Relazione di progetto C++ del 27/01/2026

Bellucci Edoardo - matricola 899795 - e.bellucci1@campus.unimib.it

Introduzione

Il progetto consiste nell'implementazione di una classe template `set<T>`, che rappresenta un insieme di elementi generici senza duplicati. La classe supporta operazioni di inserimento, rimozione, accesso in sola lettura agli elementi, confronto tra set, iterazione e funzioni globali aggiuntive richieste dalle specifiche.
Nel file `main.cpp` sono presenti test automatici che verificano il corretto funzionamento di tutte le funzionalità implementate.

Struttura della classe

La classe template `set<T>` rappresenta un insieme di elementi unici di tipo generico T. L'implementazione utilizza una struttura dati basata su una lista concatenata semplice, in modo da occupare memoria solo per gli elementi effettivamente presenti nel set.

Ogni elemento è memorizzato in una struttura interna `node`, che contiene: il valore di tipo T e un puntatore al nodo successivo.

La classe ha:

- un puntatore `_head` al primo nodo del set
- una variabile `_size` che rappresenta il numero di elementi contenuti nel set.

Il tipo T deve supportare l'operatore di uguaglianza `operator==`, utilizzato per evitare duplicati e per confrontare gli elementi.

Per quanto riguarda i costruttori, sono disponibili i seguenti:

- **Costruttore di default:** che inizializza un set vuoto.
- **Costruttore di copia:** che crea una copia profonda di un altro set, inserendo uno a uno tutti gli elementi presenti nel set sorgente. Implementazione protetta da gestione delle eccezioni.
- **Costruttore da intervallo di iteratori:** consente di creare un set a partire da una coppia generica di iteratori. Gli elementi vengono inseriti utilizzando la funzione `add(const T &value)`, garantendo automaticamente l'assenza di duplicati e lasciando al compilatore la gestione della compatibilità tra i tipi.

La classe presenta alcune funzioni aggiuntive per l'accesso e la manipolazione dei dati:

- `size()` restituisce il numero di elementi presenti nel set
- `contains(const T &value)` verifica la presenza di un elemento all'interno del set
- `clear()` svuota completamente il set, liberando tutta la memoria allocata
- `swap(set &other)` scambia il set corrente con quello passato come parametro

- `remove(const T &value)` rimuove un valore dal set, aggiorna anche l'intero set dopo la rimozione
- `add(const T &value)` aggiunge un nuovo elemento al set.

Per quanto riguarda gli operatori:

- `const T& operator[](unsigend int i) const` : accesso in sola lettura, all'i-esimo elemento.
- `T& operator[](unsigned int i)` : accesso in lettura e scrittura, non richiesto dal progetto, ma implementato per sicurezza.
- `operator+` : implementato come funzione globale. Restituisce un nuovo set contenente tutti gli elementi presenti in almeno uno dei due set passati come parametro. Ritorna un nuovo set, aggiunge gli elementi tramite iteratori costanti.
- `operator-` : implementato come funzione membro della classe set. Restituisce un nuovo set contenente gli elementi presenti sia nel set corrente sia nel set passato come parametro. L''aggiunta avviene tramite iteratori costanti.
- `operator==` : operatore di uguaglianza tra due set. Implementato come funzione membro. Due set sono considerati uguali se hanno la stessa size e se hanno gli stessi elementi indipendentemente dall'ordine.
- `operator=` : operatore di assegnamento.
- `operator<<(std::ostream &os, const set<T> &s)` : operatore di output per il set, definito come funzione globale.
- `operator==(const Attivita &a, const Attivita &b)` : operatore di uguaglianza per due Attivita. Operatore implementato per tipo custum Attivita, definito in una struct. Due attivita sono uguali se hanno lo stesso titolo, ora di inizio e di fine.
- `operator<<(std::ostream &os, const Attivita &a)` : operatore di stream di output per oggetto Attivita.

La funzione globale e generica `filter_out` permette di creare un nuovo set contenente solo gli elementi di un set di partenza che soddsfano un determinato predicato.
La funzione riceve: un set s di tipo set<T> e un predicato P, implementato come funtore, ovvero un oggettp che ridefinisce l'operatore `()`. La funzione scorre il set utilizzando un `const_iterator`, e per ogni elemento applica il predicato. Se il predicato restituisce true, l'elemento viene inserito nel set risultato tramite `add()`, che garantisce l'assenza di duplicati. Sono stati definiti, a titolo d'esempio, i funtori I`sEven` e `IsOdd`.

Supporto agli iteratori

La classe set<T> fornisce il supporto all'iterazione tramite due classi interne: `iterator e const_iterator`, entrambe implementate come **iteratori di tipo forward**. Gli iteratori permettono di scorrere sequenzialmente gli elementi del set, senza esporre direttamente la struttura interna. L'iteratore `iterator` consente l'accesso in lettura e scrittura agli elementi, mentre `const_iterator` permette esclusivamente l'accesso in sola lettura. Entrambi implementano gli operatori di deferenziazione, incremento (pre e post) e confronto. La classe mette a disposizione le funzioni `begin()` e `end()` sia in versione costante che non, permettendo l'uso degli iteratori anche in contesti `const`.

Per quanto riguarda la gestione delle eccezioni, nella classe set<T> l'accesso agli elementi tramite indice, mediante **`operator[]`**, è protetto da un controllo sui limiti e, in caso di indice non valido, viene sollevata un'eccezione di tipo **`std::out_of_range`**.

