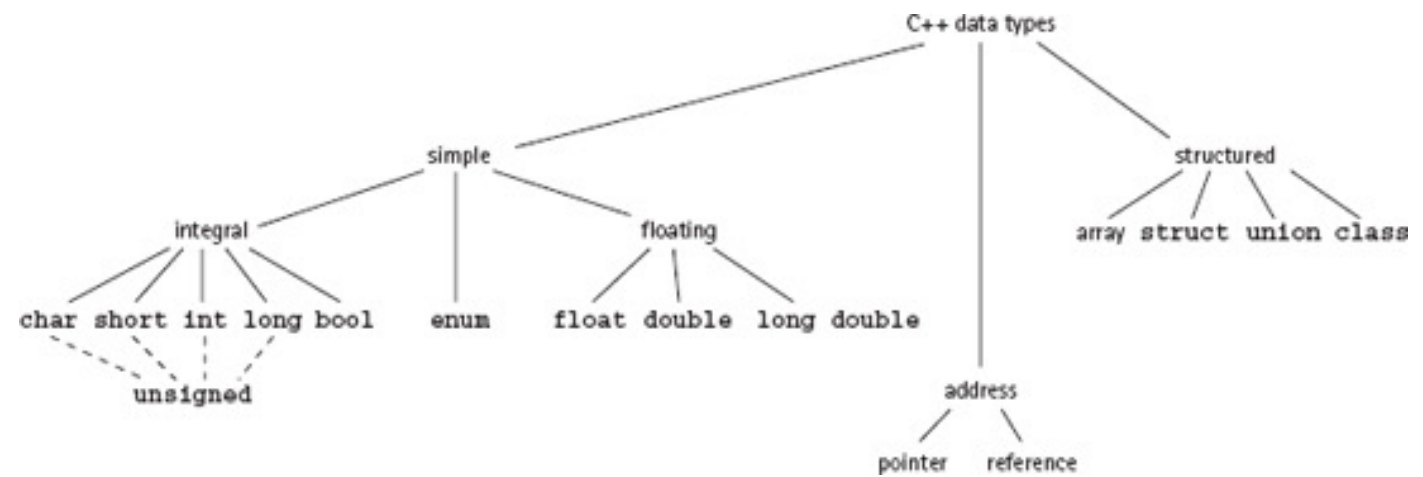


# strutture dinamiche

introduzione alla programmazione

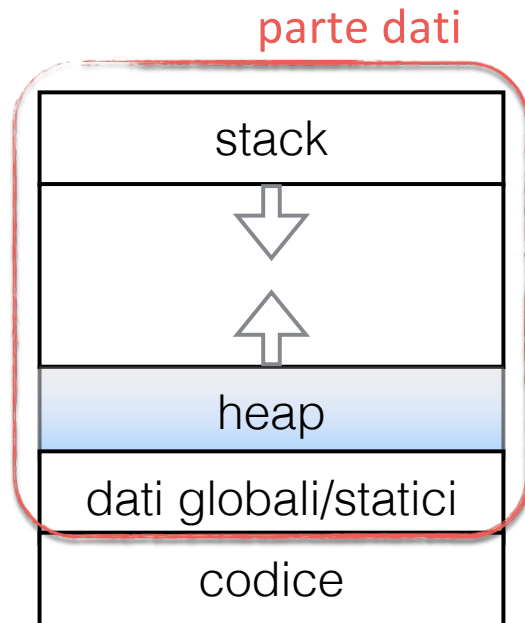
# tipi di dato



# Motivazioni

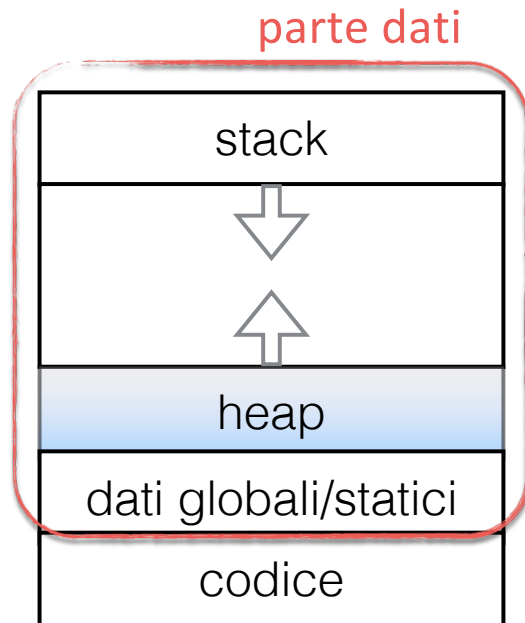
- In molti casi pratici abbiamo bisogno di manipolare strutture dinamiche
  - eg, se non siamo a conoscenza a priori delle dimensioni che una data struttura potrà raggiungere
- La scelta più efficace è di consentire al programma di allocare *dinamicamente* la memoria necessaria

# Strutture dinamiche in memoria



- Le strutture dinamiche si trovano nello *heap*
- *Esso è una sorta di serbatoio, al quale il programma può accedere a run time per richiedere spazio*
- La gestione è molto complicata
- Dal punto di vista del programmatore due operazioni:
  - richiesta di spazio
  - restituzione spazio
- Lo spazio dello heap può avere problemi di frammentazione

# Strutture dinamiche in memoria



- Le strutture dinamiche sono più efficienti nell'uso dello spazio
- Non sono necessariamente efficienti dal punto di vista del tempo di calcolo (anzi, una gestione efficiente è molto complessa da ottenere)

# Puntatori

- Un **puntatore** è una **variabile** atta a contenere come valore **l'indirizzo di un'altra variabile**
- Un puntatore è sempre associato ad un *tipo*
- In C e C++

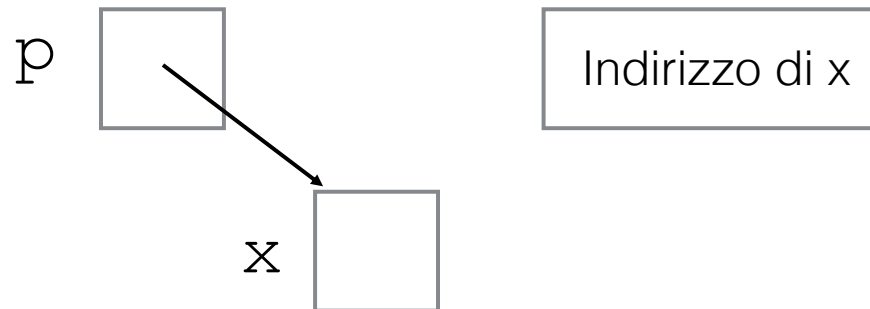
```
int* p;
```

p è una variabile puntatore  
a variabili di tipo intero

# Puntatori

- $T\ x; T^* p;$   
es: `int x; int *p;`
- Se  $x$  è una variabile di tipo  $T$  e  $p$  è un puntatore a una variabile di tipo  $T$  e il valore di  $p$  coincide con l'indirizzo di  $x$ , si dice che

$p$  punta a  $x$

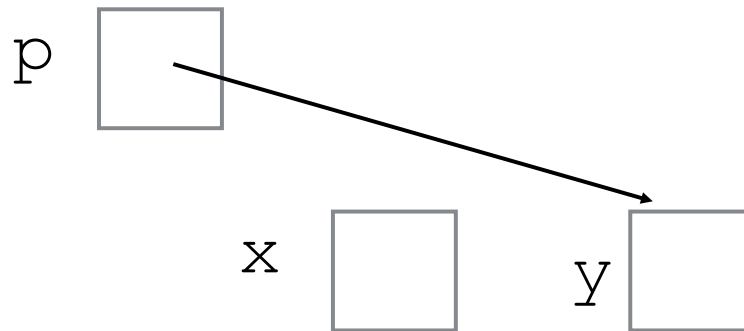


# Puntatori

- Se  $x$  è una variabile di tipo  $T$  e  $p$  è un puntatore a  $T$  e il valore di  $p$  coincide con l'indirizzo di  $x$ , si dice che

$p$  punta a  $x$

se cambio il  
valore di  $p$





# Operatori

- A livello di linguaggio abbiamo due operatori importanti

- ***OPERATORE DI REFERENZIAMENTO***

Data una variabile  $x$  restituisce un *valore indirizzo*, corrispondente all'indirizzo di  $x$

$p = \&x;$

- ***OPERATORE DI DEREFERENZIAMENTO***

Dato un puntatore  $p$  restituisce la *variabile puntata da  $p$*

$*p$

questa variabile è valida solo se il valore di  $p$  punta ad un indirizzo già allocato

