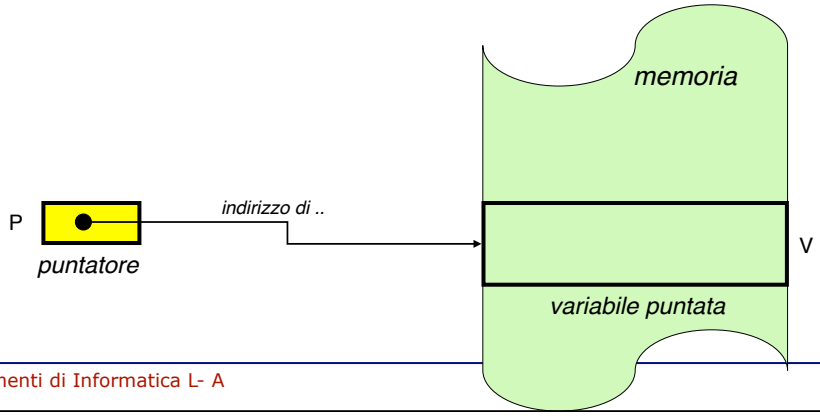
# Il linguaggio C
## I puntatori



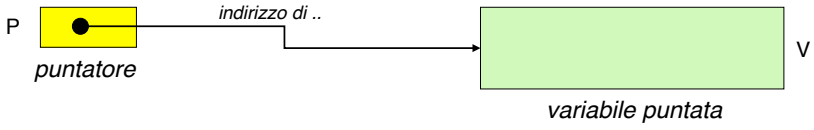



# Il puntatore

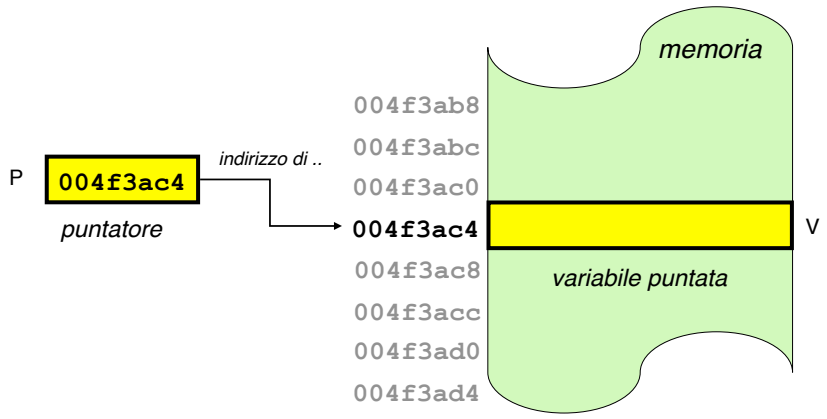
È un tipo di dato scalare, che consente di rappresentare gli **indirizzi** delle variabili allocate in memoria.

**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*** ).



# Esempio








# Il puntatore in C

In C i puntatori si definiscono mediante il costruttore *.

**Definizione di una variabile puntatore:**

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

dove:

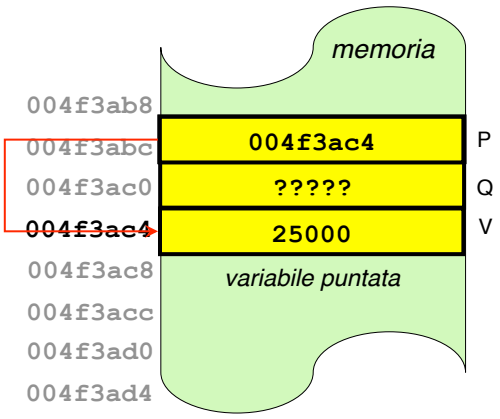
- `<TipoElementoPuntato>` è il tipo della variabile puntata
- `<NomePuntatore>` è il nome della variabile di tipo puntatore
- il simbolo * è il costruttore del tipo puntatore.

**Ad esempio:**

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

# Esempio

```
int *P,*Q,
     V=25000;
P=&V;
```







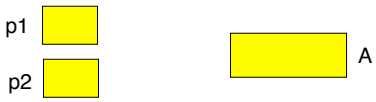

# Il puntatore in C

**Operatori**:

- ***Assegnamento* (=)**: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante NULL, per indicare l'indirizzo nullo (0).
- ***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*** (domani).
- ***Operatori relazionali***:>,<,==, !=
- **Operatore indirizzo &**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale è allocata la variabile.

