Scelte implementative per il progetto C++

Federico Salvioni nnn aaa.bbb@campus.unimib.it

2025-01-12

Introduzione

In questo documento sono riportate le scelte progettuali relative al progetto per l'esame di Programmazione C++, anno accademico 2024/2025, appello di Gennaio. Cosa contengono dei singoli file é riportato in `README.md`, mentre i dettagli di cosa fanno i singoli metodi (le eccezioni lanciate, le pre/post condizioni, ecc...) sono riportati nella documentazione generata mediante `doxygen`.

Sintassi

In linea con la sintassi in genere utilizzata nel codice C++, soprattutto nella libreria standard, é stato preferito lo *sneak case* (`hello_world`) al *camel case* (`helloWorld`).

Come viene spesso fatto nella programmazione ad oggetti, gli elementi atomici hanno un identificatore che inizia con la lettera minuscola, mentre gli oggetti container hanno un identificatore che inizia con la lettera maiuscola.

Per quanto riguarda la formattazione, non é stato seguito pedissequamente un coding style in particolare. Alcune ispirazioni, soprattutto il modo in cui sono scritte le funzioni, vengono da https://www.gnu.org/prep/standards/standards.html#Formatting. Altre idee vengono da https://google.github.io/styleguide/cppguide.html#Formatting.

Makefile

Il codice é compilabile mediante `Makefile`, che utilizza delle flag di base per istruire il compilatore nel mostrare quanti piú warning possibile. Per testare il codice é stato utilizzato `AddressSanitizer`, una funzionalitá presente nei moderni compilatori C++ che permette una ispezione ap-

profondita della presenza di memory leak e altri errori nella gestione della memoria dinamica. I flag per attivare tale funzionalitá sono ancora presenti nel `Makefile`, ma sono disabilitati per una questione di compatibilitá, dato ché non tutti i compilatori lo supportano. Se lo si desidera, é comunque riattivabile. Il codice é stato comunque testato con `Valgrind`, che ha segnalato la totale assenza di memory leak.

Struttura dati

La struttura dati che é stata usata per implementare lo stack é sostanzialmente un array statico `Items` di dimensione fissata `maximum_size`, corredato di un intero `top_pos` che indica la posizione corrente della cima dello stack.

Il valore di `top_pos` viene inizializzato a -1, anziché 0. Questa potrebbe apparire una scelta singolare, ma é necessaria per poter gestire il caso in cui lo stack é vuoto. Infatti, non é possibile[1] dichiarare un array statico vuoto. Inoltre, in questo modo, la posizione della cima dello stack segue quella di un normale array. Infatti, dato che ogni volta che si aggiunge un elemento allo stack il valore di `top_pos` aumenta di 1, uno stack che contiene un solo elemento ha la cima in posizione 0, che equivale ad avere un array che contiene un elemento solo.

`top_pos` parte da -1, mentre `maximum_size` parte da 0. Potrebbe essere quindi una scelta ragionevole dichiarare `maximum_size` come `unsigned`. Nonostante questo, entrambi sono dichiarati con lo stesso tipo `item_type` (`int`). Questo permette, nonostante tecnicamente ci sia un uso della memoria meno efficiente, di risparmiare molte annose conversioni di tipo, che fra i due sono fatte molto di frequente.

Eccezioni

La classe Stack utilizza estensivamente l'eccezione `std::bad_alloc` della libreria standard per segnalare la presenza di una allocazione di memoria fallita. É stata poi dotata di due eccezioni proprie, `Maximum_size_reached` e `Minimum_size_reached`, che rispettivamente segnalano l'aver raggiunto la massima/minima capienza dello stack e non é possibile proseguire oltre. Entrambe sono state implementate semplicemente come due `struct` vuote dato che non hanno particolari dati membro da fornire.

[1] Alcuni compilatori supportano delle estensioni che permettono di avere array statici di dimensione 0. Adottare questa scelta non é solo discutibile, ma é anche poco conveniente, perché non essendo parte dello standard rende il codice non portabile.

Costruttori

Oltre ai due costruttori principali (default constructor e copy constructor) ed al costruttore che ha due iteratori per input (requisito d'esame), ne é stato aggiunto un quarto che inizializza uno stack vuoto di una certa dimensione. Tale costruttore é dichiarato `explicit` per fare in modo che il compilatore non lo intenda come un cast implicito. Le celle vuote sono inizializzate con il valore di valore di default del tipo templato, quale che sia.

