Memoria dinamica e operatori bitwise

Matteo Spanio

23 aprile 2024

Strutture dati

Le strutture sono molto utili per definire… strutture dati. Per esempio si possono definire array dinamici, liste concatenate, alberi, pile, code…

Gli array in C sono molto limitati, infatti dobbiamo gestire la loro lunghezza *a mano*, ad esempio passandola come argomento di una funzione.

Si può creare una struttura che contenga un array e la sua lunghezza:

Lista 1 Implementazione di un array di interi con lunghezza variabile.

```
#define MAX 1024 /* <1> */

typedef struct intarray {
    int array[MAX]; /* <2> */
    int length; /* <3> */
} IntArray;
```

1. Definiamo una costante `MAX` che rappresenta il numero massimo di elementi che possono essere memorizzati nell'array;
2. La struttura dati contiene l'array di interi (che, in questo caso, è di dimensione $1024 \times 4 = 4096$ byte)
3. La struttura contiene anche un intero che rappresenta la lunghezza attuale dell'array: cioè il numero di elementi effettivamente contenuti dall'array.

Si noti che il membro `lenght` è aggiornato solo quando si aggiungono o rimuovono elementi dall'array.

Adesso si può passare la struttura come argomento di una funzione, senza dover tenere traccia in maniera esplicita della lunghezza dell'array:

La routine per aggiungere un elemento all'array è la seguente:

Lista 2 Implementazione della funzione per stampare un IntArray.

```
void stampa_array(IntArray a) {
    for (int i = 0; i < a.length; i++) {
        printf("%d ", a.array[i]);
    }
    printf("\n");
}
```

```
int append(IntArray *a, int value) {                    ①
    if (a->length < MAX) {                              ②
        a->array[a->length] = value;                    ③
        a->length++;                                    ④
        return 0;                                       ⑤
    }
    return 1;                                           ⑥
}
```

① La funzione `append` prende come argomento un puntatore a `IntArray` e un intero da aggiungere all'array, l'array è passato *by reference* perché i suoi campi vengono modificati;
② Prima di effettuare l'inserimento viene controllato che l'array non sia già pieno, per accedere ai membri di un puntatore a una struttura si usa l'operatore `->`, che è una forma abbreviata di `(*p).membro`;
③ Si aggiunge il valore all'array;
④ Si incrementa la lunghezza dell'array;
⑤ La funzione restituisce 0 se l'inserimento è andato a buon fine;
⑥ Restituisce 1 se l'array è pieno.

Esercizio

Per esercizio si implementino le funzioni: 1. `int get(IntArray a, int index)` che restituisce l'elemento in posizione `index` dell'array `a`; 2. `int insert(IntArray *a, int index, int value)` che inserisce l'elemento `value` in posizione `index` dell'array `a`; 3. `int remove(IntArray *a, int index)` che rimuove l'elemento in posizione `index` dell'array `a`.

> **Consiglio**
>
> Si noti che le funzioni `insert` e `remove` possono fallire se l'array è pieno o vuoto, rispettivamente, inoltre è necessario spostare tutti gli elementi dell'array.

Memoria dinamica

L'implementazione vista in Lista 1 ha due grossi problemi:

1. la dimensione dell'array è fissa;
2. la struttura `IntArray` funziona solo con interi, per gestire float, char, ecc. bisogna creare ogni volta una nuova struttura;

Soffermandosi sul punto 1 ci si può accorgere che le strutture dati viste fino ad ora in C hanno dimensione fissa. Infatti gli array vengono inizializzati con un valore costante. Non sempre però è possibile sapere quanta memoria sarà necessaria al programma per contenere i dati, per questo, esistono dei comandi che permettono di allocare dinamicamente zone di memoria, permettendo di fare strutture che crescono o si rimpiccioliscono durante l'esecuzione del programma.

Ad esempio si pensi a un programma che, a runtime, prenda in input dei dati dall'utente e li immagazzini in un array. Si può pensare di creare un array di dimensione molto grande, sperando che sia sufficiente per contenere i dati che inserirà l'utente. Evidentemente questa soluzione introduce facilmente errori, infatti, nel caso in cui l'utente inserisca pochi dati, ci sarà un grande spreco di memoria, al contrario, se l'utente inserisce più dei dati che possono essere immagazzinati, bisogna interrompere l'esecuzione del programma mostrando un messaggio di errore.

Per evitare tutto ciò la libreria `stdlib.h` mette a disposizione 3 funzioni per gestire *dinamicamente* la memoria:

1. `malloc` (memory allocation): alloca una zona di memoria di una certa dimensione
2. `calloc` (clear allocation): alloca una zona di memoria di una certa dimensione e la inizializza a 0
3. `realloc` (reallocate): cambia la dimensione di una zona di memoria già allocata

`malloc`, `calloc` e `realloc` restituiscono un puntatore a void, che è un puntatore generico. Questo puntatore non fa riferimento a un certo tipo di dato, ma solo a un indirizzo di memoria.

