Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

giovedì 24 novembre 2016

L'obiettivo di questo esercitazione è sperimentare definizione, implementazione e uso del tipi di dati astratti. In particolare faremo riferimento alla struttura dati chiamata *stack* (pila), in cui gli oggetti sono inseriti e rimossi attraverso una politica LIFO (last in first out, ovvero: l'ultimo a entrare è il primo ad uscire).

L'esercitazione è suddivisa in fasi, in modo da generalizzare via via le soluzioni trovate nelle fasi precedenti.

1 Tipo di dato astratto pila

Per semplicità, cominciamo col considerare programmi che usano una sola pila, contenente valori interi.

1.1 Interfaccia

Scrivete un file di intestazione `stack.h` nel quale raccogliere i prototipi delle funzioni di gestione della pila, definite come segue:

- **make_empty** svuota la pila;
- **is_empty** restituisce 1 se la pila è vuota, o altrimenti;
- **top** restituisce il valore in cima alla pila;
- **pop** estrae il valore in cima alla pila;
- **push(n)** inserisce n in cima alla pila;

Ad esempio, la funzione push avrà prototipo:

```
void push( int n );
```

Commentate opportunamente i vari prototipi e non dimenticare di proteggere il file di intestazione da inclusioni multiple con l'uso delle direttive **`#ifndef... endif`**.

1.2 Implementazioni

Le funzioni dichiarate nell'interfaccia `stack.h` vanno definite in un file `stack.c` di implementazione.

Naturalmente è possibile implementare l'interfaccia in molti modi diversi. Quando diciamo che la pila è un tipo di dato *astratto* ci riferiamo proprio al fatto che al client è nota l'interfaccia, ma non l'implementazione: il client non può accedere all'oggetto pila se non attraverso le funzioni fornite dall'interfaccia; inoltre è possibile modificare l'implementazione dell'interfaccia senza interferire in nessun modo col client.

Dovete implementare la pila in due maniere diverse; ciascuna implementazione richiederà la costruzione di un file apposito.

- Nel file `stack_array.c` rappresentate la pila attraverso un array di interi e un indice alla cima della pila;
- Nel file `stack_list.c` rappresentate la pila con una lista concatenata: in questo caso, la cima della pila è rappresentata proprio dalla testa della pila, quindi inserimenti e cancellazioni vanno fatti in testa.

Testate il funzionamento delle vostre implementazioni con un semplice file di test `test.c` opportunamente progettato. Quando compilate, scegliete quale implementazione testare. Ad esempio, per testare l'implementazione con la lista, dovrete usare il comando

```
gcc test.c stack_list.c
```

1.3 Esempio di client: calcolatrice in notazione postfissa

Ricordiamo innanzitutto cosa si intende per espressione aritmetica in notazione postfissa. Per semplicità consideriamo solo operatori binari, ovvero che si applicano ad una coppia di operandi. Un espressione è scritta in notazione postfissa quando gli operatori seguono gli operandi cui si applicano (nella usuale notazione infissa, invece, gli operatori appaiono in mezzo agli operandi cui si applicano). Ad esempio, l'espressione in notazione postfissa `3 5 +` equivale all'espressione in notazione infissa `3 + 5`.

Con la notazione postfissa si eliminano i problemi dovuti alle parentesi e alla precedenza degli operatori (prima la divisione, poi l'addizione ecc.). Inoltre non c'è bisogno di annotare i risultati intermedi.

Per calcolare il valore di un espressione postfissa, si può usare un ciclo che esegue le seguenti azioni:

```
leggi un token (operatore o numero);
se il token è un numero
  inseriscilo nella pila;
se il token è un operatore
  estrai due valori dalla pila;
  applica ad essi l'operatore;
  inserisci il risultato nella pila;
```

Scrivete un file `calc.c` che implementa il precedente algoritmo, utilizzando una pila.

Per semplicità, assumiamo che i numeri abbiano al massimo 10 cifre. Dichiarate una variabile `char token[10]` nella quale memorizzare il token corrente, attraverso la funzione `scanf( "%s", token )`. Osservate che, per stabilire se `token` è un numero oppure un operatore, basta analizzare il suo primo carattere. Ricordate infine che la funzione `scanf` restituisce il numero di elementi letti e memorizzati, oppure `EOF` se si ha un errore o se l'input è finito: potete usare questa informazione per capire quando termine l'espressione di cui calcolare il valore.

Ricordate di includere all'interno del file `calc.c` il file di intestazione `stack.h`. Ricordate inoltre di compilate il vostro programma scegliendo una delle due implementazioni della pila, ad esempio con il comando

```
gcc -o calc_array calc.c stack_array.c
```

2 Generalizzazione a pile contenenti altri tipi di oggetti

L'interfaccia e le implementazioni della pila che avete preparato possono essere riciclate per risolvere altri problemi che necessitano dell'uso di una pila di interi. Non è detto però che tutte le pile debbano contenere interi: come generalizzare il codice in modo da poter gestire anche una pila di float, o stringhe, o strutture, o puntatori ad altri oggetti...?

