



- 6° Modulo 2<sup>a</sup> parte
  - Puntatori
    - Costruttori di tipi complessi
    - Uso della memoria dinamica

<http://www.elet.polimi.it/upload/martucci/index.html>

# Uso di Puntatori

<pre>int X,Y,Z; int V[5]={ 20,21,22,23,24 } ; int A, *P, *Q, **PP;</pre>											
Indir		nome		Indir		nome		Indir		nome	
1000		X	X=10;	1000	10	X		1000	124	X	
1002		Y	Y=X+1;	1002	11	Y		1002	11	Y	
1004		Z	Z=X+Y;	1004	21	Z		1004	21	Z	
1006	20	V[0]		1006	20	V[0]		1006	57	V[0]	
1008	21	V[1]		1008	21	V[1]		1008	21	V[1]	
1010	22	V[2]		1010	22	V[2]		1010	22	V[2]	
1012	23	V[3]		1012	23	V[3]		1012	23	V[3]	
1014	24	V[4]		1014	24	V[4]		1014	24	V[4]	
1016		A	V[5]=50; !!	1016	50	A		1016	34	A	
1018		P	P=V;	1018	1006	P	*P=57;	1018	1006	P	
1020		Q	Q=&V[4];	1020	1014	Q	A=*Q+10;	1020	1014	Q	
1022		PP	PP=&Q;	1022	1020	PP	X=**PP+100;	1022	1020	PP	
1024				1024				1024			

# Pointer Types

- pointer variable definitions:

```
int *ip;           // pointer to variable of type int
char *c1, *c2, *c3; // pointer to variables of type char
float **fp;       // pointer to pointer to variable of type float
```

- dereference operator (\*) and address-of operator(&):

```
int i, *ip = &i, **ipp = &ip, ***ippp = &ipp;
// i is a variable of type int
// ip is a pointer to a variable of type int
// ipp is a pointer to a pointer to a variable of type int
// ippp is a pointer to a pointer to a pointer to a variable of type int
double d = 2.7183, *dp = &d;
// the value of d is 2.7183
// the value of &d is an address (where 2.7183 is stored...)
// the value of dp is the address of d.
// the value of *dp is the value of what dp points to (d or 2.7183)
```

- string pointers:

```
char *string = "This is a string.\n";
// strings are null ('\0') terminated by convention
char *same_string = string;
```

---

Irwin Sheer

---

Superconducting Super Collider Laboratory

---

MS 2300, 2550 Beckleymeade Ave., Dallas, TX 75237

---

Tel: (214) 708-1050; Fax: (214) 708-6354

---

e-mail: Irwin\_Sheer@ssc.gov

x	A
y	Z
temp	

```
char x,y,temp;
```

cambia i valori di x e y \*/

```
temp = x ;
x = y ;
y = temp ;
```

dopo la elaborazione

x	Z
y	A
temp	A



```
Typedef struct { int a[20]; char b[100] ..... } RECORD;
RECORD X, Y;
```

X				
Y				

1010  
1460

/\* per scambiare X ed Y si devono eseguire molte istruzioni  
... si copiano tutti i byte \*/

/\* si scrive sempre un breve programma \*/

```
RECORD TEMP;
TEMP= X;
X=Y;
Y=TEMP;
```

```
RECORD * PX, *PY, *PTEMP;
```

PX	1010
PY	1460
PTEMP	

invece senza l'asterisco ...

```
PTEMP = PX ;
PX = PY ;
PY = PTEMP;
```

dopo la esecuzione

PX	1460
PY	1010
PTEMP	1010

## Variabili Dinamiche

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

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

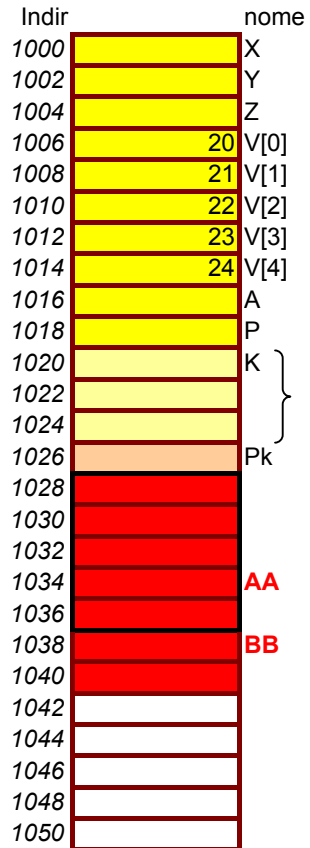
### Variabili automatiche:

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

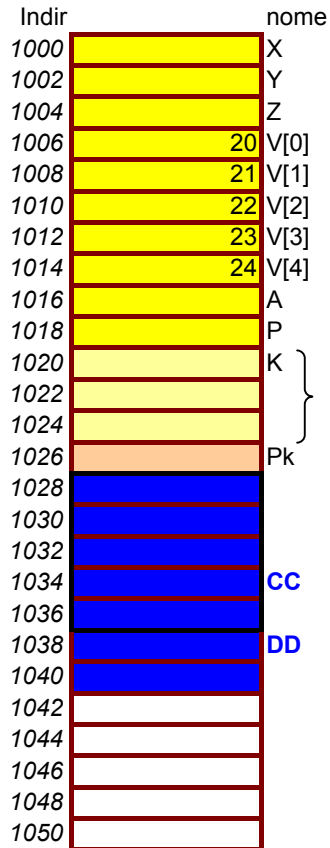
```

int X,Y,Z;
int V[5]={ 20,21,22,23,24 } ;
int A, *P, *Q, **PP;
typedef struct { int C1; float C2 } Coppia;
Coppia K, *Pk;
void F1(int P1)
{ int AA, BB; } ;
void F2(int P1)
{ int CC, DD; } ;

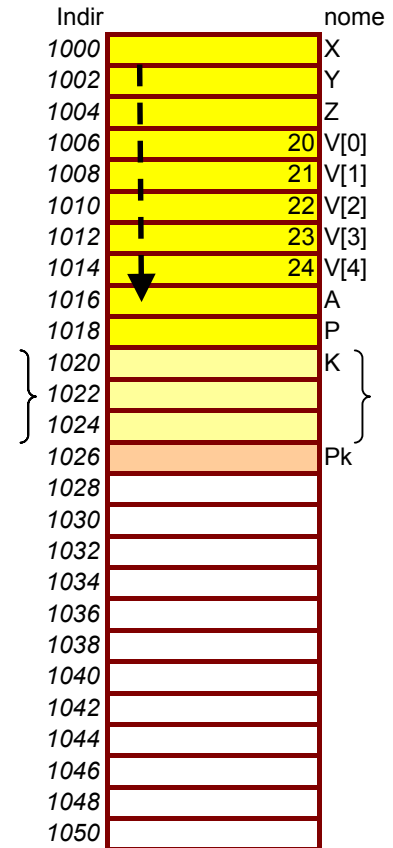
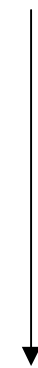
```



**F1(15);**



**F1(15);  
F2(55);**



**F1(15);  
F2(55);  
X=Y;  
eccetera**

## Variabili Dinamiche

### Variabili dinamiche:

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

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

## Variabili Dinamiche

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

- malloc
- free

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

## Variabili Dinamiche

### Allocazione di variabili dinamiche:

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

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

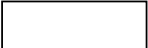
- **tipodato** e` il tipo della variabile puntata
- **punt** e` una variabile di tipo **tipodato \***
- **sizeof()** e` una funzione standard che calcola il numero di bytes che occupa il dato specificato come argomento
- e` necessario convertire esplicitamente il tipo del valore ritornato (casting):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

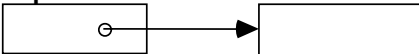
### Significato:

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

### Ad esempio:

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

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

punt                      \*punt  


\*punt=12

punt                      \*punt  



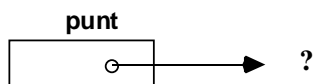

## Variabili dinamiche

### 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 deallocata: \*punt non esiste piu`.

### Esempio:

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

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

### Esempio:

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

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

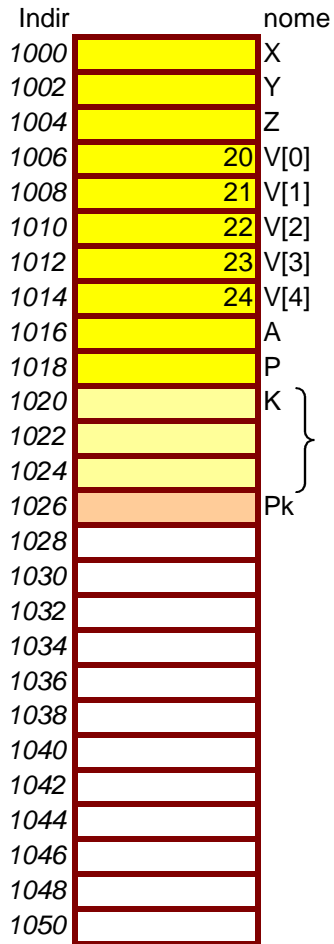
  *p = 55;
  /* assegnamento di un valore alla
  cella *p referenziata da p */