`malloc`

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

Supponendo di voler creare un array di interi di dimensione variabile, c'è bisogno di creare un puntatore ad interi e poi assegnargli l'indirizzo della nuova zona di memoria:

```c
int *a;
a = malloc(10 * sizeof(int));
```

La funzione `malloc` accetta come argomento il numero di byte da allocare. L'esempio crea 10 cellette di memoria, ciascuna grande quanto un intero (4 byte).

`calloc`

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

- Per inizializzare array torna comoda anche la funzione calloc, che ha due parametri: numero di membri e dimensione (in byte) dei membri
- Inoltre ha come effetto aggiuntivo di porre a zero tutti i byte interessati (quindi tutti i membri dell'array che creo)

```
double *a;
a = calloc(10, sizeof(double));
```

L'effetto di questo codice è lo stesso del precedente (con la malloc), ma con tutti i membri dell'array inizializzati a 0.

`realloc`

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

- Questa funzione cambia la dimensione di uno spazio allocato dinamicamente.
- L'argomento `ptr` deve essere un puntatore ottenuto da una funzione ..alloc, altrimenti porta a comportamento non definito.
- I dati già presenti nella zona di memoria vengono mantenuti.

NULL

- Quando si usa la memoria dinamica, è importante controllare che l'allocazione sia andata a buon fine.
- Se `malloc`, `calloc` o `realloc` non riescono a trovare spazio in memoria, restituiscono `NULL`.
- `NULL` è una costante che rappresenta un puntatore all'indirizzo `0x0`. `NULL` è definito in `stdlib.h`.

`malloc`, `calloc`, `realloc`

- Queste funzioni scrivono in una zona di memoria nota come **heap**;
- Le variabili locali e i parametri delle funzioni sono invece memorizzati nello **stack**;
- Usando ricorsione e molta memoria dinamica si può esaurire la memoria disponibile
- Occorre quindi fare uso oculato della memoria e pulire quella che usata che non serve più

`free`

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

- La funzione free permette di liberare un blocco di memoria (e quindi renderlo disponibile per un'altra allocazione)
- `ptr` dopo l'esecuzione di `free` continua a puntare alla stessa zona di memoria, ma il contenuto di quella zona non è più garantito, si chiama **dangling pointer**
- Occorre, subito dopo la free, assegnare a `ptr` il valore `NULL` (oppure una nuova memoria)
- accedere un blocco deallocato è un gravissimo errore

Array dinamico

Un array dinamico è un array la cui dimensione può cambiare durante l'esecuzione del programma.

Per implementare un array dinamico si può usare una struttura che contiene un puntatore all'array e la sua lunghezza:

```c
typedef struct {
    int *array;
    int capacity;
    int length;
} DynArray;
```

`DynArray`

Il nuovo tipo `DynArray` ha tre membri:

- `array` è un puntatore ad interi, che punta all'array dinamico
- `capacity` è la dimensione massima dell'array
- `length` è il numero di elementi attualmente presenti nell'array

Quando length raggiunge capacity, bisogna riallocare la memoria per l'array, aumentando la sua dimensione.

`DynArray` generico

Rimane il problema che `DynArray` funziona solo con array di interi.

Abbiamo però visto che `void *` serve proprio quando si usano indirizzi di memoria senza un tipo preciso, questo ci permette di definire un array di `void *` per decidere a runtime il tipo dell'array:

```c
typedef struct {
    void **array;
    int capacity;
    int length;
} GenericArray;
```

Operazioni bitwise

Gli operatori bitwise permettono di effettuare calcoli al livello dei bit delle variabili. Questi operatori sono molto utili quando si deve lavorare con i registri di un microcontrollore o con i dati grezzi provenienti da un sensore.

Gli operatori bitwise in C sono: - `&` AND - `|` OR - `^` XOR - `~` NOT - `<<` Shift a sinistra - `>>` Shift a destra

Calcoli

Questi operatori applicano le regole dell'algebra booleana, pertanto sarà importante conoscere le tabelle di verità delle operazioni AND, OR e XOR.

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Esempi

In elettronica digitale, spesso si utilizzano serie di bit per rappresentare lo stato di un sistema. Ad esempio, si può utilizzare un byte per rappresentare lo stato di 8 interruttori. Supponiamo che l'interruttore 3 sia acceso e gli altri spenti. Per rappresentare lo stato degli interruttori si può utilizzare una variabile di tipo `char`:

```c
unsigned char switches = 0b00001000;
// oppure
unsigned char switches = 8;
```

In questa configurazione il quarto bit interruttore è acceso, mentre gli altri sono spenti.

Ipotizzando di voler accendere un altro interruttore, si può utilizzare l'operatore OR:

```
// switches = 0b00001000
switches = switches | 0b00000100;
// oppure
switches = switches | 4;
```

Ora il valore di `switches` sarà `0b00001100`. Il che rappresenta il fatto che gli interruttori 3 e 4 sono accesi.

Per spegnere tutti gli interruttori si può utilizzare l'operatore AND:

```
switches = switches & 0b00000000;
// oppure
switches = switches & 0;
// switches: 0b00000000
```