# Consistenza della notazione

- $T^* p;$
- Significa che
  - $p$  è un puntatore a  $T$  (potremmo anche dire che  $p$  è di tipo  $T^*$ )
  - $*p$  è di tipo  $T$

Nota però che `int * p, q;`

dichiara una variabile di tipo puntatore a intero ( $p$ ) e una variabile intera ( $q$ )

# Operatori

- *NOTA IMPORTANTE:*

- L'operatore di referenziazione & produce sempre un valore destro (l'indirizzo della variabile che non posso cambiare)

```
es15.cpp:22:8: error: expression is not assignable  
    &x = 10;  
    ~ ~ ^
```

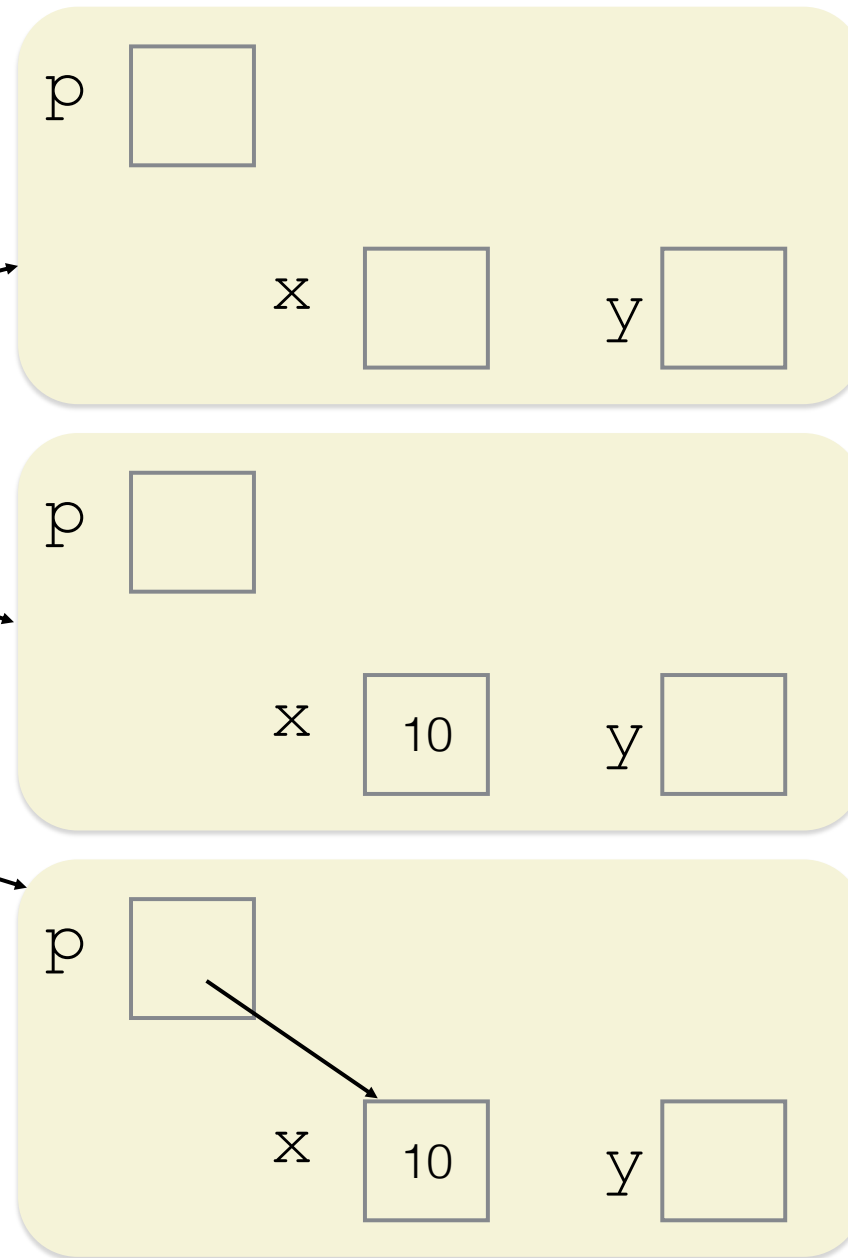
- L'operatore di deferenziazione \* può essere usato sia come valore destro (il contenuto della variabile puntata da ..) che come valore sinistro (la variabile stessa)

# Esempio

```
int x,y;  
int *p;
```

```
x=10;
```

```
p=&x;
```



una stampa:

```
x 10  y 0  p 0x7fff58030ab8  *p 10
```

# Esempio

```
int x,y;  
int *p;
```

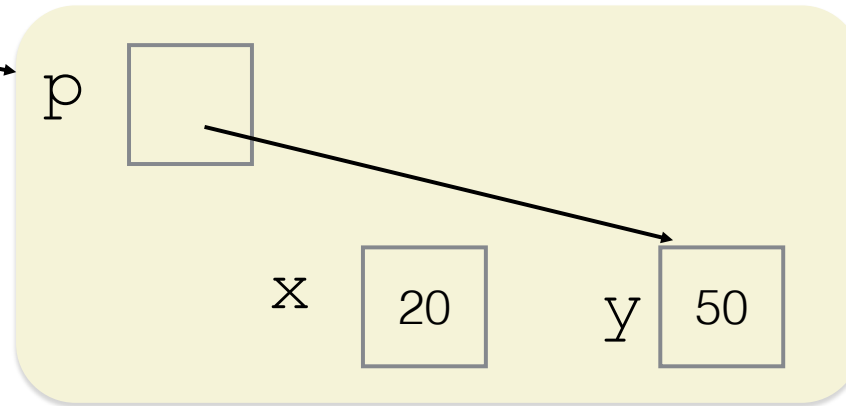
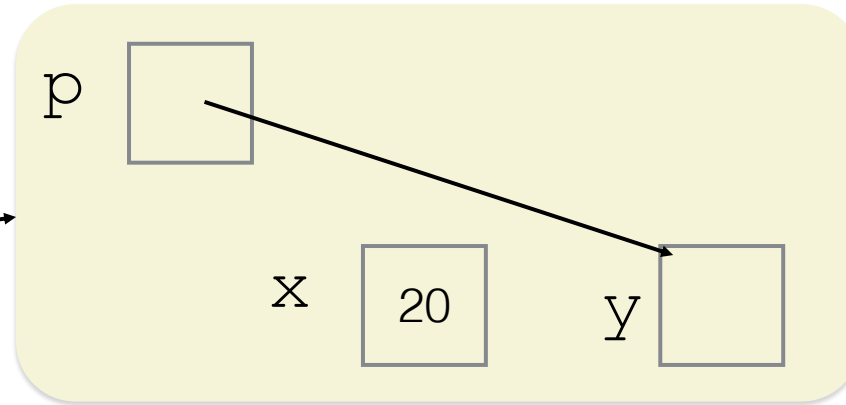
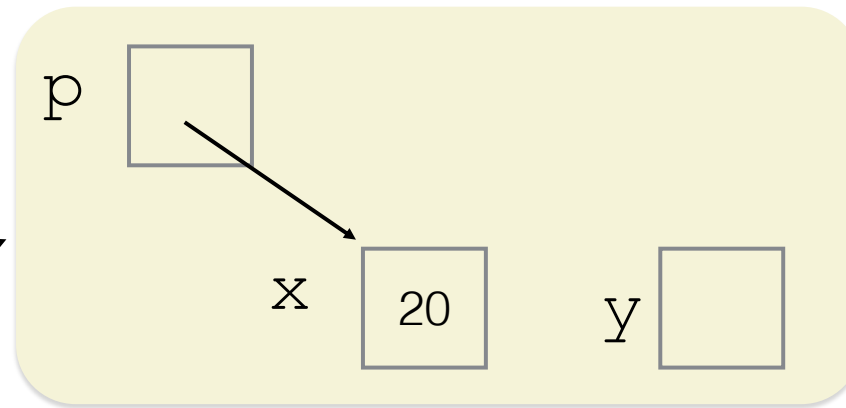
```
x=10;
```

```
p=&x;
```

```
*p = 20;
```

```
p = &y;
```

```
*p = 50;
```



