

Puntatori e Aritmetica dei puntatori

Lezione 6

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2019/2020



Argomenti del Corso

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

- Introduzione all'ambiente Linux
- Introduzione al C
- Tipi primitivi e costrutti condizionali
- Costrutti iterativi ed array
- Funzioni, stack e visibilità variabili
- Puntatori e memoria
- Debugging
- Tipi di dati utente
- Liste concatenate e librerie
- Ricorsione

Sommario

- 1 Puntatori
 - Variabili puntatore
 - Aritmetica dei puntatori
- 2 Puntatori e funzioni
 - Passaggio per riferimento
- 3 Array Multidimensionali

Errori comuni

```
#include <stdio>
#define N 10

void f(int [] a){
    ...
}

int main() {
    int a[N];
    f(a[N]);
    return 0;
}
```

Errori comuni

```
#include <stdio>
#define N 10

void f(int [] a){
    ...
}

int main() {
    int a[N];
    f(a[N]);
    return 0;
}
```

f(a) invece di f(a[N])

Errori comuni

```
float f(){  
    ...  
}  
  
int main(){  
    printf("%d\n", f());  
    return 0;  
}
```

Errori comuni

```
float f(){  
    ...  
}  
  
int main(){  
    printf("%d\n", f());  
    return 0;  
}
```

%f invece di %d

Errori comuni

```
int main() {  
    int somma, num_valori;  
    .....  
    printf("%.2f\n", somma / num_valori);  
    return 0;  
}
```


Errori comuni

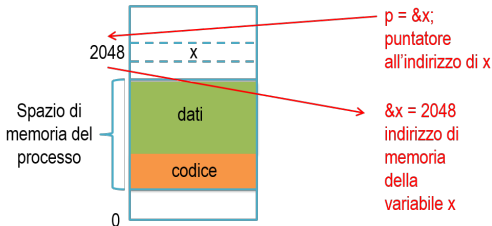
```
int main() {  
    int somma, num_valori;  
    .....  
    printf("%.2f\n", somma / num_valori);  
    return 0;  
}
```

somma / (float) num_valori

Outline

- 1 Puntatori
 - Variabili puntatore
 - Aritmetica dei puntatori
- 2 Puntatori e funzioni
 - Passaggio per riferimento
- 3 Array Multidimensionali

Indirizzi di Memoria e Puntatori



- In C è possibile conoscere l'indirizzo della cella di memoria in cui è memorizzata una variabile (o una funzione!!)
 - Operatore unario `&` restituisce l'indirizzo di memoria di una variabile, e.g. `&x`
- **Puntatore**: variabile che denota un indirizzo di memoria nello spazio di indirizzamento del processo

Variabili Puntatore

```
int a = 10; //a e' una variabile intera (iniz. a 10)
```

Variabile	Indirizzo	Contenuto
a	0x000064	10

Variabili Puntatore

```
int a = 10; //a e' una variabile intera (iniz. a 10)  
int *b; //b e' una variabile puntatore ad interi
```

Variabile	Indirizzo	Contenuto
a	0x000064	10
b	0x000068	

Variabili Puntatore

```
int a = 10; //a e' una variabile intera (iniz. a 10)  
int *b; //b e' una variabile puntatore ad interi  
b = &a; //b contiene l'indirizzo di memoria di a
```

Variabile	Indirizzo	Contenuto
a	0x000064	10
b	0x000068	0x000064



Variabili Puntatore

```
int a = 10; //a e' una variabile intera (iniz. a 10)
int *b; //b e' una variabile puntatore ad interi
b = &a; //b contiene l'indirizzo di memoria di a
...
//Attraverso l'indirizzo di memoria posso
manipolare il contenuto di una variabile
*b = *b - 2;
```



Variabili Puntatore

```
int a = 10; //a e' una variabile intera (iniz. a 10)
int *b; //b e' una variabile puntatore ad interi
b = &a; //b contiene l'indirizzo di memoria di a
...
//Attraverso l'indirizzo di memoria posso
manipolare il contenuto di una variabile
*b = *b - 2;
```



Dichiarare ed operare sui puntatori

```
nometipo * var
```

Dichiara:

- una variabile puntatore con nome `var`
- e tipo: indirizzo di variabile di tipo `nometipo`

Dichiarare ed operare sui puntatori

`nometipo * var`

Dichiara:

- una variabile puntatore con nome `var`
- e tipo: indirizzo di variabile di tipo `nometipo`
- L'operatore `&` viene usato per **restituire l'indirizzo di una variabile**
 - `b = &a;`
 - Lo posso usare per ottenere l'indirizzo di una variabile puntatore, e.g. `&b!`
- L'operatore `*` viene usato per **accedere al contenuto** di un indirizzo di memoria memorizzato da un puntatore (**dereferenziazione**)
 - `*b = *b - 2;`

I due usi del simbolo *

Usato nella **dichiarazione** significa che la variabile è di tipo **puntatore a**

```
int *a;  
int **c;
```

I due usi del simbolo *

Usato nella **dichiarazione** significa che la variabile è di tipo **puntatore a**

```
int *a;  
int **c;
```

Usato nei comandi invece esegue la **dereferenziazione**:
accesso alla variabile puntata, in particolare:

I due usi del simbolo *

Usato nella **dichiarazione** significa che la variabile è di tipo **puntatore a**

```
int *a;  
int **c;
```

Usato nei comandi invece esegue la **dereferenziazione**:
accesso alla variabile puntata, in particolare:

- all'interno di un **espressione**, da **accesso al contenuto** dell'indirizzo di memoria puntato

```
if (*a > 10) { ... } else { ... }
```

I due usi del simbolo *

Usato nella **dichiarazione** significa che la variabile è di tipo **puntatore a**

```
int *a;  
int **c;
```

Usato nei comandi invece esegue la **dereferenziazione**:
accesso alla variabile puntata, in particolare:

- all'interno di un **espressione**, da **accesso al contenuto** dell'indirizzo di memoria puntato

```
if (*a > 10) { ... } else { ... }
```

- a **sinistra** dell'assegnamento, permette di **modificare il contenuto** dell'indirizzo di memoria puntato

```
*a = 10;
```

Operatori di dereferenziazione * e di indirizzo &

- hanno priorità più elevata degli operatori binari

Operatori di dereferenziazione * e di indirizzo &

- hanno priorità più elevata degli operatori binari
- * è associativo a destra: $**p$ è equivalente a $*(*p)$

Operatori di dereferenziazione * e di indirizzo &

- hanno priorità più elevata degli operatori binari
- * è associativo a destra: $**p$ è equivalente a $*(*p)$
- & può essere applicato **solo** ad una variabile;
&a non è una variabile quindi & **non è associativo**

Operatori di dereferenziazione * e di indirizzo &

- hanno priorità più elevata degli operatori binari
- * è associativo a destra: $**p$ è equivalente a $*(*p)$
- & può essere applicato **solo** ad una variabile;
&a non è una variabile quindi & **non è associativo**
- * e & sono uno l'inverso dell'altro

Operatori di dereferenziazione * e di indirizzo &

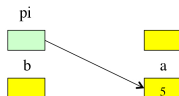
- hanno priorità più elevata degli operatori binari
- * è associativo a destra: $**p$ è equivalente a $*(*p)$
- & può essere applicato **solo** ad una variabile;
&a non è una variabile quindi & **non è associativo**
- * e & sono uno l'inverso dell'altro
 - data la dichiarazione `int a;`
`*&a` è un altro modo per denotare `a` (sono entrambi variabili)

Operatori di dereferenziazione $*$ e di indirizzo $&$

- hanno priorità più elevata degli operatori binari
- $*$ è associativo a destra: $**p$ è equivalente a $*(*p)$
- $&$ può essere applicato **solo** ad una variabile;
 $&a$ non è una variabile quindi $&$ **non è associativo**
- $*$ e $&$ sono uno l'inverso dell'altro
 - data la dichiarazione `int a;`
 $*\&a$ è un altro modo per denotare a (sono entrambi variabili)
 - data la dichiarazione `int *pi;`
 $*pi$ ha valore (un indirizzo) uguale al valore di pi
però:
 - pi è una variabile
 - $*pi$ non lo è (ad esempio, non può essere usato a sinistra di $=$)

