L'Allocazione Dinamica della Memoria nel linguaggio C

Prof. Rio Chierego
riochierego@libero.it
http://www.riochierego.it/informatica.htm

Sommario

Questo documento tratta l'allocazione dinamica della memoria in C.

Verrà evidenziato come, alcune volte, l'utilizzo di strutture dati di tipo statico non sono opportune in certi scenari applicativi.

Saranno descritte in dettaglio le funzioni C per la gestione dinamica della memoria e presentati diversi esempi.

Indice

1 Introduzione

La scelta delle appropriate strutture dati è di fondamentale importanza per la risoluzione di un certo problema almeno tanto quanto un buon programma di manipolazione.

Fin qui ci siamo occupati di strutture dati sia di tipo scalare (tipo intero, reale, carattere) sia di tipo strutturato (array di char, vettori di interi, e record o strutture).

A ciascuna di esse abbiamo associato le relative rappresentazioni interne previste dal C.

Le variabili corrispondenti ai tipi suddetti non possono mutare le loro caratteristiche in fase di esecuzione e pertanto sono dette *statiche*.

Vogliamo occuparci ora di un altro tipo di variabili: quelle *dinamiche* cioè le variabili che possono essere create e/o accresciute in fase di esecuzione.

2 Perché Allocare Dinamicamente?

Il lettore potrebbe chiedersi qual è la necessità di utilizzare l'allocazione statica piuttosto che quella dinamica.

Si consideri per esempio il seguente frammento di codice per il calcolo della media degli elementi di un vettore. Una prima versione potrebbe essere:

```
# define N 10
int main ( )
{
int  v [N ] , i , somma , media;
// i n s e r i m e n t o   d a t i
f o r  ( i = 0 ;  i <N ;  i ++)
   {
   printf ( " I n s e r i s c i   e l e m e n t o
   scanf ( "%d " ,  & v [ i ] ) ;
   }
// calcolo media e visualizzazione
somma = 0;
f o r  ( i = 0 ;  i <N ;  i ++)
   {
   somma = somma + v [ i ]
   }
printf ( " La  media è :  % d " ,  media ) ;
return 0;
}
```

Il problema in questo caso è che tale codice effettua la media esclusivamente di `N` valori.

Se si volesse per esempio calcolare la media di 20 numeri piuttosto che di 10 occorrerebbe OBBLIGATORIAMENTE modificare `N` essendo un vettore una struttura dati a carattere statico.

Quindi scriveremmo

```
# define  N 2 0
```

e ricompileremmo il programma.

Una soluzione per rendere il programma più flessibile sarebbe quella di chiedere all'utente di inserire il numero di elementi di cui calcolare la media e quindi utilizzare tale valore in luogo di `N`.

Chiaramente una soluzione del tipo:

```
. . .
int num ;
printf ( " Di q u a n t i v a l o r i v u o i f a r e l a media ? " ) ;
scanf ( "%d " , & num ) ;
int v [num] ;
. . .
```

non è corretta in C visto che la dichiarazione di un vettore richiede la conoscenza del numero di elementi a tempo di compilazione.

Nel suddetto frammento di codice, infatti, il valore della variabile `num` non è noto a tempo di compilazione ma soltanto a tempo di esecuzione (in questo caso dopo l'esecuzione della `scanf`).

Nelle sottosezioni successive presentiamo due diverse soluzioni a questo problema.

La prima sovradimensiona il vettore (detto anche metodo della FALSA DINAMICITA') mentre la seconda fa uso dell'allocazione dinamica della memoria (DINAMICITA' AUTENTICA).

2.1 Sovradimensionamento (FALSA DINAMICITA')

Una possibile soluzione potrebbe essere quella di sovradimensionare il vettore in questo modo:

```
# define MAXDIM 100
int main ( )
{
int v [MAXDIM ] , num , i , somma , media;
// falsa dinamicità e controllo max numero di elementi da trattare
do
  {
  printf ( " Di q u a n t i  v a l o r i  v u o i  f a r e  l a media ? " ) ;
  scanf ( "%d " , & num );
  if (num < 1) || (num > MAXDIM)
    {
     printf ( " dimensione del vettore errata! );
    }
  }
while (num < 1) || (num > MAXDIM);
  //inserimento dati
for ( i = 0 ; i < num ; i ++)
  {
  printf ( " I n s e r i s c i  e l e m e n t o  %d esimo : ", i+1 ) ;
  scanf ( "%d " , & v [ i ] ) ;
  }
 // calcolo della media e visualizzazione
somma = 0 ;
f o r ( i = 0 ; i < num ; i ++)
  {
  somma = somma + v [ i ] ;
  }
media = (float) somma / num ;
printf ( " La media è : % d " , media ) ;
return 0;
}
```

Il problema di questo approccio è però il potenziale ed enorme spreco di memoria.

