



## Puntatori

### Passaggio per riferimento Allocazione dinamica

Ing. Nadia Ranaldo  
Dipartimento di Ingegneria  
Università degli Studi del Sannio

## Puntatori (1)



- Fino ad ora abbiamo dichiarato variabili ed il compilatore alloca ad esse uno spazio di memoria opportuno
- Per accedere alle variabili utilizziamo gli identificatori
  - non interessa sapere dove sono in memoria
- I **puntatori** ci danno la possibilità di:
  - Conoscere dove sono memorizzate in memoria le variabili
  - Lavorare con gli indirizzi di memoria delle locazioni in cui sono memorizzate le variabili
    - Trovare l'indirizzo di una variabile dichiarata
    - Utilizzare gli indirizzi per scrivere/leggere il valore di una variabile

2

## Puntatori (2)



- Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile
- Sono utilizzati per:
  - Passaggio di parametri per indirizzo alle funzioni
  - Riferimento a grandi strutture dati in maniera compatta
  - Condivisione di dati tra più parti del programma
  - Riservare nuova memoria durante l'esecuzione del programma

3

## Dichiarazione di puntatori



### Sintassi

**type \*identifier;**

- Dichiarare una variabile puntatore di nome *identifier*, ad una variabile di tipo *type*
  - Es: `int *p;`
- Dichiarare una variabile puntatore, di nome *p*, ad una variabile di tipo *int*
- Dichiarazioni multiple: è sufficiente usare un *\** prima della dichiarazione di ogni variabile;
  - Es: `int *a, *b;`
- E' possibile dichiarare puntatori a qualsiasi tipo di dato
- E' possibile inizializzare i puntatori ad un indirizzo oppure a **NULL**
  - costante predefinita (in `stdio.h`) che denota il puntatore nullo (indirizzo nullo)

4

# Operatori unari "&" e "\*"



**&** (operatore indirizzo): applicato ad una variabile, restituisce l'indirizzo della variabile

Es: `int x = 5;`

`int *p;`

`p = &x; /* assegna a p l'indirizzo di x, "p punta ad x" */`

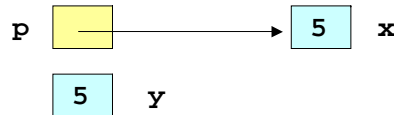


**\*** (operatore deferenziamento): applicato ad un puntatore, restituisce il contenuto della variabile da esso puntata

Es: `int x=5, y, *p;`

`p=&x`

`y = *p;`



# Esempi (1)



`int a = 50; /* una var intera */`

`int *b; /* una var puntatore a interi */`

`...`

`b = &a; /* assegna a b l'indirizzo della cella in cui è memorizzata a (b punta ad a) */`

a è memorizzata nella cella 250



b è memorizzata nella cella 444 (&b)

# Esempi (2)



`int a = 50; /* una var intera */`

`int *b; /* una var puntatore a interi */`

`...`

`b = &a; /* assegna a b l'indirizzo della cella in cui è memorizzata a (b punta ad a) */`



Dopo questo assegnamento in b è memorizzato l'indirizzo di a

# Esempi (3)



`int a = 50;`

`int *b = &a;`

`*b = *b + 5;`

Denota la variabile all'indirizzo contenuto in b

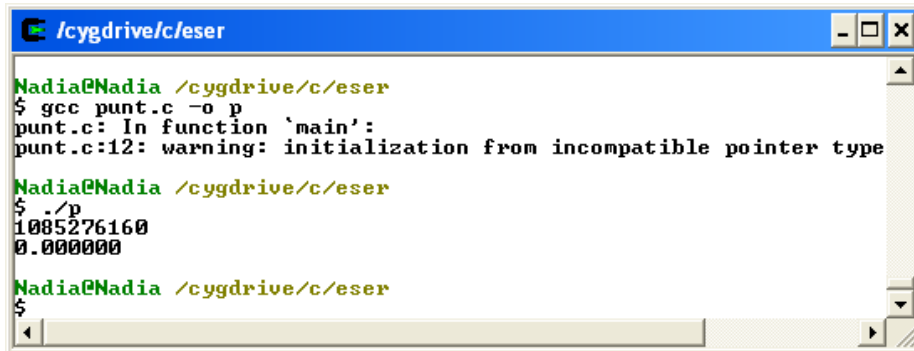


Dopo questo assegnamento in a è memorizzato il valore 50 + 5

## Esempi (4)



```
float a = 5.5;
int *b = &a; /* Warning: tipi diversi!
            (int*)(float *)
...
printf("%d\n", *b);
printf("%f\n", *b);
```



```

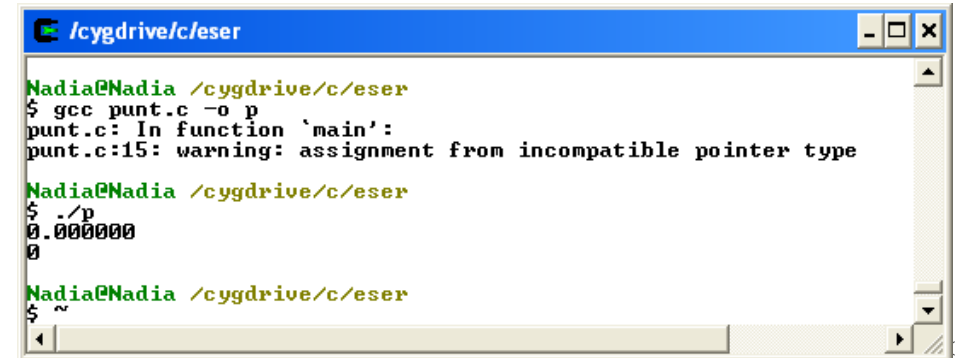
/cygdrive/c/eser
Nadia@Nadia /cygdrive/c/eser
$ gcc punt.c -o p
punt.c: In function 'main':
punt.c:12: warning: initialization from incompatible pointer type
Nadia@Nadia /cygdrive/c/eser
$ ./p
1085276160
0.000000
Nadia@Nadia /cygdrive/c/eser
$
```

9

## Esempi (5)



```
int y = 3;
float *t; /* Warning: tipi diversi!
          (float*)(int *)
t=&y;
printf("%f\n", *t);
printf("%d\n", *t);
```



```

/cygdrive/c/eser
Nadia@Nadia /cygdrive/c/eser
$ gcc punt.c -o p
punt.c: In function 'main':
punt.c:15: warning: assignment from incompatible pointer type
Nadia@Nadia /cygdrive/c/eser
$ ./p
0.000000
0
Nadia@Nadia /cygdrive/c/eser
$
```

10

## Aritmetica dei puntatori (1)



- È possibile scrivere espressioni puntatore usando operatori aritmetici binari di somma e sottrazione (+, -) e operatori aritmetici unari di incremento e decremento (--, ++)

- Un'espressione puntatore non può contenere la somma tra due puntatori
- Un'espressione contenente la differenza tra due puntatori è di tipo intera
- Ad esempio:

```
int x, *p, *p1;
...
p1 = p+p; /* Errore di sintassi! */
p1 = p+4; /* OK */
p1 = p--; /* OK */
x = p-p1; /* OK */
p = p-p1; /*Warning: i tipi non coincidono (punt/int)*/
```

- Gli operatori unari di incremento e decremento hanno priorità maggiore rispetto a tutti gli altri operatori
- Non è possibile utilizzare tutti gli altri operatori aritmetici
  - Errore di sintassi