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

```

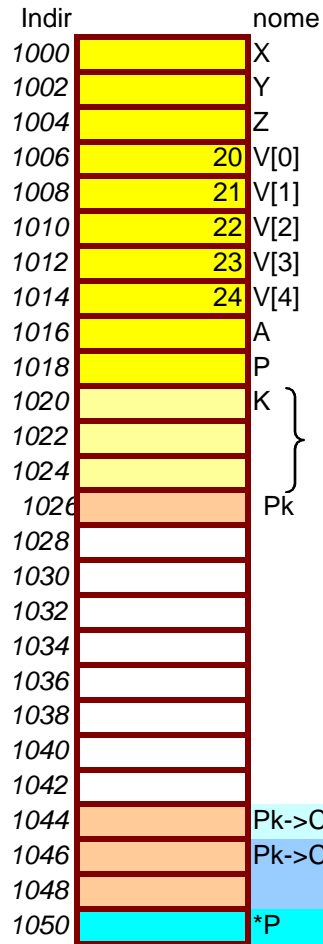
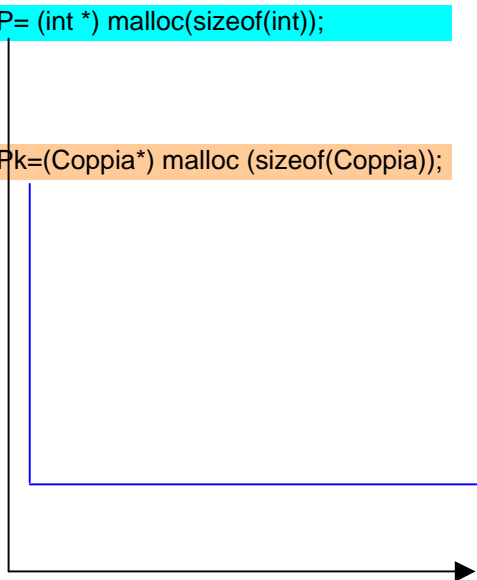
int X,Y,Z;
int V[5]={ 20,21,22,23,24 } ;
int A, *P, *Q, **PP;
typedef struct { int C1; float C2 } Coppia;
Coppia K, *Pk;

```



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

```
Pk = (Coppia *) malloc (sizeof(Coppia));
```



Allocazione stack

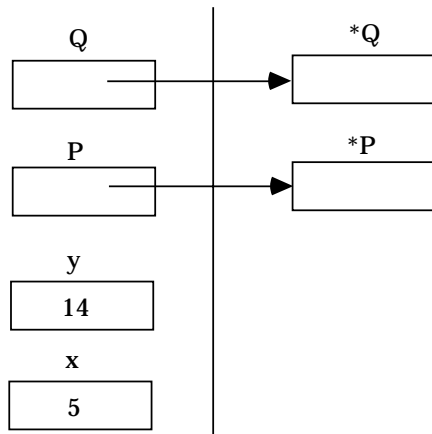


Allocazione Heap

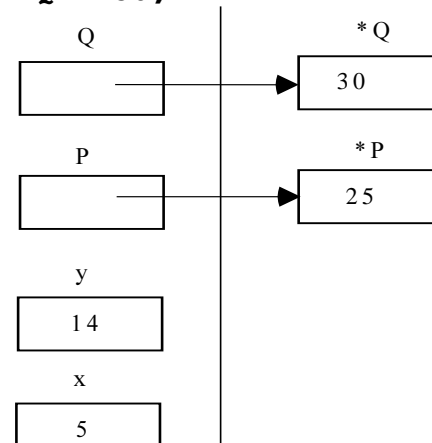
### Esempio:

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

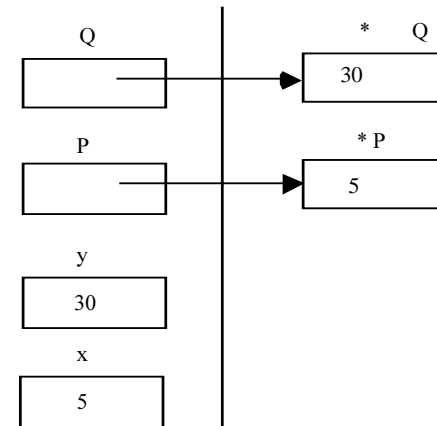
```
  x=5;
  y=14;
  P=(int *)malloc(sizeof(int));
  Q=(int *)malloc(sizeof(int));
```



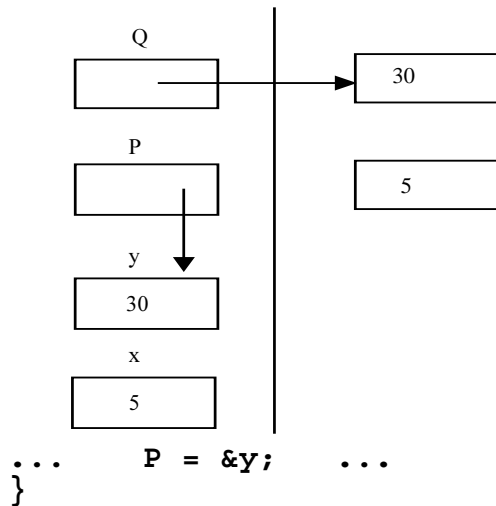
```
*P = 25;
*Q = 30;
```



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



.....