una stampa:

```
x 20 y 50 p 0x7fff58030ab4 *p 50
```

# Esempio

```
int x,y;  
int *p;
```

```
x=10;
```

```
p=&x;
```

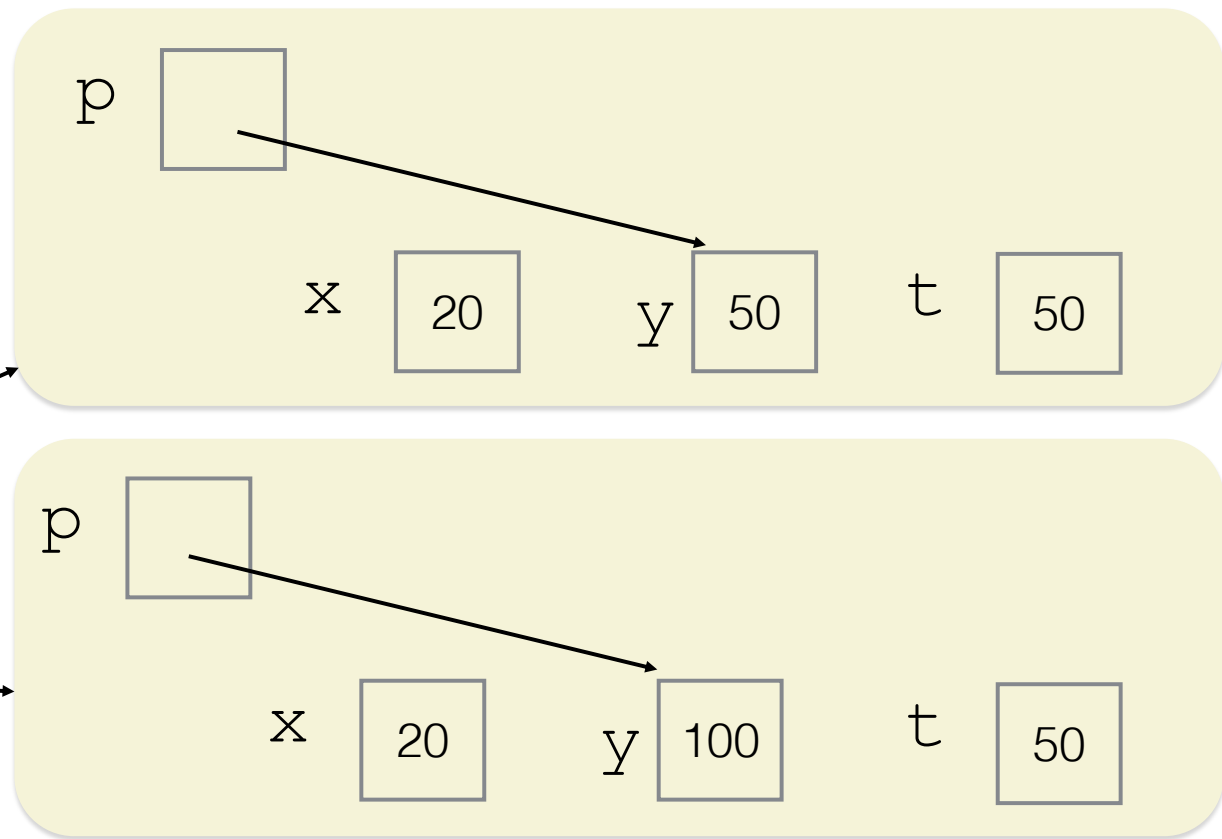
```
*p = 20;
```

```
p = &y;
```

```
*p = 50;
```

```
int t = *p;
```

```
*p = 100;
```



una stampa:

```
x 20  y 100  p 0x7fff58030ab4  *p 100  t 50
```

# Attenzione!

- Se p punta ad un indirizzo di memoria non valido non si possono prevedere gli esiti di questa operazione
  - o il programma abortisce o risultati sono privi di senso...
- Come inizializzare un puntatore?
  - Se non sono ancora pronto ad assegnargli un indirizzo vero e proprio,  
oppure se voglio che **sia chiaro che è un puntatore non valido**,  
posso usare il valore "puntatore nullo"

```
int *p=nullptr;
```

# ATTENZIONE!

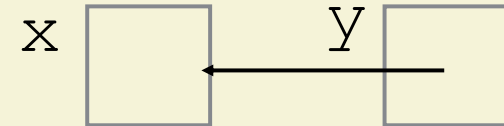
## Reference variables

- In C++ troviamo un uso un po' diverso dell'operatore &
- & usato per dichiarare **variabili reference**: ci permette di associare un nome nuovo (alias) ad una variabile già esistente

```
int x;  
int& y=x;
```

il legame deve essere creato  
contestualmente alla dichiarazione!!

- Si tratta di un legame indissolubile
- Importantissimo: y non è un puntatore!





# Reference variables

- Creare riferimenti può risultare utile in molti casi pratici
- ad esempio nomi diversi in differenti ambiti di visibilità che accedono alla stessa informazione
- questo è quello che accade quando una variabile viene passata per riferimento

```
void funzione (int& a) {...}
```

```
...
```

```
funzione(b); // alla chiamata si realizza l'alias tra il parametro  
attuale e il parametro formale
```

# aritmetica dei puntatori

- I puntatori possono essere usati come operandi in espressioni aritmetiche o logiche

- Esempi:

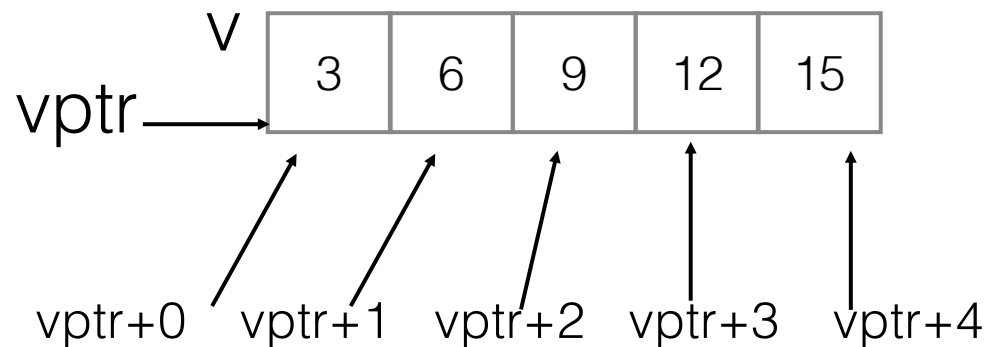
```
int v[5]={3,6,9,12,15};  
int* vptr=v; //equivalente a  
//indirizzo base di v (sia esso 3000)
```

- `vptr+=2;` //l'incremento non è di 2 celle, dipende dalla  
//dimensione dell'oggetto a cui punta  
//in questo caso  $3000+2*4$
- `*(vptr+2)=4;` //idem.

# Aritmetica dei puntatori

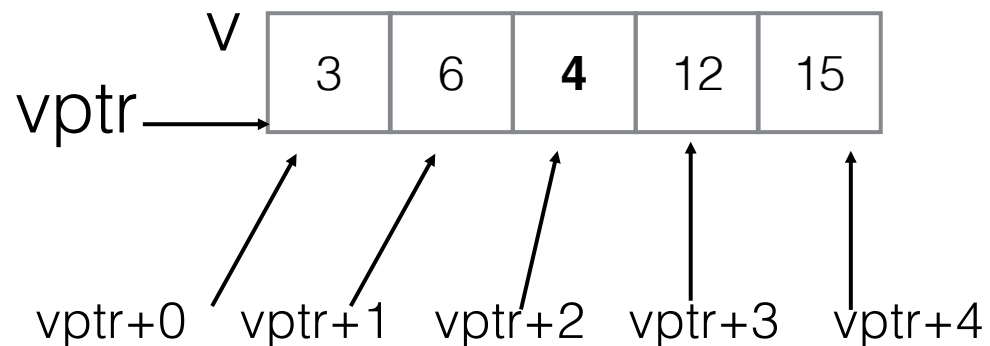
- `int v[5]={3,6,9,12,15};`  
`int *vptr=v;`  
`*(vptr+2)=4;`

- `vptr=vptr+1;`



# Aritmetica dei puntatori

- `int v[5]={3,6,9,12,15};`  
`int *vptr=v;`  
`* (vptr+2)=4;`      **NB qui non ho cambiato il valore del puntatore**
- `vptr=vptr+1;`

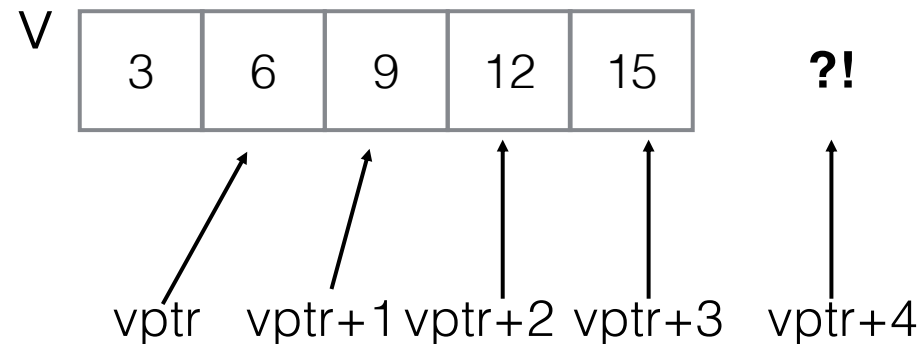


# Aritmetica dei puntatori

- `int v[5]={3,6,9,12,15};`  
`int *vptr=v;`  
`*(vptr+2)=4;`

**NB qui cambio valore del puntatore**

- `vptr=vptr+1;`



# Aritmetica dei puntatori - esempio d'uso

- Un modo più efficiente di realizzare una visita di un array con stampa

```
int v[5]={3,6,9,12,15};  
int *vpPtr=v;  
for (int i=0;i<5;++i) cout << *(vpPtr++) << " ";  
cout << endl;
```