Test della classe

Nel file main.cpp è stata implementata una serie di test automatici per verificare il corretto funzionamento della classe set<T> e di tutte le funzionalità richieste dalle specifiche. Ogni test è strutturato in modo da mostrare a schermo le operazioni eseguite e successivamente verificarne il comportamento tramite l'uso di **`assert()`**.
I test coprono:

- costruttori e operatore di assegnamento
- inserimento, rimozione e accesso agli elementi
- confronto tra set tramite **`operator==`**
- utilizzo degli iteratori
- funzioni globali (**`operator+, operator-, filter_out`**)
- utilizzo del set con tipi base e con un tipo custom (**`Attivita`**)
- salvataggio e caricamento da file tramite le funzioni **`save e load`**.

Screen di Valgrind

```
 *** TUTTI I TEST PASSATI CORRETTAMENTE ***

==11733==
==11733== HEAP SUMMARY:
==11733==     in use at exit: 0 bytes in 0 blocks
==11733==   total heap usage: 91 allocs, 91 frees, 118,794 bytes allocated
==11733==
==11733== All heap blocks were freed -- no leaks are possible
==11733==
==11733== For lists of detected and suppressed errors, rerun with: -s
==11733== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Il programma è stato testato con valgrind senza rilevare memory leak.

Progetto Qt: relazione e scelte implementative

Ho utilizzato la versione 6.8.1 della libreria Qt (progetto svolto sulla macchina virtuale messa a disposizione dall'università).

Per rappresentare il tipo di un'attività è stato utilizzato un `enum class` chiamato `TaskType`, che può
assumere i valori `Event` e `Activity`. Rispetto a un `enum` tradizionale, rende il codice più sicuro. I valori
dell'enumerazione devono essere sempre usati specificando il nome dell'`enum` (`TaskType::Event`).

La struttura `Task` rappresenta il modello di una singola attvità/evento all'interno dell'applicazione. Ho usato
una `struct` anzichè una classe perchè un `Task` è un contenitore di dati, senza logica complessa, e poi
verrà usato come elemento in una collezione, molto più semplice da gestire.

`selectedDate` : memorizza la data attualmente selezionata nel calendario
`tasksByDate` : struttura dati principale dell'applicazione.

In particolare, `tasksByDate` è una `QMap<QDate, QList<Task>>` che associa a ogni data l'elenco delle attività previste per quel giorno.

La funzione `refreshTable` aggiorna la tabella delle attività in base alla data selezionata. All'inizio la tabella viene svuotata e vengono recuperate tutte le attività associate al giorno passato come parametro. Successivamente le attività vengono ordinate in base all'orario di inizio, così da essere visualizzate in ordine cronologico durante la giornata. Dopo l'ordinamento, la funzione inserisce una riga per ogni attività nella tabella e ne visualizza le informazioni principali. Questa funzione viene richiamata ogni volta che cambia la data selezionata o quando l'elenco delle attività viene modificato.

La funzione `onSaveTaskClicked` gestisce il salvataggio di una nuova attività. Vengono letti dall'interfaccia l'orario di inizio e, se presente, l'orario di fine, verificando che non esistano sovrapposizioni con altre attività della stessa giornata. Per quanto riguarda gli eventi, non vengono presi in considerazione nel controllo di sovrapposizione; evento e attività possono sovrapporsi. In caso di conflitto viene mostrato un messaggio di avviso e il salvataggio viene annullato. In assenza dell'orario di fine, l'attività è considerata puntuale (scelta progettuale). Non è stata implementata la gestione di attività a cavallo di due giorni, scelta progettuale ritenuta non necessaria ai fini del progetto. Se i controlli hanno esito positivo, viene creato un nuovo Task, salvato su file e la tabella delle attività viene aggiornata.

La funzione `onUpdateTaskClicked` permette la modifica di un'attività esistente. L'attività da aggiornare viene individuata tramite un indice impostato alla selezione di una riga della tabella; se non è selezionata alcuna attività la funzione termina. I dati dell'attività vengono aggiornati con i nuovi valori inseriti dall'utente, salvati su file e la tabella viene ricaricata.
La funzione `onTableSelectionChanged` gestisce l'abilitazione dei pulsanti di modifica ed eliminazione in base alla presenza di una selezione nella tabella.
La funzione `onDeleteTaskClicked` consente l'eliminazione di un'attività selezionata: dopo aver verificato la validità della selezione, l'attività viene rimossa dalla lista del giorno corrente, i dati salvati su file e la tabella aggiornata.

La funzione `onTableItemClicked` viene eseguita quando l'utente seleziona una riga della tabella delle attività. L'attività corrispondente viene recuperata, l'indice della riga salvato e i campi dell'interfaccia grafica vengono compilati automaticamente per permetterne la modifica.
La funzione `esisteSovrapposizione` verifica se una nuova attività si sovrappone temporalmente ad altre attività della stessa giornata, confrontando gli intervalli di tempo;

le attività senza orario di fine sono considerate puntuali. Gli eventi, come detto in precedenza, non sono sottoposti a controllo.

La funzione `salvaSuFile` gestisce la persistenza dei dati salvando tutte le attività in un file di testo, creando il file se non esiste e scrivendo ogni attività su una singola riga.
La funzione `caricaDaFile` ripristina lo stato dell'applicazione all'avvio leggendo il file riga per riga, ricostruendo gli oggetti Task e inserendoli nella struttura dati alla data corretta.

Il progetto è stato inoltre compilato sia in modalità release e debug e analizzato tramite il Valgrind Memory Analyzer di Qt creator, non riscontrando nessun leak. (solo issues).