Relazione Progetto
Programmazione ad oggetti
a.a. 2018/19

QTravel

Battilana Jacopo
Matricola N.1162149

Ambiente di sviluppo:

- Compilatore: GCC 7.3.0, 64bit
- Versione Qt Creator: 4.9.2
- Sistema operativo: Windows 10 Pro, Version 1903, 64bit

Materiale:

- Cartella *QTravel* contenente tutti i file *header* e *cpp*; il file *QTravel.pro* per la compilazione del progetto; la cartella i*cons* con le icone della GUI
- file *esempio.xml*: file contenente dei dati di esempio in XML da importare nel programma
- file *relazione.pdf*: relazione del progetto

Compilazione ed esecuzione:

Accedere da terminale alla cartella QTravel,
eseguire il comando: *qmake QTravel.pro.* Questo genererà il file Makefile
quindi eseguire il comando *make* per creare i file oggetto e l'eseguibile *QTravel*

Introduzione:

L'applicazione QTravel propone la gestione di un registro di viaggi, composto da viaggi memorizzati in un *Container* definito dal sottoscritto, come da specifica del progetto. Le funzioni messe a disposizione dell'utente sono:

- creazione, eliminazione, modifica dei viaggi, i quali possono essere effettuati in macchina, treno o in bici;
- ricerca di una stringa di testo al loro interno;
- salvataggio e importazione di file database, con una formattazione XML

Il progetto è stato ideato con una logica MVC (Model-View-Controller) infatti, si è cercato di mantenere la parte del model(dati) separata dalla view(GUI). La comunicazione tra i due è gestita dal controller.
Il tempo richiesto complessivamente dal progetto è stato approssimativamente di 52 ore:

- Analisi preliminare: 2 ore
- Progettazione modello e GUI: 6 ore
- Apprendimento e prove del framework Qt: 13 ore, distribuite tra tutorato e documentazione durante le fasi di codifica
- Codifica modello e GUI: 20 ore, di cui una buona parte impiegata per la cura dell'aspetto grafico
- Debugging e testing: 11 ore compreso il miglioramento e revisione del codice

Si ritiene non sia necessario un manuale utente, vista la semplicità d'uso della GUI.

1. Container:

Come richiesto, è stato realizzato un template di classe Container<T>, che permette la gestione di un insieme di elementi di tipo T.

1.1 TripVector

La classe *TripVector<T>* è composta da un vettore di tipo *T**, una *size* (che indica il numero di elementi nel vettore) e una *capacity*, che indica la reale dimensione con cui è allocato il vettore.

Si è scelto di raddoppiare la capacity qualora diventasse insufficiente per contenere gli elementi del vettore, così da limitare le copie del vettore all'inserimento: il metodo *doubleCapacity* esegue questa funzione. Il container mette inoltre a disposizione due iteratori: *iterator* e *const_iterator* che contengono un puntatore agli elementi (sono completi di costruttore, operatori di uguaglianza, disuguaglianza, somma, differenza, incremento, decremento e dereferenziazione) e servono per una gestione più ad alto livello del contenitore.

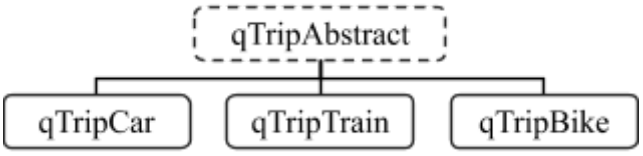
I metodi *begin* e *end* restituiscono l'iteratore rispettivamente al primo e al past-the-end elemento del container.

Altri metodi messi a disposizione dal container sono: *getSize*, *isEmpty*, *pop_back*, *push_back*, *erase*, *operator[]* con il comportamento atteso; metodo *search* che ricerca un elemento T nel container e in caso di match ne ritorna l'iteratore; metodo *insert*, che inserisce un elemento *T* nella posizione puntata da un iteratore passatogli.

Si è scelto inoltre di creare due enum per un'eventuale gestione delle eccezioni: *EmptyContainerException* e *NullPointerException*.

2. Model:

Di seguito sono elencate le classi del modello (parte logica), e le principali funzioni offerte.
Si è scelto di non realizzare una classe Smart Pointer. Infatti nel progetto non vengono fatte copie dei viaggi, inoltre il problema del garbage a seguito dell'eliminazione di un viaggio è risolto nel metodo *QTravel::remove* richiamando la delete sul puntatore corrispondente.



2.1 qTripAbstract

La classe qTripAbstract rappresenta un viaggio astratto. È una classe virtuale pura, quindi non istanziabile.

