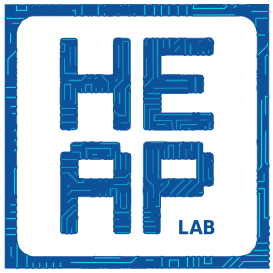
Esercizi su memoria dinamica in C

Stefano Cherubin*



21/12/2017

[**Informatica A**] Esercitazione #15

corso per Ing. Gestionale a.a. 2017/18

*<nome>.<cognome>@polimi.it

Indice

1 String Fusion

Si implementi una funzione C con il seguente prototipo che restituisca la concatenazione delle stringhe prese in input senza modifcare le stringhe originali.

È possibile avvalersi della libreria `string.h`

```
char* strfusion(char* str0, char* str1)
```

1.1 Approccio alla soluzione

1.1.1 Stringa di lunghezza indeterminata

Nella libreria `string.h` esiste già una funzione che esegue la concatenazione di due stringhe, `char* strcat(char* dst, char* src)` ma questa non preserva la stringhe originarie. Essa infatti modifica la stringa `dst`, aggiungendole in coda `src`. Per ottemperare alla richiesta del testo dell'esercizio occorre allocare uno spazio dedicato per il risultato. Così facendo non si andrà ad intaccare le due stringhe in ingresso. Lo spazio dedicato dovrà essere dimensionato in funzione delle stringhe in ingresso, la cui dimensione non è nota a priori. La soluzione farà quindi uso di memoria dinamica, allocando una stringa al momento, utilizzando le esatte dimensioni richieste dai dati.

1.2 Soluzione C

Listato 1: String Fusion

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

char* strfusion(char* str0, char* str1) {
  int len;
  char* str;
  len = strlen(str0) + strlen(str1) + 1;
  str = (char*) malloc(len * sizeof(char));
  if (!str) return str;
  strcpy(str, str0);
  strcat(str, str1);
  return str;
}
```

2 Dynamic merge

Scrivere una funzione C che esegua il merge di due vettori di interi allocati dinamicamente.

2.1 Soluzione C

Listato 2: Dynamic merge

```
1  int* dynVecMerge(int v1[], int n, int v2[], int m) {
2    int* v;
3    v = (int*) malloc((n + m) * sizeof(int));
4    if (!v) return v;
5    int i = 0;
6    int j = 0;
7    int k = 0;
8    while (i < n && j < m) {
9      if (v1[i] < v2[j]) {
10       v[k] = v1[i];
11       i++;
12     } else {
13       v[k] = v2[j];
14       j++;
15     }
16     k++;
17   }
18   while (i < n) {
19     v[k] = v1[i];
20     i++;
21     k++;
22   }
23   while (j < m) {
24     v[k] = v2[j];
25     j++;
26     k++;
27   }
28   return v;
29 }
```

3 Dynamic matrix

Si scriva una funzione `malloc2d`, in grado di allocare una matrice rettangolare di numeri reali (tipo `float`), le cui dimensioni sono ricevute come parametri. La matrice viene inizializzata ponendo il valore di tutte le celle 0.0 eccetto quelle sull'ultima riga con valore 1.0.

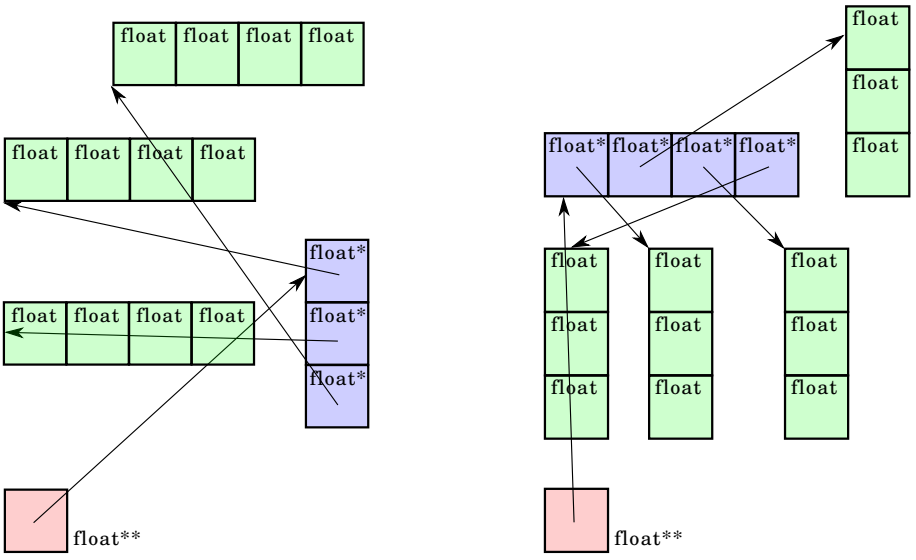
Per semplicità, si trascuri la possibilità che la allocazione dinamica non vada a buon fine.

3.1 Approccio alla soluzione

3.1.1 Array di array

Così come è possibile allocare dinamicamente una struttura di tipo primitivo, è anche possibile allocare dinamicamente dei puntatori. A loro volta i puntatori allocati dinamicamente potranno contenere indirizzi di memoria allocata dinamicamente.

Un vettore di vettori `float` per esempio può essere visto logicamente come una matrice, pur avendo allocato ogni riga (o ogni colonna) in modo indipendente. Un esempio è riportato in figura 1. Di seguito sono riportate diverse versioni di soluzione per meglio chiarire questo concetto.



(a) Vettore di «vettori riga»

(b) Vettore di «vettori colonna»

Figura 1: Matrice allocata come vettore di vettori

3.1.2 Allocazione e inizializzazione

L'allocazione dinamica consente di richiedere aree di memoria extra durante l'esecuzione. L'allocazione dinamica eseguita tramite `malloc` consente di effettuare una sorta di «dichiarazione al volo» di aree di memoria. Sebbene sussistano molte differenze tra una allocazione dinamica e una dichiarazione di variabile statica, è bene ricordare una analogia tra le due operazioni: **non è garantita l'inizializzazione** dell'area di memoria riservata. Occorre quindi inizializzare ogni volta tutta l'area di memoria allocata che si andrà ad utilizzare.

L'operazione di inizializzazione di un'area di memoria può essere fatta a valle di una istruzione di dichiarazione di variabile (o `malloc`) oppure può essere fatta *in-place* (esempio: `float a = 1.0f;`). Esiste una istruzione, simile alla `malloc` , la quale, oltre ad eseguire una richiesta di memoria a tempo di esecuzione, effettua anche una inizializzazione delle celle di memoria a `0`: tale istruzione di chiama `calloc` e vengono riportati alcuni esempi di utilizzo di questa funzione nelle soluzioni che seguono. Per una trattazione più completa, si rimanda ad una qualunque reference della libreria `stdlib.h`[1].

[1] `www.tutorialspoint.com/c_standard_library/c_function_calloc.htm`, tra i tanti disponibili online

3.2 Soluzione

3.2.1 Vettore di «vettori riga», con malloc e sovrascritture

Listato 3: vettore di «vettori riga» (malloc)