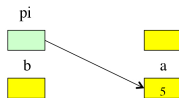
Operatori di dereferenziazione * e di indirizzo &

`pi = &a`

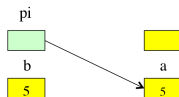


Operatori di dereferenziazione * e di indirizzo &

`pi = &a`

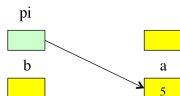


`b = *pi`

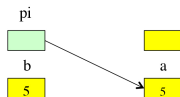


Operatori di dereferenziazione * e di indirizzo &

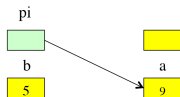
`pi = &a`



`b = *pi`



`*pi = 9`



Puntatori e tipi

E' possibile dichiarare puntatori per tutti i **tipi primitivi** (e anche le strutture)

Puntatori e tipi

E' possibile dichiarare puntatori per tutti i **tipi primitivi** (e anche le strutture)

```
int *a, *b; // Dichiarazione multipla (ripetere '*')  
float *c, d; // c e' un puntatore a float, d e' una  
variabile float
```

Puntatori e tipi

E' possibile dichiarare puntatori per tutti i **tipi primitivi** (e anche le strutture)

```
int *a, *b; // Dichiarazione multipla (ripetere '*')
float *c, d; // c e' un puntatore a float, d e' una
variabile float
double h,*e, **f; // f e' un puntatore a puntatore a
double
...
e= &h; // la sequenza di assegnamenti e' corretta
f = &e;
```

Puntatori e tipi

E' possibile dichiarare puntatori per tutti i **tipi primitivi** (e anche le strutture)

```
int *a, *b; // Dichiarazione multipla (ripetere '*')
float *c, d; // c e' un puntatore a float, d e' una
variabile float
double h,*e, **f; // f e' un puntatore a puntatore a
double
...
e = &h; // la sequenza di assegnamenti e' corretta
f = &e;
```

Che tipo ha *e? E *f e **f?

Puntatori e tipi

E' possibile dichiarare puntatori per tutti i **tipi primitivi** (e anche le strutture)

```
int *a, *b; // Dichiarazione multipla (ripetere '*')
float *c, d; // c e' un puntatore a float, d e' una
variabile float
double h,*e, **f; // f e' un puntatore a puntatore a
double
...
e = &h; // la sequenza di assegnamenti e' corretta
f = &e;
```

Che tipo ha *e? E *f e **f?

NULL: costante predefinita (in `stdio.h`) che denota il
puntatore nullo

Costanti e puntatori

Queste due dichiarazioni sono equivalenti, cioè **puntatori a costanti intere**

```
const int *a;  
int const *a;
```

Non potete fare `*a = 10!`

Costanti e puntatori

Queste due dichiarazioni sono equivalenti, cioè **puntatori a costanti intere**

```
const int *a;  
int const *a;
```

Non potete fare `*a = 10!`

```
const int *a; //Puntatore a costanti intere  
int *const a; //Puntatore costante ad interi
```

Costanti e puntatori

Queste due dichiarazioni sono equivalenti, cioè **puntatori a costanti intere**

```
const int *a;  
int const *a;
```

Non potete fare `*a = 10!`

```
const int *a; //Puntatore a costanti intere  
int *const a; //Puntatore costante ad interi
```

Non sono equivalenti!!! Nel secondo caso **non potete modificare l'indirizzo** a cui punta `a`, ma potete **cambiare il suo contenuto** con `*a!`

Aritmetica dei puntatori

E' possibile utilizzare alcuni degli **operatori aritmetici** classici (+, -, ++, -, ...) per scrivere **espressioni con i puntatori**

```
int a[4], *p; // Dichiaro un array di interi ed un  
             puntatore ad interi
```


Aritmetica dei puntatori

E' possibile utilizzare alcuni degli **operatori aritmetici** classici (+, -, ++, --, ...) per scrivere **espressioni con i puntatori**

```
int a[4], *p; // Dichiaro un array di interi ed un  
             // puntatore ad interi
```

```
p = &a[0];
```