# NOTARE LE DIFFERENZE

Ad ogni iterazione aggiorniamo il puntatore

Incremento postfisso (come ultima cosa...)

```
int v[5]={3,6,9,12,15};
int *vptr=v;
for (int i=0;i<5;++i) {
    cout << " *vptr: " << *vptr++ << " ";
    cout << endl;
}
```

```
home$ ./visit
*vptr: 3
*vptr: 6
*vptr: 9
*vptr: 12
*vptr: 15
```

Incremento prefisso (come prima cosa...)

```
int v[5]={3,6,9,12,15};
int *vptr=v;
for (int i=0;i<5;++i) {
    cout << *++vptr << " ";
    cout << endl;
}
return(0);
```

```
home$ ./visit
*vptr: 6
*vptr: 9
*vptr: 12
*vptr: 15
*vptr: 32766
```

# NOTARE LE DIFFERENZE

```
int v[5]={3,6,9,12,15};
int *vptr=v;
for (int i=0;i<5;++i) {
    cout << "vptr: " << static_cast<void*>(vptr);
    cout << " *vptr: " << *vptr++ << " ";
    cout << endl;
```

```
home$ ./visit
vptr: 0x7ffeeafa5ae0 *vptr: 3
vptr: 0x7ffeeafa5ae4 *vptr: 6
vptr: 0x7ffeeafa5ae8 *vptr: 9
vptr: 0x7ffeeafa5aec *vptr: 12
vptr: 0x7ffeeafa5af0 *vptr: 15
```

Qui aggiorno il puntatore incrementandolo di 1 pos ad ogni iterazione

```
int v[5]={3,6,9,12,15};
int *vptr=v;
for (int i=0;i<5;++i) {
    cout << "vptr: " << static_cast<void*>(vptr);
    cout << " *vptr: " << *(vptr+i) << " ";
    cout << endl;
}
```

```
home$ ./visit
vptr: 0x7ffee19c9ae0 *vptr: 3
vptr: 0x7ffee19c9ae0 *vptr: 6
vptr: 0x7ffee19c9ae0 *vptr: 9
vptr: 0x7ffee19c9ae0 *vptr: 12
vptr: 0x7ffee19c9ae0 *vptr: 15
```

Qui accedo ad un elemento che dista "i" posizioni dal puntatore  
(che rimane fisso)



# Allocazione e deallocazione dinamica

- L'uso dei puntatori fatto finora (punto a variabili allocate in modo statico) è poco interessante e in C++ può essere quasi sempre evitato
- I puntatori sono invece essenziali nella gestione dinamica della memoria

# Allocazione dinamica

- Richiedo al gestore dello heap di allocare memoria necessaria a mantenere una variabile di un certo tipo
- Il gestore alloca la memoria necessaria sullo heap e restituisce l'indirizzo
- In C++ questa operazione si realizza con il comando `new`

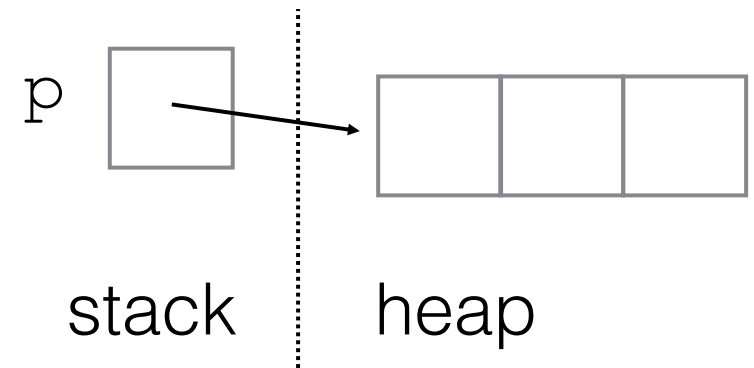
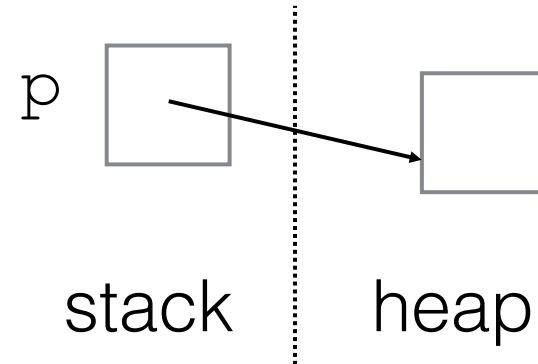
```
T *p;  
p = new T;
```

- Questa istruzione permette di allocare sullo heap una variabile di tipo `T` e di salvare in `p` l'indirizzo di tale variabile

- `T *p;`

- `p = new T;`

- `int size = 3;`  
`p = new T[size];`



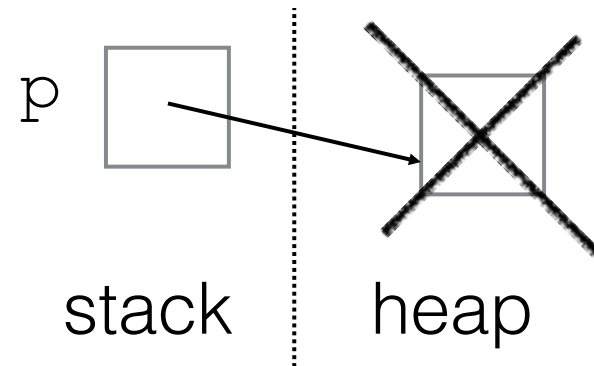
# Deallocazione dinamica

- Restituisce allo heap una porzione di memoria che non serve più
- Tale operazione invalida l'accesso alla porzione di memoria e la lascia a disposizione del gestore dello heap per usi futuri
- In C++ questa operazione si realizza con il comando `delete`

`delete p;`

- Se abbiamo una struttura con più elementi

`delete [] p;`



# Dangling pointers

- Per evitare che un puntatore non punti (più) a nulla, dopo un delete conviene

```
delete [] p1;  
p1=nullptr;
```

# Memory leak

- L'operatore `new` ci aiuta a creare le variabili dinamiche solo quando sono necessarie
- Quando abbiamo finito di usarle dovremmo liberare la memoria occupata nello heap con una `delete`
- Se non lo facciamo (ossia se teniamo variabili dinamiche anche se non servono più) produciamo un memory leak
- ***memory leak*** - perdita di spazio di memoria dovuta alla mancata deallocazione

