Azzolini Riccardo 2021-03-24

Pile

1 Operatore ->

Come si vedrà a breve, in C capita molto spesso di accedere ai campi di una struttura tramite un puntatore a tale struttura. Allora, per rendere più compatto e leggibile il codice, il linguaggio mette a disposizione una "scorciatoia": l'operatore binario infisso `->`. Esso deve avere come operando sinistro un puntatore *`pt`* a una struttura e come operando destro un'etichetta *`campo`* di un campo di tale struttura. Dati tali operandi, *`pt`*`->`*`campo`* è esattamente equivalente a `(*`*`pt`*`).`*`campo`*: si dereferenzia il puntatore *`pt`* e si seleziona il campo *`campo`* della struttura puntata, che può così essere letto e/o modificato.

2 Pila

La **pila**, o **stack**, è una delle più semplici e importanti strutture dati. Le principali operazioni su una pila sono:

- `init_stack`: crea la struttura dati in memoria;
- `push`: inserisce un nuovo elemento;
- `pop`: elimina l'elemento in cima;
- `top`: legge l'elemento in cima;
- `is_empty`: indica se la pila è attualmente vuota.

Ci sono vari modi di implementare una struttura che supporti queste operazioni. Adesso, come esempi di applicazioni della memoria dinamica, si vedranno due possibili implementazioni: una basata su un vettore allocato dinamicamente e una basata su una lista concatenata.

3 Implementazione con un vettore

Una pila basata su un vettore allocato dinamicamente può essere rappresentata dalla seguente struttura `stack`:

```
typedef int ITEM;

struct stack {
    ITEM *array;
    int max_size;
    int top;
};

typedef struct stack STACK;
```

- Una pila è una struttura dati omogenea, cioè contiene valori che sono tutti dello stesso tipo. Siccome l'implementazione della pila non dipende dal tipo degli elementi, in questo codice si usa una `typedef` per assegnare a esso il nome astratto `ITEM`: così, per cambiare il tipo degli elementi è sufficiente modificare la `typedef`.
- I campi della struttura `stack` servono a tenere traccia di tutte le informazioni necessarie per la gestione della pila:
  - `array` è un puntatore al vettore di `ITEM` allocato dinamicamente che conterrà gli elementi della pila;
  - `max_size` indica la dimensione del vettore allocato, ovvero il numero massimo di elementi che la pila può contenere;
  - `top` è l'indice della prima posizione libera del vettore.
- Il nome `stack` non corrisponde di per sé a un tipo, perché è solo l'etichetta di una struttura: il nome del tipo corrispondente è `struct stack`. Volendo evitare di scrivere `struct stack` ogni volta, si usa una `typedef` per associare a tale tipo un nome più breve: `STACK`.

Adesso verranno mostrate le implementazioni delle varie operazioni sulla pila. Si noti che, per semplicità, verranno omessi alcuni dettagli, come ad esempio la gestione degli errori e la deallocazione della memoria allocata,[1] che sarebbero importanti in un'implementazione "reale", destinata all'uso pratico, ma non riguardano strettamente la logica di gestione della struttura dati.

[1] In particolare, la deallocazione della memoria verrebbe tipicamente gestita fornendo, come ulteriore operazione sulla pila, una procedura di distruzione da chiamare quando la pila non serve più.

3.1 `init_stack`

La funzione `init_stack` crea dinamicamente una pila in grado di contenere al massimo il numero di elementi passato come argomento. La pila creata è inizialmente vuota.

```
STACK *init_stack(int n) {
    STACK *tmp = (STACK *)malloc(sizeof(STACK));
    if (tmp == NULL) {
        // Gestione dell'errore...
        return NULL;
    }
    tmp->array = (ITEM *)calloc(n, sizeof(ITEM));
    if (tmp->array == NULL) {
        // Gestione dell'errore...
        return NULL;
    }
    tmp->max_size = n;
    tmp->top = 0;
    return tmp;
}
```

1. Per prima cosa, si alloca dinamicamente (nello heap) lo spazio necessario per la struttura `stack`. Se l'allocazione fallisce, l'errore viene gestito in qualche modo (qui omesso, per semplicità — è mostrata solo la restituzione di `NULL`, che è il modo convenzionale di segnalare il fallimento di una funzione che restituisce in puntatore).

