Relazione progetto C++ Febbraio 2019

Nome: Alessandro

Cognome: Sorrentino

Matricola: 815999

Mail: a.sorrentino10@campus.unimib.it

Introduzione:

La richiesta era implementare un Set templato, ovvero una collezione di dati di tipo generico T che non fossero ripetuti.

Ho deciso di rappresentare il Set come una serie di celle collegate fra di loro (linked list) ed ogni cella contenente una coppia di valori.

L'implementazione di una struttura dati composta da celle collegate è stata decisa in base alla comodita durante l'inserimento o rimozione dei vari elementi. Avrei potuto utilizzare un array ma sarebbe stato molto scomodo e, immaginando la presenza di un numero elevato di elementi presenti nel Set, copiare tutto l'array in un altro ogni volta che veniva apposta una modifica sarebbe sicuramente risultato dispendioso sia dal punto di vista dello spazio che del costo computazionale.

Variabili membro:

Ogni cella (nodo) contiene una coppia di valori contenuti in una struct chiamata *"node"* .

Ogni nodo contiene:

- Valore: rappresenta l'effettivo valore contenuto nella collezione dati
- Next: puntatore alla cella successiva così da avere una visione contigua dei dati presenti

Sono presenti anche un puntatore alla testa della collezione (head), così da avere una referenza al primo elemento della lista nel caso la si voglia scorrere, ed una variabile size (accessibile dall'esterno con l'apposito metodo getter) così da tenere conto della dimensione della Set.

Essendo la classe templata si è rivelato necessario procedendo con lo sviluppo del progetto l'aggiunta di un nuovo parametro templato che gestisse la comparazione di eguaglianza fra dati generici T (un funtore) perchè altrimenti non saprei come confrontare due dati presenti nella collezione.

Metodi standard:

Come consono, sono stati implementati I 4 metodi basici che ogni classe deve definire.

Ogni ***nodo*** è dotato di costruttore primario e secondario di comodo allo sviluppo della classe. I nodi sono infatti inaccessibili all'esterno di questa.

Costruttore di default permette di inizializzare una Set vuota quindi con size a zero e puntatore a NULL.

Il ***copy constructor*** permette di creare una Set partendo da una Set già esistente. Di questo ne viene copiata la dimensione e tutti I dati membro a membro. Nel caso non si riesca ad allocare memoria per un nodo lo stato della Set originale viene preservato.

L'***operatore di assegnamento*** = permette di assegnare una collezione ad un altra tramite la ridefinizione dell'operatore = . Di fatto l'operatore di assegnamento non fa altro che condividere I nodi di una Set con una seconda Set condividendo il puntatore alla head e copiandone il valore di size.

Il ***distruttore*** elimina ogni allocazione nello heap dei nodi in sequenza fino a svuotare la collezione. Viene infine aggiornato il parametro di size ed il puntatore di testa viene messo a NULL così da non puntare a dati inconsistenti.
Viene utilizzato un metodo di supporto chiamato ***clear***.

Implementazione dei requisiti funzionali:

Requisiti funzionali:

1. random access operator

2. add new values to the set

3. remove values from the set

4. const iterators

5. set secondary constructor from iterators

6. print operator

7. filtering function

8. set concatenation

1. Il primo metodo che ho implementato è stato il ***random access operator***.
   Questo operatore permette l'accesso diretto ai dati mediante un indice, simulando così una locazione contigua di memoria. L'implementazione scelta, essendo la struttura di base formata da una lista di nodi collegati, è stata quella di partire dal primo elemento in testa alla lista e scorrere un certo numero di nodi corrispondente all'indice utilizzato

utilizzando il puntatore alla cella successiva contenuto nella cella corrente. Sono stati fatti preventivamente dei controlli su un possibile out of bound.

2. L'inserimento di un nuovo dato all'interno della collezione avviene tramite il metodo ***add***. L'aggiunta avviene sequenzialmente e sempre in coda visto che non è richiesto un ordine particolare. Viene scorsa quindi prima tutta la collezione fino all'ultimo elemento. Viene creato un nuovo nodo usando il costruttore secondario contenente il valore da aggiungere.
   L'inserimento deve essere tuttavia coerente con la defizione di Set. Vengono fatti dei controlli su un possibile duplicato all'interno del Set utilizzando il predicato templato di eguaglianza. In tal caso viene lanciata un'eccezione che segnala il tentativo di inserire un duplicato. In caso di esito negativo lo stato della collezione rimane invariato. Se invece l'aggiunta di un nuovo valore alla collezione ha esito positivo viene aggiornata la dimensione della lista e viene anche aggiornato il puntatore all'ultimo elemento così da puntare al nuovo dato aggiunto.
   È stata gestita anche la casistica di aggiunta dati in testa (Set vuoto).