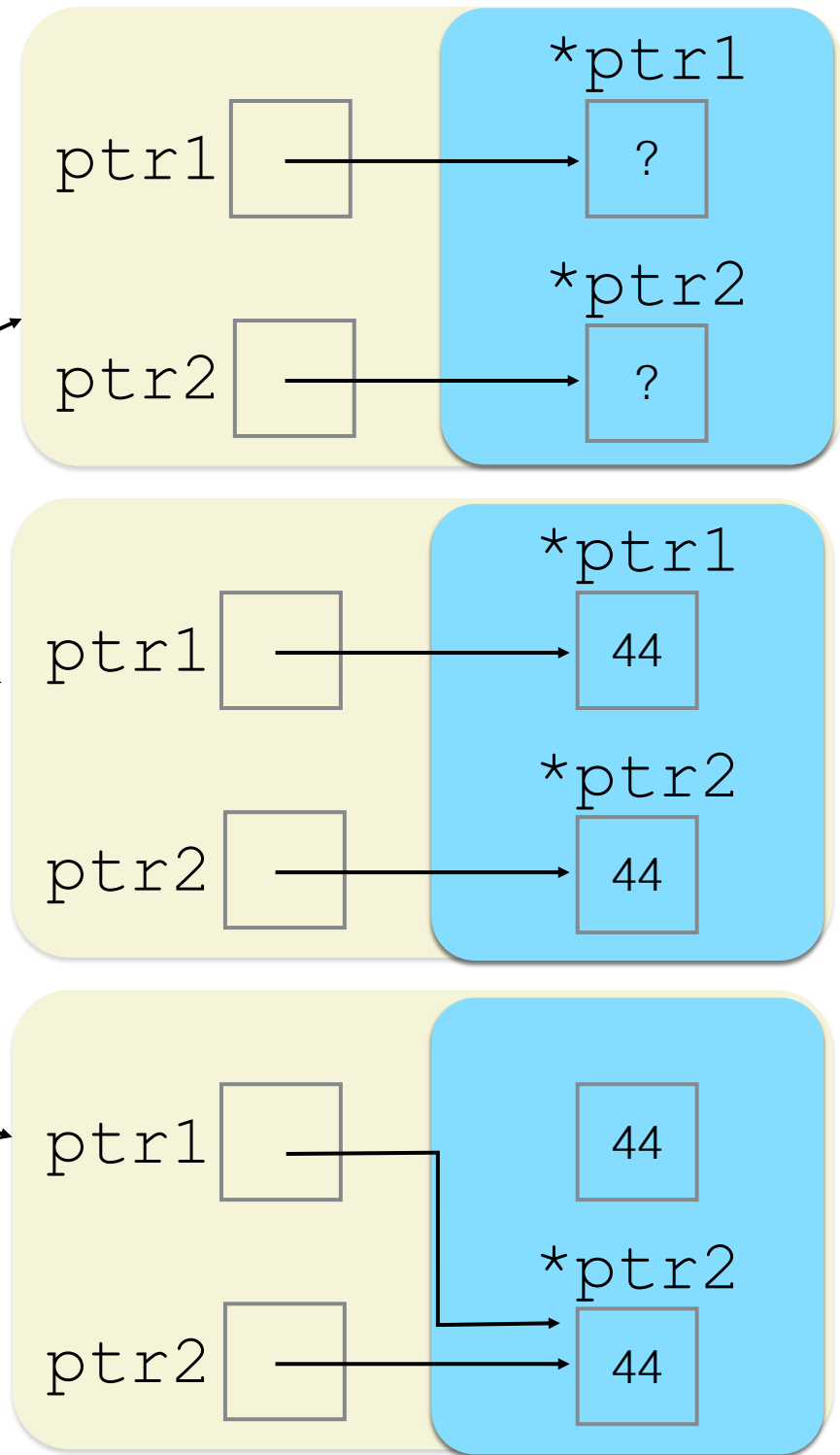
# Esempio

```
int *ptr1 = new int;  
int *ptr2 = new int;
```

```
*ptr2 = 44;  
*ptr1 = *ptr2;
```

```
ptr1 = ptr2;
```

```
delete ptr2;  
ptr1=nullptr;  
ptr2=nullptr;
```



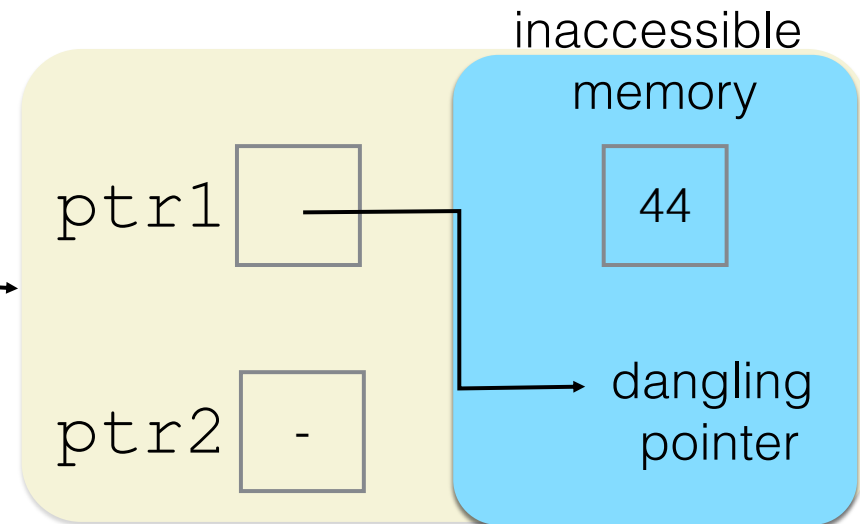
# Esempio

```
int *ptr1 = new int;  
int *ptr2 = new int;
```

```
*ptr2 = 44;  
*ptr1 = *ptr2;
```

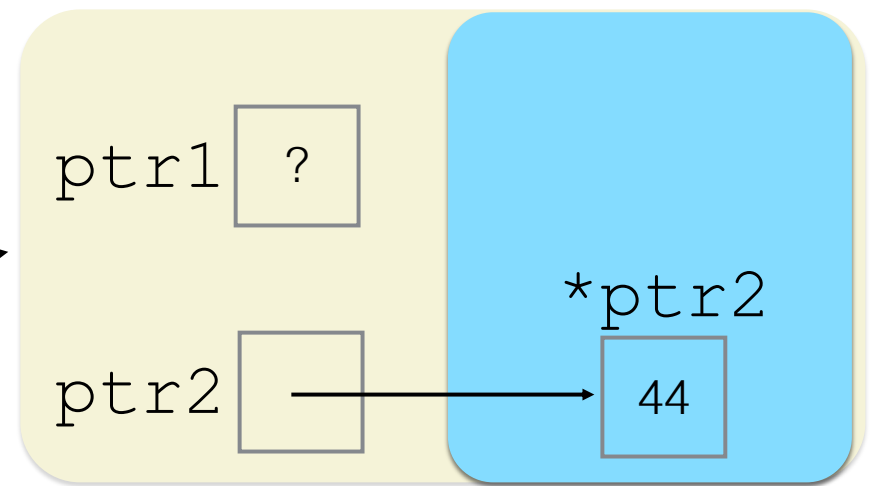
```
ptr1 = ptr2;
```

```
delete ptr2;  
ptr2 = nullptr;
```





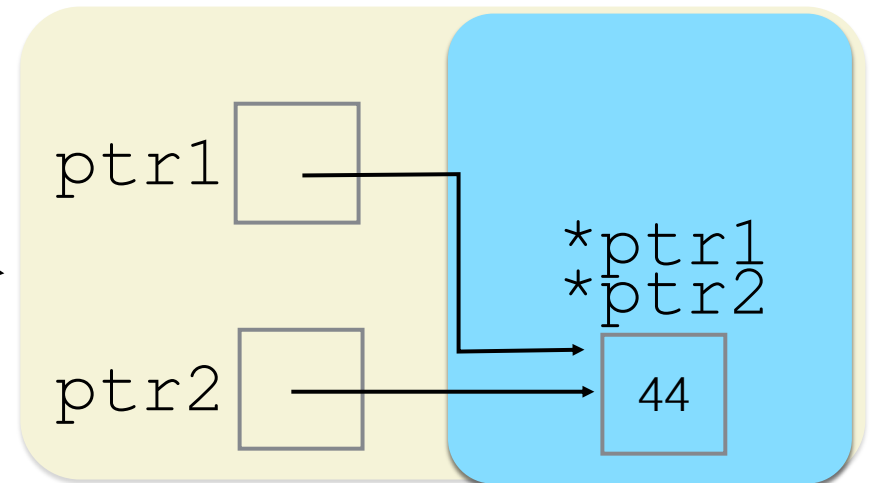
```
int *ptr1 = new int;  
int *ptr2 = new int;  
*ptr2 = 44;
```



```
*ptr1 = *ptr2;
```

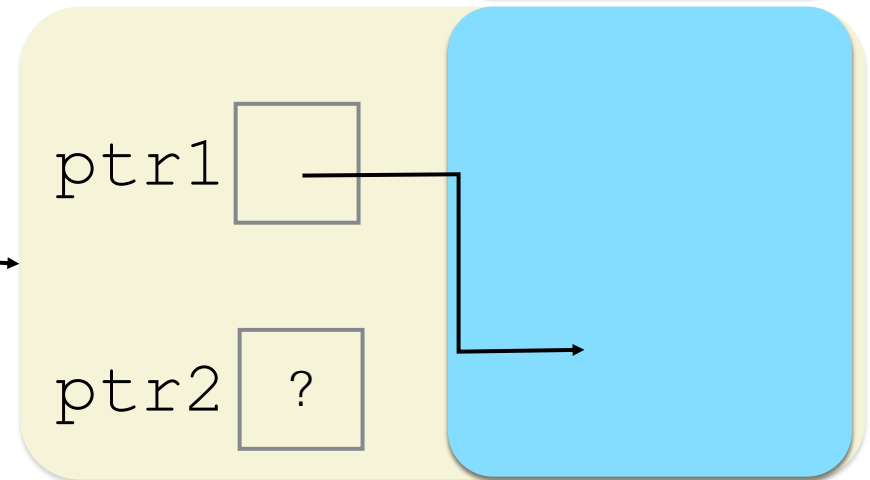
```
delete ptr1;
```

```
ptr1 = ptr2;
```



```
delete ptr2;
```

```
ptr1=nullptr;
```



```
int *ptr1 = new int;  
int *ptr2 = new int;  
*ptr2 = 44;
```