**Ad esempio:**

```
int *p1, *p2;
int A;
p1 = &A;
*p1 = 127;
p2 = p1;
p1 = NULL;/* NULL è la costante che vale 0 e
       denota il puntatore "nullo" */
```



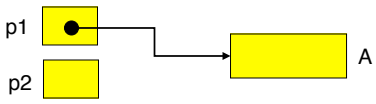
# Il puntatore in C

**Operatori**:

- ***Assegnamento* (=)**: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante NULL, per indicare l'indirizzo nullo (0).
- ***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*** (domani).
- ***Operatori relazionali***:>,<,==, !=
- ***Operatore indirizzo* (&)**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale è allocata la variabile.

**Ad esempio:**

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





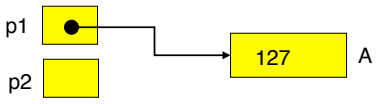

# Il puntatore in C

**Operatori**:

- ***Assegnamento* (=)**: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante NULL, per indicare l'indirizzo nullo (0).
- ***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*** (domani).
- ***Operatori relazionali***:>,<,==, !=
- ***Operatore indirizzo* (&)**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale è allocata la variabile.

**Ad esempio:**

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




Fondamenti di Informatica L- A

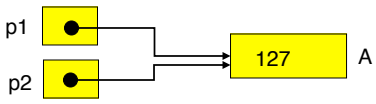
# Il puntatore in C

**Operatori**:

- ***Assegnamento* (=)**: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante NULL, per indicare l'indirizzo nullo (0).
- ***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*** (domani).
- ***Operatori relazionali***:>,<,==, !=
- ***Operatore indirizzo* (&)**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale è allocata la variabile.

**Ad esempio:**

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





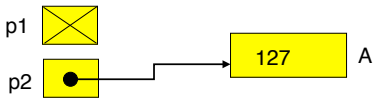

# Il puntatore in C

**Operatori**:

- ***Assegnamento* (=)**: è possibile l'assegnamento tra puntatori (dello stesso tipo). È disponibile la costante NULL, per indicare l'indirizzo nullo (0).
- ***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*** (domani).
- ***Operatori relazionali***:>,<,==, !=
- ***Operatore indirizzo* (&)**: si applica ad una variabile e restituisce l'indirizzo della cella di memoria nella quale è allocata la variabile.

**Ad esempio:**

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




Fondamenti di Informatica L- A

# Il puntatore in C: * e &

**Operatore Indirizzo &:**

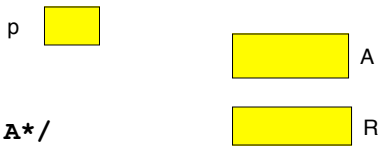
- & si applica solo ad oggetti che esistono in memoria (quindi, già definiti).
- & non è applicabile ad espressioni.

**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 &:**

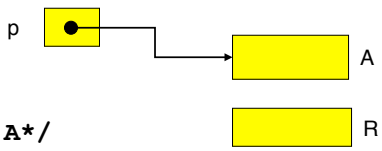
- & si applica solo ad oggetti che esistono in memoria (quindi, gia` definiti).
- & non è applicabile ad espressioni.

**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 */
```




Fondamenti di Informatica L- A

# Il puntatore in C: * e &

**Operatore Indirizzo &:**

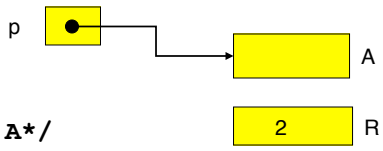
- & si applica solo ad oggetti che esistono in memoria (quindi, gia` definiti).
- & non è applicabile ad espressioni.

**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 &:**

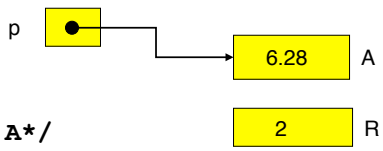
- & si applica solo ad oggetti che esistono in memoria (quindi, gia` definiti).
- & non è applicabile ad espressioni.

**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 */
```




