Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Gestione dinamica della memoria

Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

A pessimistic programmer sees the array as half empty.
An optimistic programmer sees the array as half full.
A Real Programmer sees the array as twice as big as it needs to be and calls realloc().

Introduzione

- Parliamo di **allocazione della memoria** quando un blocco di memoria (RAM) viene assegnata ad un programma in esecuzione.
- In C abbiamo tre classi di allocazione delle memoria:
  - **Allocazione automatica**. Riguarda le variabili (non `static`) con scope locale. La memoria viene *automaticamente allocata* sullo **stack** (*record di attivazione*) e *automaticamente deallocata* (rilasciata) al termine dell'esecuzione della funzione in cui la variabile è dichiarata.

Introduzione

- Parliamo di **allocazione della memoria** quando un blocco di memoria (RAM) viene assegnata ad un programma in esecuzione.
- In C abbiamo tre classi di allocazione delle memoria:
  - **Allocazione automatica**. Riguarda le variabili (non `static`) con scope locale. La memoria viene *automaticamente allocata* sullo **stack** (*record di attivazione*) e *automaticamente deallocata* (rilasciata) al termine dell'esecuzione della funzione in cui la variabile è dichiarata.
  - **Allocazione statica**. Riguarda le variabili globali e le variabili locali dichiarate `static`. La memoria viene allocata nel **data segment** e viene rilasciata al termine dell'esecuzione del programma.

Introduzione

- Parliamo di **allocazione della memoria** quando un blocco di memoria (RAM) viene assegnata ad un programma in esecuzione.
- In C abbiamo tre classi di allocazione delle memoria:
  - **Allocazione automatica**. Riguarda le variabili (non `static`) con scope locale. La memoria viene *automaticamente allocata* sullo **stack** (*record di attivazione*) e *automaticamente deallocata* (rilasciata) al termine dell'esecuzione della funzione in cui la variabile è dichiarata.
  - **Allocazione statica**. Riguarda le variabili globali e le variabili locali dichiarate `static`. La memoria viene allocata nel **data segment** e viene rilasciata al termine dell'esecuzione del programma.
  - **Allocazione dinamica**. Riguarda memoria richiesta a *run-time* esplicitamente dal programmatore. E' necessario utilizzare opportune funzioni di libreria per richiedere memoria. Il blocco di memoria richiesto viene allocato sullo **heap**. Deve essere esplicitamente rilasciata dal programmatore. L'unico modo per poter accedere alla memoria allocata dinamicamente è tramite puntatori.
- Vediamo adesso come gestire dinamicamente la memoria in un programma C.

Funzioni di libreria per la gestione dinamica della memoria

- I prototipi delle funzioni per la gestione dinamica della memoria sono dichiarati nel file header `stdlib.h`.
- Abbiamo tre aspetti legati alla gestione dinamica della memoria.
- **Allocazione della memoria**
  - Funzioni di allocazione: `malloc()`, `calloc()`
  - Allocano un generico blocco di byte contigui sull'heap.
  - Restituiscono un puntatore `void *` al primo byte del blocco o `NULL` se l'allocazione non è possibile.
  - L'unica possibilità per utilizzare la memoria allocata è tramite puntatore.

Funzioni di libreria per la gestione dinamica della memoria

- I prototipi delle funzioni per la gestione dinamica della memoria sono dichiarati nel file header `stdlib.h`.
- Abbiamo tre aspetti legati alla gestione dinamica della memoria.
- **Allocazione della memoria**
  - Funzioni di allocazione: `malloc()`, `calloc()`
  - Allocano un generico blocco di byte contigui sull'heap.
  - Restituiscono un puntatore `void *` al primo byte del blocco o `NULL` se l'allocazione non è possibile.
  - L'unica possibilità per utilizzare la memoria allocata è tramite puntatore.
- **Deallocazione della memoria**
  - Funzione di deallocazione: `free()`.
  - *Rlascia* memoria allocata tramite `malloc()`, `calloc()` o `realloc()`.
  - La memoria rilasciata può essere utilizzata per eventuali nuove allocazioni.

