Monaco Filippo

840089

filippomonaco@outlook.com

RELAZIONE PROGETTO C++

22/02/2022

Strumenti Utilizzati

- Ubuntu 20.04.3 LTS
- gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
- Valgrind-3.15.0
- Doxygen 1.8.17

Descrizione Progetto

Il progetto richiede l'implementazione e realizzazione di una classe che implementa un Set (una collezione di elementi unici) di elementi generici T.

L'ordine delle sezioni di questa relazione non segue l'ordine presente nel codice, ma ho scelto di seguire un ordine logico per facilitare la lettura e la comprensione delle mie scelte implementative.

Implementazione Set:

La classe template Set richiede il tipo di dato generico **T**. La comparazione tra due elementi di tipo T (per mantenere la struttura del Set) viene fatta attraverso il funtore **Equals**, passato come parametro templato durante la creazione del Set.
La scelta di usare un funtore di comparazione e non di usare l'operatore == definito su tipi T ricalca l'implementazione della classe set della libreria standard (set - C++ Reference (cplusplus.com)). Inoltre, rende impossibile in principio creare un Set su tipi T senza definire un comparatore tra elementi di tipo T.

Come struttura dati per l'implementazione del Set, ho scelto di usare una **Linked List**, e di gestire la richiesta di unicità degli elementi a livello programmativo.

Ci sono diversi modi per implementare un Set (Set Implementations (The Java™ Tutorials > Collections > Implementations) (oracle.com)).

A livello di prestazioni la mia scelta non è la più efficiente, ma a livello di implementazione risulta più semplice e diretta.

Altre scelte avrebbero solo reso più complicata sia l'implementazione che l'utilizzo da parte dell'utente.

Per esempio, lo HashSet sarebbe stato molto più efficiente, ma avrebbe richiesto la specifica di una HashFunction per ogni tipo di dato T passato al Set.

Una scelta come il BST (Binary Search Tree) invece non avrebbe funzionato perché avrebbe richiesto un predicato di confronto ulteriore all'uguaglianza (come < e >) per poter dividere gli elementi nei diversi rami dell'albero.

Linked List

Ho quindi scelto di usare una Linked List per implementare il Set, composto da **nodi** (implementati come struct).

Ogni **nodo** della lista è composto da un valore (**val**) e da un puntatore al successivo nodo della lista(**next**). L'ultimo nodo ha come campo next un nullptr, e quindi rappresenta la coda della lista.

L'ordine degli elementi nella lista non è significativo, ma semplicemente è dato dal momento di aggiunta degli elementi all'interno del Set.

Per esempio, risultano essere equivalenti i seguenti Set

A = {1, 5, 4, 0}

B = {0, 4, 1, 5}

In quanto contengono gli stessi (unici) elementi.

L'unicità degli elementi all'interno del Set è garantita a partire dalla funzione **add**.

Funzione add

La funzione **add**() aggiunge elementi al Set.

Per garantire l'unicità degli elementi aggiunti al Set (più specificamente, i loro valori), la funzione add chiama a sua volta la funzione **find_internal()** per verificare che non sia già presente un elemento con medesimo valore.

Come spiegato successivamente, per come è stata implementata find_internal, prima di chiamarla devo necessariamente controllare manualmente che il valore cercato non sia presente nella **_head** del Set.

Se l'elemento da aggiungere all'interno del Set non è già presente, allora questo viene aggiunto in testa al Set. Questa scelta è puramente implementativa, in quanto (come specificato in precedenza) l'ordine degli elementi all'interno del Set non è rilevante.

Funzione remove

La funzione **remove()** elimina un elemento dal Set, se presente. Ritorna true se l'elemento è presente e viene cancellato, false altrimenti.

Come per la funzione **add()**, la funzione remove() chiama la funzione **find_internal()** per verificare che un elemento con medesimo valore sia effettivamente presente nel Set.

Come spiegato successivamente, per come è stata implementata find_internal, prima di chiamarla devo necessariamente controllare manualmente che il valore cercato non sia presente nella **_head** del Set.

Se l'elemento cercato è presente nel Set (sotto forma di nodo, con valore equivalente a quello passato come parametro), questo viene cancellato, la memoria deallocata, e la struttura di Linked List viene preservata, collegando il nodo precedente (A) a quello eliminato (D) con quello successivo (B).

A --- D --- B -> A --- B

Funzione find e find_internal

La funzione **find** fa una ricerca per vedere se un elemento (internamente un nodo) è già presente all'interno del Set. La comparazione tra i valori dei due elementi viene effettuata tramite il funtore templato Equals.

