Programmazione C++

Coppola Gianluca 845120 g.coppola15@campus.unimib.it

Il progetto richiede la realizzazione di un grafo orientato, i cui nodi sono rappresentati da un generico identificativo.
Ho deciso di rappresentare i nodi con un array _nodes di tipo generico T, e quindi per effettuare il confronto tra due tipi T ho definito un funtore Equal.
Il funtore deve implementare un operator() const per permettermi di confrontare i tipi generici di dato T.
Gli archi del grafo sono implementati da una matrice bidimensionale _edges di boolean, che rappresenta la matrice d'adiacenza del grafo.
Ho anche scelto di implementare due variabili unsigned int, una per conoscere il numero di nodi nel grafo (_sizeN) e un'altra per il numero di archi (_sizeE), ed una variabile _eq che indica il funtore Equal.
Oltre i metodi richiesti dalla classe ho scelto di implementare tre metodi ausiliari:

1. Init(size):
   Mi permette di inizalizzare un grafo di dimensione data, e allocare la memoria richiesta. Crea un array _nodes di dimensione size, setta il contatore di nodi _sizeN a size e inoltre si occupa di settare la matrice d'adiacenza a false e il contatore di archi _sizeE a 0.
2. Reset():
   Questo metodo si occupa di deallocare tutta la memoria allocata per il grafo, e impostare le variabili a default. Effettivamente è ciò che farebbe il distruttore, e per questo nel metodo distruttore mi limiterò a chiamare questo metodo di reset.
3. Find(node):
   Questo metodo serve a trovare l'indice nell'array _nodes di un determinato nodo. Ritornerà -1 se il nodo non è stato trovato.

La classe grafo implementa i 4 Metodi fondamentali, ovvero il Costruttore di default che crea un grafo vuoto, il Copy Constructor che mi permette di instanziare un grafo contenente i valori di un altro grafo passato come input, l'operatore di assegnamento operator= che mi permette di copiare il contenuto di un grafo in un altro grafo e infine il destructor che dealloca la memoria occupata dal grafo.
Questi quattro metodi possiedono una stampa di debug della loro firma, per effettuare questa stampa è richiesto il flag C_DEBUG durante la compilazione del file.

Per gestire i vari problemi risultanti da input errati nei metodi add_node, remove_node, add_edge e remove_edge ho deciso di implementare due Eccezioni custom, DuplicateNodeException per l'aggiunta di nodi già esistenti e NonExistentNodeException per le operazioni su nodi non esistenti. Nei metodi add_node e remove_node è presente un blocco try-catch per evitare di modificare il grafo in caso di un lancio di eccezione.

I metodi add_node e remove_node agiscono in modo simile: spostano il grafo su una variabile temporanea tmp, resettano il grafo, e vanno a ricopiare il grafo tmp su un nuovo grafo, aggiungendo o rimuovendo il nodo passato come parametro a seconda del metodo usato.

I metodi add_edge e remove_edge, dopo essersi accertati di operare su nodi validi, andranno ad aggiungere o rimuovere un arco su di essi.

La classe implementa inoltre un const_iterator di tipo forward, che itera sugli identificativi dei nodi contenuti nel grafo.

Nel main ho effettuato vari metodi per il test tramite asserzioni dei metodi della classe graph di 3 tipi: int, string, e una classe Custom "Persona". Tutti e tre i tipi possiedono una struct contenente il loro funtore Equal.

Progetto Qt: **(Versioni: Qt Creator 4.14.2, Qt 6.0.3)**

Il Progetto consiste nella creazione di un programma per la risoluzione automatica di un Sudoku. Per fare ciò, ho iniziato creando una griglia 9 x 9 (suddivisa visivamente in box 3x3) di QLineEdit. Ho scelto questo input widget per vari motivi:

- Le celle devono contenere poca informazione, solamente un numero.
- Dispone di una proprietà isModified(), che mi permette di capire quando un utente ha modificato un QLineEdit al momento della risoluzione del sudoku e di trattarlo come un dato "Fixed".
- Tramite il QRegularExpression e QValidator posso impostare per ogni cella un "Validator" che permette come input solo i valori di una Regular Expression (in questo caso i numeri da 1 a 9).

Ho deciso di creare una matrice di QLineEdit* (_cells) per permettermi di indirizzare ogni QLineEdit con una riga e una colonna, e una matrice di int (_sudo) per risolvere il sudoku. Avere due matrici separate una per la risoluzione e una per il display mi aiuta quando dovrò andare a mostrare il sudoku step-by-step.
Per entrambi le matrici ho utilizzato la classe QVector, poichè mi permette di non preoccuparmi personalmente della gestione della memoria.

Ho creato dei pulsanti per la gestione del sudoku:

- *Back*: Mostra lo step precedente della soluzione (Se presente).
- *Forward*: Mostra lo step successivo della soluzione (Se presente).
- Pulsante "<<": Mostra il primo step della soluzione.
- Pulsante ">>": Mostra l'ultimo step della soluzione.
- Reset: Pulisce ogni QLineEdit e ne permette la modifica, resetta la matrice _sudo a 0, e disabilita i pulsanti per rivisitare le soluzioni.

Il pulsante risolvi imposta tutte le QLineEdit readOnly, se contengono un dato “Fixed” vado a scriverlo nella matrice _sudo, altrimenti imposto la QLineEdit con colore rosso per distinguere i dati “Fixed” dalle soluzioni.
Prima di iniziare con l’algoritmo backtracking per la risoluzione (solve()) controllo che i dati inseriti dall’utente siano validi, e ritorno un messaggio di errore se non lo sono.

Per effettuare questo controllo ho scritto la funzione isValid(int digit, unsigned int row, unsigned int col). La funzione mi dice se il valore è già presente in quella riga, colonna o nel suo box 3x3.

Dopodichè procedo con l’algoritmo ricorsivo solve(): L’algoritmo prova ad inserire in ogni cella che non sia “fixed” un numero a partire da 1, controllando che sia valido e procedendo ricorsivamente sulla cella successiva. Se il numero non è valido, procedo con il successivo, fino a 9. Se tutti i valori da 1 a 9 non sarano validi in una cella, l’algoritmo uscirà dal for e ritonerà false, terminando la risoluzione. Se riesce a scorrere tutta la matrice (row == 9) l’algoritmo terminerà restituendo true in output. A questo punto posso stampare il sudoku e abilitare i pulsanti per ripercorrere la soluzione “step-by-step” (ammesso che ci siano step).

Ho deciso di creare una variabile di campo int step e fare un overload del metodo print con parametro (int step), che mi permette di inserire un int corrispondente ad uno step (step di ogni cella = (i*9) + j) e quindi mostrare il sudoku step by step tramite i pulsanti back e forward. I pulsanti back, “<<” verranno disabilitati quando non esiste uno step precedente, mentre forward e “>>” verranno disabilitati quando non esiste uno step successivo.

Ho anche creato due funzioni nextStep(int step) e prevStep(int step) che restituiscono lo step precedente della soluzione, saltando ogni cella contenente un dato “Fixed”.