Allocazione dinamica della memoria: `malloc()`

In C è possibile allocare dinamicamente un'area di memoria (ossia, durante l'esecuzione di un programma) tramite le funzioni `malloc()` e `calloc()` (occorre includere il file `<stdlib.h>` che contiene i loro prototipi).

- La funzione `malloc()` richiede al sistema di allocare un'area di memoria della dimensione specificata come argomento.
  - Se la richiesta ha successo viene restituito l'indirizzo all'area di memoria allocata;
  - altrimenti viene restituito `NULL` (puntatore nullo il cui valore è 0).

  Per specificare la dimensione in modo corretto si usa l'operatore `sizeof`.

Esempio

```
double *p;
p = malloc(sizeof(double));
*p = 3.5;
```

Il valore di `p` è l'indirizzo a una locazione di memoria L di tipo double allocata dinamicamente. Il contenuto di L è 3.5.

Osservazioni

Attenzione all'*uso coerente dei tipi*.

L'istruzione

```
int *p;
p = malloc(sizeof(double));
```

può portare ad errori in esecuzione.

Infatti, avendo `p` tipo `int*`, il *valore* di `p` deve essere una locazione di memoria L di tipo `int` (in altri termini, la dimensione di L deve essere uguale a `sizeof(int)`).

L'istruzione corretta è:

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


Allocazione dinamica della memoria: `calloc()`

Se T è il nomed di un tipo, la chiamata

```
calloc(n, sizeof(T))
```

è equivalente a

```
malloc(n * sizeof(T))
```

Quindi, viene restituito (se possibile) l'indirizzo a una locazione di memoria L la cui dimensione è uguale n volte la dimensione richiesta dal tipo T.

In altri termini:

- L è un array creato dinamicamente di tipo T e dimensione n.

Esempio

```
int *p;
p = calloc(2,sizeof(int));
p[0] = 0;
p[1] = 1;
```

Il valore di `p` è l'indirizzo a un array di `int` allocato dinamicamente formato dagli elementi `p[0]` (valore 0) e `p[1]` (valore 1).

Analogamente, è sbagliato fare

```
int *p;
p = calloc(10, sizeof(double));  // array di double
```

La versione corretta è:

```
int *p;
p = calloc(10, sizeof(int));   // array di int
```

Nota

Il compilatore non rileva questo tipo di errori.

Infatti, l'indirizzo I restituito da `malloc()` e `calloc()` ha tipo `void*` (tipo *puntatore generico*), quindi I può essere assegnato a una variabile di un *qualunque* tipo `T*` senza segnalazione di warning.

Restituzione memoria allocata dinamicamente

Quando non è più necessario utilizzare un'area di memoria allocata dinamicamente, *occorre* restituirla al sistema mediante la funzione `free()`, a cui va passato come argomento l'indirizzo dell'area di memoria da liberare. Se si dimentica di rilasciare la memoria, si rischia di esaurire la memoria disponibile.

- In C non esiste alcun meccanismo automatico di *garbage collection*, ma la gestione della memoria allocata dinamicamente è a carico del programmatore, che deve preoccuparsi di gestire la memoria a "basso livello".
- Le richieste di allocazione/deallocazione della memoria avvengono in modo *impredicibile*, a differenza di quanto avviene per i record di attivazione di una funzione in cui le richieste di allocazione/deallocazione avvengonono in modalità "Last In First Out".

  Per gestire la memoria dinamica non si può usare lo stack di sistema, ma viene utilizzata un'apposita area di memoria chiamata heap.

Differenze fra stack e heap

- Tempo di vita

  Il tempo di vita di una variabile allocata nello stack è l'intervallo fra la allocazione e la deallocazione del record di attivazione della funzione in cui la variabile è definita (infatti, quando la funzione termina la variabile non serve più).

  Il tempo di vita di un'area di memoria nello heap è invece l'intervallo fra la chiamata a `malloc()` (o a `calloc()`) che alloca la memoria e la `free()` che la dealloca; può essere superiore al tempo di vita del record di attivazione della funzione che richiede l'allocazione.

- Dimensione memoria allocata

  Lo spazio richiesto dalle variabili nello stack è noto in tempo di compilazione

  Non è invece possibile prevedere in fase di compilazione l'utilizzo della memoria nello heap.