Funzioni di libreria per la gestione dinamica della memoria

- I prototipi delle funzioni per la gestione dinamica della memoria sono dichiarati nel file header `stdlib.h`.
- Abbiamo tre aspetti legati alla gestione dinamica della memoria.
- **Allocazione della memoria**
  - Funzioni di allocazione: `malloc()`, `calloc()`
  - Allocano un generico blocco di byte contigui sull'heap.
  - Restituiscono un puntatore `void *` al primo byte del blocco o `NULL` se l'allocazione non è possibile.
  - L'unica possibilità per utilizzare la memoria allocata è tramite puntatore.
- **Deallocazione della memoria**
  - Funzione di deallocazione: `free()`.
  - *Rlascia* memoria allocata tramite `malloc()`, `calloc()` o `realloc()`.
  - La memoria rilasciata può essere utilizzata per eventuali nuove allocazioni.
- **Modifica della dimensione di memoria**
  - Funzione di modifica della dimensione: `realloc()`.
  - Permette di modificare la dimensione di memoria allocata precedentemente.
  - Permette sia di diminuire che aumentare la dimensione.

Allocazione dinamica della memoria

- Le due funzioni

```
1 void *malloc(size_t size);
2
3 void *calloc(size_t nmemb, size_t size);
```

consentono di allocare un **blocco contiguo** di memoria di dimensione rispettivamente `size` e `nmemb*size` byte.

Allocazione dinamica della memoria

- Le due funzioni

```
1 void *malloc(size_t size);
2
3 void *calloc(size_t nmemb, size_t size);
```

  consentono di allocare un **blocco contiguo** di memoria di dimensione rispettivamente `size` e `nmemb*size` byte.
- L'unica differenza tra le due funzioni consiste nel fatto che la `calloc()` *inizializza* i bit della memoria allocata a zero.
- Restituiscono un puntatore `void *` al primo byte del blocco di memoria.
  - E' necessario un cast sul puntatore per poter utilizzare la memoria.
- Se la memoria non può essere allocata, restituiscono `NULL`.
  - E' sempre opportuno eseguire un check di controllo.
- Grazie alla equivalenza tra array e puntatori, possiamo accedere alla memoria allocata dinamicamente come se fosse un array.

Allocazione dinamica della memoria: esempi 1/2

- Esempio di utilizzo basilare: allochiamo spazio di memoria per un `int`.

```c
// Allochiamo 4 byte
int *p = malloc(4);
*p = 10;
```

```c
// Allochiamo sizeof(int) byte
int *p = (int *)malloc(sizeof(int));
*p = 10;
```

Nel primo esempio, il cast viene effettuato in modo implicito al tipo `int *`: è preferibile utilizzare un cast esplicito. Inoltre, se `sizeof(int) > 4`, il comportamento è non definito. Il secondo esempio mostra la modalità corretta di utilizzo.

Allocazione dinamica della memoria: esempi 1/2

- Esempio di utilizzo basilare: allochiamo spazio di memoria per un `int`.

```c
// Allochiamo 4 byte
int *p = malloc(4);
*p = 10;
```

```c
// Allochiamo sizeof(int) byte
int *p = (int *)malloc(sizeof(int));
*p = 10;
```

Nel primo esempio, il cast viene effettuato in modo implicito al tipo `int *`: è preferibile utilizzare un cast esplicito. Inoltre, se `sizeof(int) > 4`, il comportamento è non definito. Il secondo esempio mostra la modalità corretta di utilizzo.

- Come allocare un array di `int` di dimensione *variabile*.

```c
unsigned int  n, i;
int i, *list;

scanf("%u",&n);

// Allochiamo un array che possa contenere n int
list = (int *)malloc(n*sizeof(int));

if(list != NULL)
  // Inizializziamo l'array
  for(i=0; i<n; i++) list[i]=i;
```

Possiamo aggirare le limitazioni del tipo di dato array: la dimensione dell'array può essere variabile e decisa in fase di esecuzione del programma.