11

## Aritmetica dei puntatori (2)



- Per la somma o la sottrazione tra un puntatore ed una o più costanti intere, il compilatore C calcola di quanti byte incrementare (decrementare) il puntatore in base al tipo del puntatore
- Ad esempio se p è un puntatore ad intero, incrementare p di uno significa assegnargli un valore in modo che punti alla successiva variabile intera, quindi l'incremento viene effettuato di 4 (e non di uno come accade per l'aritmetica tra numeri interi!)
  - Se p è un puntatore a double, incrementare p di due significa farlo puntare a due variabili successive di tipo double, quindi l'incremento viene effettuato di 16 (8\*2)
- In generale

```
type *p;
(p+a) => p + sizeof(type)*a
```
- La differenza tra due puntatori p1 e p2 è pari al numero di variabili del tipo dei due puntatori che possono essere memorizzate nella quantità di memoria calcolata come p1-p2
  - Il risultato può essere sia positivo che negativo
  - (p1-p2)/sizeof(type)
  - Se il tipo dei due puntatori non è lo stesso si verifica un errore di sintassi
  - E' possibile eseguire un cast

12

## Aritmetica dei puntatori (3)



```
int v[3],*p=&v[0];  
p = p+1;
```

IND	v[0]
IND+4	v[1]
IND+8	v[2]
IND+12	.....

p contiene l'indirizzo IND

IND	v[0]
IND+4	v[1]
IND+8	v[2]
IND+12	.....

p contiene l'indirizzo IND + 4

13

## Aritmetica dei puntatori (4)



```
int v[3],*p=&v[0];  
p = p+1;  
p--;
```

IND	v[0]
IND+4	v[1]
IND+8	v[2]
IND+12	.....

p contiene nuovamente l'indirizzo IND

14

## Aritmetica dei puntatori (5)



```
int v[3],*p=&v[0];  
p = p+1;  
p--;  
p+=3;
```

IND	v[0]
IND+4	v[1]
IND+8	v[2]
IND+12	.....

p contiene l'indirizzo IND + 12

15

## Aritmetica dei puntatori (6)



```
int v[3],*p=&v[0],*q;  
p = p+1;  
p--;  
q = p;  
p+=3;  
v[0] = p-q;
```

IND	v[0]
IND+4	v[1]
IND+8	v[2]
IND+12	.....

v[0] contiene 3, numero di interi memorizzabili fra p e q

16

## Altri esempi



```
int v[10], *p, x, y;
x = 1;
p = &x;      /* p punta a x */
y = *p;     /* y vale 1 */

*p = 0;     /* x viene posto a 0 */

p = &v[0];   /* p punta a v[0] */
p = &v[1];   /* p punta a v[1] */

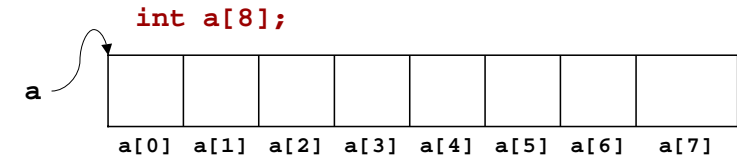
*p = *p + 10; /*si somma 10 al contenuto di ciò che è
              puntato da p */
(*p)++;      /* incremento di 1
              del contenuto di ciò che è puntato da p */
x=*p++;     /* x vale il contenuto puntato da p
              poi viene incrementato*/
x=***p;     /* x vale il contenuto puntato da (p più uno) */
```

17

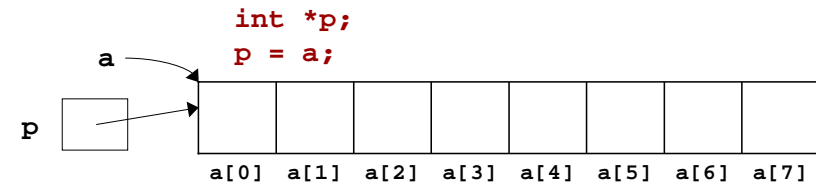
## Puntatori ed Array (1)



- Il nome di un vettore è un puntatore costante e rappresenta l'indirizzo della prima cella di memoria del vettore stesso



- Tale indirizzo può essere assegnato ad una variabile puntatore, poiché essa può contenere un indirizzo



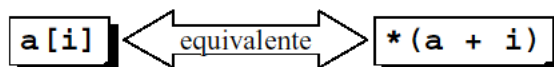
18

## Relazione tra puntatori ed array



- Nomi di array e puntatori sono in pratica equivalenti a parte che:
  - un array rappresenta un indirizzo fissato (a differenza dei puntatori che possono cambiare)
  - con un array si alloca anche della memoria, mentre con i puntatori no!

```
int a[10];
int i;
```



19

## Array e puntatori: perché equivalenti?

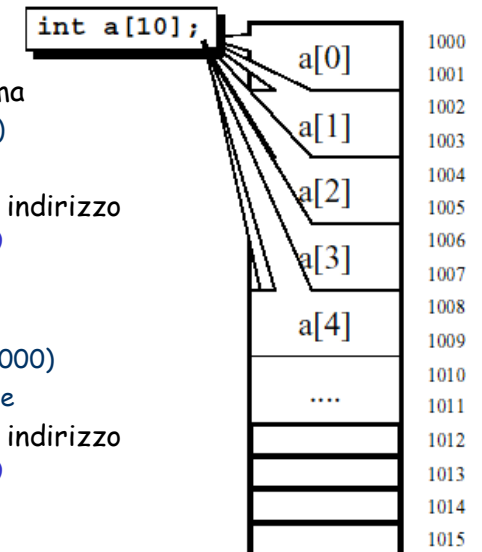


a[i]

- Per accedere ad a[i] si somma
  - indirizzo dell'array a (1000)
  - i \* sizeof(int)
- accedendo alla variabile con indirizzo  $1000 + i * \text{sizeof}(\text{int})$

\*(a+i)

- Per accedere ad \*(a+i)
  - indirizzo del puntatore a (1000)
  - spostato di i posizioni intere
- accedendo alla variabile con indirizzo  $1000 + i * \text{sizeof}(\text{int})$



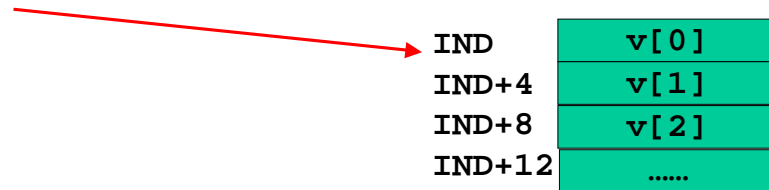
20

## Puntatori ed Array (2)



### Esempio

```
int v[3], *p=&v[0], *q;
q = v;
```



q contiene l'indirizzo IND  
v == IND

21

## Puntatori ed Array (3)

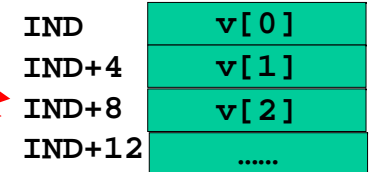


- L'operatore delle parentesi quadre [ ] è una abbreviazione dell'aritmetica dei puntatori

```
int v[3], *p=&v[0], *q, tmp;
/* le due istruzioni che seguono sono
equivalenti */
```

```
tmp = v[2];
```

```
tmp = *(v+2);
```



