Puntatori

Fondamenti di Informatica, AA 2022/23

Luca Cassano

luca.cassano@polimi.it

Puntatori

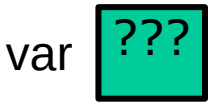
- Un puntatore è un **tipo di dato** che viene utilizzato in C per dichiarare una variabile che **deve contenere un indirizzo** di una cella di memoria.
- In gergo, si dice che la variabile puntatore “punta” alla cella di memoria, il cui indirizzo è contenuto nella variabile puntatore.
- Quando viene dichiarata una variabile puntatore, è necessario anche specificare il tipo di dato contenuto nelle celle di memoria che verranno “puntate” dalla variabile.
- La sintassi per dichiarare una variabile puntatore è:

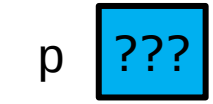
```
tipo *nome;
```

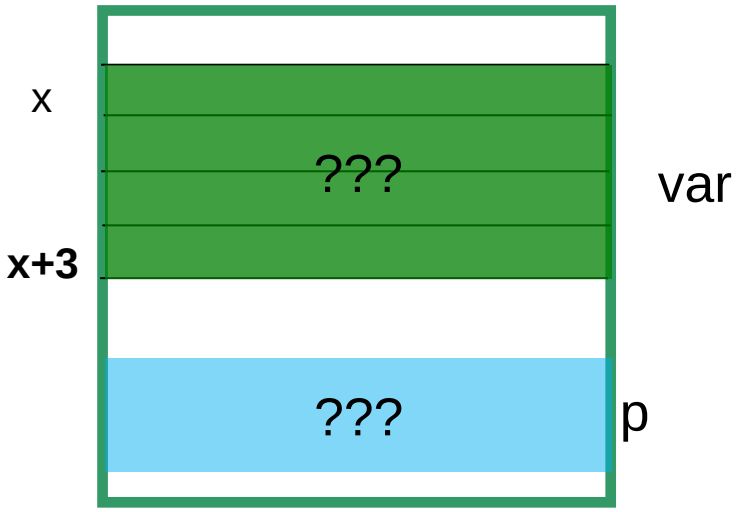
Puntatori

Come si assegna un indirizzo di memoria di una variabile ad un puntatore?

```
float var;
float *p;
```



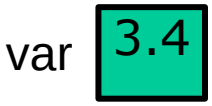


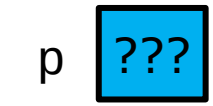


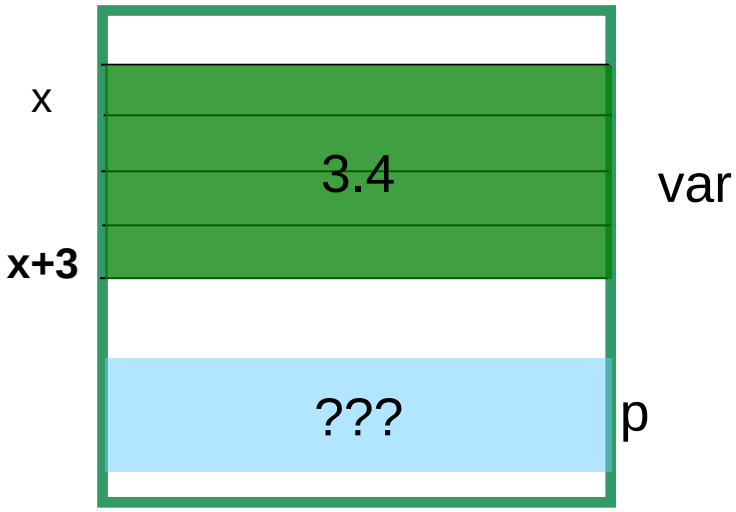
Puntatori

Come si assegna un indirizzo di memoria di una variabile ad un puntatore?

```
float var;
float *p;


var=3.4;
```



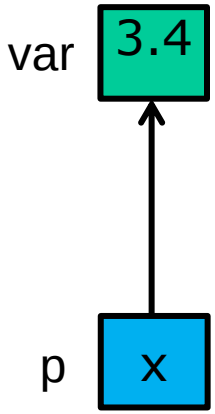


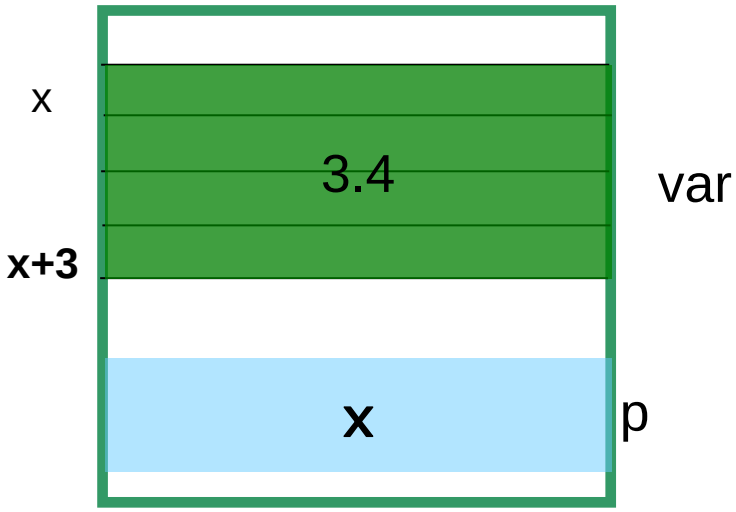


Puntatori

Come si assegna un indirizzo di memoria di una variabile ad un puntatore?

```
float var;
float *p;


var=3.4;
p = &var;
```






```
&variabile
```

L'operatore & ("indirizzo di") è un operatore unario e restituisce l'indirizzo di memoria di una variabile qualunque:

Puntatori

Abbiamo svelato il *mistero* della **`scanf`**

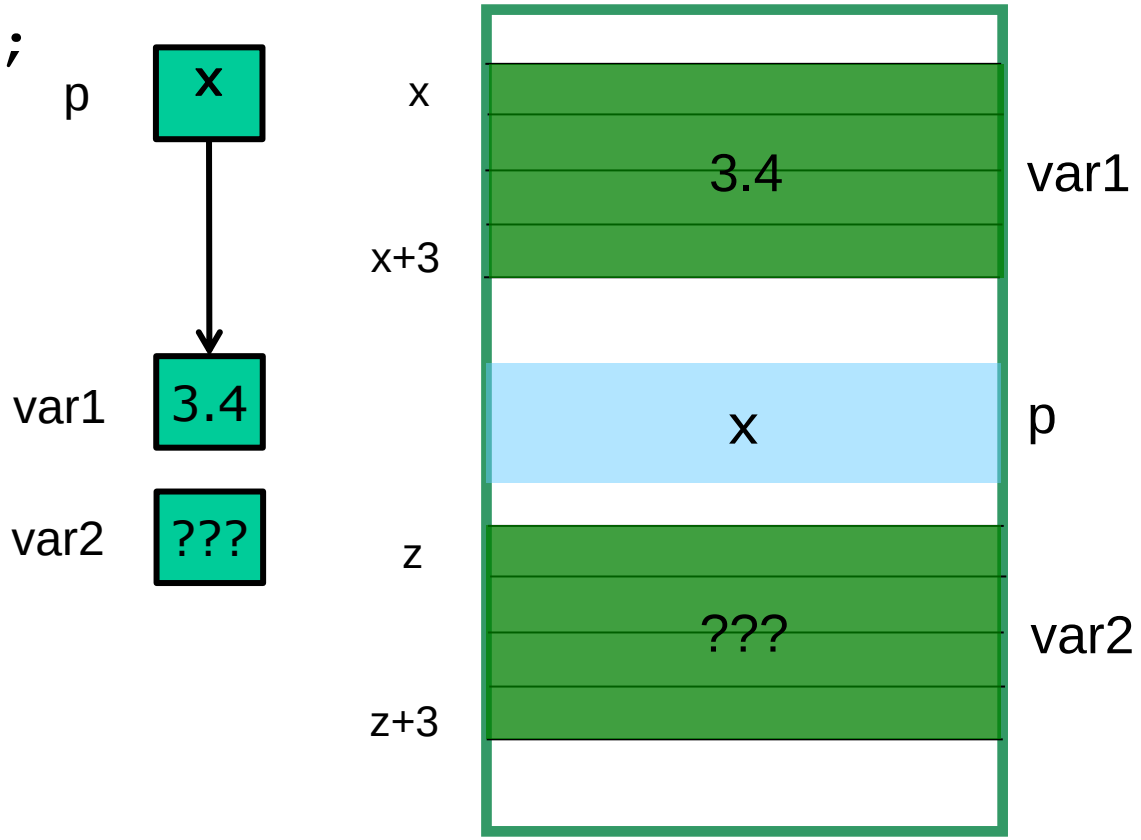
```
#define N 20

int x;
scanf("%d", &x);

char s[N + 1];
scanf("%d", s);
```

Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

```
*puntatore
```

```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
```

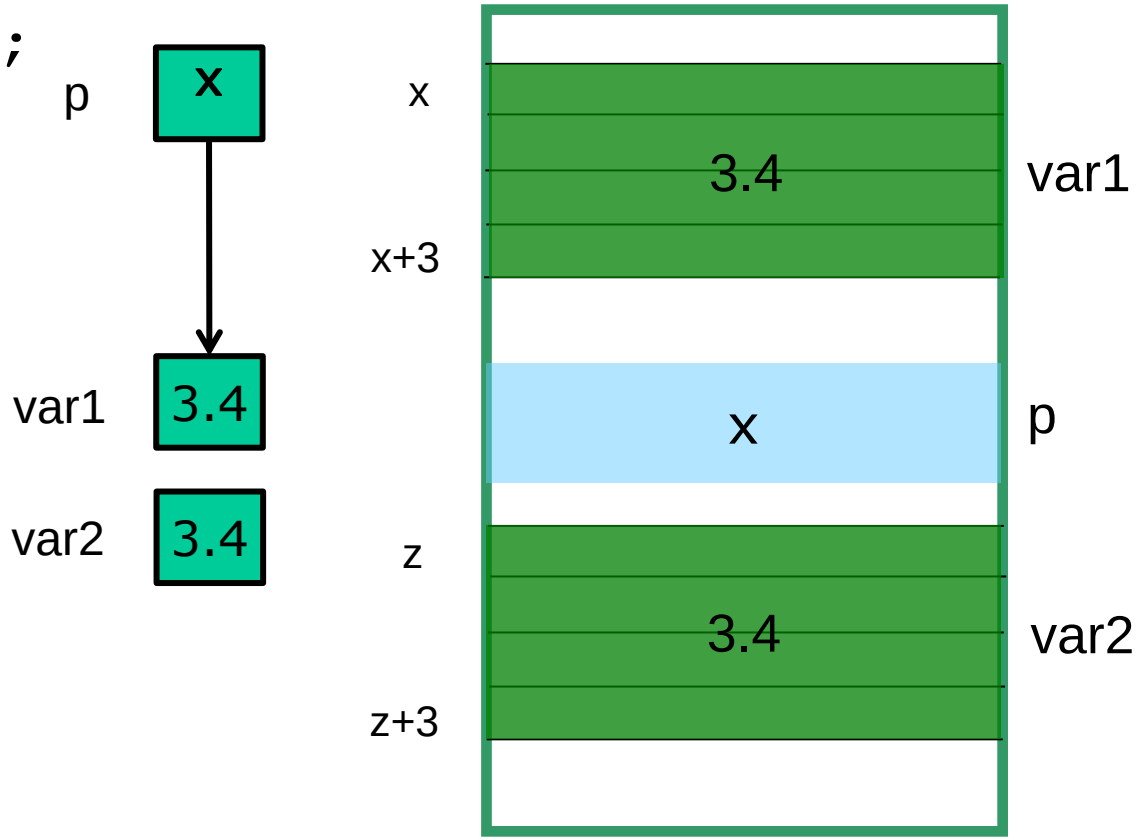


Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

`*puntatore`

```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
var2 = *p;
```