La ricerca del Set necessita quindi di una visita intera della **Linked List** (tempo di visita O(n)), e viene fatta meditante la funzione di supporto **find_internal**().

Per come ho implementato la **Linked List** (facendo uso solo del puntatore next, e non di un puntatore previous), la funzione di ricerca necessariamente deve restituire il puntatore al nodo precedente a quello trovato.

Questa necessità deriva dal fatto che se si vuole eliminare un nodo (D), bisogna collegare il nodo precedente (A) con quello successivo (B).

A --- D --- B -> A --- B

Lasciando però pubblica una funzione tale, si esporrebbe la struttura dati nodo privata alla classe, ritornando un puntatore ad un tale elemento.

Per risolvere questo problema ho creato due funzioni: **find** e **find_internal**.

La prima è pubblica e ritorna semplicemente un bool nel caso in cui l'elemento cercato compaia all'interno del Set.

La seconda è privata, e viene usata proprio per la cancellazione e l'aggiunta di nuovi elementi all'interno del Set.

Un altro problema che è sorto da questa implementazione della funzione **find_internal**, è il fatto che la testa della lista ovviamente non ha nessun nodo precedente. Per raggirare questo problema ho scelto di lasciare al chiamante la responsabilità di controllare che l'elemento che sta cercando non sia proprio nella testa della lista. Questo è un'ulteriore ragione per la creazione di un'interfaccia pubblica per la funzione **find**.

Dati membro Set

Il Set è quindi implementato come una Linked List di nodi.

Il primo **nodo** è salvato in una variabile puntatore **_head**.

La variabile **_size** memorizza la dimensione attuale del Set, e viene esclusivamente utilizzata per semplificare dei controlli durante le operazioni. Se per esempio devo verificare

l'uguaglianza tra due Set posso subito controllare che abbiano _size uguale. Se così non fosse, rifiuterei subito l'uguaglianza, senza bisogno di ulteriori calcoli.

copy constructor

Poiché il **copy constructor** fa uso della funzione **add**() per aggiungere elementi nel nuovo Set, e poiché (come descritto nella sezione sulla funzione **add**()) gli elementi vengono aggiunti in cima alla lista, il Set che viene ritornato avrà ordine degli elementi opposto rispetto a quello di partenza.

A = { a b c d}

B = { d c b a}

La proprietà di unicità degli elementi dei due Set viene comunque rispettata, e i due insiemi sono equivalenti.

operator==

Come specificato in precedenza, due Set possono essere equivalenti anche se internamente presentano un ordine diverso di elementi. Ciò che conta è la presenza di tutti e soli gli stessi elementi in entrambi i Set.

Per questo motivo, per l'implementazione dell'overloading dell'**operator==** ho dovuto fare utilizzo della funzione find_internal, per ricercare ogni elemento del primo Set nel secondo.

L'operatore è implementato come membro della classe (e non globale) per evitare che ci sia una conversione implicita a Set.

operator=

L'operatore di assegnamento **=** utilizza al suo interno un **copy constructor**, e, per come è stato implementato, inverte gli elementi del Set.

Rappresentazione Set su console

Come rappresentazione del Set su console (mediante operatore << per lo standard output) ho deciso di usare la notazione matematica di insieme, ovvero

{a, b, c, d}

const_iterator

Come iteratore per il Set ho scelto di utilizzare un forward_iterator, in quanto la struttura interna implementata (Linked List), è composta da nodi con solo un puntatore next.

Gestione Eccezioni

Per la gestione delle eccezioni ho deciso di implementare due semplici classi custom:

- **myexcp_domain_error** (derivante dalla classe standard std::domain_error)
- **myexcp_out_of_range** (derivante dalla classe standard std::out_of_range)

che si limitano a chiamare il costruttore della rispettiva classe madre con un messaggio custom.

%%% bad_alloc %%%

Le eccezioni di **bad_alloc** sono lanciate da qualunque funzione chiami **add**, in quanto è l'unica funzione che chiama new (e quindi richiede allocazione di memoria esplicitamente). Queste funzioni sono: copy constructor, constructor con iteratori, e le tre funzioni globali filter_out, operator+, e operator-.

%%% myexcp_domain_error %%%

Le eccezioni di tipo **myexcp_domain_error** sono lanciate dalla funzione dell'operator[] quando lo si cerca di invocare su un Set vuoto e dalle funzioni degli iteratori quando si cerca di muoverli o di accedere a dati puntati da un nullptr.