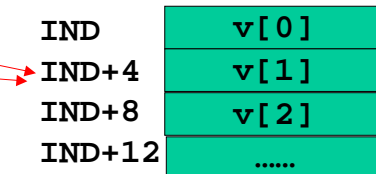
22

## Puntatori ed Array (4)



- L'operatore delle parentesi quadre [ ] può essere usato con una qualsiasi variabile puntatore

```
int v[3], *p=v, *q, tmp;
tmp = v[1];
tmp = p[1];
```



23

## Puntatori ed Array (5)



- I seguenti frammenti di codice sono equivalenti:

```
int v[N], *p=v;
int somma = 0;
```

```
/* versione 1 */
for(i=0; i<N; i++)
    somma+= v[i];
```

```
/* versione 2 */
for(i=0; i<N; i++)
    somma+= *(v+i);
```

24

## Puntatori ed Array (6)



- I seguenti frammenti di codice sono equivalenti (segue):

```
int a[N], *p=&v[0];
int somma = 0;
```

```
/* versione 1 */
for(i=0; i<N; i++)
    somma+= v[i];
```

```
/* versione 3 */
for(p=v; p<(v+N); p++)
    somma+= *p;
```

```
/* si fa scorrere p dall'inizio alla fine
dell'array */
```

25

## Passaggio di parametri per riferimento (1)



- Tutti i parametri delle funzioni C sono passati per valore
  - Il loro valore viene copiato nello stack frame della funzione chiamata
  - Ogni modifica al parametro nel corpo della funzione non modifica l'originale nella funzione chiamante
- E' possibile realizzare passaggi per riferimento utilizzando i puntatori
  - I passaggi per riferimento permettono di modificare il valore di una variabile nell'ambiente del chiamante
    - Quando si passa un array ad una funzione si realizza il passaggio per riferimento poiché il nome di un array rappresenta un puntatore (indirizzo al primo elemento dell'array)

26

## Passaggio di parametri per riferimento (2)



- La funzione `func` deve essere invocata passando l'indirizzo di una variabile, non il suo nome (o un valore dato da una espressione)

```
main() {
    int a, b[2];
    func(&a);
    func (&b[0]);
    func(b+1);
    . . .
}
```

- Al termine della funzione `func` le modifiche fatte utilizzando i puntatori corrispondono a modifiche sulle variabili della funzione chiamante
- Il passaggio per riferimento consente di avere un altro modo per restituire il risultato di una funzione, oltre a quello mediante il parametro di ritorno

27

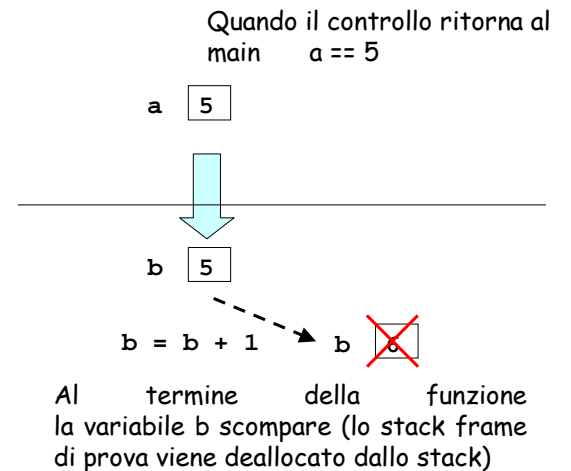
## Esempio di passaggio per valore



```
#include <stdio.h>
```

```
void main(void)
{
    int a;
    a = 5;
    prova(a);
    printf("%d\n", a);
}

void prova(int b)
{
    b = b + 1;
}
```



28

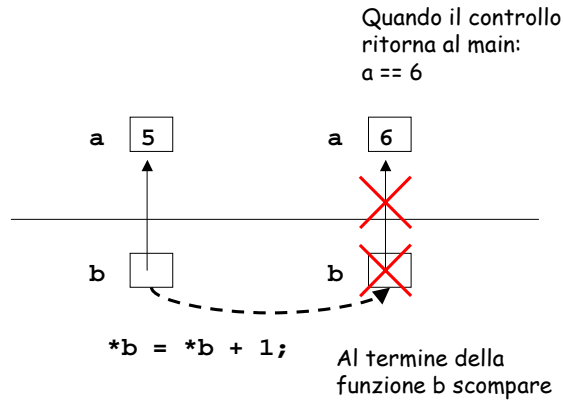
## Esempio di passaggio per riferimento



```
#include <stdio.h>

void main(void)
{
    int a;
    a = 5;
    prova2(&a);
    printf("%d\n", a);
}

void prova2(int *b)
{
    *b = *b + 1;
}
```



29

## La funzione scambia (1)



- Esempio: la funzione che scambia fra loro i valori di due variabili passate come parametro:
  - si potrebbe pensare di realizzarla nel seguente modo:
 

```
void scambia (int x, int y){
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```
  - e poi chiamare `scambia(a,b)`
  - non funziona! Perché lo scambio viene fatto sulle copie

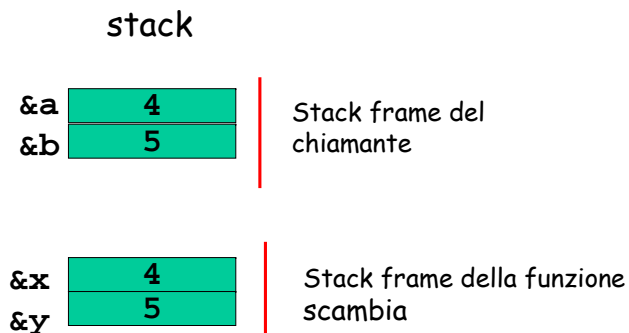
30

## La funzione scambia (2)



- Esempio di chiamata

```
int main() {
    int a=4, b=5;
    scambia (a,b);
    printf("%d,%d",a,b);
    ...
}
```



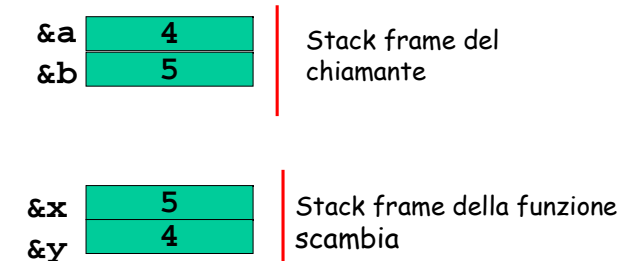
31

## La funzione scambia (3)



- Alla fine dell'esecuzione di scambia (prima di ritornare al chiamante)

```
int main() {
    int a=4, b=5;
    scambia (a,b);
    printf("%d,%d",a,b);
    ...
}
```



32

## La funzione scambia (4)



- Al momento di eseguire la printf() (lo stack frame della funzione scambia è stato deallocato dallo stack)

```
int main() {
    int a=4, b=5;
    scambia (a,b);
    printf("%d,%d",a,b);
    ...
}
```

&a	4	Stack frame del chiamante
&b	5	

33

## La funzione scambia (5)



- La versione corretta è ...