☞ 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!

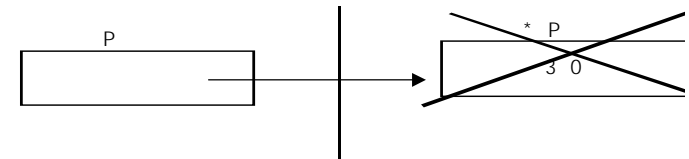
## Problemi legati all'uso dei Puntatori

### Riferimenti pendenti (dangling references):

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

### Ad esempio:

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



## Problemi legati all'uso dei Puntatori

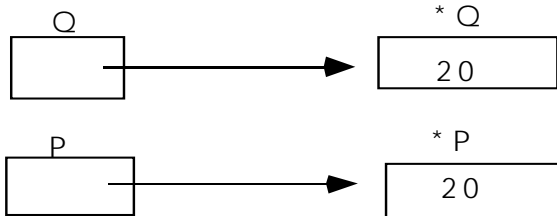
### Aree inutilizzabili:

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

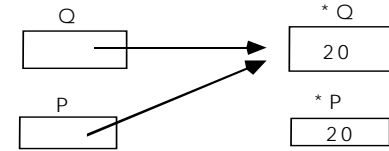
### Ad esempio:

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

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



```
P = Q;
```



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

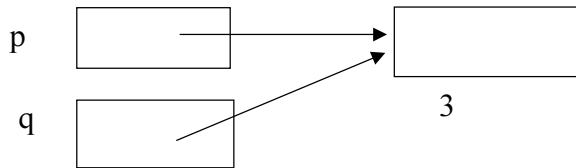
## Problemi legati all'uso dei Puntatori

### Aliasing:

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! */
```

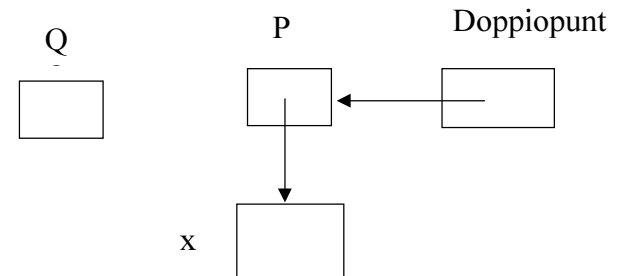
## Puntatori a puntatori (handle)

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

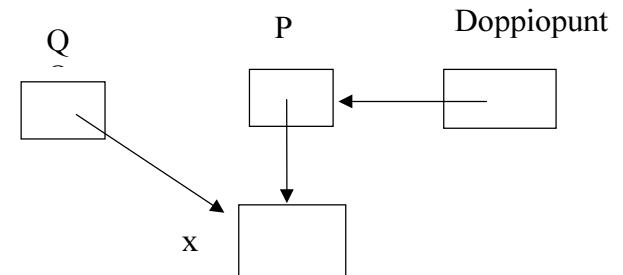
```
[typedef] TipoDato **TipoPunt;
```

### Ad esempio:

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



```
Q = *DoppioPunt;
```



lista

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#define NULL 0

typedef struct element {DATIIMPORTANTI vs;
                        struct element *next;
                        }ELEMENT ;

ELEMENT *create_list_element();
void add_element( ELEMENT *e);

ELEMENT *head =NULL;

void main( )
{
    int j;

    for (j=0; j < 10; ++j)
        add_element ( create_list_element() );
};

/* ----- */

ELEMENT *create_list_element()
{
    ELEMENT *p;

    p = (ELEMENT *) malloc( sizeof ( ELEMENT ) );
    if (p == NULL)
    {
        printf( "create_list_element: malloc failed.\n");
        exit( 1 );
    }
    p->next = NULL;
    return p;
};

/* ----- */

void add_element( ELEMENT *e)
{
    ELEMENT *p;
    /* If the first element (the head) has not been
     * created, create it now.
     */
    if (head == NULL)
    {
        head = e;
        return;
    }
    /* Otherwise, find the last element in the list */
    for (p = head; p->next != NULL; p = p->next)
        ; /* null statement */
    p->next = e;
};
```