Fondamenti di Informatica L- A

# Puntatore come costruttore di tipo

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

**Dichiarazione di un tipo puntatore:**

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

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

**Ad esempio:**

```
typedef float *tpf;
tpf p;
float f;
p=&f;
*p=0.56;
```



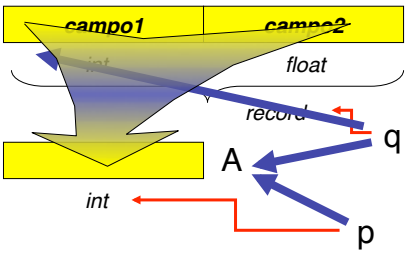
# Puntatori: controlli di tipo

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

➔ il compilatore ***può*** effettuare controlli statici sull'uso dei puntatori.

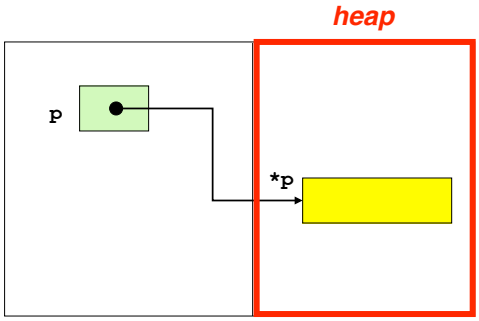
**Esempio:**

```
typedef struct{int campo1; float campo2;}record;
int A, *p;
record X, *q;

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




➔ Viene segnalato dal compilatore (***warning***) il tentativo di utilizzo di un puntatore a un tipo diverso rispetto a quello per cui è stato definito.

# Variabili dinamiche






# Variabili automatiche e dinamiche

In C è possibile classificare le variabili in base al loro tempo di vita.

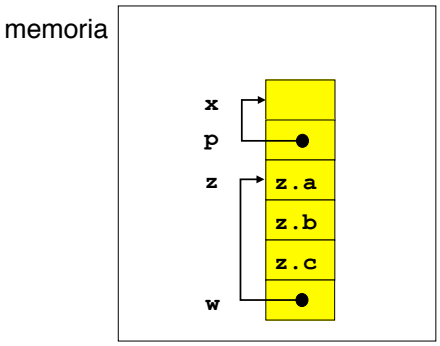
Due categorie:
- variabili ***automatiche***
- variabili ***dinamiche***

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

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

# Variabili automatiche

```
int *p, x;
struct {int a, b, c;} z, *w;
```






# Variabili dinamiche

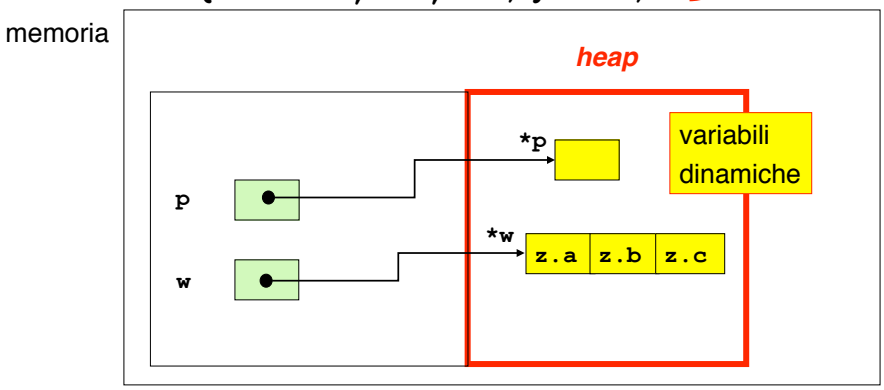
**Variabili dinamiche:**

- 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 è l'intervallo di tempo che intercorre tra l'allocazione e la deallocazione (che sono impartite esplicitamente dal programmatore).

# Variabili dinamiche