2. Quello allocato al passo precedente è *solo* lo spazio necessario per i 3 campi della struttura (un puntatore e due interi). Lo spazio per il vettore va allocato separatamente, e qui per comodità lo si fa usando `calloc` (ma si potrebbe ugualmente usare `malloc(n * sizeof(ITEM))`, dato che qui l'inizializzazione a 0 della memoria non serve). L'indirizzo di base del vettore così allocato viene salvato nel campo `array` della struttura allocata prima.

   Anche in questo caso, se l'allocazione fallisce si gestisce in qualche modo l'errore (si noti in particolare che qui un'implementazione reale dovrebbe rilasciare lo spazio riservato per la struttura `stack`: l'errore di allocazione riguarda solo il vettore, quindi tale struttura rimane in memoria, e deve essere liberata manualmente per evitare che continui a occupare inutilmente spazio fino al termine del programa).

3. Si inizializzano i restanti campi della struttura, salvando in `max_size` la dimensione del vettore allocato impostando a 0 l'indice della prima posizione libera (perché la pila è inizialmente vuota).

4. Si restituisce l'indirizzo della struttura appena allocata e inizializzata.

3.2 `is_empty`

La funzione `is_empty` restituisce il valore 1 (vero) se la pila è vuota, e 0 (falso) altrimenti.

```
int is_empty(STACK *pt) {
    if (pt == NULL) {
        // Termina segnalando un errore...
    }
    return pt->top == 0;
}
```

Innanzitutto, si controlla per sicurezza che il puntatore passato come argomento non sia nullo; se lo è, l'errore viene in qualche modo gestito/segnalato (ad esempio restituendo un qualche valore particolare come codice di errore), e la funzione termina. Poi, una volta verificato che il puntatore non è nullo, si accede al campo `top` della struttura puntata per controllare quale sia la prima posizione libera del vettore: se essa è quella di indice 0, cioè la prima, allora la pila è vuota, altrimenti contiene almeno un elemento.

3.3 `push`

La funzione `push` inserisce nella pila un nuovo elemento passato come argomento.

```
void push(STACK *pt, ITEM el) {
    if (pt == NULL) {
        // Gestione dell'errore...
        return;
    }
    if (pt->top >= pt->max_size) {
        // Gestione dell'errore...
        return;
    }
    pt->array[pt->top] = el;
    pt->top++;
}
```

1. Anche qui, come in `is_empty`, si controlla che il puntatore alla pila non sia nullo, e in caso contrario l'errore viene in qualche modo gestito, dopodiché la funzione termina.
2. Un altro errore da gestire è il caso in cui si prova ad aggiungere un elemento ma la pila è già piena. Perciò, il codice di questa funzione deve verificare che `top` (l'indice della prima posizione libera) sia un indice valido per il vettore: se non lo è (in particolare, se `top == max_size`, che non è un indice valido perché gli indici

vanno da 0 a `max_size - 1`), significa che nel vettore non ci sono più posizioni libere.

3. Dopo i controlli, si può finalmente effettuare l'inserimento vero e proprio, salvando il nuovo elemento nella prima posizione libera del vettore.
4. Ora la posizione del vettore indicata da `top` è occupata, dunque si incrementa `top` per contrassegnare come prima libera la posizione successiva (se esiste).

3.4 `top`

La funzione `top` legge e restituisce l'elemento in cima alla pila, se questa non è vuota, altrimenti segnala in qualche modo un errore.

```
ITEM top(STACK *pt) {
    if (is_empty(pt)) {
        // Termina segnalando un errore...
    }
    return pt->array[pt->top - 1];
}
```

L'elemento in cima alla pila è quello situato nella posizione precedente alla prima libera, cioè all'indice `top - 1`.

Osservazione: Qui non si controlla esplicitamente che `pt` sia non nullo, perché tale controllo viene già fatto dalla funzione `is_empty`, quindi ripeterlo sarebbe superfluo.

3.5 `pop`

La funzione `pop` rimuove l'elemento presente in cima alla pila, oppure non fa nulla se la pila è già vuota.

```
void pop(STACK *pt) {
    if (!is_empty(pt)) {
        pt->top--;
    }
}
```

Qui si esegue solo una cancellazione logica: l'elemento presente in cima alla pila non viene rimosso dal vettore, ma semplicemente la posizione che esso occupa viene contrassegnata come libera (facendo puntare a essa l'indice `top`), così un'eventuale successiva `push` potrà sovrascrivere l'elemento eliminato.