**p punta all'indirizzo
di a[0]**

a[3]	ADDR + 12
a[2]	ADDR + 8
a[1]	ADDR + 4
a[0]	ADDR

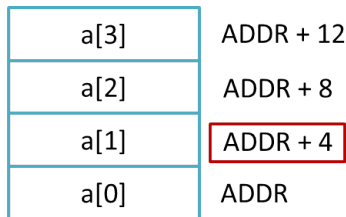
Aritmetica dei puntatori

E' possibile utilizzare alcuni degli **operatori aritmetici** classici (+, -, ++, -, ...) per scrivere **espressioni con i puntatori**

```
int a[4], *p; // Dichiaro un array di interi ed un
              // puntatore ad interi
```

```
p = &a[0];
p = p+1;
```

```
&a[1] == &a[0] + sizeof(int)
```

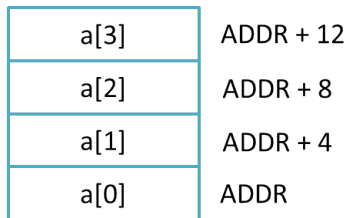


Aritmetica dei puntatori

E' possibile utilizzare alcuni degli **operatori aritmetici** classici (+, -, ++, --, ...) per scrivere **espressioni con i puntatori**

```
int a[4], *p; // Dichiaro un array di interi ed un  
              puntatore ad interi
```

```
p = &a[0];  
p = p+1;  
p--;
```

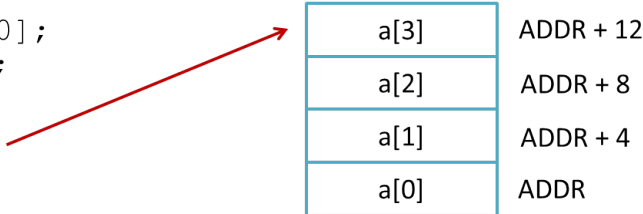


Aritmetica dei puntatori I

E' possibile utilizzare alcuni degli **operatori aritmetici** classici (+, -, ++, --, ...) per scrivere **espressioni con i puntatori**

```
int a[4], *p; // Dichiaro un array di interi ed un  
             // puntatore ad interi
```

```
p = &a[0];  
p = p+1;  
p--;  
p += 3;
```



a[3]	ADDR + 12
a[2]	ADDR + 8
a[1]	ADDR + 4
a[0]	ADDR

Aritmetica dei puntatori II

Il valore calcolato in corrispondenza di un'operazione del tipo $p+i$ **dipende dal tipo T** di p (analog. per $p-i$):

Op. Logica: $p = p+1$ Op.Algebrica: $p = p + \text{sizeof}(T)$

Aritmetica dei puntatori II

Il valore calcolato in corrispondenza di un'operazione del tipo $p+i$ **dipende dal tipo T** di p (analog. per $p-i$):

Op. Logica: $p = p+1$ Op.Algebrica: $p = p + \text{sizeof}(T)$

```
int *pi;  
*pi = 15;  
pi=pi+1; /* pi punta al prossimo int (4 byte dopo)*/
```

Aritmetica dei puntatori II

Il valore calcolato in corrispondenza di un'operazione del tipo $p+i$ **dipende dal tipo T** di p (analog. per $p-i$):

Op. Logica: $p = p+1$ Op.Algebrica: $p = p + \text{sizeof}(T)$

```
int *pi;  
*pi = 15;  
pi=pi+1; /* pi punta al prossimo int (4 byte dopo)*/
```

```
double *pd;  
*pd = 12.2;  
pd = pd+3; /* pd punta a 3 double dopo (24 byte dopo)*/
```

Aritmetica dei puntatori II

Il valore calcolato in corrispondenza di un'operazione del tipo $p+i$ **dipende dal tipo T** di p (analog. per $p-i$):

Op. Logica: $p = p+1$ Op.Algebrica: $p = p + \text{sizeof}(T)$

```
int *pi;  
*pi = 15;  
pi=pi+1; /* pi punta al prossimo int (4 byte dopo)*/
```

```
double *pd;  
*pd = 12.2;  
pd = pd+3; /* pd punta a 3 double dopo (24 byte dopo)*/
```

```
char *pc;  
*pc = 'A';  
pc = pc - 5; /* pc punta a 5 char prima (5 byte prima)*/
```


Puntatori ed array (I)

In generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.