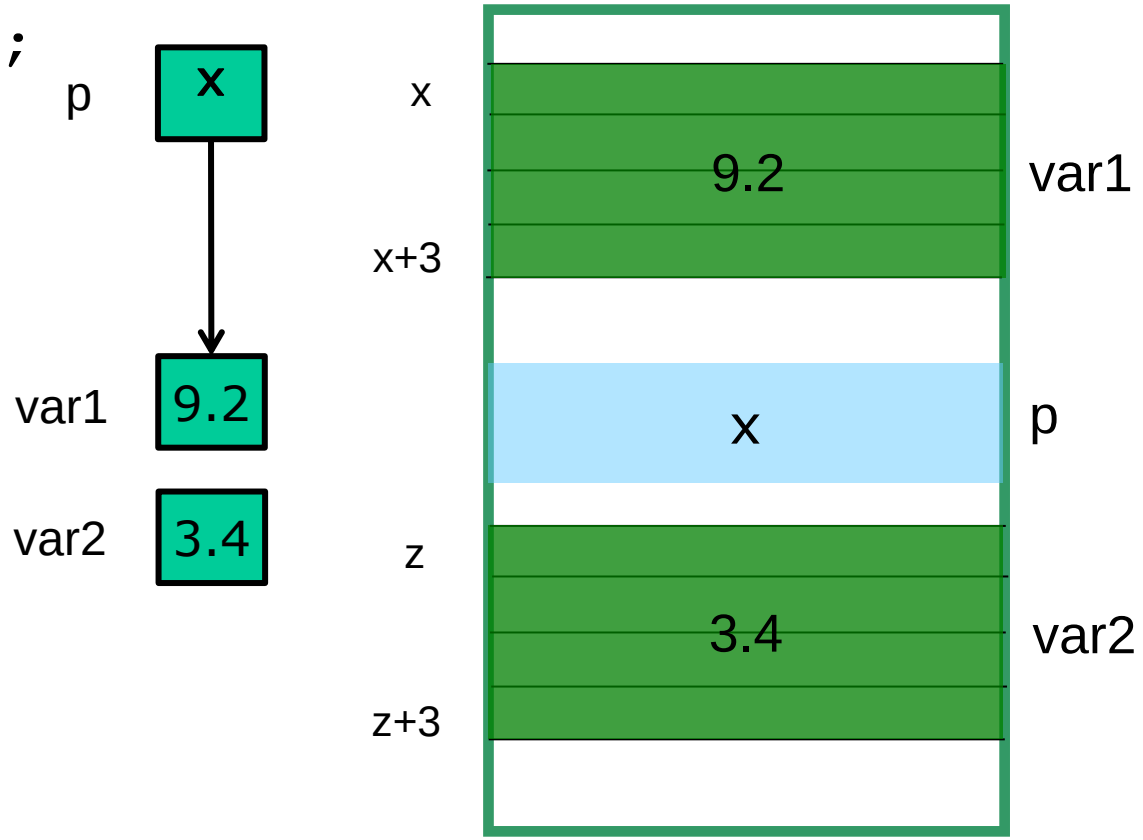


Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

`*puntatore`

```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
var2 = *p;
*p = 9.2;
```

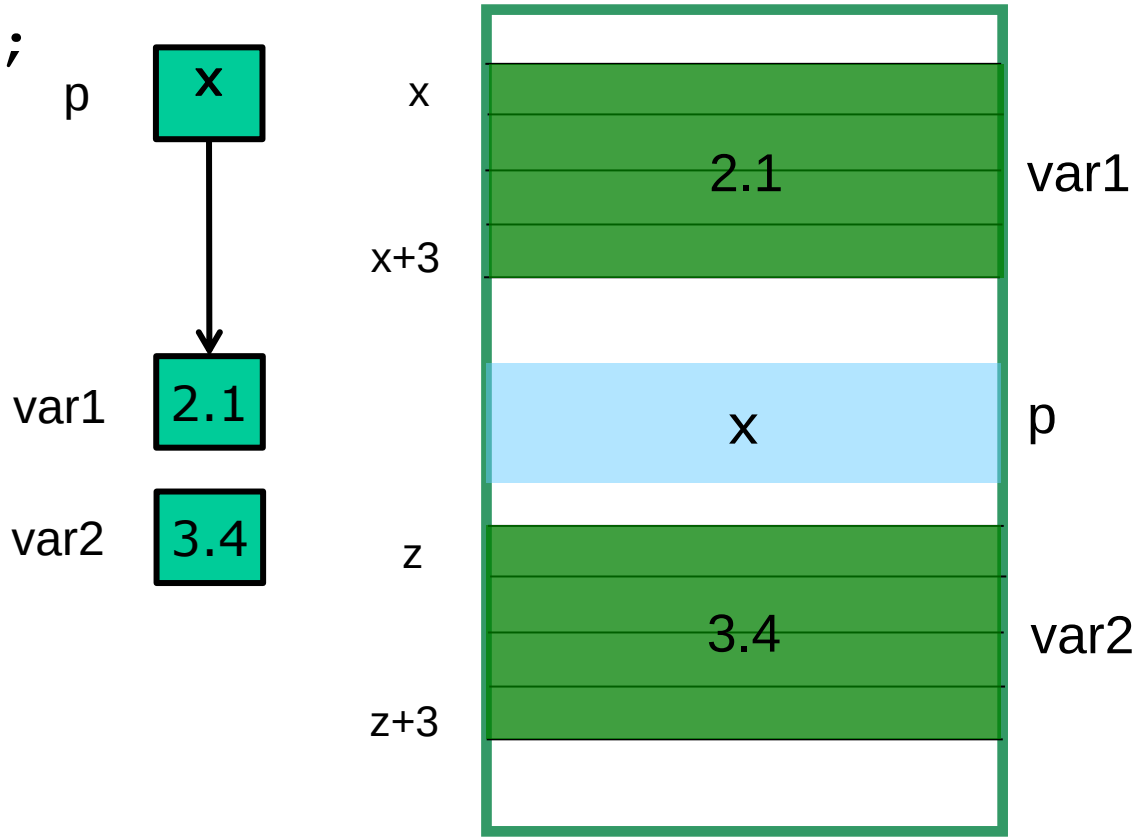


Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

`*puntatore`

```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
var2 = *p;
*p = 9.2;
var1 = 2.1;
```

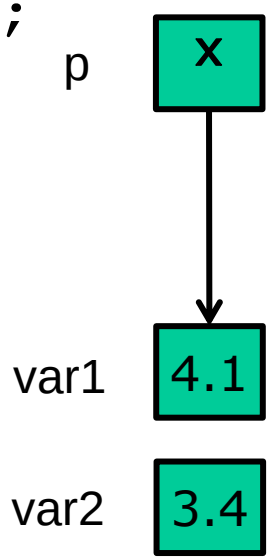


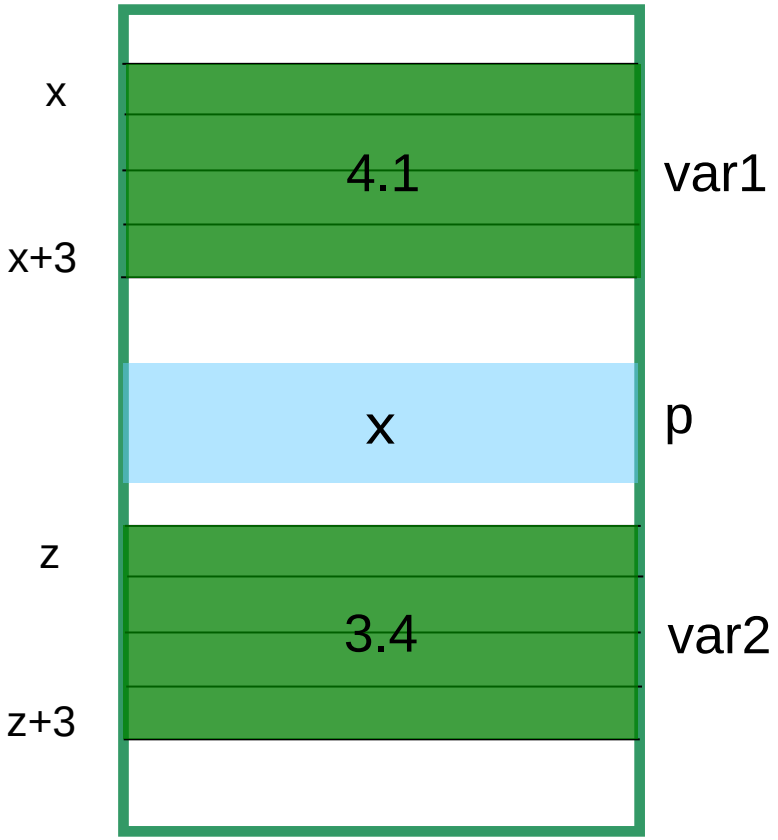
Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

`*puntatore`

```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
var2 = *p;
*p = 9.2;
var1 = 2.1;
*p = var1 + 2;
```