Allocazione dinamica della memoria: esempi 2/2

- Possiamo utilizzare la memoria dinamica per allocare array di strutture.

```
1  #include <stdlib.h>
2
3  struct person {
4    char  *name;
5    int    age;
6    double height;
7    double weight;
8  };
9
10 struct person *alloc_list(size_t n) {
11   struct person *tmp;
12   tmp = (struct person *)calloc(n,sizeof(struct person));
13   if (tmp!=NULL) { // Inizializziamo i campi
14     size_t i;
15     for(i=0; i<n; i++) {
16       tmp[i].name   = NULL;
17       tmp[i].age    = -1;
18       tmp[i].height = -1;
19       tmp[i].weight = -1;
20     }
21   }
22   return tmp;
23 }
```

La funzione alloca, inizializza e restituisce un puntatore ad un array di `n` elementi di tipo `struct person`. Restituisce `NULL` se l'allocazione non ha avuto successo.

Deallocazione della memoria

- La funzione

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

  consente di *rilasciare* la memoria allocata dinamicamente.
- La memoria deallocata ritorna disponibile per eventuali nuove allocazioni.
- Se `ptr` punta a `NULL`, la `free()` non fa nulla.
- Se `ptr` non è un puntatore valido, cioè se non punta ad un indirizzo di memoria ritornato precedentemente da una chiamata di `malloc()`, `calloc()`,`realloc()` o se la memoria puntata da `ptr` è stata precedentemente deallocata, il comportamento è non definito.

Deallocazione della memoria

- La funzione

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

  consente di *rilasciare* la memoria allocata dinamicamente.
- La memoria deallocata ritorna disponibile per eventuali nuove allocazioni.
- Se `ptr` punta a `NULL`, la `free()` non fa nulla.
- Se `ptr` non è un puntatore valido, cioè se non punta ad un indirizzo di memoria ritornato precedentemente da una chiamata di `malloc(), calloc(),realloc()` o se la memoria puntata da `ptr` è stata precedentemente deallocata, il comportamento è non definito.
- E' buona norma liberare la memoria allocata dinamicamente quando non è più necessaria.
  - A differenza di altri linguaggi di programmazione, il linguaggio C non dispone di un *garbage collector*.
  - Se perdiamo il riferimento (indirizzo del primo byte) alla memoria allocata, non saremo più in grado di accedervi e questa resterà inutilizzabile (garbage) per tutta la durata del programma.

Deallocazione della memoria: esempio

- Vediamo come liberare la memoria allocata dinamicamente per l'array di `struct person` definito precedentemente.

```
1 void free_list(struct person *ptr, size_t n) {
2   if (ptr!=NULL) {
3     size_t i;
4
5     for(i=0; i<n; i++)
6        free(ptr[i].name);
7     free(ptr);
8   }
9 }
```

- Notiamo che prima di liberare la memoria puntata da `ptr` (salvato in `tmp`) è necessario scorrere l'intero array e liberare la (eventuale) memoria puntata dal campo `name` nella struttura.
- Assumiamo implicitamente che l'array di caratteri del nome sia allocato dinamicamente durante l'esecuzione del programma (oppure punti a `NULL`).
- Se non eseguiamo questo passaggio, la memoria puntata dal campo `name` resterà allocata e inutilizzabile (garbage) per l'intero tempo di vita del programma dopo la chiamata a `free_list()`

Modifica della dimensione di memoria

- La funzione

```
1 void *realloc(void *ptr, size_t size);
```

  consente di modificare la dimensione della memoria puntata da `ptr` nella dimensione specificata da `size`.
- Possiamo sia aumentare che diminuire la dimensione del blocco puntato da `ptr`.