Contiene il testo del *titolo,* della *partenza* e dell'*arrivo* in delle stringhe*;* un campo *Data* di tipo QDate (si è scelto di utilizzare il metodo QDate per pura comodità, mentre per garantire una completa separazione da modello a view si sarebbe potuto usare i metodi della libreria standard di C++, ma avrebbero comportato una considerevole modifica di alcuni frammenti di codice); un campo per la *durata* del viaggio e uno per la *distanza* percorsa di tipo *Double*.

La classe presenta due costruttori, di cui uno richiede il campo durata in secondi invece che in ore, minuti e secondi come da costruttore normale, in modo da poter costruire il viaggio a partire dal testo prelevato da un file di salvataggio. Dispone inoltre di un metodo *clone* virtuale puro che restituisce una copia della nota, un metodo

virtuale puro *type* che restituisce un *enum type,* un metodo virtuale puro *calcolo* che restituisce un *double* e opportuni metodi get e set..

2.2 qTripCar

È una classe che eredita da qTripAbstract. Rappresenta un viaggio effettuato in macchina.

Ai campi dati di qTripAbstract aggiunge un booleano che rappresenta il tipo di carburante che la macchina usa (diesel o benzina) e i relativi metodi get e set. Sono definite le classi pure *clone*, *type e calcolo*.

2.3 qTripTrain

È una classe che eredita da qTripAbstract. Rappresenta una viaggio effettuato in treno.

Ai campi dati di qTripAbstract aggiunge un booleano che rappresenta il tipo di treno impiegato (Ad alta velocità o regolare) e i relativi metodi get e set. Sono definite le classi pure *clone*, *type e calcolo*.

2.4 qTripBike

Eredita dalla classe qTripAbstract . Rappresenta un viaggio in bicicletta.

Ai campi dati di qTripAbstract aggiunge un Double che rappresenta il numero di calorie consumate per kilometro e i relativi metodi get e set. Sono definite le classi pure *clone, type e calcolo.*

2.5 QTravel

La classe QTravel rappresenta l'intero registro di viaggi, memorizzato in un *TripVector* di puntatori polimorfi a *qTripAbstract*. Mette a disposizione costruttore di default, distruttore profondo (che richiama il metodo *clearTrips* per eliminare la memoria puntata dai puntatori del container), metodi di aggiunta e rimozione, metodo *getAll Trips* che restituisce tutti i viaggi in sola lettura, metodo *trip* che restituisce il viaggio all'indice passato, in lettura/scrittura e metodo *getSize*.

3. Controller:

La classe controller gestisce tutte le richieste che vengono lanciate dalla view, tramite degli handler del modello:

3.1 Controller

Contiene un puntatore a *QTravel*, uno a MainWindow (la classe principale della view) e un *TripVector* di puntatori a *tripWidgets* (cioè l'integrazione grafica di un singolo viaggio).

Il costruttore (che è chiamato dal *main*) crea questi oggetti e avvia la *MainWindow*. I metodi *getTrips* e *trip* sono handler dei rispettivi metodi nella classe *QTravel*. La classe contiene un metodo *calcolo* che fa una stima del costo di tutti i viaggi presenti nel registro e delle calorie consumate nei viaggi in bici; gli slot *add* e *remove* che agiscono sia sul model che sulla view; uno slot *search* che controlla per ogni viaggio nel model, la presenza di una stringa nei campi testuali del viaggio: se non matcha, il viaggio viene nascosto dalla view; gli slot *save* e *load* che eseguono il salvataggio/apertura su file usando la classe *XmlController*.

4. View:

La GUI è stata realizzata senza l'ausilio del tool QtDesigner, ma richiamando testualmente gli oggetti del framework Qt necessari, in modo da evitare l'autogenerazione, e avere così un maggior controllo sul codice eseguito.

L'interfaccia principale è la *MainWindow*, la quale contiene i pulsanti, le barre, e un insieme di *TripWidget* che rappresentano ognuno un singolo viaggio. Tramite il pulsante di inserimento, verrà aperta una finestra ausiliaria (NewTripDialog) con la funzione di creare e aggiungere (sia al model che alla view) un nuovo viaggio con i dati inseriti dall'utente.

4.1 MainWindow

La classe *MainWindow* eredita dalla classe di Qt *QWidget*, ed è la finestra principale della GUI.

Ha come fields privati un puntatore al *Controller* e un punatore al *layout* in cui vengono visualizzati i viaggi. Il costruttore imposta tutto ciò che riguarda la parte grafica (menu bar, pulsanti, layouts, ecc..), poi connette ogni azione al relativo slot. In particolare: *addTripWidget* aggiunge il *TripWidget* passatogli al *layout*; *newTrip* avvia un *QDialog* per l'inserimento di un nuovo viaggio, e se cliccato il pulsante di conferma, aggiunge tramite

il *Controller* il nuovo viaggio creato; *save* e *load* aprono un nuovo *QFileDialog* per la selezione del percorso del file, e avviano i relativi slot di I/O nel *Controller*.