%%% myexcp_out_of_range %%%

Le eccezioni di tipo **myexcp_out_of_range** sono lanciate esclusivamente dalla funzione dell'operator[] quando si cerca di accedere ad indici invalidi (indici negativi o indici superiore alla dimensione del Set).

Funzioni Globali

Per l'implementazione delle tre funzioni globali (filter_out, +, -) ho fatto uso dei const_iterators.
Il Set ritornato da queste funzioni è passato come copia, per rispettare il principio RAII (Resource Acquisition Is Initialization), e quindi evitare di passare pointers o references per oggetti (in questo caso Set) con scope locale (all'interno della funzione).

filter_out

La funzione filter_out() filtra da un Set passato come parametro tutti e soli gli elementi che verificano un certo predicato P.
Ciclando sul Set di partenza usando gli iteratori della classe stessa, ogni elemento viene passato attraverso il predicato P e, se verificato, viene aggiunto al Set che verrà ritornato al termine della funzione.

operator+

L'operator+ fa la concatenazione tra due Set, ovvero crea un nuovo Set contenente tutti e soli gli elementi che sono presenti in un Set o nell'altro. Ciò implica che se un medesimo elemento è presente in entrambi i Set, verrò riportato una sola volta nel Set di output.
Il Set di output inizialmente è riempito con tutti gli elementi del secondo Set (scelta arbitraria), e successivamente vengono aggiunti tutti gli elementi del primo Set, utilizzando gli iteratori della classe. La proprietà di unicità degli elementi nel Set viene rispettata dalla funzione add().

operator -

L'operator- fa l'intersezione tra due Set, ovvero crea un nuovo Set contenente tutti e soli gli elementi comuni ai due Set.
Il primo Set viene ciclato utilizzando gli iteratori della classe e, utilizzando la funzione find() (interfaccia pubblica della funzione find_internal()), se trova lo stesso elemento nel secondo Set, questo viene aggiunto al Set di output.

Testing

Per il testing della classe templata Set ho creato diverse funzioni con numerosi tipi di dati. In particolare, ho testato:

- Set di interi
- Set di interi costanti (per testare che la proprietà read-only fosse soddisfatta)
- Set di floating point
- Set di stringhe
- Set di bool
- Operazioni su Set vuoti
- Set di person (tipo custom basato su una struct)
- Set di Set di interi
- Interfaccia degli iteratori su Set

Per ciascun tipo di dati ho dovuto creare un funtore di uguaglianza e un funtore di predicato da passare alla funzione filter_out.

In ciascuna funzione di testing vengono quindi chiamate tutte le funzioni dell'interfaccia pubblica della funzione Set, e le funzioni globali filter_out, + e -.

In particolare, vengono chiamati:

- **default constructor,** per testare la creazione di nuovi Set
- **add()**, per testare l'aggiunta di nuovi elementi all'interno del Set e per verificare la proprietà di unicità degli elementi all'interno del Set
- **remove**(), per testare la cancellazione degli elementi dal Set e per verificare la corretta deallocazione di memoria
- **clear()**, per verificare la cancellazione di tutti gli elementi del Set e per verificare la corretta deallocazione di memoria
- **copy constructor**, per testare la copia di un Set in un altro
- **operator=**, per testare l'assegnamento di un Set da un altro
- **constructor con coppia di iteratori**, per testare la copia di un Set in un altro con gli iteratori della classe
- **operator[]**, per testare l'accesso agli elementi del Set
- **operator==**, per testare il confronto tra due Set
- **filter_out**, per testare la funzione filtro di Set con Predicati definiti dall'utente
- **operator+**, per testare la concatenazione tra due Set
- **operator-**, per testare l'intersezione tra due Set
- **find()**, per testare la funzione pubblica di ricerca all'interno del Set

Per la classe const_iterator vengono invece chiamati:

- **default constructor**, per testare la creazione di nuovi iteratori
- **begin()** e **end()**, per testare l'assegnamento e il funzionamento degli iteratori
- **operator***, per testare il deferenziamento
- **operator+**, per testare il pre-incremento
- **operator+(int)**, per testare il post-incremento
- **copy constructor**, per testare la copia di un iteratore in un altro
- **operator=**, per testare l'assegnamento di un iteratore da un altro
- **operator== e operator!=**, per testare il confronto tra due iteratori
- **operator->**, per testare l'accesso a dati membro di una classe/struct