L'unico caso in cui lo sappiamo è quando utilizziamo dei vettori.

Il **nome di un array** è un **puntatore costante** al primo elemento del vettore (`int *const`)

Puntatori ed array (I)

In generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.

L'unico caso in cui lo sappiamo è quando utilizziamo dei vettori.

Il **nome di un array** è un **puntatore costante** al primo elemento del vettore (`int *const`)

```
int a[4], *p;  
// I due comandi che seguono, sono equivalenti  
p = &a[0]; // Indirizzo del primo elemento dell'array  
p = a; // Puntatore al (primo elemento del) array
```

Puntatori ed array (I)

In generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.

L'unico caso in cui lo sappiamo è quando utilizziamo dei vettori.

Il **nome di un array** è un **puntatore costante** al primo elemento del vettore (`int *const`)

```
int a[4], *p;  
// I due comandi che seguono, sono equivalenti  
p = &a[0]; // Indirizzo del primo elemento dell'array  
p = a; // Puntatore al (primo elemento del) array
```

L'operatore `array[·]` è un'abbreviazione per un'operazione **aritmetica su puntatori**

```
int tmp;  
// I due assegnamenti che seguono sono equivalenti  
tmp = a[2]; // Terzo elemento dell'array  
tmp = *(a+2); // Contenuto dell'indirizzo puntato da a +  
           (2 * sizeof(int))
```

Puntatori ed array (II)

Un esempio che riassume i modi in cui si può accedere agli elementi di un vettore.

```
int vet[] = {11, 22, 33, 44, 55};  
int *pi = vet, offset = 3;
```

/ i seguenti assegnamenti sono equivalenti */*

```
vet[offset] = 88;  
*(vet + offset) = 88;  
pi[offset] = 88;  
*(pi + offset) = 88;
```

Puntatori ed array (III)

```
#define DIM 100
```

```
int a[DIM],*p, i;
```

```
/*Modi alternativi per scandire un vettore*/
```

```
for (i=0; i<DIM; i++)  
    printf("%d", a[i]);
```

```
for (i=0; i<DIM; i++)  
    printf("%d", p[i]);
```

```
for (i=0; i<DIM; i++)  
    printf("%d", *(a+i));
```

```
for (i=0; i<DIM; i++)  
    printf("%d", *(p+i));
```

```
for (p=a; p<a+DIM; p++)  
    printf("%d", *p);
```

Il Codice che non Compila!!! Perchè?

```
#define DIM 100  
int a[DIM], *p;  
for (p=a; a<p+DIM; a++) /*Non compila! Perche? */  
    printf("%d", *a);
```



Il Codice che non Compila!!! Perchè?

```
#define DIM 100  
int a[DIM], *p;  
for (p=a; a<p+DIM; a++) /*Non compila! Perche? */  
    printf("%d", *a);
```



`int a[DIM]` dichiara un **puntatore costante** a interi (`int *const`)
⇒ Non posso modificare dove punta a!

Cast esplicito

E' possibile fare il **cast esplicito** di un puntatore **ad un qualsiasi altro tipo di puntatore**

Cast esplicito

E' possibile fare il **cast esplicito** di un puntatore ad un qualsiasi altro tipo di puntatore

```
int a = 8;  
int *b; //Puntatore a interi  
double *c; //Puntatore a double  
...  
b = &a;  
c = (double*) b;
```