  Ad esempio, in

```
char *p;
scanf("%d", &n);
p = calloc(n, sizeof(char));
```

  la dimensione dell'array creato dinamicamente è nota solo durante l'esecuzione del programma.

Per quesi motivi la memoria dinamica non può essere gestita tramite lo stack.

Esercizio

Scrivere un programma che legge una sequenza di parole da standard input e le stampa su standard output in ordine inverso. La sequenza termina con enf-of-file (conviene quindi redirigere lo standard input da file).

Ad esempio, se lo standard input contiene

```
uva  pera  uva  ananas
```

deve essere stampato

```
ananas  uva  pera  uva
```

Si assume che la lunghezza massima di una parola sia `WORD` e che il numero massimo di parole sia `N`.

Soluzione 1

Si utilizza un procedimento ricorsivo simile a quello visto per stampare in ordine inverso una sequenza di interi letta da standard input.

Quindi:

- Se lo standard input non contiene parole, non si compie alcuna operazione.
- Altrimenti, si legge la prima parola w_1, *ricorsivamente* si stampano le rimanenti parole, quindi si stampa w_1.

Soluzione 2

Se non si vuole usare la ricorsione, occorre definire in `main()` un array `parole[]` per memorizzare le parole lette.

- Ogni elemento dell'array contiene l'indirizzo di una parola. Quindi il tipo dell'array è `char*`.
- Poiché le parole da leggere sono al massimo `N`, la dimensione dell'array è `N`.

```
#define N  100

int main(){
  char *parole[N];
  ...
}
```


- Per ogni $0 \leq k < N$, il *valore* di `parole[k]` è l'indirizzo della k-esima parola letta da standard input.
- Ogni parola letta da standard input va memorizzata in un array di `char` creato dinamicamente nello heap.

Per creare dinamicamente gli array che contengono le parole definiamo la funzione

```
char* newWord(char  *w)
```

La funzione `newWord()` crea dinamicamente un array di `char` contenente la parola `w` passata come parametro e restituisce l'indirizzo del nuovo array.

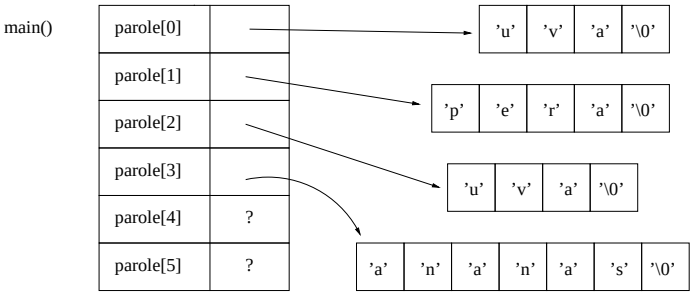
Per evitare spreco di memoria, l'array deve avere la *minima* dimensione richiesta per contenere `w`.

Esempio

Dopo la lettura di

```
uva  pera  uva  anana
```

si ha:



La funzione `newWord()`

Occorre fare i seguenti passi *nell'ordine indicato*.

(i) Si crea dinamicamente un array di `char` in grado di contenere la parola `w`. L'indirizzo del nuovo array viene memorizzato nella variabile `new` di tipo `char*`

```
char *new;
new = calloc(strlen(w)+1, sizeof(char));
```

La funzione `strlen()` della libreria standard `<string.h>` restituisce il numero di caratteri della parola passata come parametro (escludendo il carattere di fine stringa `'\0'`).

Quindi, se `w` è una parola di n caratteri, la chiamata

```
calloc(strlen(w)+1, sizeof(char))
```

restituisce l'indirizzo a un array di `char` di $n+1$ elementi.

(ii) Si copia la parola `w` nell'array `new`. Si può usare la funzione la funzione `strcpy()` della libreria standard `<string.h>`.

```
strcpy(new, w);   // copia w in new
```

(iii) La funzione `newWord()` può ora restituire l'indirizzo della nuova parola, che è `new`.