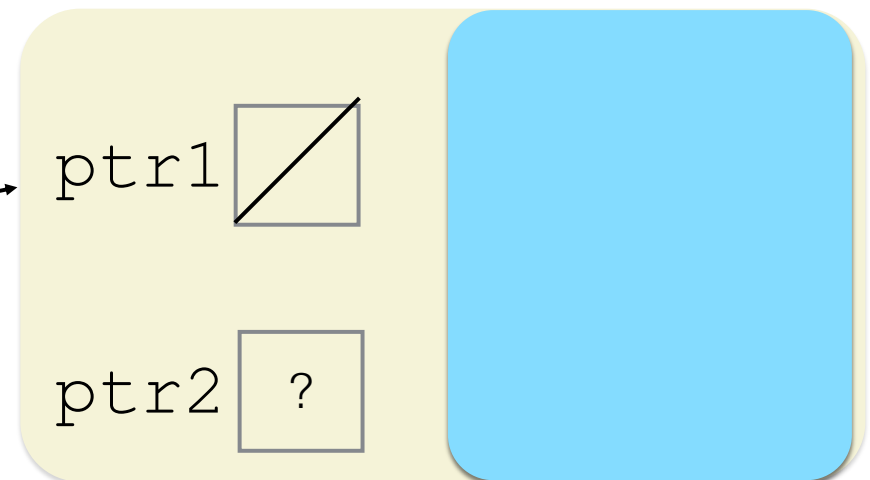
```
*ptr1 = *ptr2;
```

```
delete ptr1;
```

```
ptr1 = ptr2;
```

```
delete ptr2;
```

```
ptr1=nullptr;
```



**IMPORTANTE!**

## Array dinamici

- `int a[10];`  
`int *p;`  
`p = new int[10];`

<b>a</b>	<b>p</b>
risiede nello stack	risiede sullo heap
dimensione costante	dimensione variabile
binding statico tra nome e array (a è un <i>puntatore costante</i> )	binding dinamico tra nome e array

# array dinamici e accesso agli elementi

- l'accesso agli elementi può essere svolto come per gli array statici (compresa aritmetica dei puntatori)

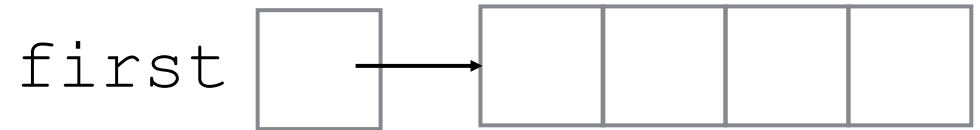
•	<pre>int *p; p = new int[5]; for (int i=0;i&lt;5;++i)     p[i]=i*i;</pre>	<pre>int *p; p = new int[5]; for (int i=0;i&lt;5;++i)     *(p+i)=i*i;</pre>
---	---	---

- come nel caso statico accedere fuori dalla memoria allocata per l'array (out of range) può produrre uno dei due seguenti scenari
  - la cella si trova fuori dall'area allocata dallo heap, il programma abortisce
  - la cella si trova nell'area allocata, il programma prosegue ma produce risultati imprevedibili

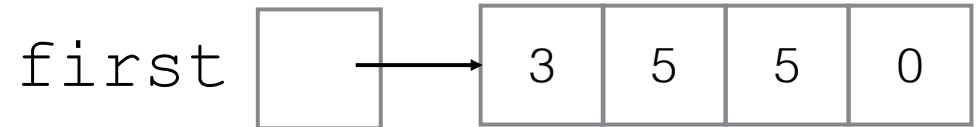
# Copia superficiale e copia profonda

- Consideriamo le seguenti istruzioni

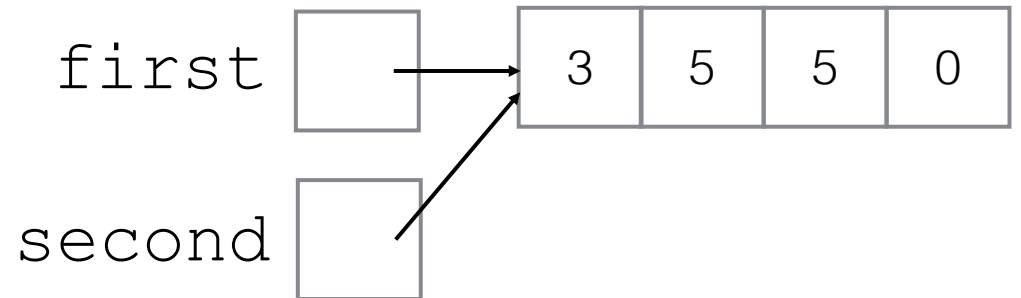
```
int *first;  
int *second;  
first = new int[10];
```



- ... inizializzo first...

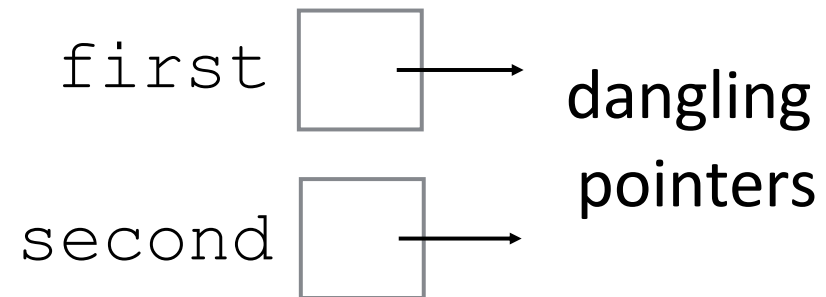


- copia superficiale  
second=first;



# Copia superficiale e copia profonda

- Cosa succede se...  
`delete [] second;`  
?



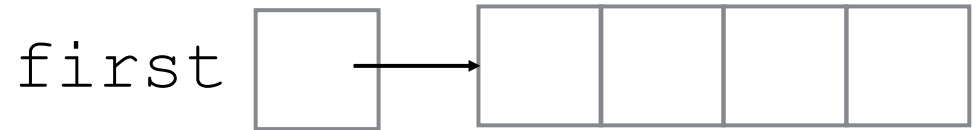
# Copia superficiale e copia profonda

- Consideriamo le seguenti istruzioni

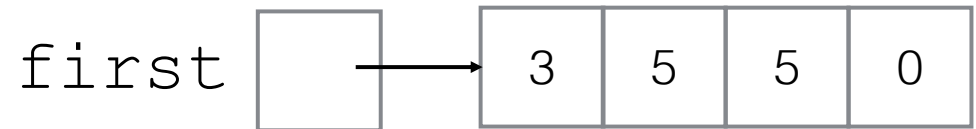
```
int *first;
```

```
int *second;
```

```
first = new int[10];
```



- ... inizializzo first...

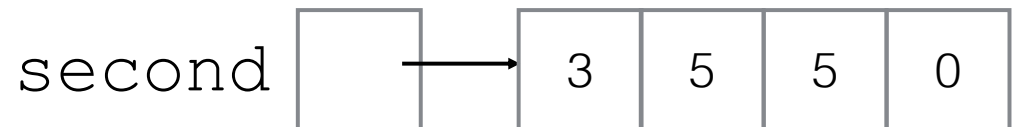
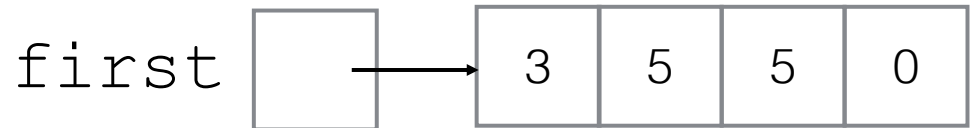


- copia profonda

```
second = new int[10];
```

```
for (int i=0; i<10; i++)
```

```
    second[i]=first[i];
```



# Allocazione dinamica - esempio

- immaginiamo di dover mantenere un numero indeterminato di valori omogenei di un dato tipo T
- si alloca un array dinamico p di piccola dimensione
- quando non ho più spazio devo allocarne di più:
  1. allocare un array temporaneo di dimensione maggiore
  2. copiare il vecchio array nel nuovo (copia profonda)
  3. cancellare l'array puntato da p e far puntare p al nuovo array



# Allocazione dinamica - esempio

```
int size = 5;           // non è più un const
int* a = new int[size]; // allocazione sullo heap
int n = 0;

//--- Scrivo nell'array
while (cin >> a[n]) {
    n++;
    if (n >= size) {
        // quando ho finito lo spazio
        size = size * 2; // raddoppio il valore della variabile size
        int* temp = new int[size]; // creo un array temp grande il doppio
        for (int i=0; i<n; ++i) {
            temp[i] = a[i]; // copia profonda
        }
        delete [] a; // libero lo spazio vecchio (piccolo)
        a = temp;    // aggiorno il puntatore
    }
}

//--- stampo quello che ho inserito (potrebbe essere meno di size!)
for (int i=0; i<n; ++i)
    cout << a[i] << endl;
```

NB: la capacità dell'array dinamico e la sua effettiva dimensione non sono necessariamente uguali!! (perché?)