```
void scambia (int* x, int* y){
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

- con chiamata scambia(&a,&b)

34

## La funzione scambia (6)



- Esempio di chiamata

```
int main() {
    int a=4, b=5;
    scambia (&a,&b);
    printf("%d,%d",a,b);
    ...
}
```

stack

&a	4	Stack frame del chiamante
&b	5	

Ogni modifica a \*x modifica il valore di a nell'ambiente del chiamante

&x	&a	Stack frame della funzione scambia
&y	&b	

35

## La funzione scambia (7)



- Alla fine dell'esecuzione di scambia (prima di ritornare al chiamante)

```
int main() {
    int a=4, b=5;
    scambia (&a,&b);
    printf("%d,%d",a,b);
    ...
}
```

&a	5	Stack frame del chiamante
&b	4	

&x	&a	Stack frame della funzione scambia
&y	&b	

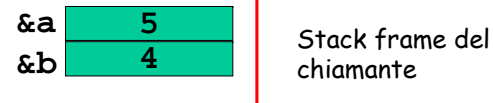
36

## La funzione scambia (8)



- Al momento di eseguire la `printf()` (lo stack frame della funzione `scambia` è stato deallocato dallo stack)

```
int main() {
    int a=4, b=5;
    scambia (&a,&b);
    printf("%d,%d",a,b);
    ...
}
```



37

## La funzione scambia (9)



```
void main(void){
    int a=3, b=7;
    scambia1(a,b); /* Al termine di scambia1 a [3] b [7] */

    scambia2(&a,&b); /* Al termine di scambia2 a [7] b [3] */
}

void scambia1(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

void scambia2(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

38

## Passaggio di parametri per riferimento (10)



- La funzione `scanf` utilizza il passaggio per riferimento `scanf("%d",&x);`
- Il valore letto da tastiera viene memorizzato nella variabile contenuta all'indirizzo passato come parametro
- ATTENZIONE:** gli array sono passati sempre per riferimento perchè quello che si passa è il nome dell'array

```
void inizializza (int x[]){
    x[0] = 11;
    ...
}
```

- con chiamata

```
int a[10];
inizializza(a);
/* qui a[0] vale 11 */
```

39

## Passaggio di parametri per riferimento (11)



- Inoltre le due scritture:

```
void inizializza (int x[]){
    x[0] = 13;
}
```

e

```
void inizializza (int* x){
    x[0] = 13;
}
```

- sono equivalenti!
- si preferisce usare la prima per leggibilità

40

## Esempi (1)



- Scrivere una funzione che legge da tastiera le coordinate cartesiane di un punto e le restituisce al chiamante
- P(x,y)
- Per scrivere una funzione che restituisce al chiamante una sola coordinata è possibile utilizzare il parametro di ritorno della funzione

```
float leggi_coordinata();
```

- Per scrivere una funzione che restituisce al chiamante due valori posso utilizzare il meccanismo del passaggio per riferimento

- i due valori vengono restituiti modificando direttamente il contenuto di due variabili del chiamante

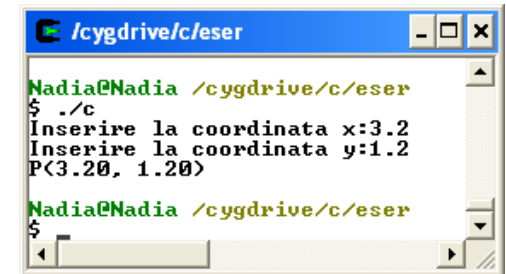
```
void leggi_coordinate(float *x, float *y);
```

41

## Esempi (2)



```
#include<stdio.h>
void leggi_coordinate(float *x,float *y);
main() {
    float x,y;
    leggi_coordinate(&x,&y);
    printf("P(%.2f, %.2f)\n",x,y);
}
void leggi_coordinate(float *x,float *y){
    printf("Inserire la coordinata x:");
    scanf("%f",x);
    printf("Inserire la coordinata y:");
    scanf("%f",y);
}
```



Notare la scanf

## Esempi (3)



- Scrivere una funzione che cerca un elemento in un array di interi

```
int * ricerca(int *a,int elem, int riemp);
```

- La funzione deve restituire l'indirizzo all'elemento trovato nell'array, NULL se l'elemento non è presente
- Utilizzare tale informazione per stampare a video la posizione in cui è stato trovato l'elemento

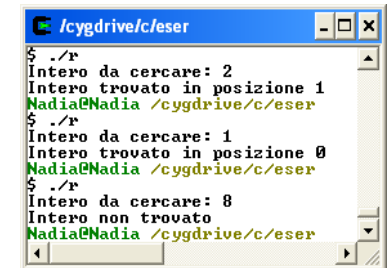
43

## Esempi (4)



```
#include<stdio.h>
int * ricerca(int *a, int elem, int n);
main() {
    int a[] = {1,2,3,4};
    int *p, elem;
    printf("Intero da cercare: ");
    scanf("%d",&elem);
    p=ricerca(a,elem, 4);
    if(p==NULL)
        printf("Intero non trovato");
    else
        printf("Intero trovato in posizione %d",(p-a));
}
int * ricerca(int *a, int elem, int n){
    int i;
    for (i=0; i<n; i++){
        if(*(a+i)==elem)
            return a+i;
    }
    return NULL; }

```



44

## Esempi (5)



- Riscrivere la seguente funzione utilizzando i puntatori e l'aritmetica dei puntatori in sostituzione degli array

```
void mystery(int a[], int n){
    int i, tmp1, tmp2;

    tmp1 = a[n-1];

    for (i=0; i<n-1; i=i+2){
        tmp2 = a[i];
        a[i] = tmp1;
        tmp1 = a[i+1];
        a[i+1] = tmp2;
    }
}
```

- Capire cosa fa la funzione

45

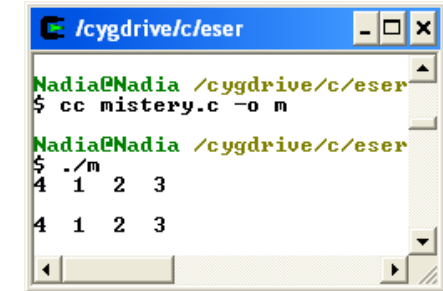
## Esempi (6)



```
#include<stdio.h>
void mystery(int a[], int n);
void mystery2(int *a, int n);
void stampa(int *a, int r);
```

```
main() {
    int a[] = {1,2,3,4};
    int b[] = {1,2,3,4};
    mystery(a,4);
    stampa(a,4);
    mystery2(b,4);
    stampa(b,4);
}

void stampa(int *a, int r){
    int i;
    for (i=0; i<r; i++)
        printf("%d ", *(a++));
    printf("\n");
}
```



```

/cygdrive/c/eser
Nadia@Nadia /cygdrive/c/eser
$ cc mystery.c -o m
Nadia@Nadia /cygdrive/c/eser
$ ./m
4 1 2 3
4 1 2 3

```

46

## Esempi (7)



```
void mystery(int a[], int n){
    int i, tmp1, tmp2;
    tmp1 = a[n-1];
    for (i=0; i<n-1; i=i+2){
        tmp2 = a[i];
        a[i] = tmp1;
        tmp1 = a[i+1];
        a[i+1] = tmp2;
    }
}

void mystery2(int *a, int n){
    int i, tmp1, tmp2;
    tmp1 = *(a+n-1);
    for (i=0; i<n-1; i=i+2){
        tmp2 = *(a+i);
        *(a+i) = tmp1;
        tmp1 = *(a+i+1);
        *(a+i+1) = tmp2;
    }
}
```

47

## Allocazioni statiche e dinamiche



- Allocare una struttura dati significa riservare spazio (byte) in memoria centrale
- Con l'allocazione statica, il compilatore, al momento della dichiarazione e fino alla fine dell'esecuzione della funzione in cui è dichiarata, riserva uno spazio in byte sufficiente a contenere le variabili allocate (scalari o vettoriali) di un determinato tipo

Es: `int a;` /\*si riservano byte necessari per un int \*/  
`float v[10];` /\* byte necessari per 10 float \*/

- Con l'allocazione dinamica è possibile:
  - Decidere il momento in cui allocare e deallocare la struttura dati
  - Decidere la quantità di memoria strettamente necessaria da riservare alla struttura dati
- Per le allocazioni dinamiche viene utilizzata l'area di memoria chiamata **heap**

48

## Allocazioni dinamiche



- Si possono ottenere mediante la funzione di libreria contenuta in `<stdlib.h>`
- Una prima funzione è `malloc`:  

