Azzolini Riccardo 2021-03-22

Gestione della memoria dinamica

1 Gestione della memoria dinamica

Quando in un programma si vogliono creare dinamicamente degli oggetti[1] che rimangano in vita anche dopo il termine della funzione/procedura in cui sono creati, tali oggetti devono essere memorizzati nell'area di memoria chiamata **free store** o **heap**, che viene appunto gestita dinamicamente dal programma stesso.

Per motivi di semplicità ed efficienza, il linguaggio C non fornisce strumenti automatici di allocazione e rilascio della memoria dinamica (come ad esempio la garbage collection di Java): spetta al programmatore richiedere i blocchi di memoria che servono e poi rilasciarli quando non sono più necessari. È allora fondamentale fare molta attenzione a gestire correttamente la memoria dinamica, poiché le conseguenze di eventuali errori possono essere disastrose.

L'unico strumento che il C fornisce è un **gestore della memoria** (dinamica), al quale è appunto possibile richiedere e poi restituire blocchi di memoria per mezzo di apposite funzioni di libreria, tra cui le principali sono `malloc`, `calloc` e `free`. I prototipi di queste funzioni sono contenuti nel file di intestazione `stdlib.h`, che è dunque necessario includere per usarle.

2 `malloc`

La funzione `malloc` ("Memory ALLOCation") ha il seguente prototipo:

```
void *malloc(size_t size);
```

Essa viene utilizzata per allocare nello heap una blocco di memoria costituito dal numero di byte specificati come parametro[2] (il tipo di tale parametro, `size_t`, è un tipo intero senza segno che può assumere un intervallo di valori adeguato per rappresentare le possibili dimensioni degli oggetti in memoria).

[1] Qui il termine "oggetto" è inteso in senso generico, come una qualunque entità rappresentata in memoria, e non nel senso della programmazione orientata agli oggetti.

[2] Gli algoritmi utilizzati per la gestione della memoria potrebbero stabilire una dimensione minima per i blocchi di memoria allocati: in tal caso, se si richiede un blocco più piccolo del minimo verrà semplicemente allocato un blocco più grande del necessario.

`malloc` restituisce un valore di tipo *puntatore a void*, `void*`, che rappresenta un indirizzo privo di tipo, perché la memoria allocata può essere utilizzata per memorizzare qualunque tipo di oggetto. Il valore restituito è:

- un indirizzo valido, e in particolare l'indirizzo di inizio del blocco allocato, se l'allocazione ha avuto successo;
- il valore nullo se l'allocazione non ha avuto successo, tipicamente perché la quantità di memoria richiesta non è disponibile.

Il valore nullo, corrispondente all'intero `0` e tipicamente rappresentato dalla macro `NULL`, è il valore convenzionalmente usato per indicare un puntatore che non punta a niente, e quindi non può essere dereferenziato (di solito, dereferenziare un puntatore nullo genera un errore in esecuzione).

Se una chiamata a `malloc` ha successo, è garantito che tutti i byte del blocco di memoria allocato siano utilizzabili e liberi (non usati per rappresentare altri oggetti già esistenti), ma non c'è alcuna garanzia sullo stato delle celle di memoria, che potrebbero contenere valori qualsiasi (ad esempio, se la memoria fosse stata precedentemente in uso, conterrebbe ancora i valori che vi erano stati scritti, poiché non viene azzerata quando la si rilascia e poi la si rialloca).

2.1 Esempi

Se si vuole allocare nello heap lo spazio necessario a contenere un valore intero, si usa la chiamata

```
malloc(sizeof(int));
```

per richiedere un blocco di memoria della dimensione di una variabile intera, e poi si assegna a un puntatore il risultato di tale chiamata, convertendolo (esplicitamente o implicitamente) da `void*` a `int*`:

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

A questo punto, si può usare `pt` per accedere alla memoria allocata, esattamente come se fosse un puntatore a una "normale" variabile:

```
*pt = 17;
// ...
printf("%d\n", *pt);
// ...
```

È anche possibile allocare, ad esempio, lo spazio necessario per un vettore di 10 interi:

```
int *pt = (int*)malloc(10 * sizeof(int));
```

Gli accessi all'array allocato avvengono come qualunque accesso a un array tramite un puntatore:

```
pt[0] = 56;
pt[9] = -15;
// ...
++pt;
printf("Secondo elemento: %d\n", *pt);
// ...
```