Modifica della dimensione di memoria

- La funzione

```
1 void *realloc(void *ptr, size_t size);
```

  consente di modificare la dimensione della memoria puntata da `ptr` nella dimensione specificata da `size`.
- Possiamo sia aumentare che diminuire la dimensione del blocco puntato da `ptr`.
- Se `ptr` è `NULL`, la `realloc()` si comporta come la `malloc()`.
- Se `ptr` non punta ad un indirizzo di memoria valido, il comportamento è non definito.
- Se `ptr` punta ad un indirizzo valido e la dimensione specificata è minore della precedente, la funzione ritorna `ptr`.
- Se `ptr` punta ad un indirizzo valido e la dimensione specificata è maggiore della precedente, la funzione potrebbe:
  - ritornare `ptr`, se è possibile estendere in modo contiguo la memoria,
  - ritornare un nuovo indirizzo di memoria (il blocco di memoria puntato da `ptr` viene ricopiato nella nuova locazione),
  - ritornare `NULL` se non è possibile allocare contiguamente al memoria richiesta (il blocco di memoria puntato da `ptr` resta invariato).
- Se `ptr` punta ad un indirizzo valido e `size` è 0, la `realloc()` equivale alla `free()`.

Modifica della dimensione di memoria: esempio

- Vediamo come estendere dinamicamente l'array di `struct person`.

```
1  int extend_list(struct person **ptr, size_t old_size, size_t new_size) {
2    struct person *tmp = NULL;
3
4    if (*ptr != NULL && old_size < new_size)
5      tmp = realloc(*ptr,new_size*sizeof(struct person));
6
7    if(tmp != NULL) {          // Memoria allocata
8      size_t i;
9      for(i=old_size; i<new_size; i++) { // Inizializziamo i nuovi elementi
10       tmp[i].name   = NULL;
11       tmp[i].age    = -1;
12       tmp[i].height = -1;
13       tmp[i].weight = -1;
14     }
15     *ptr = tmp;              // Aggiorniamo il puntatore al primo elemento
16   }
17   return tmp != NULL;        // Estensione effettuata se tmp != NULL
18 }
```

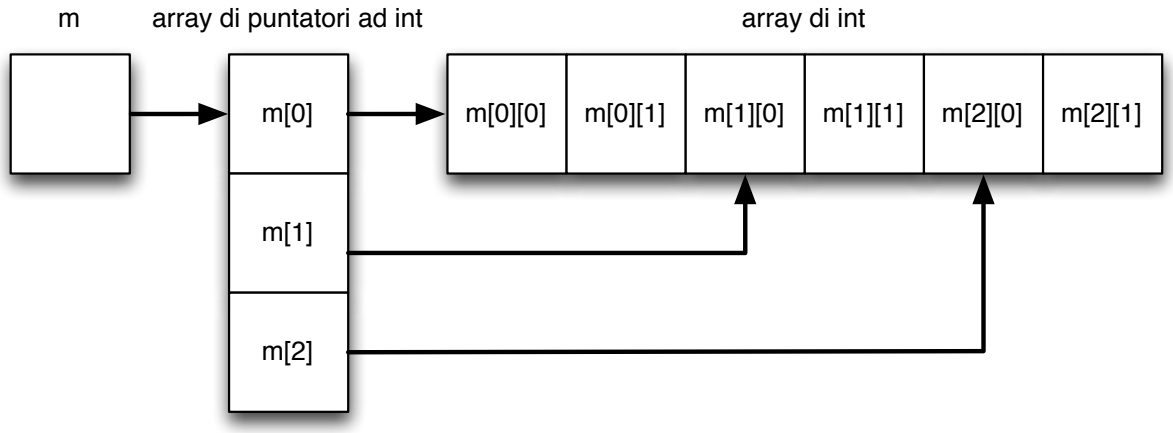
- Questa implementazione permette unicamente di estendere la dimensione dell'array: se la nuova dimensione è minore della precedente, la funzione non esegue nulla.
- Ritorna 0 se l'estensione è stata effettuata, 1 altrimenti.
- Passiamo alla funzione l'indirizzo del puntatore all'array, non l'indirizzo dell'array: in questo modo possiamo modificare direttamente nella funzione il puntatore all'array, nel caso in cui la `realloc()` abbia spostato il tutto in una nuova locazione di memoria.

Esempio: allocazione dinamica di matrici 1/4

- Come possiamo allocare dinamicamente una matrice di `int` di dimensione generica?