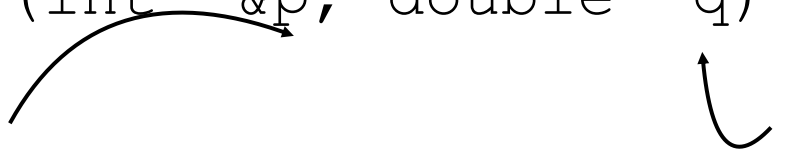
# Funzioni e puntatori

- una variabile puntatore può essere passata come parametro di funzione, sia per valore che per riferimento

```
void example(int* &p, double *q)
{
  ..
}
```

per riferimento

per valore



- un tipo di dato restituito da una funzione può essere un puntatore

```
int* example2 (...)
```

```
{
}
```

# Puntatori, array e funzioni

- Avevamo detto “Un array non conosce la propria dimensione”
- In realtà questo è vero (ed e' un problema) quando passiamo l'array come parametro ad una funzione
- Questo perche' in realtà alla funzione passiamo un puntatore al primo elemento dell'array  
**serve per indicare da dove l'array inizia**

```
• void initialize(int *list, int size)
  {
    for (int i=0; i<size; i++)
      list[i]=0;
  }
```

# Funzioni, puntatori, array...

```
const int CHUNK_SIZE = 10;

void print_array(int *a, int size) {
    for (int i=0; i<size; ++i)
        std::cout << *(a+i) << " ";
    std::cout << std::endl;
}

void init_array(int *a, int size, int K) {
    for (int i=0; i<size; ++i)
        *(a+i)=K;
}

int main ()
{
    int *p1;
    p1 = new int[CHUNK_SIZE]; //array dinamico (heap)

    int v[CHUNK_SIZE]; //array statico (stack)

    init_array(v, CHUNK_SIZE, 1);
    print_array(v, CHUNK_SIZE);

    init_array(p1, CHUNK_SIZE, 3);
    print_array(p1, CHUNK_SIZE);
}
```

Funzioni che genericamente  
prendono un puntatore...

Proviamo a incapsulare le informazioni

# Funzioni, puntatori, array dinamici

```
struct dynamic_array {  
    int *store;  
    unsigned int size;  
}
```

```
void read_d_array(dynamic_array& d) {  
    // definire una variabile intera s a un valore negativo  
    int s=-1;  
    while (s<0)  
    {  
        std::cout << "inserisci la dimensione dell'array " << std::endl;  
        std::cin>>s;  
    }  
    d.size=s;  
    d.store = new int [s];  
    for (unsigned int i=0;i<s;++i) {  
        std::cout << "inserisci un valore intero " << std::endl;  
        std::cin>>d.store[i];  
    }  
}
```

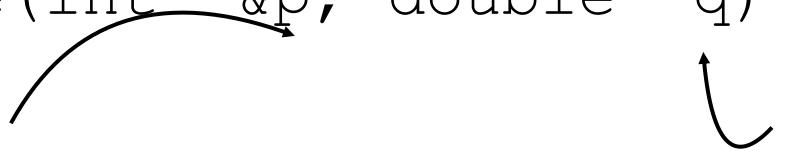
```
void print_d_array(const dynamic_array& d) {  
  
    int *p;  
    p=d.store;  
    for (unsigned int i=0;i<d.size;++i) {  
        std::cout << *p++ << "//"  
    }  
    std::cout << std::endl;  
}
```

# Approfondimento

# Funzioni, puntatori allocazione dinamica della memoria

- una variabile puntatore può essere passata come parametro di funzione, sia per valore che per riferimento

```
void example(int* &p, double *q)
{
  ..
}
```

 **per riferimento** **per valore**

- un tipo di dato restituito da una funzione può essere un puntatore

```
int* example2 (...)
```

```
{
}
```

# Tricky! passaggio per valore

```
#include <iostream>

using namespace std;

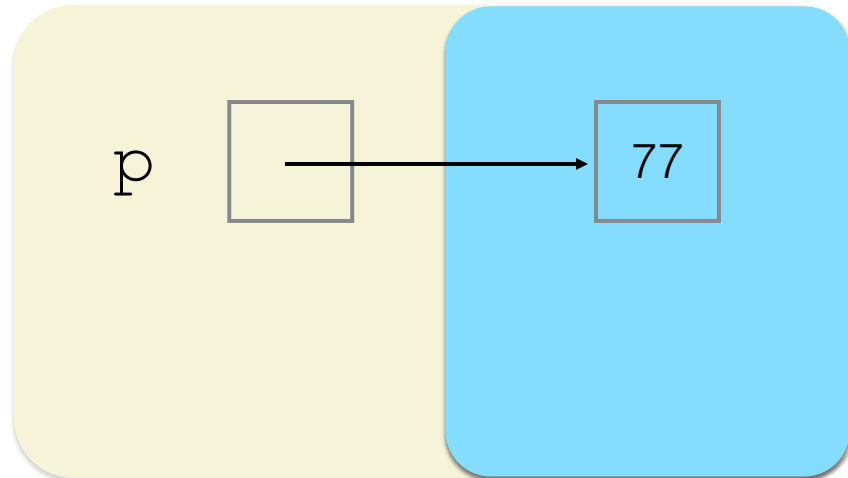
void tricky(int* );

int main( )
{
    int* p;
    p = new int;
    *p = 77;
    cout << "Before call to function *p == " << *p << endl;
    tricky(p); //chiamata
    cout << "After call to function *p == " << *p << endl;
    return 0;
}

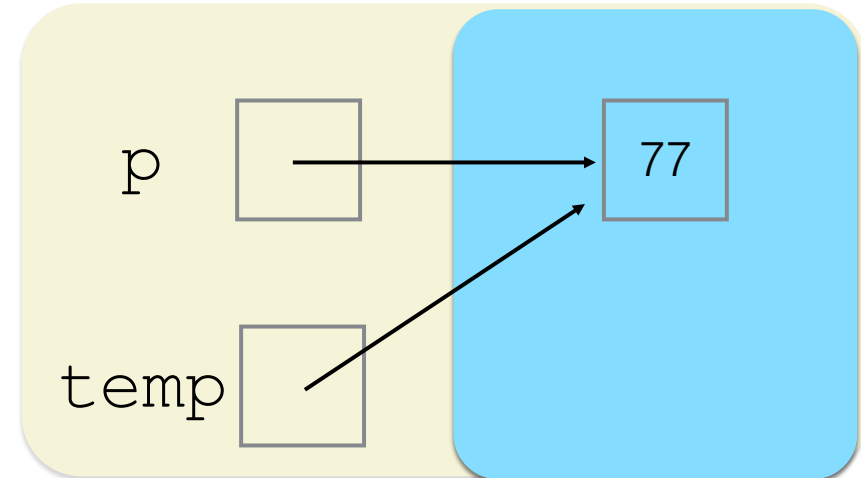
void tricky(int* temp) //passaggio per valore di puntatore
{
    *temp = 99;
    cout << "Inside function call *temp == " << *temp <<
endl;
}
```