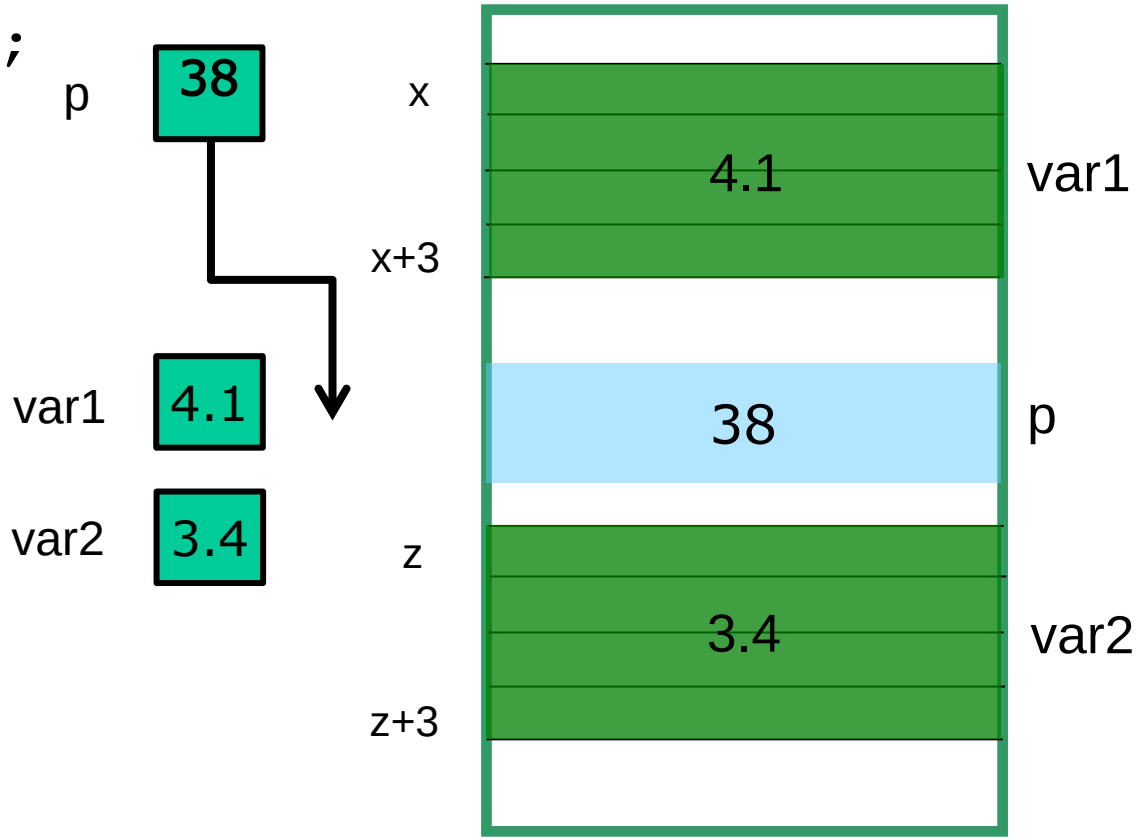


Puntatori

Come si accede al **contenuto** della cella di memoria puntata dal puntatore

`*puntatore`

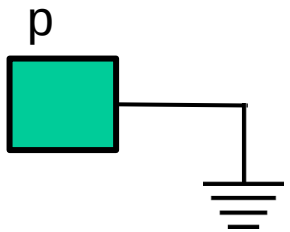
```
float var1,var2;
float *p;
var1 = 3.4;
p = &var1;
var2 = *p;
*p = 9.2;
var1 = 2.1;
*p = var1 + 2;
p = 38;
```



Puntatori

Inizializzazione

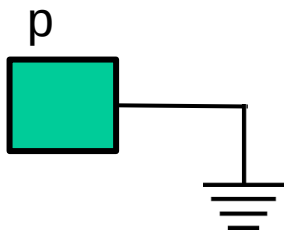
```
float *p = NULL;
```



Puntatori

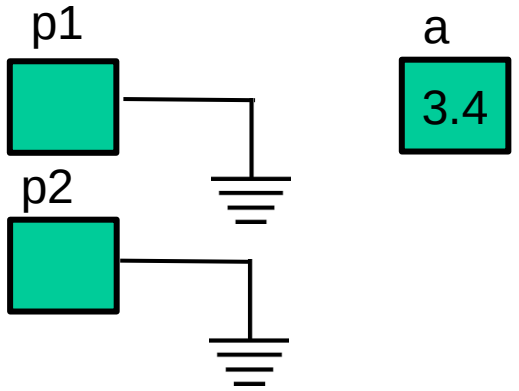
Inizializzazione

```
float *p = NULL;
```



Assegnamento fra puntatori

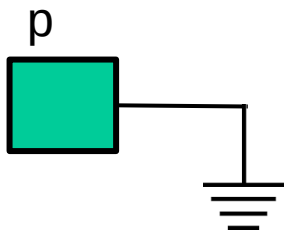
```
float a=3.4;
float *p1=NULL,*p2=NULL;
```



Puntatori

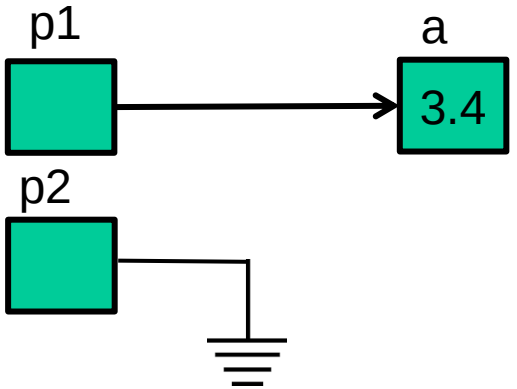
Inizializzazione

```
float *p = NULL;
```



Assegnamento fra puntatori

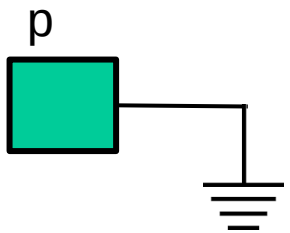
```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
```



Puntatori

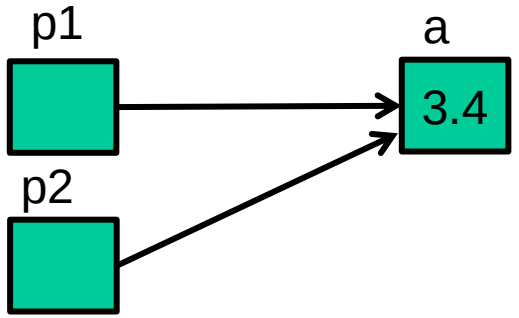
Inizializzazione

```
float *p = NULL;
```



Assegnamento fra puntatori

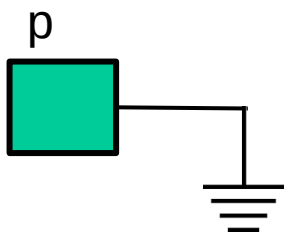
```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = p1;
```



Puntatori

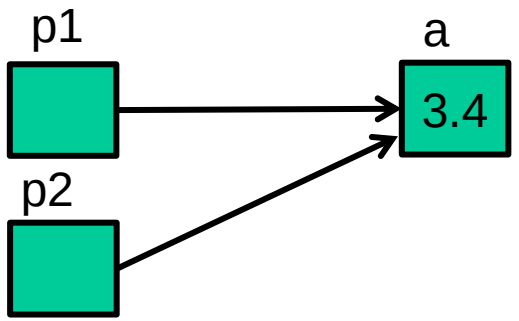
Inizializzazione

```
float *p = NULL;
```



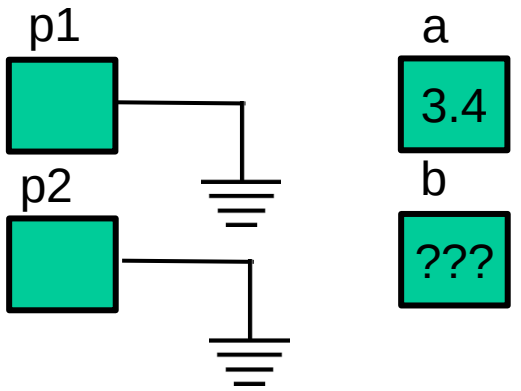
Assegnamento fra puntatori

```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = p1;
```



Assegnamento con dereferenziazione

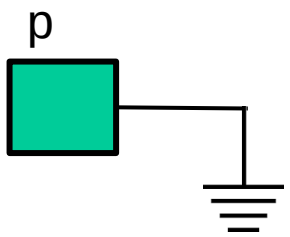
```
float a=3.4,b;
float *p1=NULL,*p2=NULL;
```



Puntatori

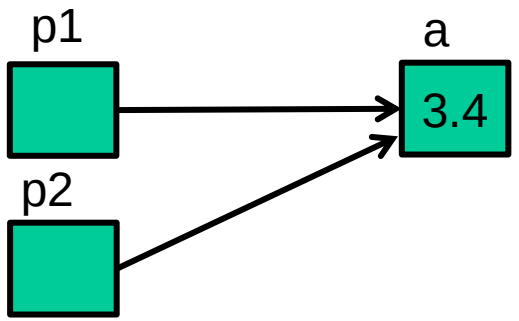
Inizializzazione

```
float *p = NULL;
```



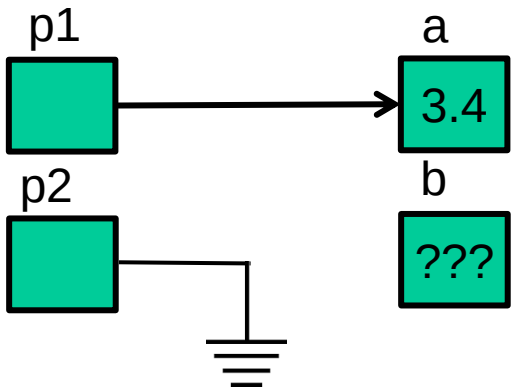
Assegnamento fra puntatori

```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = p1;
```



Assegnamento con dereferenziazione

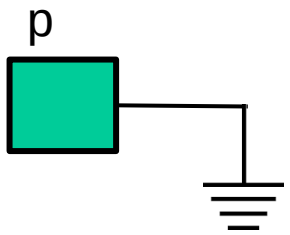
```
float a=3.4,b;
float *p1=NULL,*p2=NULL;
p1 = &a;
```



Puntatori

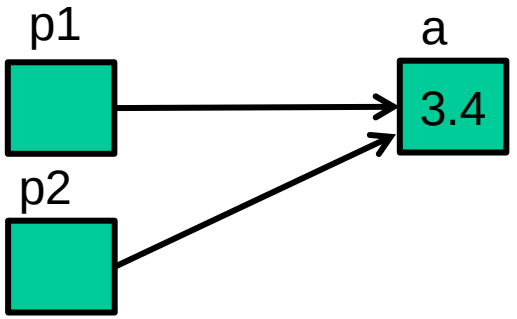
Inizializzazione

```
float *p = NULL;
```