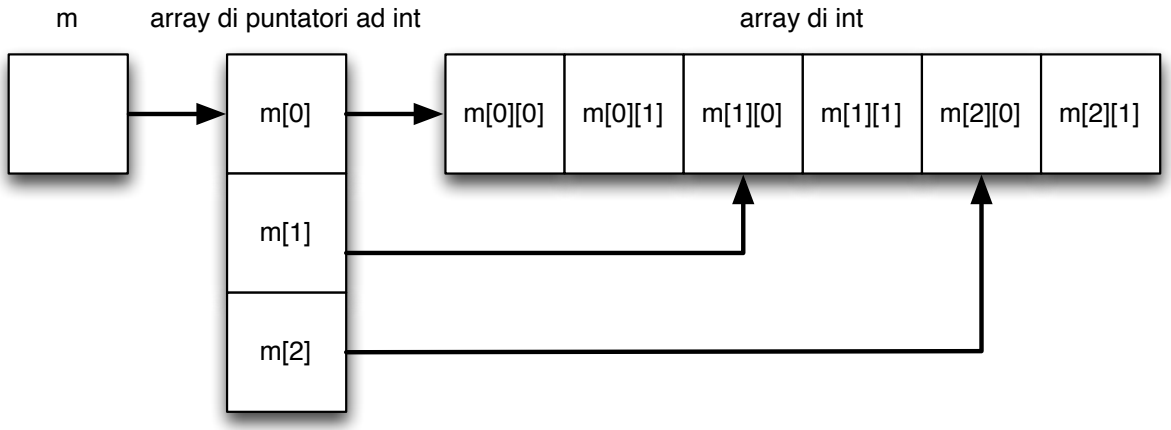
Esempio: allocazione dinamica di matrici 1/4

- Come possiamo allocare dinamicamente una matrice di `int` di dimensione generica?
- Approccio tramite array (dinamico) di puntatori ad unico array (dinamico) di `int`.

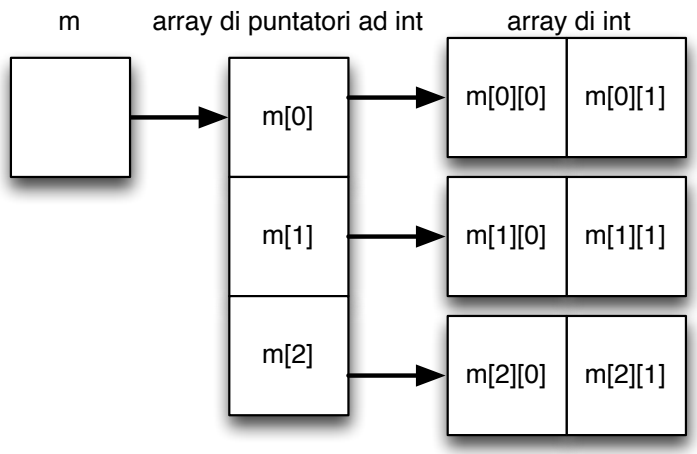


Esempio: allocazione dinamica di matrici 1/4

- Come possiamo allocare dinamicamente una matrice di `int` di dimensione generica?
- Approccio tramite array (dinamico) di puntatori ad unico array (dinamico) di `int`.



- Approccio tramite array (dinamico) di puntatori a molteplici array (dinamici) di `int`.



- La funzione di allocazione ritorna il puntatore `m`. Qual è il tipo puntato da `m`?