Codice di `newWord()`

```
/*  Restituisce un puntatore a un array creato dinamicamente
    contente la parola w   */

char *newWord(char *w){
  char *new;
  new = calloc(strlen(w)+1, sizeof(char));
  strcpy(new,w);
  return new;
}
```


Struttura di `main()`

In `main()` vanno definiti:

- L'array `parole[]` che contiene *tutte* le parole lette da standard input.
- L'array `word[]` da utilizzare per poter leggere da standard input *una* parola.

```
#define N 100  // numero massimo  di parole
#define WORD  30  // lunghezza massima di una parola

char *newWord(char *w){
   ...
}

int main(){
   ...
}
```



```
int main(){
  char *parole[N];
  char word[WORD+1];
  int k,i;

 /***  LETTURA  ***/

/* All'inizio di ogni iterazione, sono state lette k parole */

 for(k=0 ;  k<N  &&  scanf("%s", word) == 1 ; k++)
    parole[k] = newWord(word);

/* sono state lette k parole */

 /***  STAMPA  ***/

  for(i=k-1; i >= 0 ; i--)
    printf("%s\n", parole[i]);
  return 0;
}
```

All'inizio di ogni iterazione del ciclo

```
for(k=0;  k<N  &&  scanf("%s", word) == 1 ; k++)
    parole[k] = newWord(word);
```

il valore di k è il numero di parole già lette (quindi il valore iniziale deve essere 0).

- Se $k = \texttt{N}$ il ciclo deve terminare (sono già state lette `N` parole).
- Altrimenti:

  La chiamata

  ```
  scanf("%s", word)
  ```

  legge da standard input una parola nell'array `word[]`.

  - Se la lettura ha successo, `scanf()` restituisce 1, quindi l'espressione
    ```
    k<N  &&  scanf("%s", word) == 1
    ```
    vale 1 e viene eseguita l'istruzione nel ciclo.
  - Altrimenti, il ciclo termina.

Al termine del ciclo sono state lette k parole.

L'ultima parola letta è `parole[k-1]`.

Per stamparle in ordine inverso:

```
for(i=k-1; i >= 0 ; i--)
  printf("%s\n", parole[i]);
```

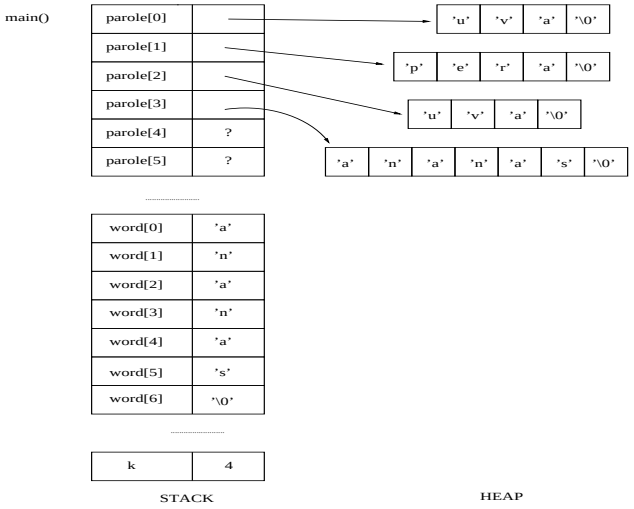
Esempio di esecuzione

Supponiamo che la sequenza in input sia

`uva  pera  uva  ananas  mela`

e che siano già state lette le prime quattro parole.

In memoria si ha:



(1) Viene valutata l'espressione

```
k<N  &&  scanf("%s", word) == 1
```

che controlla l'uscita dal ciclo.

Essendo $k < \texttt{N}$, viene eseguita la chiamata

```
scanf("%s", word);
```

che pone nell'array `word[]` la prossima parola su standard input, che è `"mela"`.

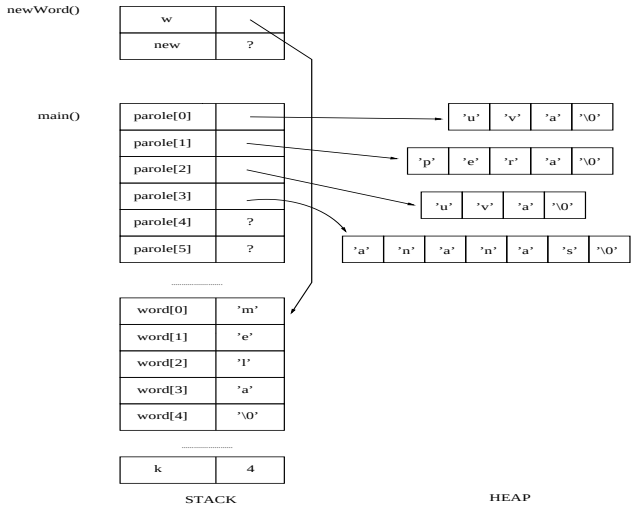
Poiché `scanf()` restituisce 1, vengono eseguite le istruzioni nel ciclo.