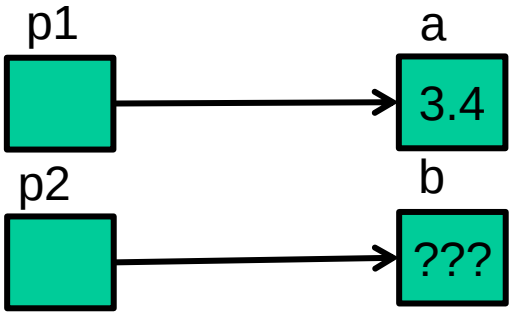
Assegnamento fra puntatori

```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = p1;
```



Assegnamento con dereferenziazione

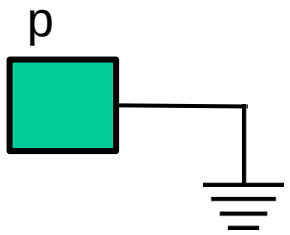
```
float a=3.4,b;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = &b;
```



Puntatori

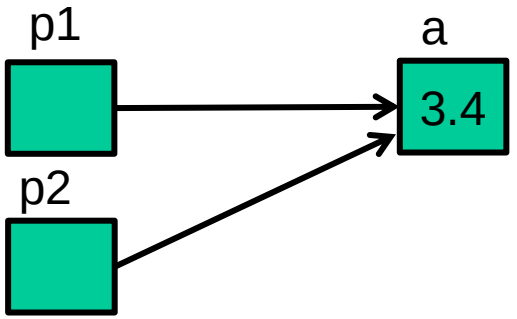
Inizializzazione

```
float *p = NULL;
```



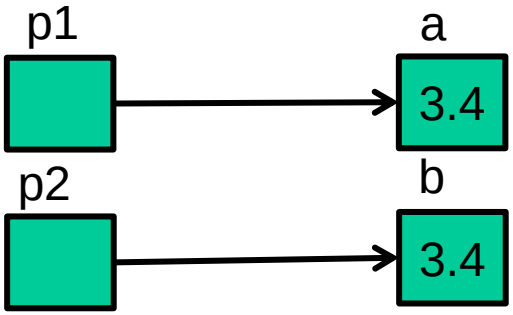
Assegnamento fra puntatori

```
float a=3.4;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = p1;
```



Assegnamento con dereferenziazione

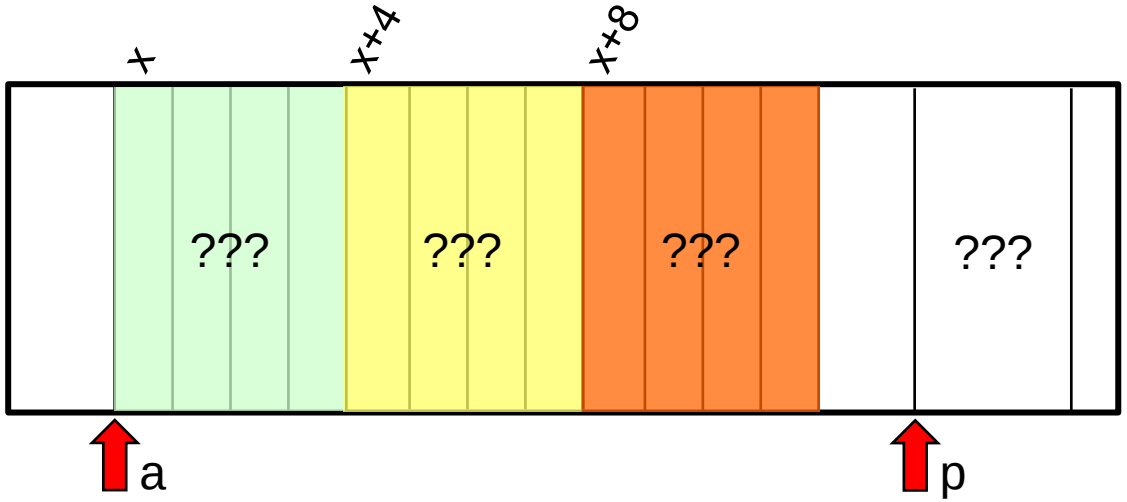
```
float a=3.4,b;
float *p1=NULL,*p2=NULL;
p1 = &a;
p2 = &b; *p2=*p1;
```



Aritmetica dei puntatori

Il C consente di effettuare somme e sottrazioni sui puntatori

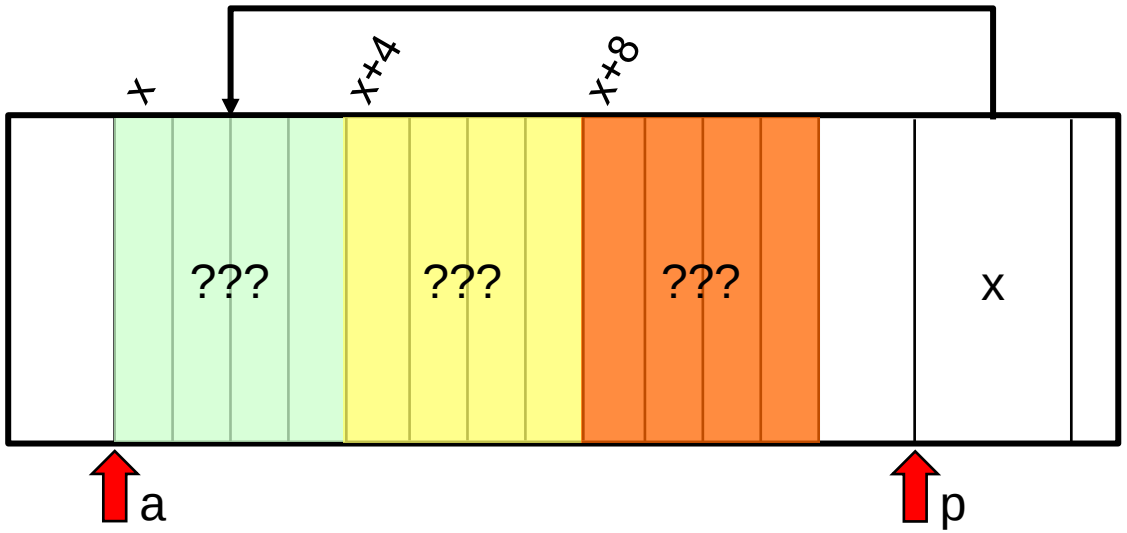
```
float *p;
float a;
```



Aritmetica dei puntatori

Il C consente di effettuare somme e sottrazioni sui puntatori

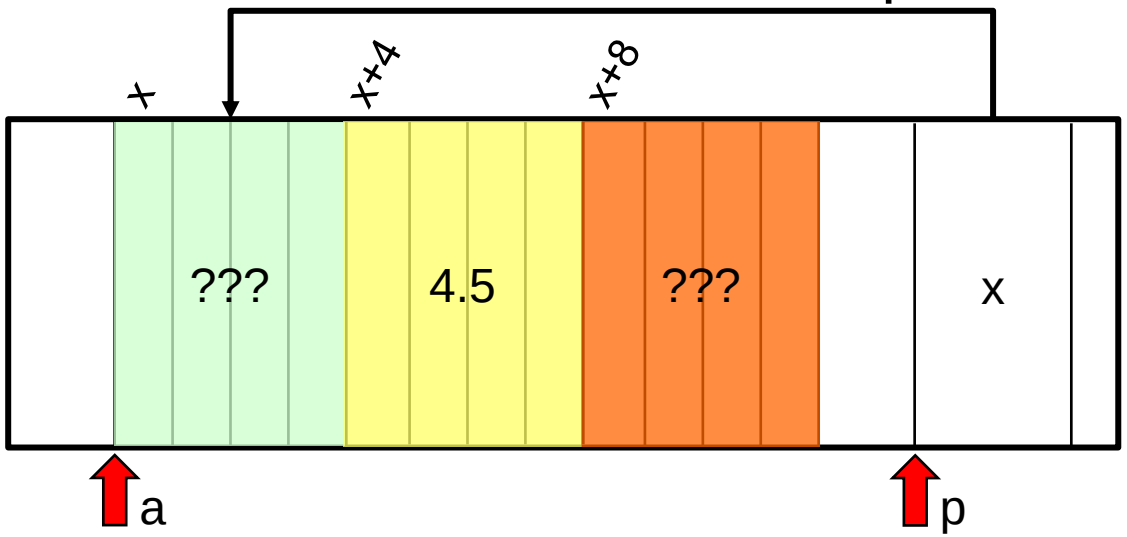
```
float *p;
float a;
p=&a;
```



Aritmetica dei puntatori

Il C consente di effettuare somme e sottrazioni sui puntatori

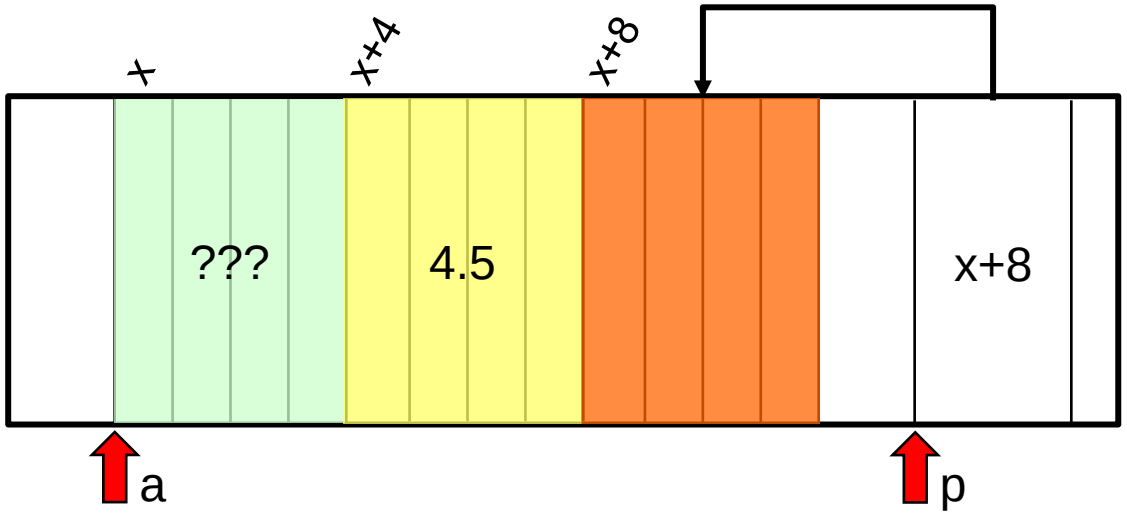
```
float *p;
float a;
p=&a;
*(p+1)=4.5;
```