```
void *malloc(n° byte della struttura dati da allocare);
```

  - `malloc` restituisce l'indirizzo di base della struttura dati allocata
  - Per ottenere la dimensione in byte della struttura dati da allocare si può utilizzare la funzione:  

$$n^{\circ} \text{ byte} = \text{sizeof}(\text{tipo di dato});$$
- Per liberare la memoria allocata si usa la funzione `free`:  

```
free(puntatore alla struttura dati);
```

49

## Array dinamici - malloc() (1)

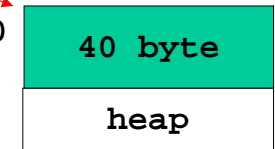


- Creare dinamicamente un array di 10 posizioni

```
int *a; /*conterrà il puntatore al
        primo elemento dell'array*/
a = (int *)malloc(10*sizeof(int));
```

Punta all'indirizzo iniziale della nuova area allocata

100  
...



50

## Array dinamici - malloc() (2)



```
int *a;
a = (int *)malloc(10*sizeof(int));
if(a==NULL)
    printf("fallimento!\n");
```

Se la funzione `malloc` non riesce ad allocare l'area di memoria richiesta restituisce `NULL`

heap

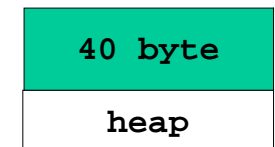
51

## Array dinamici - malloc() (3)



```
int *a;
a = (int *)malloc(10*sizeof(int));
if(a==NULL)
    printf("fallimento!\n");
else {
    a[4] = 345;
    ...
}
```

Si può accedere all'array con i consueti operatori (come se fosse statico)



52

## Array dinamici - malloc() (4)



- `malloc()` non inizializza la memoria a 0!
  - Possiamo inicializzarla esplicitamente oppure
  - usare `calloc()`
  - Restituisce un puntatore allo spazio per un array di un certo numero di elementi e di un certo tipo

```
int *a;
a = (int *)calloc(10,sizeof(int));
if(a==NULL)
    printf("fallimento!\n");
else {
    a[4] = 345;
    ...
}
```

53

## Array dinamici - realloc()



- `realloc()` modifica la lunghezza di un'area allocata precedentemente
- I contenuti restano invariati per uno spazio pari al minimo fra la nuova e la vecchia lunghezza
- Se la nuova lunghezza è maggiore della vecchia, il nuovo spazio non è inizializzato

```
int *a, *b; /*puntatori al primo elemento
            dell'array*/
a = (int *)malloc(10*sizeof(int));
...
b = (int *)realloc(a,20*sizeof(int));
/* adesso b punta ad un'array di 20 elementi */
```

54

## Array dinamici - realloc() (2)



- Meglio usare sempre due puntatori diversi (a,b)!
  - Altrimenti in caso di fallimento `NULL` sovrascrive il vecchio puntatore

55

## Array dinamici - free()



- Lo spazio allocato sullo heap non viene deallocato all'uscita delle funzioni
- La deallocazione deve essere richiesta esplicitamente usando `free()`

```
int *a;
a = (int *)malloc(10*sizeof(int));
...
free(a);
/* se a questo punto si accede nuovamente ad a
può succedere di tutto! */
```

56

## Tipo puntatore generico: void \* (1)



- Non si può dereferenziare
- E' necessario una conversione prima di manipolare la variabile puntata

- Es:

```
void *c;  
int a;  
c = &a;  
c++; /* incrementa sempre di 1 */  
*c = 5; /* errore a tempo di compilazione */  
*(int *)c = 5; /* OK */
```

57

## Tipo puntatore generico: void\* (2)



- Utilizzato per scrivere funzioni 'polimorfe'
- Per restituire o ricevere come parametri puntatori a tipi differenti

• Es:

- il prototipo della malloc() è  
`void *malloc (unsigned int size);`
- quando scrivo  
`int *a;`  
`a = (int *)malloc(10*sizeof(int));`
- viene effettuato un cast esplicito a `(int *)`

58

## Tipo puntatore generico: void\* (3)



- Prototipi delle altre funzioni di allocazione e deallocazione:

```
void * calloc (unsigned int size, unsigned  
int size);
```

```
void * realloc (void * ptr, unsigned int  
size);
```

```
void free (void * ptr);
```

59

## Passaggio di un array dinamico ad una funzione



- Si può ottenere passando alla funzione l'indirizzo della prima locazione di memoria;

Esempio:

```
void main(void){  
    int a[10]; /* Allocazione statica */  
    float *v;  
    ...  
    v = (float *)malloc(sizeof(float) * 20); /*Allocaz.  
                                                Dinamica */  
    funz(a, v);  
    ...  
}  
void funz(int *b, float *w) {  
    /* Usando "b" e "w" si accede rispettivamente alle  
    aree di memoria di "a" e "v" */  
}
```

60

## Esempio (1)



```
void inputarray (int *, int);
void printarray (int *, int);
main()
{int *a, dim;
  printf ("Che dimensione?");
  scanf ("%d",&dim);
  a = (int *) calloc ( dim, sizeof(int) );
  inputarray (a, dim);
  printarray (a, dim);
  free (a);
}
void inputarray(int *p, int size)
{ int i;
  for (i=0; i < size; i++) {
    printf ("a[%d]=", i);
    scanf ("%d", p++ );
  }
}
```

- Due funzioni che utilizzano l'aritmetica dei puntatori:
  - inputarray()
  - printarray()
 (è possibile anche utilizzare in alternativa funzioni che utilizzano la notazione con le parentesi quadre)

### Main

- Input della dimensione
- Allocazione:
  - cast a (int \*)
  - dim interi
- Chiamata di inputarray
  - scanf sul puntatore p
  - incremento postfisso di p

61

## Esempio (2)



```
void inputarray (int *, int);
void printarray (int *, int);
main()
{int *a, dim;
  printf ("Che dimensione?");
  scanf ("%d",&dim);
  a = (int *) calloc (dim, sizeof(int));
  inputarray (a, dim);
  printarray (a, dim);
  free (a);
}
.....
void printarray (int *p, int size)
{ int i;
  for (i=0; i < size; i++)
    printf ("a[%d]=%d\n", i, *(p++));
}
```

- Chiamata di printarray
- La stampa di ogni elemento
  - stampa l'intero puntato da p e poi incrementa il puntatore
- Viene liberata la memoria allocata

62

## Esercizio



- Individuare il tipo del lato destro e sinistro delle seguenti istruzioni di assegnamento e gli eventuali errori a tempo di compilazione o a tempo di esecuzione