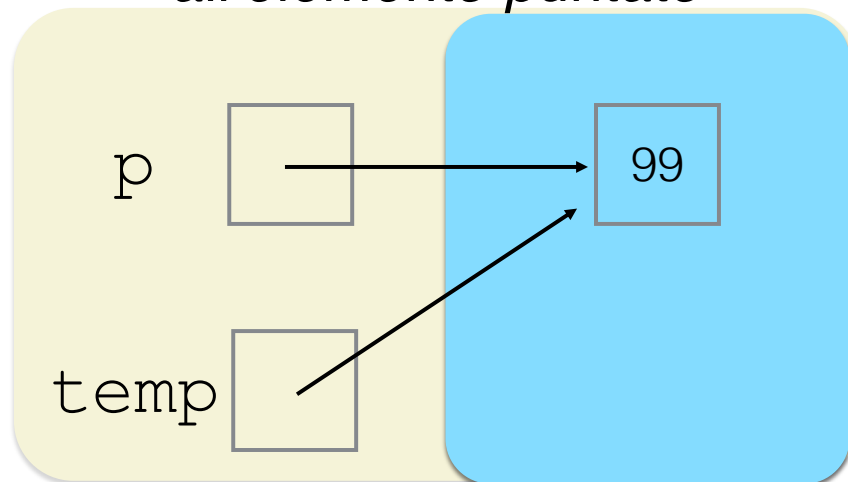
*Prima della chiamata di tricky*



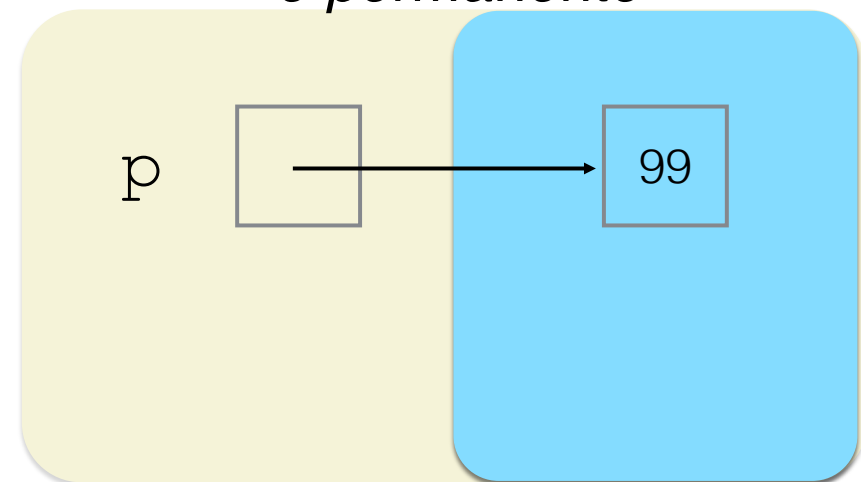
*Il passaggio per valore crea copia del puntatore ma non della variabile puntata (copia superficiale)*



*La modifica viene applicata all'elemento puntato*



*Di ritorno al main, la modifica è permanente*



# Tricky! passaggio per valore

```
#include <iostream>
```

```
using namespace std;
```

```
void tricky(int* );
```

```
int main( )
```

```
{
```

```
    int* p;
```

```
    p = new int;
```

```
    *p = 77;
```

```
    cout << "Before call to function *p == " << *p << endl;
```

```
    tricky(p); //chiamata
```

```
    cout << "After call to function *p == " << *p << endl;
```

```
    return 0;
```

```
}
```

```
void tricky(int* temp) //passaggio per valore di puntatore
```

```
{
```

```
    *temp = 99;
```

```
    cout << "Inside function call *temp == " << *temp <<
```

```
endl;
```

```
}
```

Before call to function \*p == 77  
Inside function call \*temp == 99  
After call to function \*p == 99

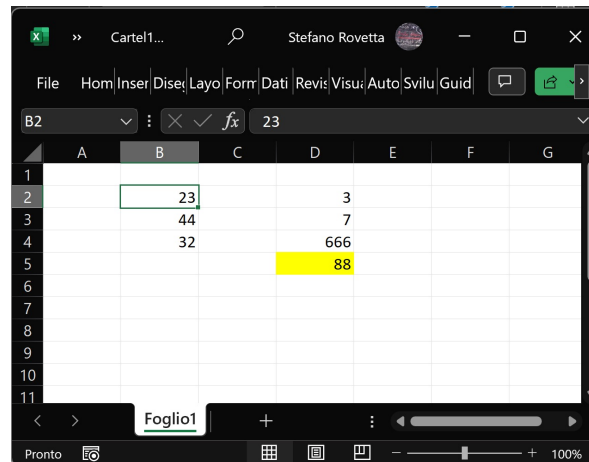
# ..che fare?

- Come risolvere questa “incongruenza”?
- Strumenti di creazione di tipo piu’ avanzati ci permetteranno di definire apposite funzioni di creazione (costruttori di copia), che creano una copia nuova ad una dichiarazione di variabile
- Un esempio di tipo “ben fatto” in questo senso sono i vector

# Strutture bidimensionali

	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4			X							
5						X	X			
6		X						X		X
7				X						X
8	X	X						X		
9										
10										

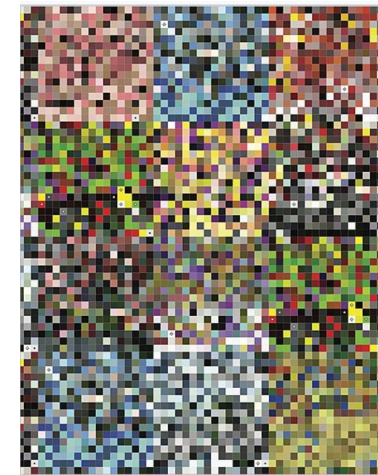
Battaglia navale



The screenshot shows a Microsoft Excel window with the title 'Cartel1...'. The active cell is B2, containing the value 23. The formula bar shows '=23'. The spreadsheet grid shows the following values:

	A	B	C	D	E	F	G
1							
2		23			3		
3		44			7		
4		32			666		
5					88		
6							
7							
8							
9							
10							
11							

Foglio di calcolo



Pixel (immagine)

Caratterizzate da due indici:  
un «numero di righe» e un «numero di colonne»

# Array bidimensionali (statici)

- `float a[nr][nc];` // nr numero righe, nc numero colonne
- Sono *array di array* – Array i cui elementi sono a loro volta array

`float a[2][3];`

a					
a[0]			a[1]		
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

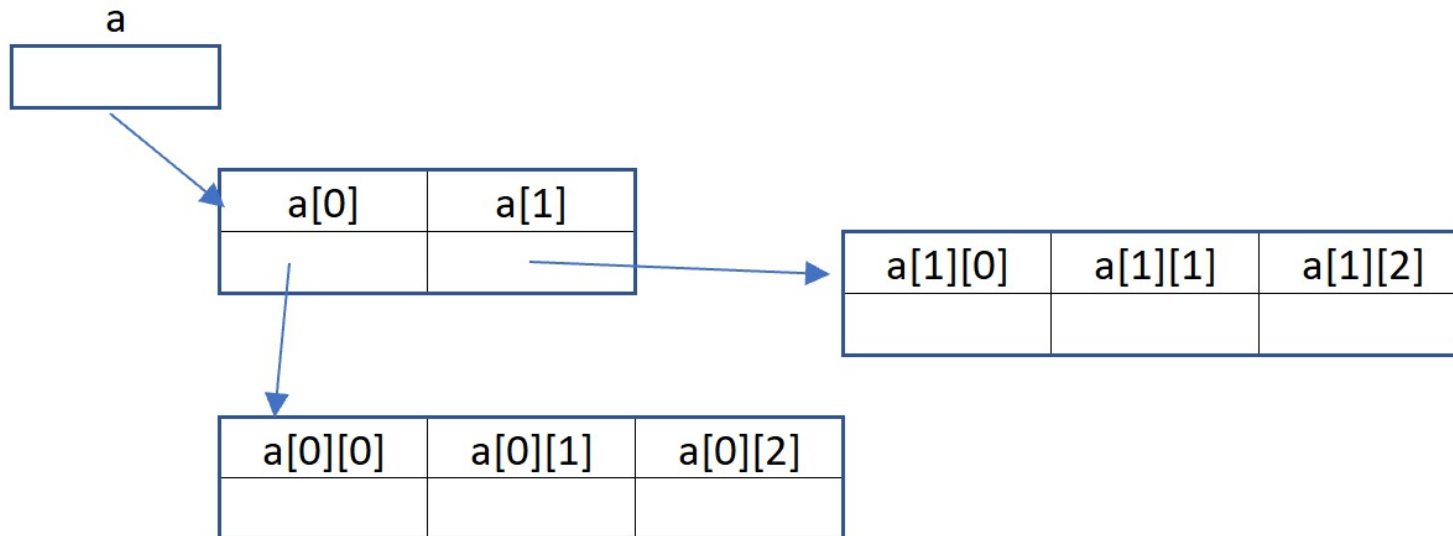
# Array bidimensionali (statici): caratteristiche

- Sempre allocati su stack
- Non si possono passare come argomento a funzione in modo semplice

# Alternativa (dinamica): puntatore a puntatore

- `float ** a;`  
`a = new float * [nr]; // un array dinamico di puntatori a float`  
`for (int i=0; i<nr; i++) a[i] = new float [nc];`  
`// un array dinamico di float`

`float ** a;`



# Puntatore a puntatore: caratteristiche

- Sempre allocati su heap
- E' possibile avere righe di dimensioni diverse