4 Implementazione con una lista concatenata

L'implementazione di una pila appena vista, basata su un vettore, ha un limite: la dimensione massima della pila è limitata a quella specificata al momento dell'inizializzazione, poiché la dimensione di un vettore è fissa una volta che esso è stato allocato. Uno dei modi per realizzare invece una pila di dimensione (teoricamente) illimitata è implementarla mediante una **lista concatenata**, allocando separatamente lo spazio per gli elementi man mano che questi vengono aggiunti (dunque lo spazio occupato dalla struttura varia dinamicamente).

La lista concatenata può essere rappresentata, ad esempio, tramite le seguenti definizioni:

```
typedef int ITEM;

struct node {
    ITEM el;
    struct node *next;
};
typedef struct node NODE;

struct stack {
    NODE *top;
};
typedef struct stack STACK;
```

- Come nell'implementazione basata su un vettore, si usa una `typedef` per astrarre il tipo degli elementi.
- La struttura `node` rappresenta un nodo della lista, che contiene il valore di un singolo elemento della pila e un puntatore al nodo successivo (se questo nodo non è l'ultimo, altrimenti tale puntatore è `NULL`).
- La struttura `stack`, che rappresenta l'intera pila, contiene semplicemente il puntatore al primo nodo della lista, ovvero alla cima della pila. Si noti che il puntatore è `NULL` quando la pila è vuota, perché una lista vuota non alcun nodo (in questo caso — più avanti si vedrà un'implementazione per le liste in cui ciò non è vero).
- Per entrambe le strutture vengono definiti dei nomi "più comodi": `NODE` e `STACK`.

Ora verranno presentate le implementazioni delle stesse operazioni implementate in precedenza per la rappresentazione basata su un vettore. Anche qui verranno omessi alcuni dettagli relativi alla gestione degli errori, ecc.

4.1 `init_stack`

In questo caso, si è scelto di *non* gestire l'allocazione della memoria per la struttura `stack` direttamente all'interno della funzione di inizializzazione `init_stack`: invece, l'allocazione spetta all'utente, e `init_stack` si occupa solo di inizializzare poi i campi. Tale soluzione è forse un po' più "scomoda" da usare, ma lascia all'utente la scelta di come allocare la struttura (ad esempio, essa può essere allocata nello heap, oppure come variabile locale in un record di attivazione, o ancora in una variabile globale, ecc.).

```
void init_stack(STACK *pt) {
    if (pt == NULL) {
        // Gestione dell'errore...
        return;
    }
    pt->top = NULL;
}
```

L'unico campo della struttura `stack` è il puntatore `top`, che viene inizializzato a `NULL` perché la pila è inizialmente vuota.

4.2 `is_empty`

Siccome una pila vuota è rappresentata da una lista senza nodi, determinare se la pila sia vuota è immediato: basta controllare se il puntatore `top` è nullo.

```
int is_empty(STACK *pt) {
    if (pt == NULL) {
        // Termina segnalando un errore...
    }
    return pt->top == NULL;
}
```

4.3 `top`

La funzione `top` usa `is_empty` per controllare che la pila non sia vuota, poi restituisce il valore `pt->top->el`. Siccome, per le regole di associatività, occorrenze multiple dell'operatore `->` si leggono da sinistra a destra, l'accesso `pt->top->el` significa che:

1. `pt->top`: si accede al campo `top` della struttura puntata da `pt` per ottenere l'indirizzo del primo nodo della lista;
2. `pt->top->el`: si accede al campo `el` della struttura puntata dal precedente indirizzo per ottenere il valore dell'elemento contenuto nel primo nodo della lista.

```
ITEM top(STACK *pt) {
    if (is_empty(pt)) {
        // Termina segnalando un errore...
    }
    return pt->top->el;
}
```