```
double x, *px, a[5];
x = *px; /*1*/
*px = x; /*2*/
px = &x; /*3*/
&x = px; /*4*/
&(x+1) = x; /*5*/
&x + 1 = x; /*6*/
*(&(x+1)) = x; /*7*/
*(&(x)+1) = x; /*8*/
x = a; /*9*/
x = a[0]; /*10*/
x = *(a[1]); /*11*/
x = (*a)[2]; /*12*/
x = a[3+1]; /*13*/
x = a[3] +1; /*14*/
x = &((a[3])+1); /*15*/
x = &(a[3]) +1; /*16*/
x = *(&(a[3])+1); /*17*/
px = a; /*18*/
px = a[0]; /*19*/
px = &(a[4]); /*20*/
```

63

## Soluzione



```
double x, *px, a[5];
x = *px; /*1*/
• double / double. No Errori a tempo di compilazione
*px = x; /*2*/
• double / double. No Errori a tempo di compilazione
px = &x; /*3*/
• punt. a double /punt. a double. No Errori a tempo di compilazione
&x = px; /*4*/
• - /punt. a double. Errore a tempo di compilazione - &x non può essere assegnato
&(x+1) = x; /*5*/
• - /double. Errore a tempo di compilazione - non si può accedere all'indirizzo di una espressione!
&x + 1 = x; /*6*/
• - /double. Errore a tempo di compilazione - &x non può essere assegnato
```

64



```
*(amp(x+1)) = x; /*7*/
```

- - /double. Errore a tempo di compilazione - l'operatore & non può essere applicato a (x+1)

```
*(amp(x)+1) = x; /*8*/
```

- double /double. No errori a tempo di compilazione. Si accede alla variabile memorizzata dopo x sullo stack, possibile errore a tempo di esecuzione!

```
x = a; /*9*/
```

- double / punt. a double. Errore a tempo di compilazione. I tipi non corrispondono

```
x = a[0]; /*10*/
```

- double / double. No errori a tempo di compilazione

```
x = *(a[1]); /*11*/
```

- double / - . Errore a tempo di compilazione. a[1] non è un puntatore

```
x = (*a)[2]; /*12*/
```

- double / - . Errore a tempo di compilazione. \*a è un double

65



```
x = a[3+1]; /*13*/ x = a[3] +1; /*14*/
```

- double / double. No errori a tempo di compilazione

```
x = amp((a[3])+ 1); /*15*/
```

- double / -. Errore a tempo di compilazione - non si può calcolare l'indirizzo di una espressione

```
x = amp(a[3]) + 1; /*16*/
```

- double / -. Warning a tempo di compilazione - I tipi non corrispondono

```
x = amp(amp(a[3])+1); /*17*/
```

- double / double. No errori a tempo di compilazione - Corrisponde a a[4]

```
px = a; /*18*/
```

- punt. a double / punt. a double. No Errori a tempo di compilazione

```
px = a[0]; /*19*/
```

- punt. a double / double. Warning a tempo di compilazione - I tipi non corrispondono

```
px = amp(a[4]); /*20*/
```

- punt. a double / punt. a double. No Errori a tempo di compilazione

66



- Scrivere una funzione che restituisca un array dinamico inizializzato con valori letti da tastiera

### Input

nessuno

### Output

1. array mediante un puntatore int \*
2. capacità int

- Per restituire due parametri possiamo usare per uno (in genere si preferisce l'array) il valore di ritorno della funzione (return) e per l'altro (capacità) una variabile passata per riferimento (mediante l'indirizzo, quindi bisogna usare un puntatore)

```
int * creaInputArray(int *cap);
```

```
int[] creaInputArray(int *cap); /* errore sintattico */
```

- Il main per usare questa funzione deve dichiarare:

- Un puntatore ad intero per contenere il riferimento all'array dinamico
- Una variabile intera per contenere la capacità dell'array creato mediante la funzione

67



```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
int* creaInputArray(int *cap);
```

```
void stampa(int a[], int r);
```

```
main() {
```

```
    int *arr;
```

```
    int cap;
```

```
    arr = creaInputArray(&cap);
```

```
    printf("Capacità: %d\n",cap);
```

```
    stampa(arr, cap);
```

```
}
```

```
void stampa(int *a, int r){
```

```
    int i;
```

```
    for (i=0; i<r; i++) {
```

```
        printf("Elemento n.%d : %d\n", i, *(a+i));
```

```
    }
```

```
}
```

68



```
int* creaInputArray(int *cap) {
    int i = 0;
    int *a;
    printf("Quanti interi vuoi inserire?: ");
    scanf("%d", cap);
    a = (int *)malloc(sizeof(int)*(*cap));
    for(i=0; i<*cap; i++){
        printf("Inserisci l'elemento n. %d: ", i);
        scanf("%d", (a+i));
    }
    return a;
}
```

```
cygdrive/c/eser
Nadia@Nadia /cygdrive/c/eser
$ ./a
Quanti interi vuoi inserire?: 5
Inserisci l'elemento n. 0: 2
Inserisci l'elemento n. 1: 4
Inserisci l'elemento n. 2: 3
Inserisci l'elemento n. 3: 2
Inserisci l'elemento n. 4: 6
Capacita': 5
Elemento n.0 : 2
Elemento n.1 : 4
Elemento n.2 : 3
Elemento n.3 : 2
Elemento n.4 : 6
Nadia@Nadia /cygdrive/c/eser
$
```



- Scrivere una funzione che restituisce un array dinamico inizializzato con valori letti da tastiera e la sua capacità mediante due variabili passate per riferimento (senza utilizzare l'istruzione return per restituire l'array)
- **Input**  
nessuno
- **Output**  
array mediante un puntatore int \*  
capacità int
- Per restituire l'array e la capacità utilizziamo due variabili passate per riferimento (mediante l'indirizzo)

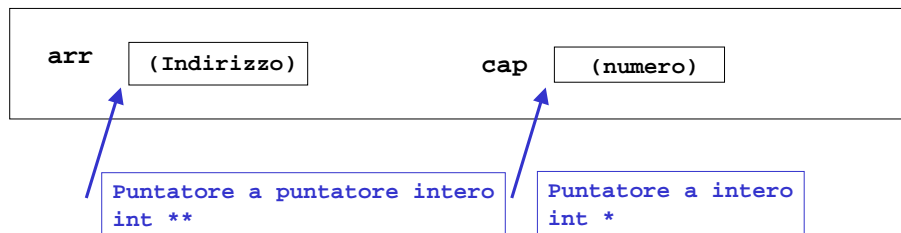
```
void creaInputArray(???????, int *cap);
```



- Per utilizzare tale funzione il main deve dichiarare le stesse variabili

- Un puntatore ad intero per contenere il riferimento all'array dinamico
- Una var. intera per contenere la capacità dell'array creato dalla funzione

main



- Per passare l'indirizzo del puntatore a intero (usato per l'array dinamico) occorre usare un puntatore di che tipo?

• **Puntatore a puntatore intero** indicato con  
`int **`  
`void creaInputArray(int ** a, int *cap);`