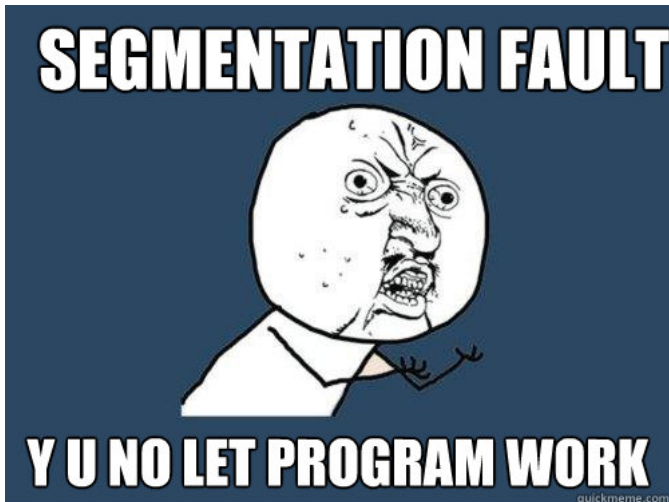
Cast esplicito

E' possibile fare il **cast esplicito** di un puntatore ad un qualsiasi altro tipo di puntatore

```
int a = 8;  
int *b; //Puntatore a interi  
double *c; //Puntatore a double  
...  
b = &a;  
c = (double*) b;
```

Che succede se ora dereferenzio `c`?

Segmentation Fault!!! Perchè?

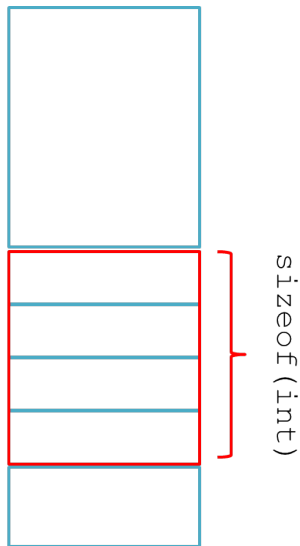


Violazione memoria con cast esplicito

```
b = &a;
```



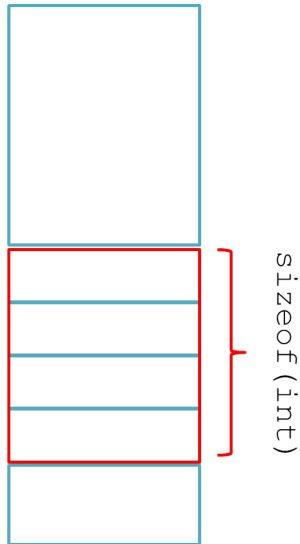
`&a`



Violazione memoria con cast esplicito

```
b = &a;  
c = (double*)b;
```

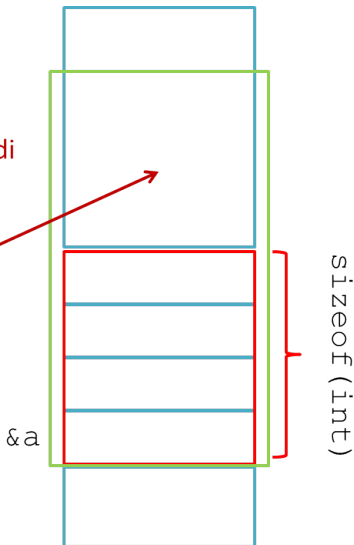
 &a



Violazione memoria con cast esplicito

`*c` va a dereferenziare un area di memoria `sizeof(double)` violando le regole di accesso

```
b = &a;  
c = (double*)b;  
if (*c > 0.1){  
    ...
```



Outline

- 1 Puntatori
 - Variabili puntatore
 - Aritmetica dei puntatori
- 2 Puntatori e funzioni
 - Passaggio per riferimento
- 3 Array Multidimensionali

Passaggio dei parametri - Valore

I parametri delle funzioni C sono **passati per valore**

- Il loro valore viene copiato sullo stack
- Ogni **modifica del parametro** nel corpo della funzione **non modifica l'originale**

Passaggio dei parametri - Valore

I parametri delle funzioni C sono **passati per valore**

- Il loro valore viene copiato sullo stack
- Ogni **modifica del parametro** nel corpo della funzione **non modifica l'originale**

```
void swap(int a, int b) {  
    int c = a;  
    a = b;  
    b = c;  
}  
void main() {  
    ...  
    swap(x, y);  
}
```

Passaggio dei parametri - Valore

I parametri delle funzioni C sono **passati per valore**

- Il loro valore viene copiato sullo stack
- Ogni **modifica del parametro** nel corpo della funzione **non modifica l'originale**

```
void swap(int a, int b) {  
    int c = a;  
    a = b;  
    b = c;  
}  
void main() {  
    ...  
    swap(x, y);  
}
```

swap non scambia il contenuto di x,y perchè lo **scambio viene fatto sulle copie!**