```
1  float** malloc2d(int nr, int nc) {
2    /* vettore di "array riga" */
3    float** m;
4    int i, j;
5    m = (float**) malloc(nr * sizeof(float*));
6    for (i = 0; i < nr; ++i) {
7      m[i] = (float*) malloc(nc * sizeof(float))
           ;
8      for (j = 0; j < nc; j++)
9        m[i][j] = 0;
10   }
11   for (j = 0; j < nc; ++j)
12     m[nr - 1][j] = 1.0; /* equivale a
13     *(*(m + (nr - 1) * sizeof(float*)) + j *
           sizeof(float)) = 1.0; */
14   return m;
15 }
```

3.2.2 Vettore di «vettori riga», con calloc

Listato 4: vettore di «vettori riga» (calloc)

```
1  float** malloc2d(int nr, int nc) {
2    /* vettore di "array riga" */
3    float** m;
4    int i, j;
5    m = (float**)malloc(nr * sizeof(float*));
6    for (i = 0; i < nr; ++i) {
7      m[i] = (float*)calloc(nc, sizeof(float));
8    }
9    for (j = 0; j < nc; ++j)
10     m[nr - 1][j] = 1.0;
11   return m;
12 }
```

3.2.3 Vettore di «vettori colonna»

Listato 5: vettore di «vettori colonna»

