Relazione Progetto queue

Per implementare la classe queue (coda FIFO), ho deciso di definirla con due parametri template: T per il tipo generico dei valori ed E per il funtore di uguaglianza che può essere deciso dall’utente (necessario per il metodo contains).

# element

La struttura interna di supporto è element, definita con due dati membro:

* Il valore dell’elemento.
* Il puntatore all’elemento successivo.

Nel costruttore di default, i dati membro sono inizializzati tramite initialization list, per migliorare l'efficienza dell'inizializzazione. Il valore dell'elemento vuoto è inizializzato tramite il costruttore di default del tipo T. Il puntatore all'elemento successivo della coda è inizializzato a nullptr.

Nel costruttore secondario, i dati membro sono inizializzati tramite initialization list, per migliorare l'efficienza dell'inizializzazione. Il valore dell'elemento è inizializzato al valore v di tipo T passato come parametro al costruttore. Il puntatore all'elemento successivo della coda è inizializzato a nullptr.

Nell’altro costuttore secondario, i dati membro sono inizializzati tramite initialization list, per migliorare l'efficienza dell'inizializzazione. Il valore dell'elemento è inizializzato al valore v di tipo T passato come primo parametro al costruttore. Il puntatore all'elemento successivo della coda è inizializzato al valore n passato come secondo parametro al costruttore.

Nel distruttore, viene chiamato automaticamente il distruttore del valore dell'elemento, corrispondente al distruttore del tipo T. Al puntatore all'elemento successivo della coda viene assegnato il valore di default nullptr.

Gli altri metodi fondamentali (Copy Constructor e operatore di assegnamento) coincidono con quelli di default, pertanto è inutile e ridondante specificarli.

# Dati membro

I dati membro della coda sono un puntatore alla testa (necessario per la rimozione in tempo costante), un puntatore alla coda (per l’inserimento in tempo costante), la dimensione e il funtore per il confronto di uguaglianza (per contains).

# clear e clear\_helper

Queste due funzioni private vengono utilizzate sia dal distruttore, sia in alcuni casi di eccezione di allocazione di memoria.

clear rimuove tutti gli elementi della coda. La rimozione viene effettuata ricorsivamente, chiamando la funzione helper privata ricorsiva clear\_helper(element \*e), a partire dall'elemento in testa alla coda. Al puntatore alla coda viene assegnato il valore di default nullptr (poiché, al termine dell'esecuzione della funzione helper ricorsiva, punta ad una locazione di memoria ora non più valida).

clear\_helper rimuove ricorsivamente gli elementi della coda a partire da un elemento specificato attraverso il puntatore ad element passato come parametro (compreso). Ogni volta che la funzione viene chiamata ricorsivamente, vengono svolte le seguenti operazioni: la funzione rimuove prima gli elementi successivi all'elemento da cui partire, richiamando la stessa funzione ricorsivamente; dopodiché, la memoria allocata dall'elemento da cui partire viene deallocata e al puntatore all'elemento da cui partire viene assegnato il valore di default nullptr (poiché l'elemento a cui puntava è stato rimosso, evitando, così, che il puntatore punti ad una locazione di memoria non più valida); infine, il numero di elementi inseriti nella coda viene decrementato di un'unità.

# Costruttore di default

I dati membro sono inizializzati tramite initialization list, per migliorare l'efficienza dell'inizializzazione. Il puntatore all'elemento in testa alla coda vuota è inizializzato a nullptr (la coda vuota non ha alcun elemento alla propria testa). Il puntatore all'elemento in coda alla coda vuota è inizializzato a nullptr (la coda vuota non ha alcun elemento alla propria coda). Il numero di elementi inseriti nella coda vuota è inizializzato a 0 (la coda vuota non presenta alcun elemento al suo interno).

# Costruttore di copia

Prima di effettuare la copia dei dati da una coda all'altra, i dati membro della coda da istanziare (ovvero la coda corrente) sono inizializzati con valori di default tramite initialization list: il puntatore all'elemento in testa alla coda è inizializzato a nullptr; il puntatore all'elemento in coda alla coda è inizializzato a nullptr; il numero di elementi inseriti nella coda è inizializzato a 0. Dopodiché, i valori degli elementi della coda da copiare vengono inseriti singolarmente, uno alla volta, nella coda corrente, mantenendo l'ordine originale (i puntatori alla testa e alla coda della coda vengono aggiornati all'interno del metodo di inserimento dei singoli elementi).

Nel caso in cui, durante l'inserimento degli elementi nella coda corrente, venga lanciata un'eccezione di allocazione di memoria, quest'ultima viene gestita eliminando completamente la coda corrente, insieme ai suoi singoli elementi (per evitare memory leak), e l'eccezione viene propagata alla funzione chiamante. Pertanto, spetta anche all'utente gestirla adeguatamente nella funzione chiamante.

Per operatore di assegnamento, distruttore, size, le due versioni di enqueue e dequeue, guardare la documentazione nel codice.

Per l’accesso in lettura e scrittura agli elementi in testa e in coda, ho implementato dei semplici getter e setter, che lanciano un’eccezione se la coda è vuota.

contains utilizza il funtore di uguaglianza \_equals di tipo E per confrontare il valore passato e quelli presenti nella coda.

# Iteratori

Per gli iteratori, ho deciso di implementare sia quelli in lettura e scrittura, sia quelli costanti (dato che, su code const, gli iteratori normali non possono essere istanziati).

Gli iteratori sono di tipo forward, in quanto seguono la logica FIFO di ritornare i valori in un certo ordine (quindi non possono né accedere randomicamente ai valori, né procedere bidirezionalmente).

Se gli iteratori sforano le locazioni di memoria della coda, viene lanciata un’eccezione apposita, che impedisce di accedere a delle locazioni non valide e sconosciute.

# transformif

Per la funzione globale, ho deciso di sfruttare gli iteratori in lettura e scrittura per verificare il predicato e per sovrascrivere gli elementi con il risultato dell’operazione data dall’operatore sui valori stessi.

# Operatore di stream

Non era richiesto, ma per questioni di test ho ritenuto necessario implementarlo.

# main

Il file sorgente contenente la funzione main contiene vari tipi custom, con relativi operatori di stream e costruttori e funtori di uguaglianza, ma anche predicati e operatori appositi per i vari tipi. La funzione di test è templata, permettendo di scriverla una sola volta e di istanziare facilmente tutti gli elementi da testare in funzioni a parte.