Passaggio per Riferimento

Possiamo sfruttare i puntatori per realizzare una versione funzionante di `swap` (**passaggio per riferimento**)

Passaggio per Riferimento

Possiamo sfruttare i puntatori per realizzare una versione funzionante di `swap` (**passaggio per riferimento**)

```
void swap(int *a, int *b) {  
    int c = *a;  
    *a = *b;  
    *b = c;  
}  
void main() {  
    ...  
    swap(&x,&y);  
}
```

Passaggio per Riferimento

Possiamo sfruttare i puntatori per realizzare una versione funzionante di `swap` (**passaggio per riferimento**)

```
void swap(int *a, int *b) {  
    int c = *a;  
    *a = *b;  
    *b = c;  
}  
void main() {  
    ...  
    swap(&x,&y);  
}
```

Nello stack viene **copiato l'indirizzo di memoria** di `x` e `y`.
Le modifiche eseguite da `swap` vengono fatte andando a **scrivere nell'indirizzo di memoria di `x` e `y`** e sono quindi **disponibili all'uscita della funzione**.

Passaggio per Riferimento - Array

Attenzione! Gli array sono sempre passati per riferimento

Passaggio per Riferimento - Array

Attenzione! Gli array sono sempre passati per riferimento

- Si passa il **nome dell'array**, che è un **puntatore al primo elemento**
- Ogni **modifica ad elementi dell'array** eseguita nel corpo di una funzione **modifica anche l'originale**

Passaggio per Riferimento - Array

Attenzione! Gli array sono sempre passati per riferimento

- Si passa il **nome dell'array**, che è un **puntatore al primo elemento**
- Ogni **modifica ad elementi dell'array** eseguita nel corpo di una funzione **modifica anche l'originale**

Le segnature

```
void arrayFun(int x[])
```

```
void arrayFun(int *x)
```

sono del tutto **equivalenti** (tipicamente si usa la prima per leggibilità)

Esempio

Problema: calcolare $\sum_{i=0}^{i=DIM-1} |array[i]|$.

```
int sumabs(int *a, int dim) {
    int i, somma;
    for (i=0; i<dim; i++)
        if (a[i]<0)
            a[i]=-a[i];
    for (i=0; i<dim; i++)
        somma=somma+a[i];
    return somma;
}
void main() {
    ...
    somma= sumabs(array, DIM);
}
```

Esempio

Problema: calcolare $\sum_{i=0}^{i=DIM-1} |array[i]|$.

```
int sumabs(int *a, int dim) {
    int i, somma;
    for (i=0; i<dim; i++)
        if (a[i]<0)
            a[i]=-a[i];
    for (i=0; i<dim; i++)
        somma=somma+a[i];
    return somma;
}
void main() {
    ...
    somma= sumabs(array, DIM);
    printf("Stampa di array: \n");
    for (i=0; i<DIM; i++)
        printf("%d ", array[i] ); /*Cosa stampa?*/
}
```

Outline

- 1 Puntatori
 - Variabili puntatore
 - Aritmetica dei puntatori
- 2 Puntatori e funzioni
 - Passaggio per riferimento
- 3 **Array Multidimensionali**

Array Multidimensionali

Sintassi:

tipo-elementi nome-array [*lung*₁]... [*lung*_{*n*}];

- `mat[3][4]`; matrice 3×4
- Per ogni dimensione *i* l'indice va da 0 a *lung*_{*i*}-1.

		colonne			
		0	1	2	3
righe	0	?	?	?	?
	1	?	?	?	?
	2	?	?	?	?

- Esempio `int marketing[10][5][12]`
(indici potrebbero rappresentare: prodotti, venditori, mesi dell'anno)

Accesso agli elementi di una matrice

		colonne			
		0	1	2	3
righe	0				
	1				
	2				

```
int i, mat[3][4];
```

```
....
```

```
i = mat[0][0];           elemento di riga 0 e colonna 0 (primo elemento)
```

```
mat[2][3] = 28;         elemento di riga 2 e colonna 3 (ultimo elemento)
```

```
mat[2][1] = mat[0][0] * mat[1][3];
```

Come per i vettori, l'unica operazione possibile sulle matrici è l'accesso agli elementi tramite l'operatore `[]`.