Esempio: allocazione dinamica di matrici 2/4

- Approccio tramite array di puntatori ad unica lista di `int`.
- Implementazione base della funzione di allocazione: non controlliamo se la memoria sia stata effettivamente allocata.

```
1  int **int_matrix_alloc(size_t row, size_t col) {
2    int **m;
3    size_t i;
4
5    m    = (int **)malloc(row*sizeof(int *));  // Array di int *
6    m[0] = (int *)calloc(row*col,sizeof(int)); // Array di int
7
8    for(i = 1; i < row; i++) // Settiamo i puntatori alle celle
9      m[i] = &m[0][i*col]; // Equivalente a: *m+i*col.
10
11   return m;
12 }
```

Esempio: allocazione dinamica di matrici 2/4

- Approccio tramite array di puntatori ad unica lista di `int`.
- Implementazione base della funzione di allocazione: non controlliamo se la memoria sia stata effettivamente allocata.

```
1  int **int_matrix_alloc(size_t row, size_t col) {
2    int **m;
3    size_t i;
4
5    m    = (int **)malloc(row*sizeof(int *));  // Array di int *
6    m[0] = (int *)calloc(row*col,sizeof(int)); // Array di int
7
8    for(i = 1; i < row; i++) // Settiamo i puntatori alle celle
9      m[i] = &m[0][i*col]; // Equivalente a: *m+i*col.
10
11   return m;
12 }
```

- Per deallocare la matrice è sufficiente conoscere l'indirizzo dell'array di puntatori.

```
1  void int_matrix_free(int **m) {
2    if(m != NULL)
3      free(m[0]); // Deallochiamo l'array di int
4    free(m);      // Deallochiamo l'array di int *
5  }
```

Esempio: allocazione dinamica di matrici 3/4

- Approccio tramite array di puntatori a differenti liste di `int`.
- Implementazione base della funzione di allocazione: non controlliamo se la memoria sia stata effettivamente allocata.

```
1  int **int_matrix_alloc(size_t row, size_t col) {
2    int **m;
3    size_t i;
4
5    m = (int **)malloc(row*sizeof(int *));   // Array di int *
6
7    for(i = 0; i < row; i++)
8       m[i] = (int *)calloc(col,sizeof(int)); // Array di int
9
10   return m;
11 }
```

Esempio: allocazione dinamica di matrici 3/4

- Approccio tramite array di puntatori a differenti liste di `int`.
- Implementazione base della funzione di allocazione: non controlliamo se la memoria sia stata effettivamente allocata.

```
1  int **int_matrix_alloc(size_t row, size_t col) {
2    int **m;
3    size_t i;
4
5    m = (int **)malloc(row*sizeof(int *));   // Array di int *
6
7    for(i = 0; i < row; i++)
8       m[i] = (int *)calloc(col,sizeof(int)); // Array di int
9
10   return m;
11 }
```

- Per deallocare la matrice è necessario conoscere il numero di righe

```
1  void int_matrix_free(int **m, size_t row) {
2    if(m!=NULL) {
3      size_t i;
4      for(i = 0; i < row; i++)
5        free(m[i]); // Deallochiamo gli array di int
6    }
7    free(m);          // Deallochiamo l'array di int *
8  }
```

Esempio: allocazione dinamica di matrici 4/4

- Indipendentemente dalla scelta implementativa, possiamo accedere agli elementi della matrice utilizzando la notazione array.
- Funzione di inizializzazione di una matrice bidimensionale di dimensione generica.

```
1 void int_matrix_init(int **m, size_t row, size_t col, int init) {
2   size_t i, j;
3
4   for(i = 0; i < row; i++)
5     for(j = 0; j < col; j++)
6       m[i][j] = init;
7 }
```

Esempio: allocazione dinamica di matrici 4/4

- Indipendentemente dalla scelta implementativa, possiamo accedere agli elementi della matrice utilizzando la notazione array.
- Funzione di inizializzazione di una matrice bidimensionale di dimensione generica.

```
1 void int_matrix_init(int **m, size_t row, size_t col, int init) {
2   size_t i, j;
3
4   for(i = 0; i < row; i++)
5     for(j = 0; j < col; j++)
6       m[i][j] = init;
7 }
```

- Funzione di stampa di una matrice bidimensionale di dimensione generica.

```
1 void int_matrix_print(int **m, size_t row, size_t col) {
2   size_t i, j;
3
4   for(i = 0; i < row; i++) {
5     for(j = 0; j < col; j++)
6       printf("%d ",m[i][j]);
7     printf("\n");
8   }
9 }
```