SCALCO DANIELE 844762 danielescalco@hotmail.it

RELAZIONE PROGETTO C++

INTRODUZIONE

La traccia del progetto richiede la costruzione di un multinsieme di elementi di un tipo generico T, ovvero un insieme che però può contenere elementi duplicati.

STRUTTURA MULTISET

Dal momento in cui è espressamente richiesto di non dover memorizzare direttamente i duplicati, ma soltanto di tenerne traccia ho pensato di realizzare la classe *MultiSet* che rappresenta un multinsieme come una lista di elementi in cui ogni elemento ha l’attributo *\_quantita* che dice il numero occorrenze di quell’elemento presenti nel multinsieme. Sebbene fosse scritto che l’ordine degli elementi non è rilevante ho deciso di tenere ordinati gli elementi, dal minore al maggiore, in modo da rendere molto più semplice e veloci alcune delle funzioni richieste, ovvero la ridefinizione dell’operatore di uguaglianza e il metodo *contains()* , e per maggiore chiarezza.

Sebbene non espressamente richiesti, la classe MultiSet è templata oltre che sul tipo T de valore degli elementi del multinsieme anche per i funtori di confronto e uguaglianza ritenuti utili, se non necessari, per la realizzazione di tutte le funzionalità richieste.

ITERATORE

La traccia prevedeva l’implementazione di un iteratore di sola lettura, perciò ho implementato esclusivamente un *const\_iterator* e la tipologia che ho scelto è quella *forward* perché è in grado di fornire tutti i supporti necessari alle implementazioni delle varie funzioni richieste. Oltre alle classiche ridefinizioni degli operatori tipici di un iteratore forward ne ho aggiunta una, quella dell’operatore *!* poiché, visto che non sono effettivamente mantenuti in memoria gli elementi duplicati, avevo la necessità di dover sapere la quantità di tali elementi restituita per l’appunto dall’operatore *!*. In questo modo sono riuscito ad implementare l’operatore di stream in modo da rappresentare ogni elemento con la sua quantità.

FUNZIONI

La funzione *aggiungiElemento(const T &e)*, che permette l’aggiunta di un singolo elemento al multinsieme, utilizza la funzione privata di supporto *aggiungiElemento(const T &e, int q)* più generica che permette invece l’aggiunta al multinsieme di più elementi uguali allo stesso tempo. Sebbene quest’ultima non fosse espressamente richiesta è stata implementata poiché ritenuta utile in quanto semplifica e velocizza l’assegnamento e la costruzione per copia dei multinsiemi per come li ho strutturati io.

Per simmetria ho deciso di implementare con lo stesso principio la funzione opposta *rimuoviElemento(cont T &e)* che utilizza la funzione privata di supporto *rimuoviElemento(const T &e, int q).*

MAIN

Nel main sono chiamate quattro metodi che testano le varie funzioni del file *multiset.h*:

* *testMultiSetInt()*: test che prova le funzionalità per un MultiSet templato su un *int*, un tipo primitivo;
* *testMultiSetPunto2D()*: test che prova le funzionalità per un MultiSet templato su un *Punto2D*, un tipo custom realizzato da me mediante una *struct* che rappresenta un punto in due dimensioni;
* *testCostruttoreIteratori()*: test che prova il costruttore templato *MultiSet(I begin, I end)* che permette la costruzione di un multinsieme con elementi di tipo T a partire dagli iteratori di inizio e fine di una sequenza di un generico tipo Q;
* *testConstMultiSetInt()*: test che prova le funzionalità per un MultiSet costante templato su un *int*;

Nei vari test sono stati usati gli *assert* per verificare il corretto funzionamento delle funzioni

VALGRIND

Ho eseguito una prova per visualizzare eventuali memory leak utilizzando Valgrind con la shell del sottosistema Linux per Windows con le direttive di compilazione *-O3* e *-DNDEBUG* e non risultano problemi come mostrato nell’immagine seguente:

Immagine che contiene testo, elettronico, screenshot, schermo

Descrizione generata automaticamente