```
int *p;
struct {int a, b, c;} *w;
```

variabili automatiche






# 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 è:

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

dove:

- **`tipodato`** è il tipo della variabile puntata
- **`punt`** è una variabile di tipo **`tipodato *`**
- **`sizeof()`** è una funzione standard che calcola il numero di byte che occupa il dato specificato come argomento
- è 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.

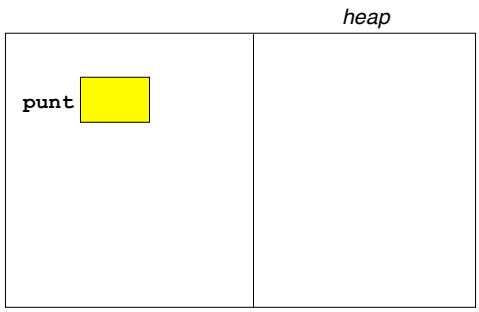


# Variabili Dinamiche

## Esempio:

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

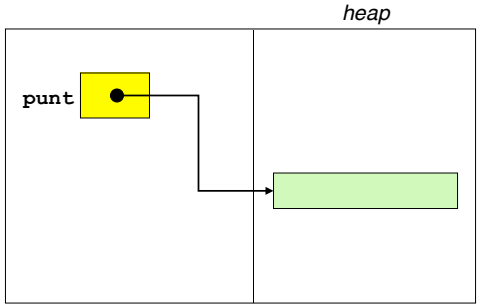
```
*punt=12;
```



# Variabili Dinamiche

## Esempio:

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

*punt=12;
```

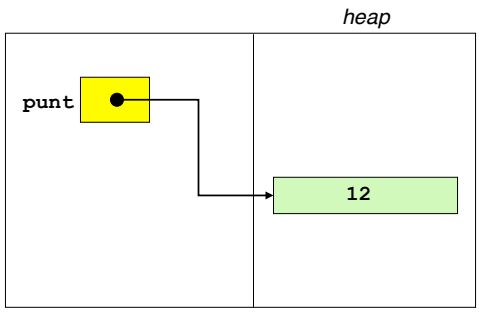





# Variabili Dinamiche

## Esempio:

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

*punt=12;
```



# Variabili dinamiche



**Deallocazione**:

Si rilascia la memoria allocata dinamicamente con:

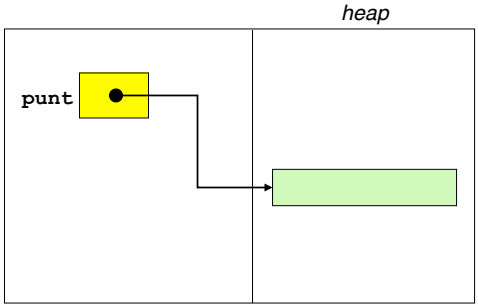
```
free(punt);
```

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

➔ Dopo questa operazione, la cella di memoria occupata da `*punt` viene liberata: `*punt` *non esiste* più.

**Esempio:**

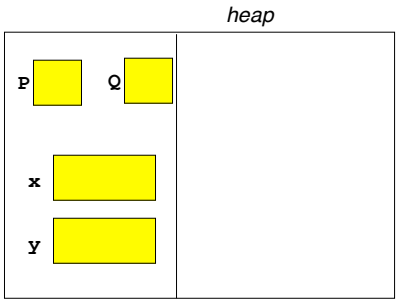
```
#include <stdlib.h>
typedef int *tp;
tp punt;
 ...
punt=(tp )malloc(sizeof(int));
*punt=12;
...<uso di punt>...
free(punt);
```





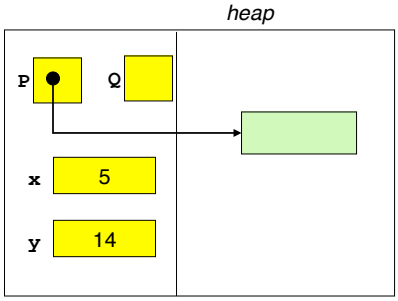

# Puntatori e variabili dinamiche: esempio

```
#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

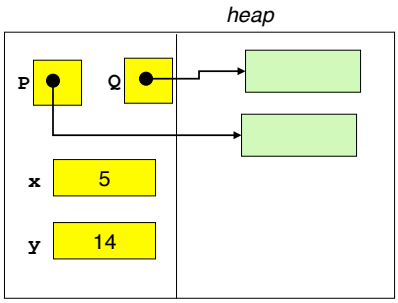
```
#include <stdio.h>

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;
}
```