(2) Viene eseguita la chiamata

```
newWord(word)
```

Il record di attivazione della funzione `newWord()` è posto sullo stack.

Il valore iniziale del parametro `w` è l'indirizzo dell'array `word`.



(3) Inizia l'esecuzione di `newWord()`.

Poiché `w` è la stringa `"mela"`, eseguendo

```
new = calloc(strlen(w)+1, sizeof(char));
```

viene creato nello heap un array di `char` di 5 elementi e il suo indirizzo è assegnato alla variabile `new`.

Si noti che i 5 elementi dell'array `new` hanno indirizzo

`new` `new+1` `new+2` `new+3` `new+4`

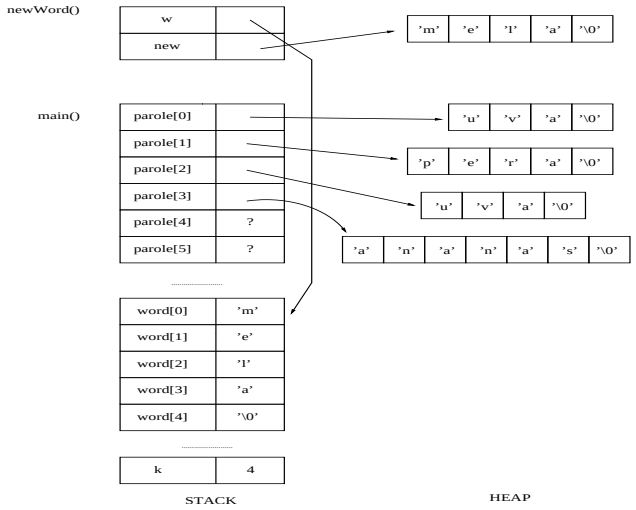
e le corispondenti locazioni dell'array possono essere denotate con

`new[0]` `new[1]` `new[2]` `new[3]` `new[4]`

Dopo l'istruzione

```
strcpy(new, w);
```

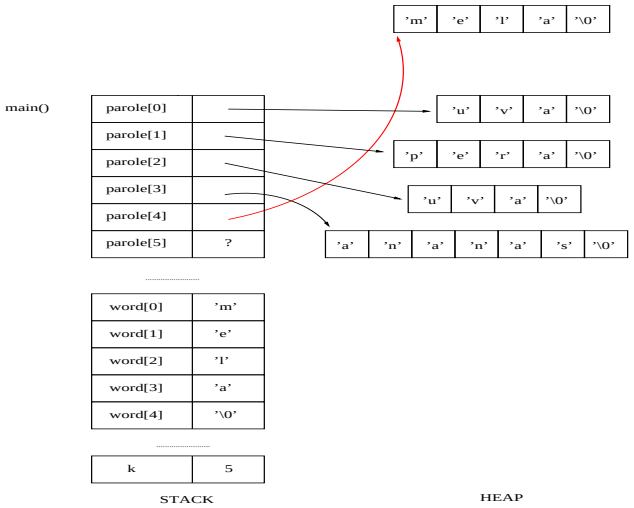
in `new` viene copiata la parola `"mela"`.



(4) L'esecuzione di `newWord()` termina e il valore restituito dalla funzione (ossia, l'indirizzo dell'array contenente `"mela"`) è assegnato a `parole[4]`.

Il record di attivazione di `newWord()` è tolto dallo stack.

L'array contenente `"mela"` rimane invece allocato nello heap.
Per eliminarlo, occorre chiamare la funzione `free()` passando come argomento il suo indirizzo.



(5) Viene di nuovo valutata l'espressione

```
k<N  &&  scanf("%s", word) == 1
```

che controlla il ciclo.

L'espressione `k<N` è verificata, ma la chiamata a `scanf()` restituisce `EOF` (infatti, la sequenza di input è terminata, quindi il prossimo carattere letto è end-of-file).

Essendo `EOF` diverso da 1, il ciclo termina.

Il valore di k è 5.
Sono state lette 5 parole e l'ultima parola letta è `parole[4]`.

(6) Il ciclo

```
for(i=k-1 ; i >= 0 ; i--)
   printf("%s\n", parole[i]);
```

stampa le parole in ordine inverso di lettura.