Aritmetica dei puntatori

Il C consente di effettuare somme e sottrazioni sui puntatori

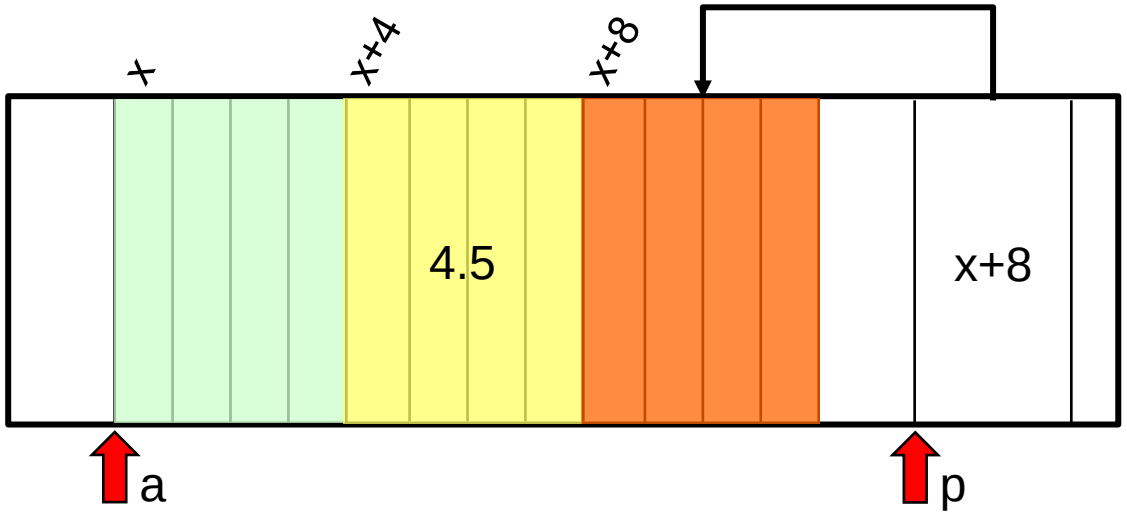
```
float *p;
float a;
p=&a;
*(p+1)=4.5;
p=p+2;
```



Aritmetica dei puntatori

Il C consente di effettuare somme e sottrazioni sui puntatori

```
float *p;
float a;
p=&a;
*(p+1)=4.5;
p=p+2;
```



Sintassi

```
puntatore = puntatore + x
```

Semantica

- incrementa/decrementa l'indirizzo contenuto nel puntatore di x **posizioni**
- la dimensione di ogni posizione, dipende dal tipo del puntatore
- questo è il motivo per cui è obbligatorio dichiarare il tipo del puntatore

La funzione sizeof

```
sizeof(<arg>)
```

Se <arg> è

- un tipo di dato, ritorna la quantità di memoria (in byte) necessaria per rappresentare un valore di quel tipo
- una variabile scalare, ritorna la quantità di memoria (in byte) occupata da quella variabile
- un array, ritorna la quantità di memoria (in byte) occupata dall'intero array

La funzione sizeof

```
type *pun;
pun = address + n;
```

Sposta il puntatore (a partire dall'indirizzo address) in avanti di **n * sizeof(type)** indirizzi…

…quindi di n elementi di tipo type

Puntatori a struct

```
typedef struct {
    float a;
    int b;
} mioTipo;


mioTipo var, *p
p = &var;

//le seguenti notazioni sono equivalenti
(*p).a = 4.5; /* diverso da *p.a */
p->a = 4.5;
```

Puntatori a struct

```
typedef struct {
    float a;
    int b;
} mioTipo;
```

Per i puntatori a struct valgono entrambe le sintassi

`(*p).campo` = `p->campo`

```
mioTipo var, *p
p = &var;

//le seguenti notazioni sono equivalenti
(*p).a = 4.5; /* diverso da *p.a */
p->a = 4.5;
```

Puntatori e Array

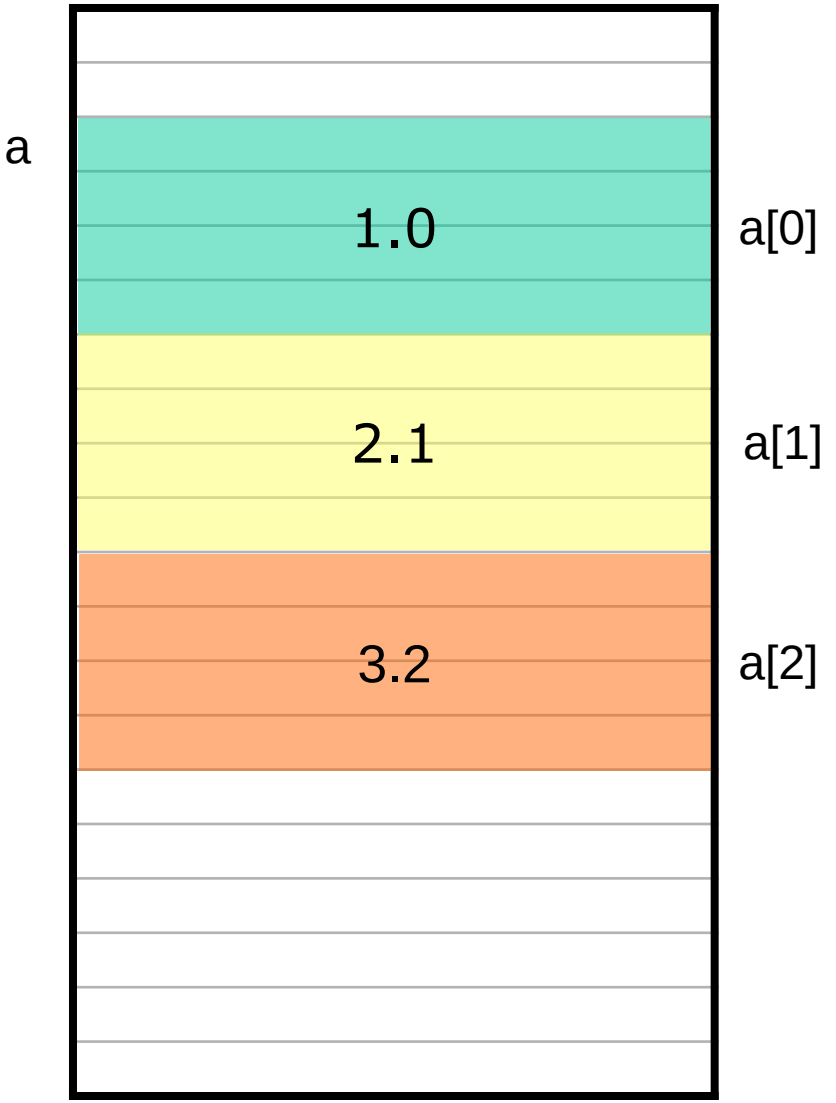
Puntatori e array

Un array viene memorizzato come un *blocco contiguo* a partire da un indirizzo di partenza (**indirizzo base**)

Il nome della variabile array (senza specificare la posizione con []) equivale l'indirizzo di partenza (del primo elemento dell'array)

Esempio

```
float a[3];

a[0]=1.0;

a[1]=2.1;

a[2]=3.2;
```

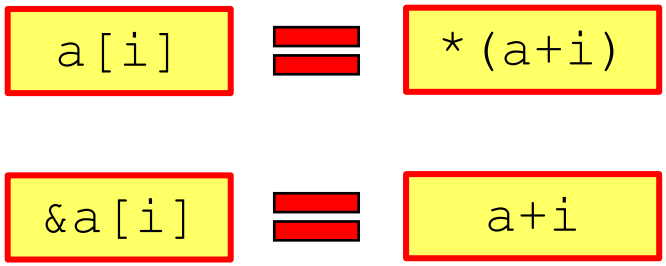


Puntatori e array

E' possibile accedere ad un array attraverso i puntatori

```
float a[5];

for (int i=0; i<5; i++)

        scanf("%f",a+i);
```

Più in generale, se `a` è una variabile di tipo array



Puntatori e array

```
int *p, a[5];

p = a;

a[2] = 3; /*scrivo ne terzo elemento*/

*(a+4) = 2; /*scrivo nel quinto elemento*/

*(p+1) = 23; /*scrivo nel secondo elemento*/
```

In generale si preferisce usare l'operatore [ ] per gli oggetti dichiarati come array/matrici e l'aritmetica dei puntatori per gli oggetti dichiarati come puntatore

Puntatori e array

```
int *p, a[5];

p = a;
```

Tuttavia `a` e `p` non sono proprio la stessa cosa: a è un valore puntatore (un indirizzo) costante (non modificabile)

- a `p` è possibile assegnare un valore (un indirizzo di memoria) mentre non è possibile farlo con `a`

Puntatori e array

Ci siamo spiegati perchè

- Non è possibile usare l'operatore = per copiare il contenuto di un array o di una stringa

```
float a[5],b[5];

b=a; //errore: b ed a sono solo gli indirizzi base
```

- Non si può usare l'operatore == per confrontare due array o due stringhe

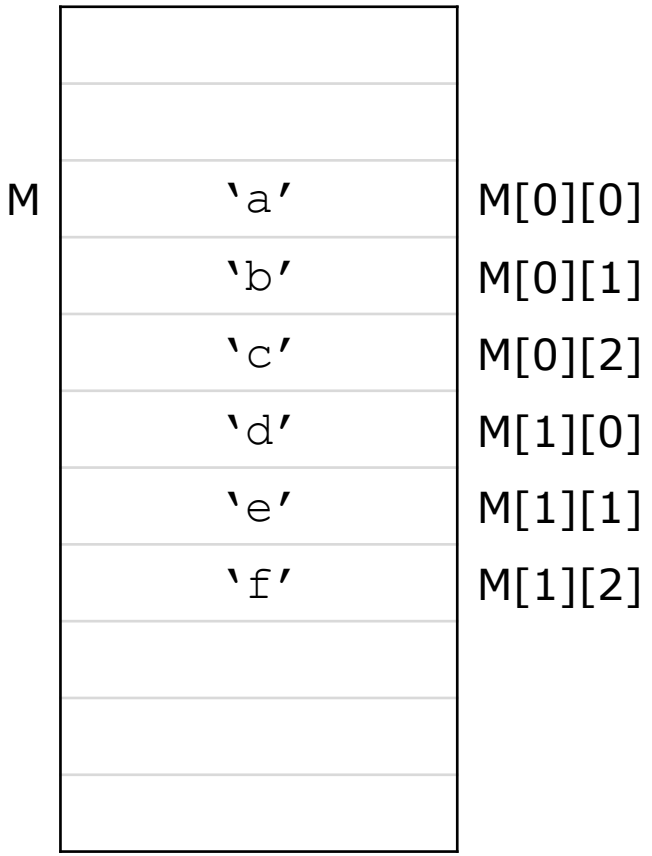
```
char a[20],b[20];

if(a==b) //vero se a e b hanno lo stesso indirizzo
```

Puntatori e Matrici

Puntatori e matrici

In maniera simile agli array, le matrici vengono memorizzate come un *blocco contiguo* (linearizzato) a partire da un **indirizzo base**, **riga dopo riga.**

Il nome della variabile matrice (usato senza specificare un indice di riga e colonna con []), è l'indirizzo base della matrice.