4.2 TripWidget

La classe *TripWidget* eredita da *QWidget*, e rappresenta la GUI di un viaggio. I principali fields privati sono: il puntatore al viaggio di riferimento di tipo *qTripAbstract*, l'indice della nota all'interno del *Container* (il quale si è mostrato necessario per poter eliminare una nota da un pulsante sulla stessa), il puntatore al *Controller* e un *unsigned int* che rappresenta il *tipo* di viaggio (Necessario per poter aprire una corretta view per la modifica). I metodi *carTrip*, *trainTrip*, *bikeTrip* servono solo per separare la parte del viaggio standard (il titolo, la partenza, l'arrivo, la data, la distanza e la durata del viaggio) dalla parte caratterizzante per il tipo di viaggio (il tipo di carburante, il tipo di treno o le calorie per km) eseguendo uno static_cast al tipo giusto.

Il costruttore di *TripWidget* crea la GUI: contiene le varie *QLabel* con le informazioni, un bottone per l'eliminazione e uno per la modifica che alla pressione richiamano i rispettivi metodi del controller. Tramite uno switch-case sul tipo, viene invocato il giusto metodo per visualizzare il "corpo" della nota. Lo slot edit() serve a richiamare un *EditTripDialog* tramite il quale viene modificato un viaggio. Il campo rappresentante il *tipo* serve a creare adeguatamente questa finestra.

4.3 NewTripDialog e EditTripDialog

La classe *NewTripDialog* eredita da *QDialog*. Viene invocata al momento della creazione di un nuovo viaggio: permette di selezionare il tipo di viaggio tramite una *QComboBox* e uno *QStackedWidget* e di inserire i dati necessari alla creazione. I suoi campi sono: un *int* che rappresenta il tipo di viaggio(si è scelto di usare un intero per facilitarne l'integrazione con gli indici all'interno dello StackedWidget – questo field è mantenuto aggiornato tramite una connessione tra il signal *activated* della *QComboBox* e lo slot *tabChanged*), i campi per la memorizzazione dei dati relativi al viaggio. Il costruttore crea la GUI, composta da: delle *QLineEdit* per il titolo, la partenza, l'arrivo, la distanza e la durata (divisa in ore, minuti e secondi), un oggetto *QComboBox* che serve per cambiare la visualizzazione di uno QStackedWidget, le cui pagine contenenhono gli oggetti specifici per l'inserimento: le pagine per la macchina e il treno aggiungono due *QRadioButton* ciascuno per selezionare la variante desiderata, mentre la pagina per la bici aggiunge un ulteriore *QLineEdit* per impostare le calorie consumate al Km. Infine, i due bottoni *Ok* e *Cancel*, sono collegati agli slot di accettazione e chiusura del *Dialog*. A creare il viaggio è il metodo *getTrip*, chiamato nella *MainWindow* una volta ricevuta la conferma dal *Dialog*: con uno switch-case sul tipo, viene creato il viaggio con i campi dati precedentemente compilati. Il nuovo viaggio è un puntatore a *qTripAbstract*, che punta dinamicamente al tipo di viaggio corretto.

La classe *EditTripDialog* eredita da *QDialog.* Viene invocata alla pressione del *editBtn* su un *TripWidget* e permette di modificare i valori di un viaggio. Sia i campi che la GUI sono molto simili a *NewTripDialog* con la differenza che viene passato al costruttore il viaggio del *TripWidget* da modificare e il tipo di viaggio corrispondente, in modo da poter creare in modo adatto la visualizzazione (ovvero con i *QRadioButton* o il *QLineEdit* a seconda del tipo di viaggio) e inserirne i dati già esistenti.

4.4 XmlController

La classe XmlController ha il compito di gestire il salvataggio/l'importazione su/da file XML. L'unico campo dati privato è un puntatore al controller, necessario per prelevare e memorizzare i dati sul modello. Il costruttore quindi crea solo questo field di copia.

Il metodo *save* chiede come parametro un *QString* contenente il percorso del file XML su cui salvare. Crea uno stream output *QXmlStreamWriter* con il file. Quindi usa uno switch-case per identificare il tipo dei viaggi e memorizza sullo stream i dati prelevati dal controller.

Il metodo *load* chiede come parametro un *QString* contenente il percorso del file XML da aprire. Crea uno stream input *QXmlStreamReader* con il file. Leggendo i tag di apertura, ne verifica la corretta formattazione. In caso di successo, crea i viaggi e li aggiunge tramite il *Controller*. In caso di insuccesso di uno o più passaggi, viene saltato il tag corrente, e si passa alla verifica del successivo.