Eliminazione di elementi dallo heap

Si vuole sostituire la prima parola nell'array `parole[]` con la parola `"noce"`.

Facendo

```
parole[0] = newWord("noce");
for(i=0 ; i < k  ; i++)
    printf("%s\n", parole[i]);
```

viene stampato

```
noce
pera
uva
ananas
mela
```


Apparentemente, il codice è corretto.

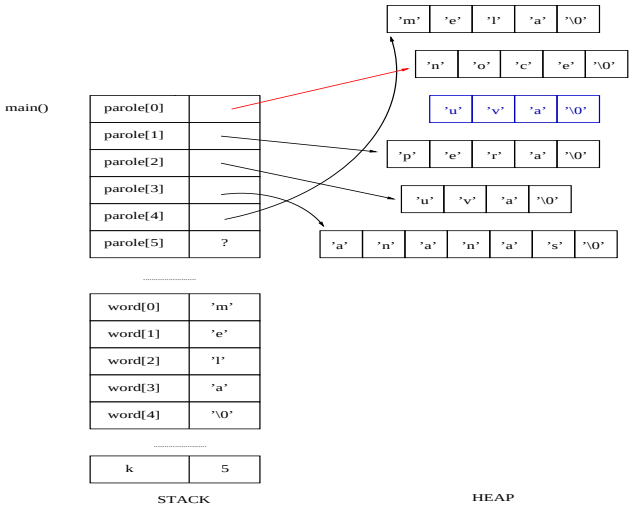
Tuttavia, con l'istruzione

```
parole[0] = newWord("noce");
```

il vecchio valore di `parole[0]` (ossia, l'indirizzo dell'array A contenete la parola `"uva"`) è *perso* (vedi figura nel prossimo lucido).

Questo significa che la memoria nello heap occupata da A non può più essere utilizzata da successive richieste di allocazione di memoria.

Può quindi capitare che il programma si blocchi per esaurimento della memoria disponibile nello heap.



Uso di `free()`

Quando un'area di memoria allocata dinamicamente non serve più, va *esplicitamente* liberata mediante una chiamata a `free()`.

Occorre quindi fare:

```
free(parole[0]);
parole[0] = newWord("noce");
for(i=0 ; i < k  ; i++)
    printf("%s\n", parole[i]);
```

Nota

Controllare che il parametro passato a `free()` sia effettivamente l'indirizzo di un'area di memoria allocata dinamicamente.

In

```
free(parole[0]);
free(parole[0]);
```

la seconda chiamata a `free()` dà errore in esecuzione, in quanto `parole[0]` non contiene più l'indirizzo di una locazione allocata dinamicamente.

Esercizi

1. Scrivere una funzione `swap()` che permette di scambiare le parole contenute in `parole[i]` e `parole[j]`.

2. Facendo riferimento alla configurazione di memoria al termine dell'esempio precedente, dire qual è il significato delle seguenti espressioni.

`parole+1` `*(parole+1)` `(*(parole+1))+3`

`*((*(parole+1))+3)` `*((*(parole+3))+1)`

Rappresentazione di matrici

Una matrice è un array bidimensionale.

Ad esempio, la dichiarazione

`int a[4][3];`

dichiara una matrice avente 4 righe e 3 colonne.

La matrice è composta da 4×3 locazioni di tipo `int` che possono essere denotate dalle espressioni:

```
    a[0][0]        a[0][1]       a[0][2]    //  <---  riga 0

    a[1][0]        a[1][1]       a[1][2]    //  <---  riga 1

    a[2][0]        a[2][1]       a[2][2]    //  <---  riga 2

    a[4][0]        a[4][1]       a[4][2]    //  <---  riga 3


//  colonna        colonna       colonna
//     0              1             2
```

Mappa di memorizzazione

L'aritmetica dei puntatori per array multidimensionali (chiamata anche mappa di memorizzazione) è piuttosto complicata e scomoda da utilizzare.

Veder i dettagli sul libro (par. 6.12) o su un manuale.

Nell'esempio precedente:

- L'indirizzo base della matrice è `&a[0][0]`
- L'espressione `a[r]` ($r = 0, 1, 2, 3$) denota tutta la riga r dell'array. Quindi, `a[r]` è il nome di una locazione di memoria di dimensione $3 \times$ `sizeof(int)`.