Esempio

```
char M[2][3];

M[0][0]='a';

M[0][1]='b';

M[0][2]='c';

M[1][0]='d';

M[1][1]='e'; M[1][2]='f';
```

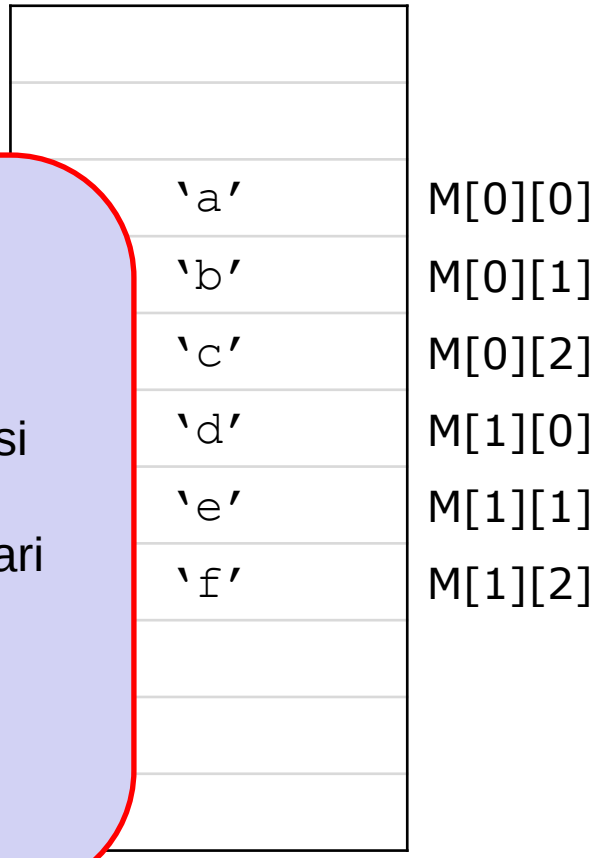
Puntatori e matrici

In maniera simile agli array, le matrici vengono memorizzate come un *blocco contiguo* a partire da un **indirizzo base**, **riga dopo riga.**

Il nome della
specificare u
è l'indirizzo

Esempio

```
char M[2][3]
M[0][0]='a';
M[0][1]='b';
M[0][2]='c';
M[1][0]='d';
M[1][1]='e'; M[1][2]='f';
```



Il nome di una matrice (senza parentesi quadre) rappresenta un puntatore costante ad un array (di dimensione pari al numero di colonne della matrice)

Puntatori e matrici

```
char M[2][3];
```

`M` **non** è un puntatore char: `char *` ma un puntatore `char (*) [3]`

- `char *p = M; /* genera un warning */`

`M` è un puntatore ad un array di 3 char: `char (*)[3]`

- `char (*p)[3] = M; /* p punta alla prima riga di M */`

Le parentesi tonde `(*p)` sono obbligatorie altrimenti non dichiaro un puntatore a array ma un array di puntatori

`p == &M[0][0]`

```
(*p)[1] = 'x'; /* assegna 'x' a M[0][1] */
(*(p+1))[0] = 'E'; /* assegna 'E' a M[1][0]
```

Puntatori e matrici

```
char M[2][3];
```

In alternativa:

```
char *p;

p = &M[0][0];

*(p+3*0+1) = 'x'; /* assegna 'x' a M[0][1] */

*(p+3*1+0) = 'E'; /* assegna 'E' a M[1][0]

*(p+3*1+2) = 'f'; /* assegna 'f' a M[1][2]
```

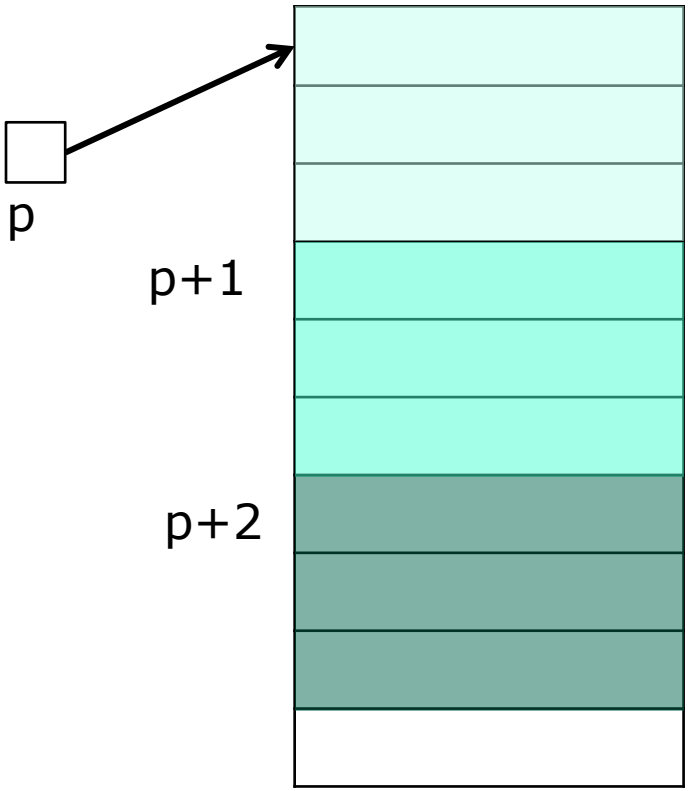
In generale, dati:

```
type M[R][C]; type *p = &M[0][0];

*(p+i*C+j) /* accede a M[i][j] */
```

Puntatori ad array e array di puntatori

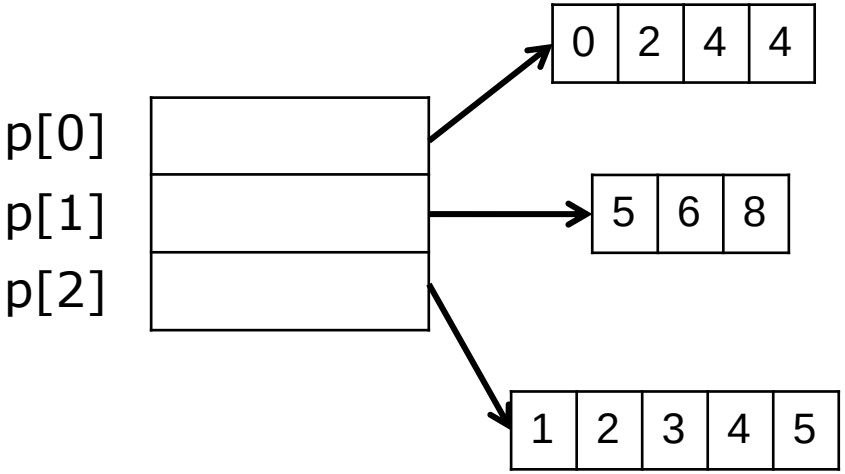
Un puntatore ad un array/matrice è un puntatore a un'area di memoria

```
float (*p)[3];
```



Un array di puntatori è più flessibile e permette di gestire aree di memoria di dimensione diversa

```
float *p[3];
```

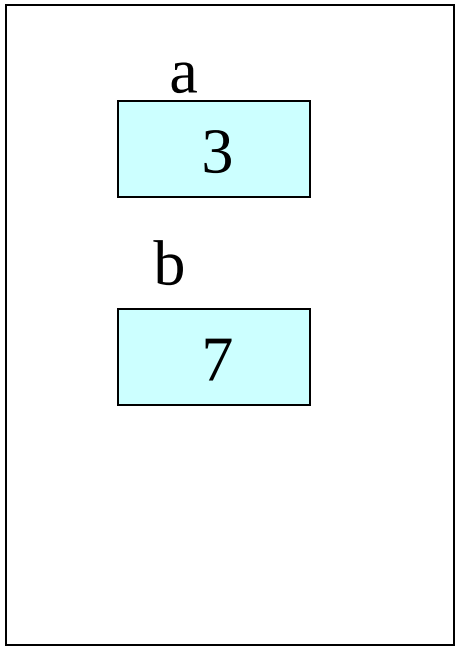


Puntatori e Funzioni

Puntatori e funzioni

```
void swap (int p, int q) {
          int temp;
          temp = p;
          p = q;
          q = temp;
}
```