3 calloc

Un'altra funzione per l'allocazione dinamica della memoria è `calloc`, che ha il prototipo:

```
void *calloc(size_t nmemb, size_t size);
```

Essa funziona in modo simile a `malloc`, ma con due importanti differenze:

- La memoria allocata con `calloc` viene inizializzata mettendo tutti i bit a 0, il che in certi casi è utile/necessario, ma comporta inevitabilmente un tempo di esecuzione maggiore rispetto a `malloc`, soprattutto se il blocco di memoria richiesto è grande.
- `calloc` permette di ragionare a un livello leggermente più alto rispetto a `malloc`, perché invece di richiedere direttamente un certo numero di byte si richiede l'allocazione di uno spazio sufficiente a memorizzare un vettore di `nmemb` oggetti, ciascuno di dimensione `size` (dunque vengono allocati `nmemb * size` byte).

Ad esempio, l'allocazione del vettore di 10 interi fatta prima con `malloc` può essere riscritta con `calloc` in questo modo:

```
int *pt = (int*)calloc(10, sizeof(int));
```

Ciascun elemento del vettore così allocato conterrà inizialmente il valore 0.

4 free

Al contrario di ciò che accade per i record di attivazione, la memoria allocata dinamicamente non viene in automatico restituita al sistema al termine della funzione in cui è stata allocata: se il programmatore non fa nulla, essa rimane riservata fino al termine del programma. Allora, per evitare che un programma consumi una quantità eccessiva di memoria, rischiando potenzialmente di esaurirla (soprattutto se essa viene allocata ripetutamente in un ciclo), è di fondamentale importanza rilasciare i blocchi di memoria che non servono più. A tale scopo, si usa la procedura `free`, il cui prototipo è:

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

Come si può vedere, `free` riceve un unico parametro, che è l'indirizzo di inizio del blocco di memoria da deallocare. Esso deve corrispondere *esattamente* a un indirizzo che in precedenza è stato restituito da una funzione di allocazione di memoria, altrimenti la deallocazione non funziona correttamente (tipicamente, si ha un errore in esecuzione). Il motivo di questo vincolo è che il gestore della memoria tiene traccia dei blocchi allocati memorizzando l'indirizzo di inizio e la dimensione di ciascuno; allora, quando si vuole deallocare un blocco, il gestore ha bisogno proprio dell'indirizzo di inizio — *e non di un altro indirizzo appartenente al blocco* — per risalire alla dimensione del blocco da rilasciare e per poter aggiornare correttamente le sue strutture dati interne. Un'altra conseguenza di questo meccanismo è che un blocco di memoria può essere rilasciato solo per intero, e non in parte (per evitare di complicare eccessivamente gli algoritmi di gestione della memoria).

4.1 Esempi

Il vettore di 10 interi allocato prima con `malloc` (o `calloc`)

```
int *pt = (int*)malloc(10 * sizeof(int));
```

va deallocato come segue quando non serve più:

```
free(pt);
```

Si consideri ora questo frammento di codice:

```
int *p = (int*)malloc(10 * sizeof(int));
*p = 30;
++p;
*p = 40;
// ...
free(p); // Errore
```

La chiamata `free` appena mostrata *non* è corretta, perché dopo l'istruzione `++p;` il puntatore `p` contiene un indirizzo che, pur essendo compreso nel blocco di memoria allocato, non è l'indirizzo restituito da `malloc`. Una possibile correzione sarebbe decrementare `p` prima di chiamare `free`, in modo da annullare l'operazione di incremento,

```
int *p = (int*)malloc(10 * sizeof(int));
*p = 30;
++p;
*p = 40;
// ...
--p;
free(p);
```

ma in molti casi è più semplice mantenere sempre un puntatore all'indirizzo originale, e creare un'altra variabile puntatore da modificare:

```c
int *p, *q;
p = q = (int*)malloc(10 * sizeof(int));
*q = 30;
++q;
*q = 40;
// ...
free(p);
```

Anche la chiamata `free` nel seguente frammento di codice è errata, perché l'indirizzo passato come argomento non corrisponde a un blocco di memoria allocata dinamicamente (nello heap), ma piuttosto a una variabile locale (situata nello stack):

```c
int i = 7;
// ...
free(&i); // Errore
```