Metodi

Tutti i metodi della classe `Stack` sono pubblici, dato che non vi sono particolari problemi se usati su una istanza di `Stack` al di fuori della classe stessa.

Lo stack é stato innanzitutto dotato dei seguenti metodi di base:

- `push`, che aggiunge un elemento in cima allo stack;
- `pop`, che rimuove l'elemento sulla cima dello stack e lo ritorna;
- `peek`, che ritorna l'elemento in cima allo stack senza rimuoverlo;
- `stack_empty`, che ritorna se lo stack sia vuoto oppure no.

Sebbene non esista uno standard per quali debbano essere i metodi che una implementazione di uno stack deve fornire, quelle sopra citate sono le piú comuni. Inolre, con l'eccezione di `peek`, tali metodi figurano nell'implementazione dello stack della libreria standard del C++ (https://en.cppreference.com/w/cpp/container/stack).

Oltre a questi metodi di base sono stati introdotti i seguenti metodi di supporto:

- `size`, che restituisce la dimensione massima dello stack;
- `head`, che restituisce la posizione della cima dello stack;
- `wipe`, che cancella il contenuto dello stack e lo ridimensiona a 0.

Si é cercato di evitare di introdurre ogni possibile metodo, preferendo invece dotare la classe del minimo numero possibile di metodi che fossero effettivamente utili. Ad esempio, per conoscere il numero di elementi al momento presenti nello stack `S`, anziché introdurre un metodo `S.current_capacity()` é sufficiente calcolare `S.head() + 1`.

Infine, i seguenti metodi sono stati introdotti perché requisiti d'esame:

- `load`, che carica lo stack a partire da una coppia di iteratori ad una sequenza;

- `clear`, che svuota lo stack del suo contenuto ma lascia intatta la sua dimensione;
- `filter_out`, che rimuove dallo stack tutti gli elementi che non rispettano un predicato.

Tutti i metodi hanno un assert che confronta `top_pos` con `maximum_size` per assicurarsi che il primo sia minore del secondo, dato che non potrá mai verificarsi una situazione dove questo non accade. Nei costruttori questo assert é assente perché irrilevante, dato che i valori vengono automaticamente inizializzati.

Iteratori

Come iteratore (sia in lettura che in lettura/scrittura) é stato scelto il `forward iterator`. Questo perché é l'iteratore che meglio si allinea con il modo in cui si accede agli elementi di uno stack, dato che lo stack puó crescere in una sola direzione.

Il metodo `begin()` restituisce un iteratore che punta al primo elemento dell'array interno allo stack, che corrisponde all'elemento "in fondo" allo stack. D'altra parte, l'elemento `end()` restituisce un iteratore che punta all'elemento immediatamente successivo alla posizione della cima dello stack. Questa é una scelta consapevole, perché un iteratore di fine conforme allo standard C++ deve puntare al $N + 1$-esimo elemento di una sequenza lunga N.

Tecnicamente, gli elementi di uno stack ad eccezione di quello in cima non dovrebbero essere accessibili, ma il supporto agli iteratori era un requisito d'esame.

Funzioni globali

La classe `Stack` é stata dotata di una funzione globale `transform` (requisito d'esame) che applica una certa operazione ad ogni elemento di uno stack. Inoltre, é stato ridefinito l'operatore `<<` per poter stampare a schermo il contenuto dello stack senza dover accedere ai suoi dati interni.

La stampa mediante `<<` restituisce i valori all'interno di una coppia di parentesi quadre; se quel valore é oltre `top_pos`, le parentesi quadre non racchiudono nulla. Lo stack é restituito in orizzontale anziché in verticale per una semplice questione di leggibilitá.

Essendo funzioni globali e non metodi di classe, sia `operator<<` che `transform` accedono al contenuto dello stack mediante iteratori. Questo permette sia di evitare un *coupling* non necessario fra la classe `Stack` e tali funzioni, sia di avere maggiore flessibilitá, di modo che se la strut-

tura interna della classe dovesse cambiare le funzioni non debbano venire aggiornate di conseguenza.