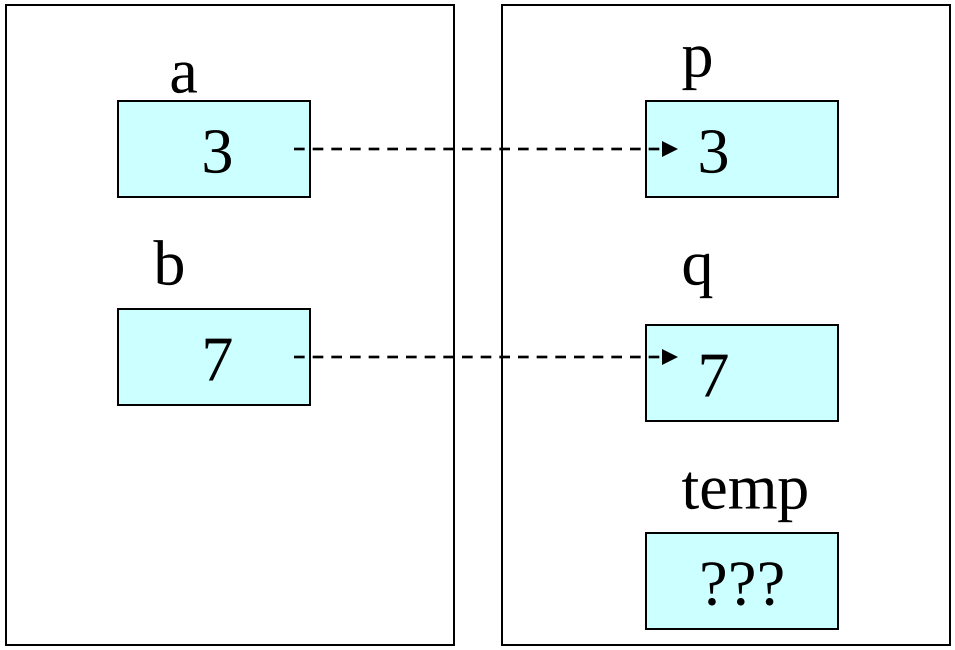
Nel main: `swap(a,b);`



Puntatori e funzioni

```
void swap (int p, int q) {
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

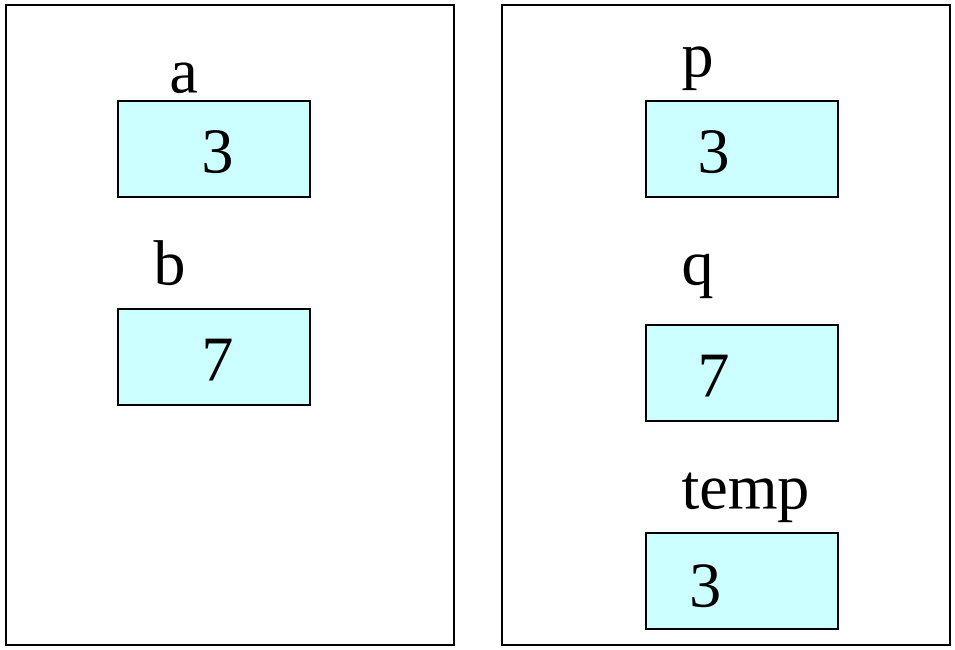
Nel main: `swap(a,b);`



Puntatori e funzioni

```
void swap (int p, int q) {
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

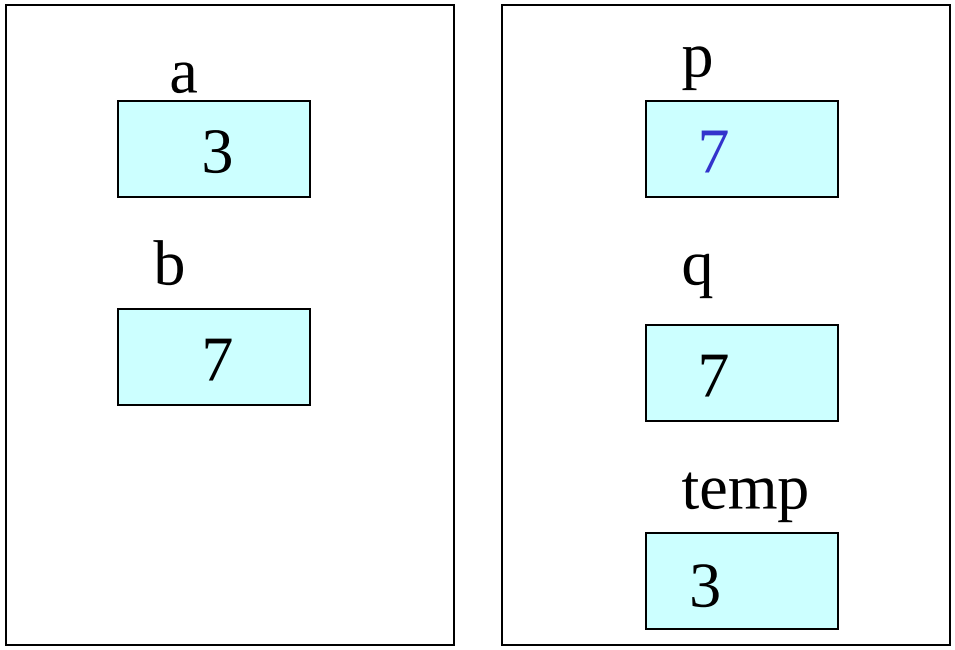
Nel main: `swap(a,b);`



Puntatori e funzioni

```
void swap (int p, int q) {
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

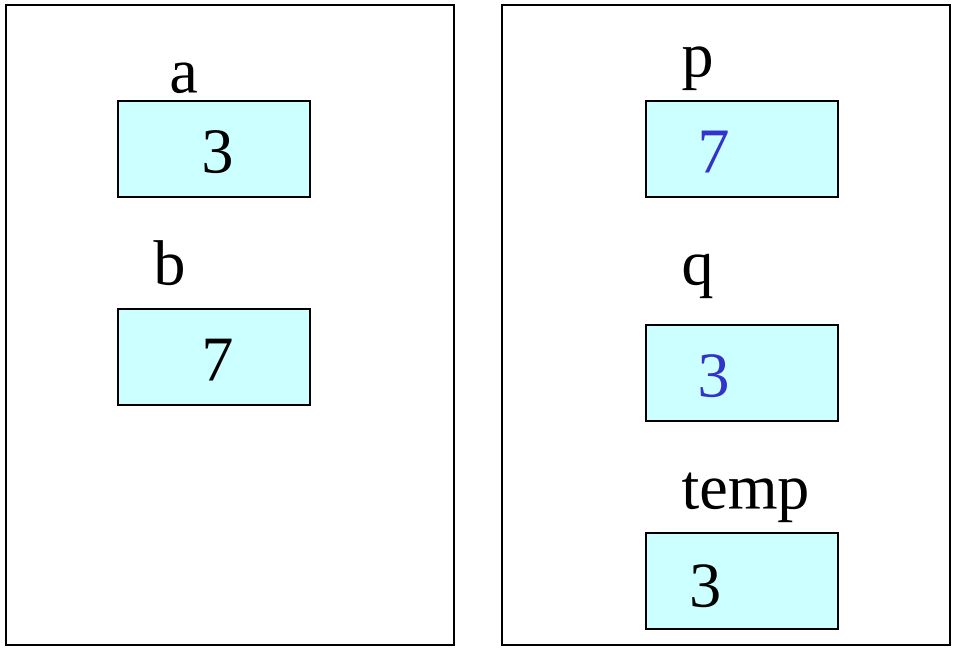
Nel main: `swap(a,b);`



Puntatori e funzioni

```
void swap (int p, int q) {
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

Nel main: `swap(a,b);`

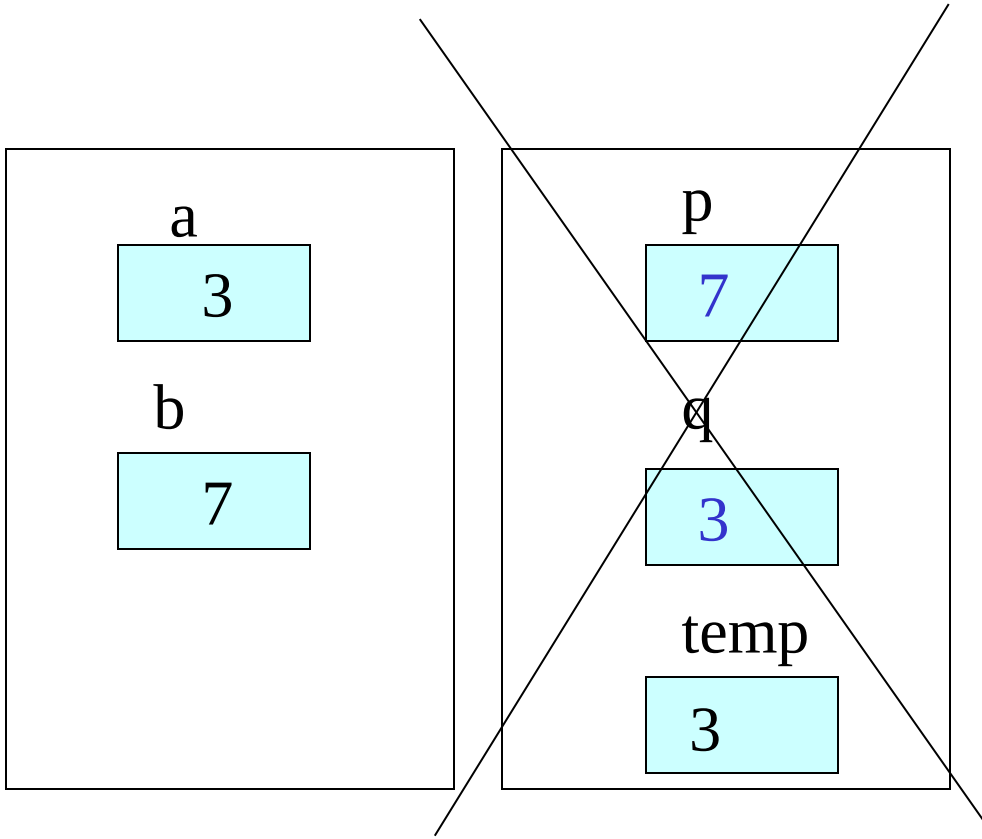


Puntatori e funzioni

```
void swap (int p, int q) {
      int temp;
      temp = p;
      p = q;
      q = temp;
}
```

Nel main: `swap(a,b);`

Al termine dell'esecuzione di `swap` le variabili nel main restano inalterate!