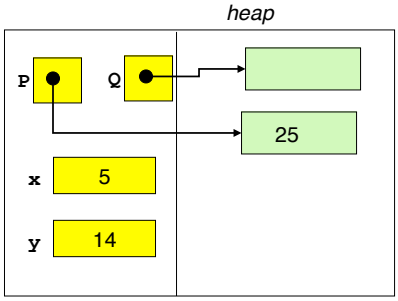

# Puntatori e variabili dinamiche: esempio

```
#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

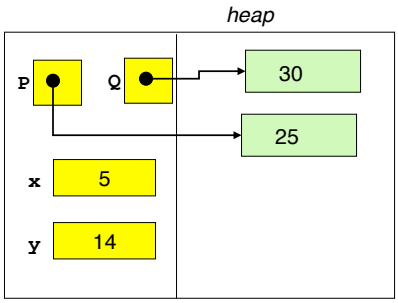
```
#include <stdio.h>

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;
}
```





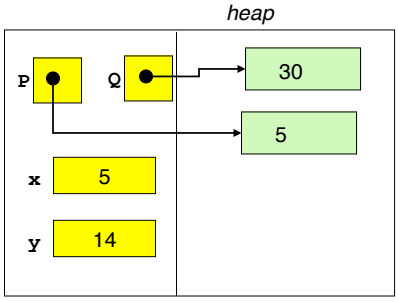

# Puntatori e variabili dinamiche: esempio

```
#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

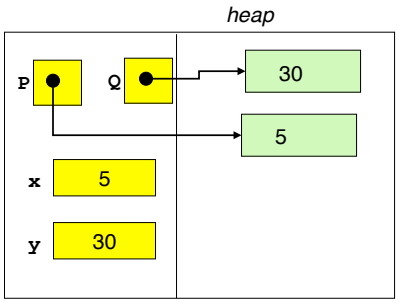
```
#include <stdio.h>

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;
}
```





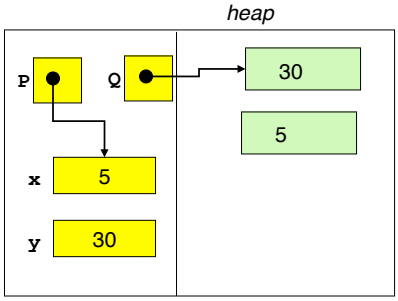

# Puntatori e variabili dinamiche: esempio

```
#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 é più utilizzabile**!



3

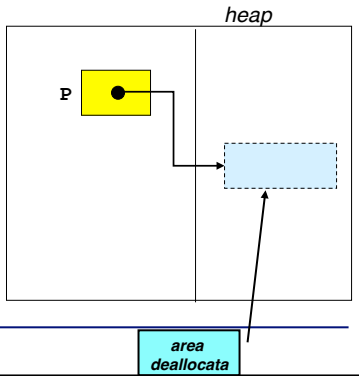
# Problemi legati all'uso dei Puntatori

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!  */
```



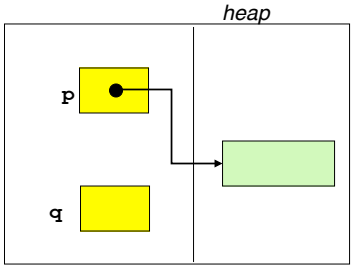
# Problemi legati all'uso dei Puntatori

**3. Aliasing:**

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! */
```





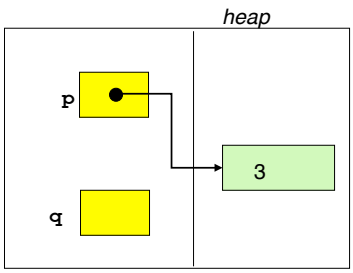

# Problemi legati all'uso dei Puntatori

**3. 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! */
```



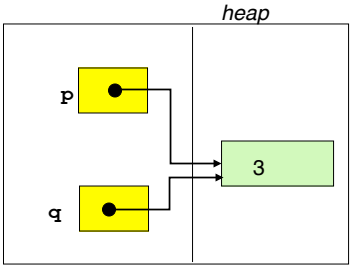
# Problemi legati all'uso dei Puntatori

**3. 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! */
```






# Problemi legati all'uso dei Puntatori

**3. 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! */
```