Un metodo consiste nell'utilizzare un ulteriore file di intestazione `item.h` e di definire (tramite `typedef`) un tipo `Item`, rappresentante il generico oggetto contenuto nella pila. Il file `item.h` andrà incluso sia nel client che nel file di implementazione della pila `stack.c`.

Ad esempio, nel caso degli interi, useremo

```
typedef int Item;
```

mentre per i caratteri useremo

```
typedef char Item;
```

Naturalmente bisogna anche modificare l'interfaccia `stack.h` e l'implementazione `stack.c`, in modo da generalizzarle. Ad esempio, il prototipo della funzione push diventerà

```
void push( Item item );
```

A questo punto, modificare il tipo di contenuto della pila è facile: basta modificare la typedef del file `item.h`.

[Facoltativo] Funzioni per elaborare oggetti di tipo `item`

Il file `item.h` potrebbe contenere anche i prototipi di funzioni che elaborano oggetti di tipo `Item`. In questo caso, dovrà esistere anche un'implementazione dell'interfaccia `item.h` con le definizioni di queste funzioni.

Ad esempio, aggiungete in `item.h` la funzione

```
void print_item( Item item );
```

e scrivetene due implementazioni: la prima `item_int.c` per il caso in cui `Item` sia stato definito come intero. la seconda `item_char.c` per il caso in cui `Item` sia stato definito come stringa.

Ricordate di aggiungere anche l'implementazione di `item.h` quando compilate, ad esempio scrivendo

```
gcc calc.c stack.c item_int.c
```

[Facoltativo] Macro definita da linea di comando

Per non dover modificare a mano il file `item.h` a seconda del tipo di contenuto che ci serve, possiamo sfruttare l'opzione `-D` del compilatore gcc, che consente di definire una macro del preprocessore tramite lina di comando. Più precisamente, in `item.h` definiamo `Item` come segue:

```
typedef ITEMTYPE Item;
```

e in fase di compilazione definiamo il valore della macro `ITEMTYPE`. Ad esempio, se il file `test.c` usa una pila di stringhe, compileremo con il comando

```
gcc test.c stack.c item.c -DITMTYPE='char *'
```

2.1 Esempio di client: trasformazione da notazione infissa a notazione postfissa

La pila è utile anche per convertire un'espressione da notazione infissa (con parentesi che racchiudono ogni operazione) a notazione postfissa. In questo caso, la pila non conterrà i numeri, ma gli operatori, rappresentati da caratteri.

L'algoritmo è semplice:

```
leggi un token;
se il token è un numero
  stampa il token;
se il token è un operatore
  inserisci il token in cima alla pila;
se il token è una parentesi aperta
```

```
    ignora il token
  se il token è una parentesi chiusa
    estrai l'operatore in cima alla pila;
    stampalo;
```

Notate che gli operandi appaiono nello stesso ordine in entrambe le notazioni; inoltre si può osservare che le parentesi aperte non sono necessarie nella notazione infissa (lo sarebbero invece nel caso di operazioni tra più di due operandi).

2.2 Esempio di client: documenti html ben formati

Semplificando un po' le cose, in questo contesto chiamiamo documento html una sequenza di tag del tipo <a> (detti *tag di apertura*) oppure (detti *tag di chiusura*, dove a è una qualsiasi stringa di caratteri alfabetici.

Diciamo che un documento html è ben formato se i tag sono correttamente annidati, ovvero l'ordine dei tag soddisfa questi due criteri:

- per ogni tag di apertura esiste uno e un solo un tag di chiusura corrispondente
- se due tag di apertura compaiono in un determinato ordine, i corrispondenti tag di chiusura devono comparire nell'ordine opposto.

Ad esempio, la sequenza `<a> <b> </b> <c> <d> </d> </c> </a>` costituisce un documento html ben formato. Al contrario, la sequenza `<a> <b> </a> </c>` non lo è perchè il tag `<b>` non è mai chiuso e il tag `</c>` non è mai aperto. Analogamente, la sequenza `<a> <b> </a> </b>` non è un documento html perchè il tag `</a>` viene chiuso prima del tag .

Scrivete un programma che legga una sequenza di tag e stabilisca se costituisce un documento html ben formato oppure no. Per farlo, potete usare una pila contenente tag e fare riferimento a questo ciclo:

```
leggi un tag t;
se t è un tag di apertura
  inserisci t nella pila;
se t è un tag di chiusura
  estrai il tag t2 dalla cima della pila;
  se t e t2 non corrispondono
    il documento non ben formato;
```

3 Tipo di dato di prima categoria

In tutti gli esempi precedenti, il client utilizza una sola pila. Esistono ovviamente casi in cui avremo bisogno di avere a disposizione più di una pila, ovvero più istanze dello stesso tipo di dato che abbiamo chiamato pila. Parliamo in questo caso di tipo di dato *di prima categoria*.