Per rappresentare correttamente la mappa di memorizzazione, l'espressione `a[r][c]` deve essere uguale a

$$*(\&a[0][0] \; + \; r \times 3 \; + \; c)$$

La mappa di memorizzazione richiede quindi la conoscenza del numero di colonne della matrice (per passare da una riga all'altra occorre saltare 3 locazioni di tipo `int`).

Questo significa che, se si vuole passare la matrice a come parametro a una funzione, il numero di colonne va *espressamente* indicato nella definizione del parametro.

Mappa di memorizzazione della matrice

Sia $l = \&a[0][0]$ (indirizzo base della matrice). Allora:

Locazione	Indirizzo
$a[0][0]$	l
$a[0][1]$	$l + 1$
$a[0][2]$	$l + 2$
$a[1][0]$	$l + 3 = l + 1 \times 3$
$a[1][1]$	$l + 4 = l + 1 \times 3 + 1$
$a[1][2]$	$l + 5 = l + 1 \times 3 + 2$
$a[2][0]$	$l + 6 = l + 2 \times 3$
$a[2][1]$	$l + 7 = l + 2 \times 3 + 1$
$a[2][2]$	$l + 8 = l + 2 \times 3 + 1$
$a[3][0]$	$l + 9 = l + 3 \times 3$
$a[3][1]$	$l + 10 = l + 3 \times 3 + 1$
$a[3][2]$	$l + 11 = l + 3 \times 3 + 2$

Esempio

Per stampare la matrice a si può usare la funzione

```
/* Stampa una matrice di row righe e 3 colonne */

void  print3(int m[][3], int row){
  int r,c; // r: indice per le righe
           // c: indice per le colonne
  for(r=0 ; r < row; r++){
    for(c=0 ; c < 3 ; c++)
      printf("%d ", m[r][c]);
    putchar('\n'); // fine riga r
  }
}
```

Occorre fare la chiamata

```
print3(a, 4);
```

Nota

La funzione `print3()` può essere usata solo per stampare matrici con 3 colonne (e un numero arbitrario di righe).

Matrici allocate dinamicamnete

La soluzione precedente è piuttosto scomoda e in genere non viene usata.

È preferibile utilizzare matrici create *dinamicamente*.

Esempio

La matrice a di 4 righe e 3 colonne, è rappresentabile mediante un vettore a di tre elementi, `a[0]`,... `a[3]` allocato dinamicamente.

Per $0 \leq r < 4$, il valore di `a[r]` è l'indirizzo a un vettore di `int` di dimensione 3 che rappresenta la riga r della matrice.

Quindi

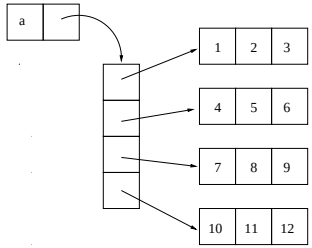
- Il tipo di `a[r]` è `int*`.
- Il tipo di `a` è `int**`.

Esempio

Sia A la matrice

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

La rappresentazione di A mediante array allocati dinamicamente è:




```
 int **a, r;

// CREAZIONE MATRICE 4 X 3

a = calloc(4, sizeof(int*));
for(r=0 ; r < 4 ; r++)
  a[r] = calloc(3, sizeof(int));

// INIZIALIZZAZIONE

a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
...
```


Rispetto a prima, per costruire la mappa di memorizzazione corretta non è richiesta la conoscenza del numero di colonne della matrice.

Infatti:

- L'indirizzo della riga 0 (ossia, del vettore di `int` che rappresenta la prima riga) è `*a`.
- L'indirizzo della riga 1 è `*(a+1)` (quindi non dipende dal numero di colonne della matrice).
- L'indirizzo della riga 2 è `*(a+2)`
- L'indirizzo della riga 3 è `*(a+3)`

Di conseguenza, quando un parametro p di una funzione rappresenta una matrice allocata dinamicamente, non è richiesto che la definizione di p contenga il numero di colonne della matrice.

Esempio

Per stampare una qualunque matrice creata dinamicamente si può usare la funzione

```
 /* Stampa una matrice dinamica di row righe e col colonne */

void  print(int** m, int row, int col){
  int r,c;
  for(r=0 ; r < row; r++){
    for(c=0 ; c < col ; c++)
      printf("%d ", m[r][c]);
    putchar('\n'); // fine riga r
  }
}
```

Per stampare la matrice a di prima occorre fare la chiamata

```
 print(a, 4, 3);
```