2.2 Allocazione Dinamica

Un approccio più efficiente dal punto di vista dell'utilizzazione della risorsa memoria sarebbe quello di utilizzare sempre un vettore formato da un numero di elementi esattamente uguale a quelli da elaborare.

```
int main ( )
{
int*   v , num, i, somma, media;

do
  {
  printf ( " Di quanti valori vuoi fare la media? " ) ;
  scanf ( "%d " ,  & num );
  if  (num < 1)
    {
     printf ( " Il numero degli elementi del vettore deve essere maggiore di 1" );
    }
  }
while (num < 1);
// alloco una quantità di memoria per memorizzare esattamente num elementi
// il puntatore punterà esattamente all'inizio di questa zona di memoria
v  =  ( int) malloc ( num * sizeof (int) );

if (v != NULL)
  {
  //inserimento dati
  for  ( i = 0 ;  i <num ;  i ++)
    {
    printf ( " Inserisci elemento %d esimo :  ", i+1 ) ;
    scanf ( "%d " ,  v + i ) ;
    }
 // calcolo della media e visualizzazione
  somma  =  0 ;
  for  ( i = 0 ;  i <num ;  i ++)
    {
    somma  =  somma  +  *(v + i) ;
    }
  media  =  (float) somma / num ;
  printf ( " La  media è :  % d " ,  media ) ;
 // ora posso liberare la zona di memoria precedentemente allocata
  free (v);
  }
else
  {
     printf ( " Errore di allocazione" );
  }
return 0;
}
```

Le funzioni del linguaggio C malloc(), free() e diverse altre saranno discusse nella sezione successiva.

3 Gestione Dinamica della Memoria

In questa sezione descriveremo in dettaglio le funzioni messe a disposizione dalla libreria standard C per la gestione dinamica della memoria.
Le principali sono quattro:

- `malloc()`
- `calloc()`
- `free()`
- `realloc()`

e sono accessibili includendo il file header `stdlib.h`:

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

`malloc()` e `calloc()` allocano un blocco di memoria (seppur in modalità differente)
`free()` libera un blocco di memoria precedentemente allocato e
`realloc()` modifica le dimensioni di un blocco di memoria precedentemente allocato.

3.1 `calloc()` e `malloc()`

Il prototipo delle funzioni `malloc` e `calloc` è il seguente:

```
void* calloc(int num_elementi, int dim_elemento);
void* malloc(int dim_totale);
```

`calloc()` alloca memoria per un vettore di `num_elementi` elementi di `dim_elemento` bytes ciascuno e restituisce un puntatore alla memoria allocata. Inoltre inizializza ogni byte di tale blocco di memoria a zero

`malloc()` alloca `dim_totale` bytes di memoria e restituisce un puntatore a tale blocco. In questo caso la memoria non è inizializzata a nessun valore.

Per determinare la dimensione in byte di un tipo di dato è possibile utilizzare la funzione `sizeof()`. Quindi, per esempio, per allocare un vettore di `n` elementi di tipo

intero basta fare: `int* v;`

```
v = (int*)calloc(n, sizeof(int));
```

Il puntatore `v` punterà al primo elemento di un vettore di `n` elementi interi. Si noti che è stato utilizzato il casting per assegnare il puntatore restituito da `calloc` a `v`. Infatti `calloc` restituisce un `void*` mentre `v` è un `int*`.
Analogamente si può utilizzare la funzione `malloc` per ottenere lo stesso risultato:

```
int v;
v = (int*)malloc(n*sizeof(int));
```

In questo caso occorre determinare il numero totale di bytes da allocare che è pari al numero di elementi del vettore per la dimensione in bytes del generico elemento. In questo caso poichè il vettore contiene interi, la dimensione del generico elemento è `sizeof(int)`.

Sia `calloc()` che `malloc()` restituiscono `NULL` se non riescono ad allocare la quantità di memoria richiesta.

3.2 free()

Il prototipo della funzione `free()` è: `void free(void* ptr);`

Essa rilascia (libera o dealloca) lo spazio di memoria puntato da `ptr` il cui valore proveniva da una precedente `malloc()` o `calloc()` o `realloc()`.

Se `ptr` è `NULL` nessuna operazione viene eseguita.

3.3 realloc()

Il prototipo della funzione `realloc()` è:
`void* realloc(void* ptr, int dim_totale);`

Essa modifica la dimensione di un blocco di memoria puntato da `ptr` a `dim_totale` bytes.

Se `ptr` è `NULL`, l'invocazione è equivalente ad una `malloc(dim_totale)`.

Se `dim_totale` è 0, l'invocazione è equivalente ad una `free(ptr)`.

Naturalmente il valore di `ptr` deve provenire da una precedente invocazione di `malloc()` o `calloc()` o `realloc()`.