Come fare per poter avere più istanze dello stesso tipo di dato? Scrivete l'interfaccia `Stack.h` a partire da `stack.h`, definendo nell'interfaccia il tipo di dato `Stack` e modificando i prototipi delle funzioni in modo da passare come argomento delle funzioni l'istanza su cui eseguire l'operazione. Inoltre, è necessario introdurre due nuove funzioni, una per costruire una nuova pila, l'altra per distruggerla.

Nota. Purtroppo, la definizione del tipo di dato `Stack` all'interno di `Stack.h` deve dipendere dall'implementazione della pila stessa e questo rende di fatto il tipo di dato non più astratto. L'interfaccia andrà modificata se scegliamo di usare una diversa implementazione. Per approfondire questo aspetto, si veda il paragrafo intitolato Information hiding.

Vediamo come fare nella pratica. Per quanto riguarda l'implementazione con array, nel file di intestazione `Stack.h` avremo questa definizione del tipo `Stack`:

```
typedef struct stack {
```

```
  Item content[N];
  int top;
} Stack;
```

Naturalmente, anche l'implementazione `Stack.c` va rifatta coerentemente.

Ricordate che il passaggio dell'argomento pila sarà sempre per valore, quindi se una funzione deve modificare la pila, sarà necessario passare come argomento il suo indirizzo, ovvero la funzione avrà come parametro un puntatore a `Stack`. Ad esempio, la funzione push diventerebbe

```
void push( Stack *s, Item item );
```

Per motivi di efficienza (evitare la copia di tutta la struttura), in tutti i prototipi si può scegliere di usare come parametri dei puntatori a `Stack`.

3.1 Esempio di client: intersezione di pile ordinate

Scrivete una funzione che, date in ingresso:

- due pile P e Q contenenti interi, tutti diversi tra loro e ordinati in ordine non decrescente (l'elemento più piccolo si trova in cima alla pila),
- una pila R vuota,

inserisca in R tutti e soli gli elementi che compaiono sia P che in Q, in modo tale che gli elementi di R risultino anch'essi tutti distinti e ordinati in ordine non descrescente.

Attenzione: al termine dell'esecuzione della funzione, le pile P e Q devono rimanere invariate!!

Facoltativo. Provate a scrivere la funzione precedente utilizzando come unica struttura dati d'appoggio una pila temporanea T, inizialmente vuota.

[Facoltativo] Information hiding e definizioni incomplete

Purtroppo, la definizione del tipo di dato `Stack` all'interno di `Stack.h` deve dipendere dall'implementazione della pila stessa e questo rende di fatto il tipo di dato non più astratto. Infatti, in questo modo anche il client conosce l'implementazione di `Stack` (attraverso l'inclusione di `Stack.h`) e potrebbe accedere al contenuto della pila direttamente, non soltanto attraverso le funzioni dell'interfaccia.

In questo caso si dice che non c'è *information hiding*. Il linguaggio C non è particolarmente adatto a gestire questo tipo di questioni, a differenza di quanto succede nei linguaggi di programmazione orientati ad oggetti. Tuttavia esiste la possibilità di dare *definizioni incomplete* di tipi e questo consente di risolvere il problema: nel file di intestazione `Stack.h` diamo una definizione incompleta del tipo, e completiamo la definizione all'interno dell'implementazione `Stack.c`.

Questo trucco consente di *nascondere informazioni* al client e tutelarsi da eventuali cambiamenti futuri nell'implementazione, dal momento che tutti i file che includono `Stack.h` potranno usare il tipo `Stack` e i prototipi dichiarati in `stack.h`, ma non avendo informazioni complete non potranno accedere direttamente alla struttura dati.

Vediamo come fare nella pratica.

Nell'interfaccia `Stack.h`, diamo una definizione *incompleta* del tipo `Stack` (che potremo usare nei prototipi):

```
typedef struct stack *Stack;
```

e la completiamo, definendo il tipo `struct stack`, solo nelle implementazioni `Stack.c`. Ad esempio, in `Stack_array.c` scriveremo

```
struct stack {
  int content[N];
  int top;
};
```

mentre in `stack_list.c` scriveremo

```
struct stack {
  Element *head;
  int count;
};
```

Attenzione: quando si usano tipi incompleti, bisogna fare particolare attenzione all'allocazione di memoria. Infatti non può essere il client ad allocare la memoria necessaria (non ha le informazioni per farlo), ma bisognerà fornire nell'interfaccia una funzione di costruzione e una di distruzione.