Puntatori e funzioni

Il problema è che dal chiamante al chiamato posso passare N parametri, ma dal chiamato al chiamante posso passare soltanto il **valore** specificato nella return

Non a caso si parla di passaggio dei parametri **per valore**

Passaggio dei parametri per indirizzo

- All’atto della chiamata l’indirizzo dei parametri attuali viene associato ai parametri formali
  - Il parametro attuale e il parametro formale **si riferiscono alla stessa cella di memoria**
- Il sottoprogramma in esecuzione lavora nel suo ambiente sui parametri formali (e di conseguenza anche sui parametri attuali) e ogni modifica sul parametro formale è una modifica del corrispondente parametro attuale
- Gli effetti del sottoprogramma si manifestano nel chiamante con modifiche al suo ambiente locale di esecuzione
- **Meccanismo per implementare uno scambio di informazioni bidirezionale con i sottoprogrammi**

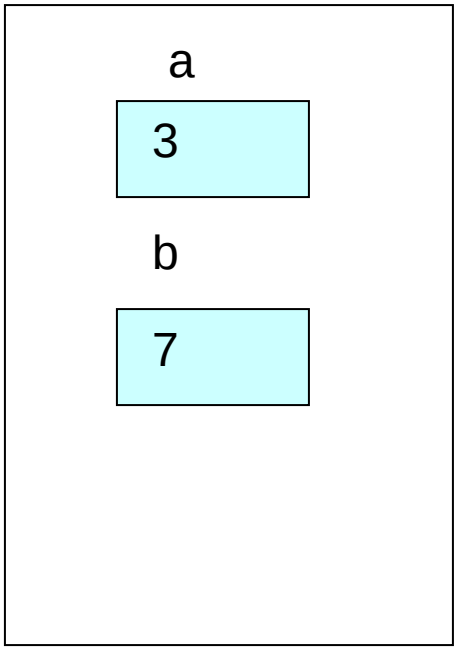
Passaggio dei parametri per indirizzo

- Il passaggio di indirizzo è realizzato mediante l'utilizzo dei puntatori
  - Nell'intestazione del sottoprogramma si specifica il parametro formale di tipo puntatore
  - All'atto della chiamata si passa come parametro attuale l'indirizzo della variabile (con l'operatore &)
- È possibile passare variabili di tutti i tipi di dato già visti con il passaggio per valore
- Il passaggio delle struct è più efficiente poiché si copia solo un indirizzo e non l'intero contenuto della struttura (che può avere dimensioni considerevoli)
- È possibile passare anche array (come vi vedrà nel seguito)

Passaggio dei parametri per indirizzo

```
...
void swap (int *p, int *q);

void swap (int *p, int *q){
   int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}

int main(){
   int a, b;
   a=3;
   b=7;
   swap(&a, &b);
   ...
}
```

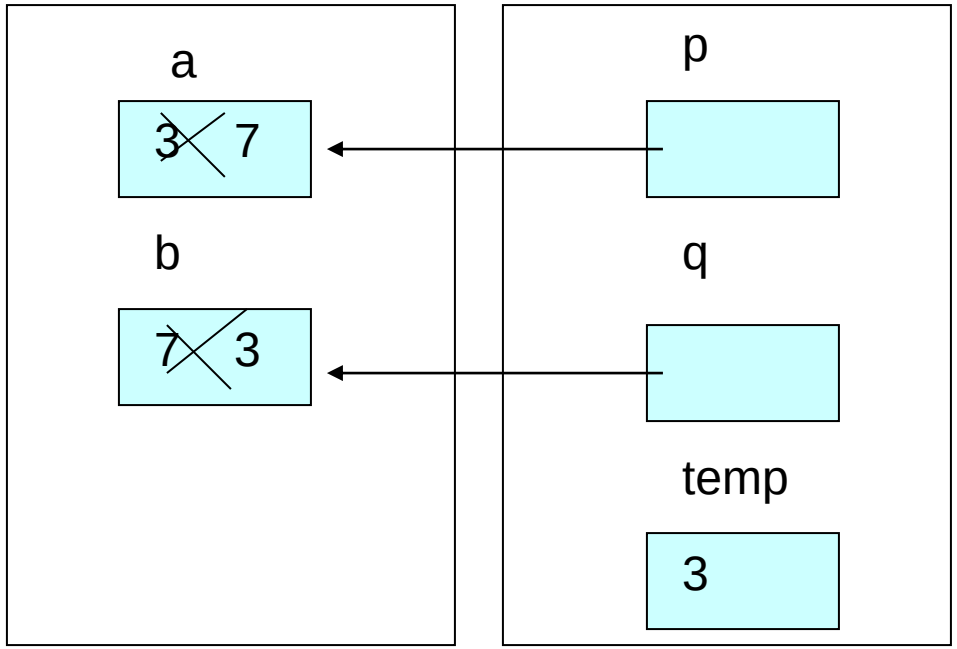
Prima dell'esecuzione di `swap`



Passaggio dei parametri per indirizzo

```
...
void swap (int *p, int *q);

void swap (int *p, int *q){
   int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}

int main(){
   int a, b;
   a=3;
   b=7;
   swap(&a, &b);
   ...
}
```

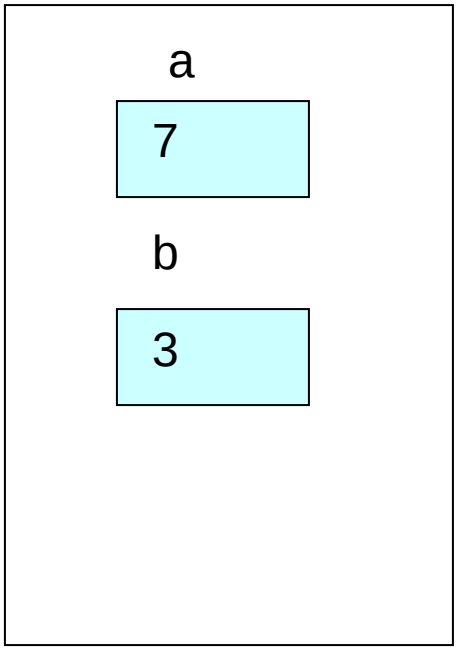
Alla fine dell'esecuzione di `swap`



Passaggio dei parametri per indirizzo

```
...
void swap (int *p, int *q);

void swap (int *p, int *q){
   int temp;
   temp = *p;
   *p = *q;
   *q = temp;
}

int main(){
   int a, b;
   a=3;
   b=7;
   swap(&a, &b);
   ...
}
```

Dopo l'esecuzione di `swap`



Passaggio dei parametri di tipo array

Per gli array monodimensionali non va specificata la dimensione tra le parentesi sia nel prototipo che nell'intestazione:

```
void invertiArray(int [], int);
```

È necessario passare come parametro la dimensione dell'array

- Il sottoprogramma in alternativa dovrebbe utilizzare la costante utilizzata nella definizione dell'array
- È sempre nota quando si realizza un sottoprogramma di libreria?

Per le stringhe non serve passare la dimensione come parametro poiché il contenuto valido termina con il valore `'\0'`

Non si può specificare un array come valore di `return`

- Verrebbe restituito un indirizzo. Ha senso?

Passaggio dei parametri di tipo array

L'array può essere passato soltanto per indirizzo

Prototipo del sottoprogramma:

```
void invertiArray(int [], int);
```

Sottoprogramma:

```
void invertiArray(int a[], int dim){
  int tmp;
  for(i=0; i<dim/2; i++){
    tmp=a[i];
    a[i]=a[dim-1-i];
    a[dim-1-i]=tmp;
  }
}
```

Invocazione:

```
invertiArray(array, dimArray);
```

Passaggio dei parametri di tipo array

È possibile utilizzare anche un puntatore:

```
void invertiArray(int* , int);
```

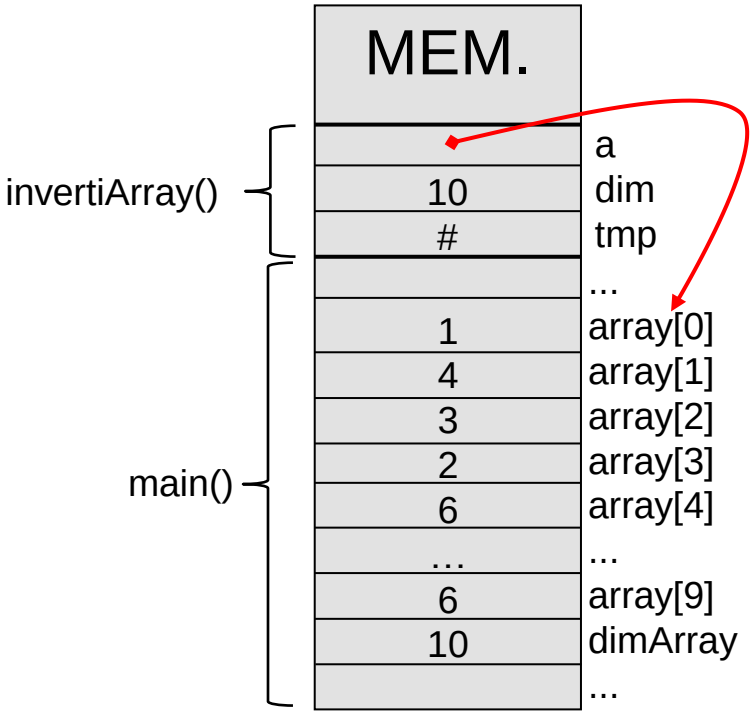
E quindi nel sottoprogramma utilizzare l'aritmetica dei puntatori:

```
void invertiArray(int *a, int dim){
  int tmp;
  for(i=0; i<dim/2; i++){
    tmp=*(a+i);
    *(a+i)=*(a+dim-1-i);
    *(a+dim-1-i)=tmp;
  }
}
```

Poiché il nome dell'array rappresenta un puntatore costante, le due soluzioni sono equivalenti!

Passaggio dei parametri di tipo array

Stato della memoria durante l'invocazione del sottoprogramma `invertiArray` eseguita nel `main`



Il parametro formale `a` contiene l'indirizzo del parametro attuale `array`. Ogni modifica viene effettuata direttamente su `array`

Passaggio dei parametri di tipo array multidimensionali

Per gli array multidimensionali si deve specificare tra `[]` tutte le dimensioni dalla seconda in poi (per permettere la linearizzazione della matrice), sia nel prototipo che nell'intestazione:

```
void trasponiMatrice(int a[][N], int dim1, int
dim2);
```

Si ricordi che il nome della matrice è un puntatore a vettore di dimensione costante, quindi il seguente prototipo è equivalente

```
void trasponiMatrice(int (*a)[N], int dim1, int
dim2);
```

Nell'invocazione specificheremo solo il nome della matrice

```
int b[M][N];
int righe, col;
...
trasponiMatrice(b, righe, col);
```

Passaggio dei parametri di tipo array multidimensionali

Dichiarazione alternativa: si può passare l'indirizzo della prima cella

```
void trasponiMatrice2(int *a , int dim1, int
dim2);
```

Nell'invocazione specificheremo l'indirizzo della prima cella della matrice

```
int b[M][N];
...
trasponiMatrice2(&b[0][0], M, N);
```

Passaggio dei parametri di tipo array multidimensionali

Nel caso di sottoutilizzo di array e array multidimensionali al sottoprogramma chiamato vanno passate anche le dimensioni effettive

Passaggio dei parametri di tipo struct

Parametri di tipo `struct` possono essere passati sia per valore che per indirizzo

Si suggerisce di passare le strutture per indirizzo per evitare onerose copie di valori dal record di attivazione del chiamante a quello del chiamato specialmente nel caso di strutture dati molto grandi