```
...
void creaInputArray(int **a, int *cap);
void stampa(int a[], int r);
main() {
    int cap, *arr;
    creaInputArray(&arr, &cap);
    printf("Capacita': %d\n", cap);
    stampa(arr, cap);
}
void creaInputArray(int **a, int *cap){
    int i = 0;
    int *arr;
    printf("Quanti interi vuoi inserire?: ");
    scanf("%d", cap);
    arr = (int *)malloc(sizeof(int)*(*cap));
    for(i=0; i<*cap; i++){
        printf("Inserisci l'elemento n. %d: ", i);
        scanf("%d", (arr+i));
    }
    *a=arr;
}
```



- Scrivere una funzione che riceve un array di interi e il suo riempimento e restituisce un array allocato dinamicamente contenente solo gli elementi pari
- Input  
array di interi `int a[]` es. [1 3 4 2 8 5]  
riempimento `riemp` es. 6
- Output  
array mediante un puntatore `int *` es. [4 2 8]  
capacità `int` es. 3
- Per restituire due parametri possiamo usare per uno il valore di ritorno della funzione (`return`) e per l'altro un parametro passato per riferimento (mediante l'indirizzo)

```
int * array_pari(int a[], int riemp, int * new_cap);  
int * array_pari(int *a,int riemp,int * new_cap);/*equivalente */  
int[] array_pari(int *a,int riemp,int * new_cap);/* ERRORE!!! */
```

73



```
int * array_pari(int a[], int riemp, int * new_cap);
```

Algoritmo:

- Per allocare un array dinamicamente occorre prima conoscerne la dimensione (uso della funzione `malloc` o `calloc`)
- Occorre quindi prima calcolare il numero di elementi pari contenuti nell'array `a`
  - Si scorre l'array `a` e si utilizza una variabile contatore `nPari` (inizializzata a zero) che viene incrementata ogni volta che si trova un numero pari
- Dopo aver allocato l'array `arrPari` si scorre nuovamente l'array `a` e ogni volta che si trova un numero pari lo si inserisce nell'array
  - Occorre un indice per scorrere l'array `a` (in lettura) e un altro per scorrere l'array `arrPari` (in scrittura) in modo da sapere in che punto fare l'inserimento
- Si aggiorna il valore di `new_cap` a `nPari` e si restituisce il puntatore `arrPari`

74



```
#include<stdio.h>  
#include<stdlib.h>  
  
int* array_pari(int *a, int riemp, int * new_cap);  
void stampa(int a[], int r);  
  
main() {  
    int a[10]={1,3,5,6,5,3,8,14,1,10};  
    int *pari;  
    int cap;  
    pari = array_pari(a,10,&cap);  
    printf("Capacita': %d\n",cap);  
    stampa(pari,cap);  
}  
  
void stampa(int a[], int r){  
    int i;  
    for (i=0; i<r; i++) {  
        printf("Elemento n.%d : %d\n", i, a[i]);  
    }  
}
```

75



```
int* array_pari(int *a, int riemp, int * new_cap){  
    int i,j = 0, nPari=0;  
    int *array_pari;  
    for(i=0; i<riemp; i++){  
        if (a[i] % 2 == 0)  
            (nPari)++;  
    }  
    array_pari = (int *)malloc(nPari*sizeof(int));  
    for(i=0; i<riemp; i++){  
        if (a[i] % 2 == 0){  
            array_pari[j]= a[i];  
            j++;  
        }  
    }  
    *new_cap=nPari;  
    return array_pari;  
}
```

76



- Riscrivere la funzione `array_pari` in modo da restituire sia l'array che il riempimento mediante due variabili passate per riferimento

- La funzione non deve restituire nulla (funzione void)

```
void array_pari(int a[], int riemp, int * new_cap, ???);
```

- L'array allocato dinamicamente è rappresentato dal puntatore al suo primo elemento (restituito dalla funzione `malloc`)
- Quindi occorre restituire un puntatore con il meccanismo di passaggio per riferimento, ovvero un puntatore ad un puntatore ad intero!

```
void array_pari(int a[], int riemp, int * new_cap,  
               int **arrPari);
```

77



```
#include<stdio.h>  
#include<stdlib.h>
```

```
void array_pari(int *a, int riemp, int * new_cap, int **arrPari);  
void stampa(int a[], int r);
```

```
main() {  
    int a[10]={1,3,5,6,5,3,8,14,1,10};  
    int *arrPari;  
    int cap;  
    array_pari(a,10,&cap, &arrPari);  
    printf("Capacita': %d\n",cap);  
    stampa(arrPari, cap);  
}
```

78



```
void array_pari(int *a, int riemp, int * new_cap, int** arrPari){  
    int i,j = 0, nPari=0;  
    int *arr;  
    for(i=0; i<riemp; i++){  
        if (a[i] % 2 == 0)  
            (nPari)++;  
    }  
    arr = (int *)malloc(nPari*sizeof(int));  
    for(i=0; i<riemp; i++){  
        if (a[i] % 2 == 0){  
            arr[j]= a[i];  
            j++;  
        }  
    }  
    *new_cap=nPari;  
    *arrPari=arr;  
}
```

79



1. Riscrivere la funzione

```
int* array_pari(int *a, int riemp, int * new_cap);  
utilizzando l'aritmetica dei puntatori
```

2. Scrivere una funzione che riceve un array di interi e il suo riempimento e restituisce due array allocati dinamicamente, uno contenente solo gli elementi pari e l'altro solo gli elementi dispari. Restituire inoltre la capacità dei due array. Utilizzare per tutte e quattro i valori di ritorno variabili passate per riferimento
3. Scrivere una funzione che ricevuto un array e il suo riempimento in ingresso, restituisca un array allocato dinamicamente e la sua capacità. Il nuovo array dovrà contenere solo i numeri positivi
4. Scrivere un insieme di funzioni su array dinamici di interi che effettuino:
  - la ricerca del minimo, del massimo
  - la somma degli elementi
  - l'ordinamentoper tutto o solo parte dell'array
  - Si utilizzi l'aritmetica dei puntatori

80

## Allocazione dinamica di un array bidimensionale (1)



- Per allocare dinamicamente un array bidimensionale (con uso delle `[][]`) occorre considerarlo come un insieme di array, in cui ogni array (monodimensionale) rappresenta una riga
- Ogni riga può essere allocata dinamicamente utilizzando la funzione `malloc` (o `calloc`) specificando la dimensione di ciascuna riga (ovvero il numero di colonne dell'array bidimensionale)
- I puntatori alle righe vengono memorizzati insieme in un array di puntatori (anch'esso allocato dinamicamente)
- Il puntatore all'array di puntatori (righe) rappresenta la matrice

Esempio di allocazione dinamica di una matrice 2x3