4.4 push

Per aggiungere un nuovo elemento in cima alla pila, la funzione `push` deve creare un nuovo nodo e inserirlo all'inizio della lista.

```
void push(STACK *pt, ITEM el) {
    NODE *n = (NODE *)malloc(sizeof(NODE));
    if (n == NULL) {
        // Gestione dell'errore...
        return;
    }
    n->el = el;
    n->next = pt->top;
    pt->top = n;
}
```

1. Si alloca lo spazio necessario per il nodo, controllando che l'allocazione vada a buon fine.
2. Si inizializzano i campi del nuovo nodo, inserendo in `el` il valore del nuovo elemento (passato come parametro alla procedura `push`) e facendo puntare `next` all'attuale primo nodo della lista, che diventerà il secondo.
3. Si aggiorna il puntatore `top` al primo nodo della lista, in modo che punti al nuovo nodo appena creato.

4.5 pop

La procedura `pop` elimina l'elemento in cima alla pila tramite l'operazione inversa di quella eseguita dalla `push`, cioè la rimozione del primo nodo della lista.

```
void pop(STACK *pt) {
    if (is_empty(pt)) return;
    NODE *n = pt->top;
    pt->top = pt->top->next;
    free(n);
}
```

1. Se la pila è già vuota, la procedura termina senza fare nulla.
2. Si salva in una variabile temporanea `n` l'indirizzo dell'attuale primo nodo della lista, cioè l'indirizzo contenuto in `top`, al fine di evitare che il nodo diventi inaccessibile quando il puntatore `top` verrà aggiornato.
3. Si aggiorna il puntatore `top`, spostandolo all'attuale secondo nodo (`pt->top->next` è appunto il puntatore che contiene l'indirizzo del nodo successivo al primo). Così, il secondo nodo diventa di fatto il primo, e il primo nodo viene rimosso dalla lista.
4. Si libera lo spazio occupato in memoria dal nodo rimosso. Si noti che ciò è possibile proprio perché l'indirizzo di tale nodo è stato salvato nella variabile `n`.

Poiché lo spazio occupato dal nodo viene effettivamente liberato, la cancellazione eseguita qui può essere considerata fisica, a differenza della cancellazione logica che si faceva nell'implementazione basata su un vettore. Il vantaggio è che, con la cancellazione fisica, lo spazio liberato può essere riutilizzato per altre allocazioni, anche relative a strutture dati diverse da questa, mentre con la cancellazione logica lo spazio degli elementi rimossi rimane comunque occupato finché non si distrugge l'intera struttura dati, e nel frattempo può essere riutilizzato solo per gli inserimenti all'interno della stessa struttura.

4.6 Esempio di utilizzo

La seguente funzione `main` mostra un semplice esempio di utilizzo della struttura dati appena implementata: si richiede all'utente di inserire dei numeri che vengono memorizzati nella pila e poi estratti per stamparli. Siccome il primo elemento estratto da una pila è l'ultimo inserito (*LIFO*, *Last In First Out*), e così via, i numeri saranno stampati in ordine inverso.

```c
int main(void) {
    STACK *st = (STACK *)malloc(sizeof(STACK));
    init_stack(st);

    for (int i = 0; i < 10; i++) {
        printf("Immetti un valore: ");
        ITEM val;
        scanf("%d", &val);
        push(st, val);
    }

    while (!is_empty(st)) {
        printf("%d\n", top(st));
        pop(st);
    }
```

```
    free(st);
    return 0;
}
```

Un'osservazione importante su questo codice è che `free(st)` rilascia *solo* lo spazio usato per rappresentare i campi della struttura `stack`, e *non* i nodi della lista concatenata, che anzi diventerebbero inaccessibili, e quindi impossibili da deallocare, poiché i loro indirizzi si perderebbero. In questo caso, ciò non è un problema per due motivi:

- questo programma rimuove dalla pila tutti gli elementi inseriti, dunque i nodi vengono già deallocati nelle operazioni di rimozione;
- se anche ci fossero nodi non deallocati, la memoria da loro occupata verrebbe automaticamente recuperata dal sistema operativo al termine del programma, che qui avviene subito dopo l'istruzione `free(st)`.

Ci sono invece molti altri casi (ad esempio, se ci fosse un ciclo che crea pile e non le svuota completamente) nei quali è concretamente importante deallocare i nodi, dunque in un'implementazione pratica sarebbe utile definire una procedura che consenta di distruggere correttamente i nodi di una pila.