Esempio: Lettura e stampa di una matrice

```
#include <stdio.h>
#define RIG 2
#define COL 3
main() {
    int mat[RIG][COL];
    int i, j;
    /* lettura matrice */
    printf("Lettura matrice %d x %d;\n", RIG, COL);
    for (i = 0; i < RIG; i++)
        for (j = 0; j < COL; j++)
            scanf("%d", &mat[i][j]);
    printf("La matrice e':\n"); /* stampa matrice */
    for (i = 0; i < RIG; i++) {
        for (j = 0; j < COL; j++)
            printf("%d  ", mat[i][j]);
        printf("\n"); /* a capo dopo ogni riga */
    }
}
```

Passaggio di matrici come parametri I

Una funzione che opera su un **vettore** ha bisogno del puntatore (costante) all'elemento di indice **0**.

Non serve la dimensione del vettore nel parametro formale.

Passaggio di matrici come parametri I

Una funzione che opera su un **vettore** ha bisogno del puntatore (costante) all'elemento di indice **0**.

Non serve la dimensione del vettore nel parametro formale.

Una funzione che opera su una **matrice** ha invece bisogno anche del **numero di colonne della matrice nel parametro formale**.

Passaggio di matrici come parametri I

Una funzione che opera su un **vettore** ha bisogno del puntatore (costante) all'elemento di indice **0**.

Non serve la dimensione del vettore nel parametro formale.

Una funzione che opera su una **matrice** ha invece bisogno anche del **numero di colonne della matrice nel parametro formale**.

```
void stampa(int mat[][5], int righe)
```

Per accedere all'elemento, `mat[i][j]`, la funzione deve **calcolare** l'indirizzo di tale elemento.

Passaggio di matrici come parametri I

Una funzione che opera su un **vettore** ha bisogno del puntatore (costante) all'elemento di indice **0**.

Non serve la dimensione del vettore nel parametro formale.

Una funzione che opera su una **matrice** ha invece bisogno anche del **numero di colonne della matrice nel parametro formale**.

```
void stampa(int mat[][5], int righe)
```

Per accedere all'elemento, `mat[i][j]`, la funzione deve **calcolare** l'indirizzo di tale elemento.

L'indirizzo di `mat[i][j]` è infatti:

$$\text{mat} + (i \cdot C \cdot \text{sizeof}(\text{int})) + (j \cdot \text{sizeof}(\text{int}))$$

dove `C` è il numero di colonne (gli elementi in ciascuna riga).

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:

- valore di `mat`, ovvero l'indirizzo del primo elemento della matrice

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:

- valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga `i` dell'elemento

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento $\text{mat}[i][j]$ è necessario conoscere:

- valore di mat , ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga i dell'elemento
- l'indice di colonna j dell'elemento

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento $\text{mat}[i][j]$ è necessario conoscere:

- valore di mat , ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga i dell'elemento
- l'indice di colonna j dell'elemento
- il numero C di colonne della matrice

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento $\text{mat}[i][j]$ è necessario conoscere:

- valore di mat , ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga i dell'elemento
- l'indice di colonna j dell'elemento
- il numero C di colonne della matrice

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento $\text{mat}[i][j]$ è necessario conoscere:

- valore di mat , ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga i dell'elemento
- l'indice di colonna j dell'elemento
- il numero C di colonne della matrice

In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.

1. **vettore**: non serve specificare il numero di elementi

Passaggio di matrici come parametri II

Riassumendo:

per calcolare l'indirizzo dell'elemento `mat[i][j]` è necessario conoscere:

- valore di `mat`, ovvero l'indirizzo del primo elemento della matrice
- l'indice di riga `i` dell'elemento
- l'indice di colonna `j` dell'elemento
- il numero `C` di colonne della matrice

In generale, in un parametro di tipo array vanno specificate tutte le dimensioni, tranne eventualmente la prima.

1. **vettore**: non serve specificare il numero di elementi
2. **matrice**: bisogna specificare il numero di colonne, ma non serve il numero di righe