```
...
int **a; /* "a" è un puntatore ad un puntatore a intero */
int i;
a = (int **)malloc(sizeof(int *) * 2);
for (i = 0; i < 2; i++) {
    a[i] = (int *)malloc(sizeof(int) * 3);
} /* a rappresenta la matrice */
```

81

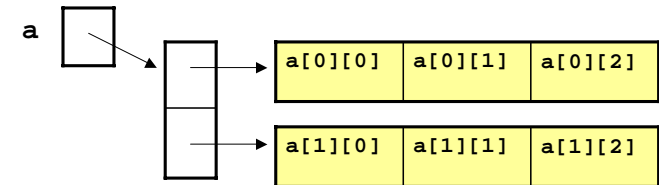
## Allocazione dinamica di un array bidimensionale (2)



- Su un array bidimensionale allocato dinamicamente è possibile continuare ad usare la notazione con le doppie parentesi quadre `[][]` per accedere agli elementi dell'array
- La riga *i*-esima è un array, e può essere acceduta con le parentesi quadre singole `[]` (ad esempio con `a[1]` si indica l'array corrispondente alla riga di indice 1)

`a[0][0] = ...`

`inputArray(a[1], 3);`

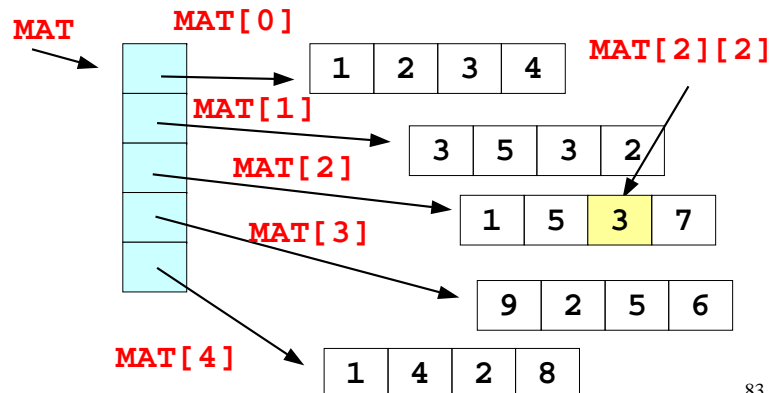


82

## Allocazione dinamica di un array bidimensionale (3)

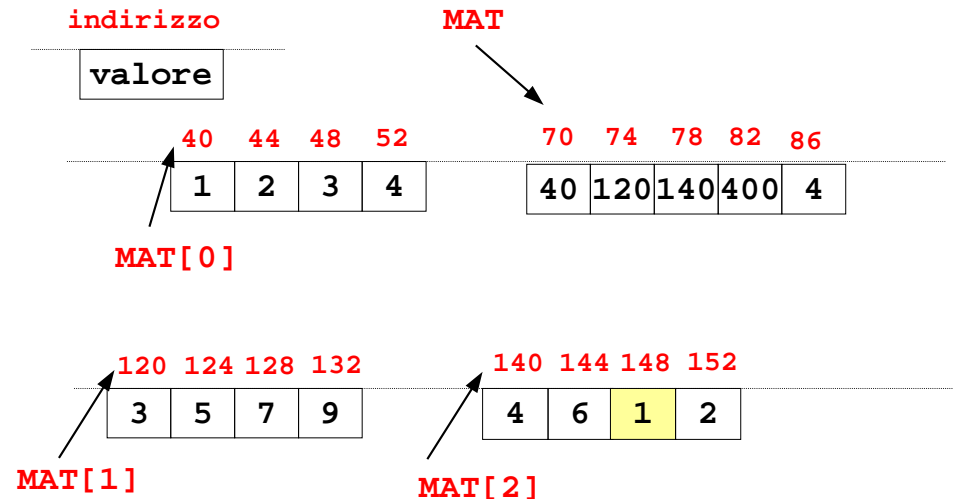


- Nel caso di array bidimensionale allocato staticamente, la memoria (nello stack frame della funzione) viene allocata in maniera contigua, in particolare la memorizzazione avviene per righe
- Nel caso di allocazione dinamica non è più utilizzata l'allocazione contigua per righe
  - Le righe sono memorizzate nell'heap in posizioni anche distanti tra loro (la posizione è determinata a tempo di esecuzione)



83

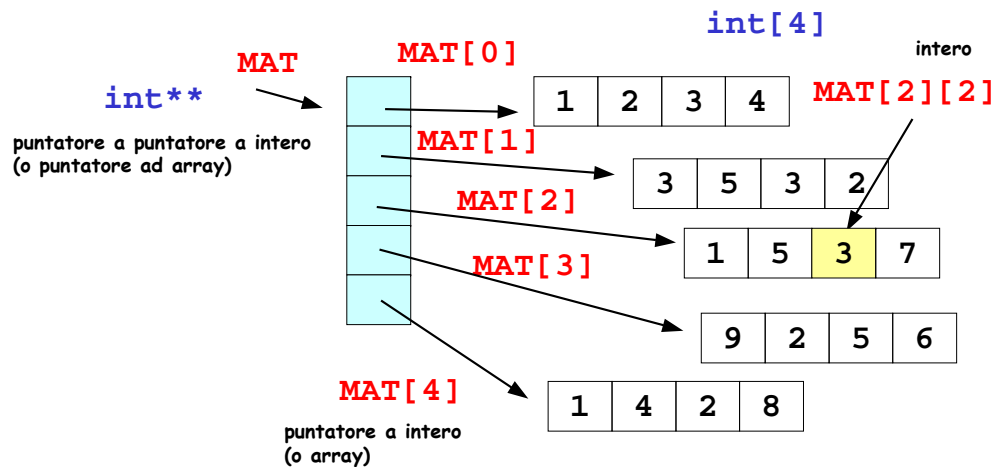
## Allocazione dinamica di un array bidimensionale (4)



84



### • Riassumendo i tipi ...



85



- Per passare un array bidimensionale statico ad una funzione è necessario specificare la seconda dimensione (in generale tutte le dimensioni successive alla prima)
- Solo in questo modo il compilatore è in grado di calcolare gli indirizzi per accedere agli elementi dell'array

```
#include<stdio.h>
void stampa (int mat[][2],int n,int m);
main(){
int mat[2][2];
int i,j,z=0;
for (i=0;i<2;i++)
for (j=0;j<2;j++)
mat[i][j] =z++;
stampa(mat,2,2);
}
void stampa(int mat[][2],int n,int m){
int i,j;
for (i=0; i<n;i++) {
for (j=0;j<m;j++)
printf("%d ",mat[i][j]);
printf("\n");
}
}
```

```
mat[i][j] =>
ind = mat+(i-1)*2+j
```

86



- Utilizzando un array bidimensionale allocato dinamicamente come un puntatore ad un array di puntatori, non è più necessario specificare la seconda dimensione
- E' quindi possibile scrivere funzioni più generiche, che non dipendono dal valore delle dimensioni successive alla prima

```
#include<stdio.h>
void stampa (int **mat,int n,int m);
int main() {
int **mat;
int i,j,z=0;
mat = (int **)malloc(sizeof(int *) * 2);
for (i = 0; i < 2; i++)
mat[i] = (int *)malloc(sizeof(int) * 2);
for (i=0;i<2;i++)
for (j=0;j<2;j++)
mat[i][j] =z++;
stampa(mat,2,2);
}
void stampa(int **mat,int n,int m){
int i,j;
for (i=0; i<n;i++) {
for (j=0;j<m;j++)
printf("%d ",mat[i][j]);
printf("\n");
}
}
```

```
mat[i][j] =>
ind = mat+i*2+j
```

87



- Perché funziona?

```
int sum_mat (int ** MAT,
int n, int m){
...
MAT[i][j] = ...;
}
*(*(MAT + i) + j)
```

88



- Funzione di allocazione di una matrice

```
int** mat_new(int m,int n){
    int i, ** mat;
    int errore = FALSE;
    mat = (int **)malloc(m*sizeof(int*));
    if (mat==NULL)
        return NULL;
    for(i=0;(i<m)&(!errore); i++){
        mat[i] = malloc(n*sizeof(int));
        if (mat[i] ==NULL)
            errore = TRUE;
    }
    if (errore) /* ...gestione errori */
    else
        return mat;
}
```



- Funzione di deallocazione

```
void mat_free(int ** mat, unsigned int m) {
    int i;
    /* dealloco tutte le righe */
    for(i=0;i<m; i++)
        free(mat[i]);
    /* dealloco il vettore dei puntatori alle righe
    */
    free(mat);
}
```