```
1  float** malloc2d(int nr, int nc) {
2    /* vettore di "array colonna" */
3    float** m;
4    int i, j;
5    m = (float**) malloc (nc * sizeof(float*));
6    for (j = 0; j < nc; ++j) {
7      m[j] = (float*) calloc(nr, sizeof(float));
8      m[j][nr - 1] = 1.0; /* equivale a
9      *(*(m + j * sizeof(float*)) + (nr - 1) *
           sizeof(float)) = 1.0; */
10   }
11   return m;
12 }
```

4 Matrix product

1. Si scriva una funzione che prenda come parametri di input due matrici di numeri reali e restituisca:

   - una matrice data dal prodotto delle due matrici in input
   - NULL quando le matrici non sono moltiplicabili
   - NULL quando non vi è sufficiente memoria per allocare il risultato

2. Si modifichi la funzione precedente per fare in modo che il valore di ritorno sia 0 in caso di matrice moltiplicabile e un valore intero diverso da 0 in caso di matrice non moltiplicabile. La matrice risultato deve comunque essere calcolata e restituita.

4.1 Approccio alla soluzione

4.1.1 Controllo di corretta allocazione

La memoria dinamica è uno strumento molto potente ma deve essere maneggiato con cura. Dopo ogni tentativo di allocazione dinamica è necessario controllare se l'allocazione è andata o meno a buon fine. In caso di allocazione non riuscita è necessario gestire questa situazione segnalando opportunamente l'errore e fermando il normale flusso di esecuzione del programma. Se accadesse che il programma provasse ad accedere a un'area di memoria che non è stato possibile allocare dinamicamente si verificherebbe un errore irreversibile con conseguente crash del programma (nella migliore delle ipotesi).

Nel caso di molteplici allocazioni di memoria, come ad esempio un array di array dinamicamente allocati, se si verifica un problema di allocazione su uno qualsiasi degli array, esso può influire anche sulle precedenti allocazioni andate a buon fine. Quando non ha senso avere una struttura allocata solo in parte, deve essere resa inutilizzabile tutta la struttura di memoria ed è necessario liberare eplicitamente la memoria già allocata con opportune `free`.

4.1.2 Passare un indirizzo per indirizzo

È sempre possibile aggiungere un parametro di output ad una funzione passandolo per indirizzo. Nel caso di un puntatore a memoria da allocare, è opportuno passare per indirizzo il puntatore stesso. Se un normale puntatore a numero reale avrà tipo `float*`, un parametro di output che conterrà memoria allocata di tipo `float*` sarà dichiarato come `float**` nel prototipo di funzione. Allo stesso modo la cosa può essere applicata anche a puntatori a puntatori così un `float**` che deve essere allocato nella funzione sarà `float***` nel prototipo.

4.2 Soluzione C - matrix return

Listato 6: Matrix product (return result matrix)

```
1  float **matrixprod(float** m1, int r1, int c1, float**
       m2, int r2, int c2) {
2    float** res;
3    int i, j, k;
4    if (c1 != r2) return NULL;
5    res = (float**)malloc(r1 * sizeof(float*));
6    if (!res) return NULL;
7    /* allocation */
8    for (i = 0; i < r1; i++) {
9      res[i] = (float*)calloc(c2, sizeof(float));
10     /* check if allocation is successful */
11     if (!res[i]) {
12       /* ALL OR NONE: if one fails, free all
             previously allocated rows */
13       for (i = i - 1; i >= 0; i--) {
14         free(res[i]);
15       }
16     /* free also the vector of pointers */
17     free(res);
18     return NULL;
19     }
20   }
21   /* compute matrix product */
22   for (i = 0; i < r1; i++) {
23     for (j = 0; j < c2; j++) {
24       /* res[i][j] already initialized from calloc */
25       for (k = 0; k < c1; k++) {
26         res[i][j] += m1[i][k] * m2[k][j];
27       }
28     }
29   }
30   return res;
31 }
```

4.3 Soluzione C - check return

Listato 7: Matrix product (return check value)

```
1  int matrixProd(float** m1, int r1, int c1, float** m2,
       int r2, int c2, float*** res) {
2    int i, j, k;
3    if (c1 != r2) return -1;
4    *res = (float**) malloc(r1 * sizeof(float*));
5    if (!*res) return 0;
6    /* allocation */
7    for (i = 0; i < r1; i++) {
8      *res[i] = (float*) calloc(c2, sizeof(float));
9      /* check if allocation is successful */
10     if (!*res[i]) {
11       /* ALL OR NONE: if one fails, free all
              previously allocated rows */
12       for (i = i - 1; i >= 0; i--) {
13         free(*res[i]);
14       }
15       /* free also the vector of pointers */
16       free(*res);
17       *res = NULL;
18       return 0;
19     }
20   }
21   /* compute matrix product */
22   for (i = 0; i < r1; i++) {
23     for (j = 0; j < c2; j++) {
24       /* *res[i][j] already initialized from calloc */
25       for (k = 0; k < c1; k++) {
26         *res[i][j] += m1[i][k] * m2[k][j];
27       }
28     }
29   }
30   return 0;
31 }
```

Licenza e crediti

Crediti

Quest'opera contiene elementi tratti da materiale di Gerardo Pelosi redatto per il corso di Fondamenti di Informatica per Ingegneria dell'Automazione a.a. 2014/15.

Licenza beerware[2]

Quest'opera è stata redatta da Stefano Cherubin. Mantenendo questa nota, puoi fare quello che vuoi con quest'opera. Se ci dovessimo incontrare e tu ritenessi che quest'opera lo valga, in cambio puoi offrirmi una birra.

[2] http://people.freebsd.org/~phk/