3. La rimozione di un elemento dalla collezione avviene mediante la chiamata al metodo ***remove***. La collezione viene scorsa sequenzialmente finchè non si trova l'elemento da dover eliminare. Anche qui si è rivelato fondamentale (come nella add) l'introduzione di un nuovo parametro templato che serve come predicato per il confronto fra due dati generici contenuti in una Set. Se l'oggetto non viene trovato viene lanciata un eccezione. Sono stati gestiti tutti I casi di rimozione in testa, nel mezzo ed in coda.
   Per ovviare il problema della rimozione in coda ho tenuto in puntatore locale al metodo chiamato prev che puntava sempre al nodo precedente in cui si era arrivati scorrendo la lista.

4. Gli iteratori per iterare l'intera collezione sono stati implementati costanti come indicato. Gli iteratori che ho scelto di implementare sono stati I ***const forward iterator*** perchè per le operazioni che dovevo implementare mi sembravano più che sufficienti.
   Questi permettono lo scorrimento unidirezionale in avanti del Set.
   Gli iteratori così definiti sfruttano semplicemente la definizione di puntatori. Una volta associato il puntatore degli iteratori alla head, lo scorrimento è analogo a quello di tutti gli altri metodi definiti in precedenza, dunque andando di cella in cella seguendo il puntatore successivo presente nel nodo.

5. Una volta implementati gli iteratori mi è stato possibile definire un ***costruttore secondario*** che prende come parametri due iteratori. Per ogni elemento referenziato dagli iteratori viene chiamata in supporto la ***add*** che aggiunge l'elemento corrente al Set.

6. La ridefinizione dell'***operatore di stampa*** << templato permette di stampare in output il contenuto di una Set. Essendo l'operatore di stampa appartenente ad una classe esterna a quella corrente, il metodo è stato dichiarato fuori dalla classe. Questo mi impediva di scorrere la Set utilizzando il puntatore alla testa non avendo accesso ai nodi e ai dati membro privati della classe. Avrei potuto dichiarare il metodo friendly alla classe ma ho preferito utilizzare gli iteratori che avevo appena definito così da poter scorrere la lista.

7. La funzione ***filter_out*** è un metodo templato che permette di generare una nuova set partendo da un'altra già esistente e filtrando tutti I valori che non soddisfano un certo predicato.
   La funzione templata prende 3 parametri templati di cui terzo è il predicato booleano P responsabile per il filtraggio dei valori all'interno della Set.
   Viene scorsa la Set originale e ne viene creata una aggiungendo man mano tutti I valori che non soddisfano il predicato P.
   Ritorna il Set generato all'interno del metodo per copia.

8. La concatenazione è gestita mediante la ridefinizione dell'***operatore di concatenazione +*** che permette, date due Set di partenza, di generarne una nuova contenente sia I dati della prima che della seconda.
   La nuova set viene creata inizialmente per copia utilizzando la prima set da concatenare. Dopodichè viene scorsa la seconda Set da concatenare e viene chiamata la ***add*** su tutti I valori rimanenti.
   Anche questa funzione è templata sul tipo di dato contenente la Set e il predicato di eguaglianza e ritorna il nuovo Set generato per copia.

File di test (main):

Ho effettuato due funzioni di test differenti per testare la mia classe.
La prima comprende test su dati primitivi, ovvero interi.
La seconda invece comprende test su dati custom, in particolare ho utilizzato la classe *"complex"* facente parte della standard library anch'essi templati (ho utilizzanto dei double).

Sono stati definiti nel main I due funtori responsabili per la filter out (come spiegato in precedenza, questi sono I due predicati booleani).
I due funtori sono: ***is_odd*** relativo a tipi di dati primitivi interi e ritorna true se il numero è pari.
L'altro è ***is_major_complex*** relativo a tipi di dati custom, più specificatamente numeri complessi, e ritorna true se la componente numerica reale è maggionre della componente numerica immaginaria.

È stato definiti anche il funtore ***is_complex_equal*** responsabile per l'eguaglianza fra due dati di tipo complesso.
Per quanto riguarda gli interi nei testi con dati primitivi ho utilizzato il funtore facente parte della standard library.

Per le eccezioni ho utilizzato eccezioni della libreria standard, in particolare la runtime_error.
Per controllare la correttezza nei test ho usato le asserzioni.