Project 2

Mastermind - Bulls and Cows Game

CIS 17C Section 48942

Name: Marlo Zeroth

Date: 12/10/2015

**Introduction**

Title: Bulls and Cows Game

Mastermind is a game that had its origins in Bulls and Cows. A code breaking game similar to developed earlier. When Mastermind begins the player is presented the option of choosing among easy, medium and hard difficulty. The difficulty determines playing with or without duplicates animals in the puzzle and the length of code. The code is a sequence of animals icons either generated by the computer or another player. Cows are animals that are in the puzzle, but not in the right spot. Bulls are animals that are in the puzzle in the right spot. The player can enter letters from 4 to 8 animals depending on the length of the code chosen. After the player enters his/her choices for the game the program creates a random code for the player to break. The player can enter up to 10 guesses to win the game. If an animal of the guess is in the puzzle, but not in the right spot the program shows the number of cows to the screen. If the player entered an animal that is in the puzzle in the right position, the number of bulls is shown to the screen. If no animals of the guess are in the puzzle 0 is printed for cows and also 0 for bulls. If the players runs out of turns before the game ends, the player looses and the computer or player 2 wins. The player is given the choice of starting another game after he/she breaks the code or runs out of guesses. The game is fun to play and I wanted to see if I could create a program that could take images and take the user input and check it against the code the computer chose at random.

**Summary**

Project size:

Number of variables: About 60 major variables. The most important where in my classes. I used a lot of QMap<String> types to calculate whether a code has duplicates, to generate the code, and calculate whether a guess and the puzzle had similar animals and if they were in the correct position. Hashing was used for the login dialog, although it is still not able to connect remotely to the server database. Trees were used to keep track of player game statistics. The most important class was the list class I created followed by the QMap and Qset classes. The most important object is the playSpace instantiated as part of the main window. This is where the user interacts with the game.

Number of methods: 69

I implemented most of the concepts we have learned up to hashing. The game still needs to be completed. It still needs the ability to connect with another player remotely and control and sink the input and output by both players. The game reads from files to load the image paths for the icons and uses a value given to each icon as a key value pair in all Qmaps. The save and load game functions have not been correctly implemented.

The program itself heavily uses a lot of the concepts for linked lists and templates. The guess interface is held in a linked list. The individual puzzle animals are held in a Qmap. I used several of the concepts from lectures and the online book to implement lists, stacks, queues etc for use some areas of the game. The online book on algorithms was particularly useful for templating code.

**Description**

The main point of the game is how it functions when a player enters a guess and how the program delivers output that the players can use to base their next guess.

**Pseudo Code**

Initialize

Get user input for game play choices for code length, allow duplicates, and player role.

Set size of puzzle and guess to size specified by the difficulty given by player.

If the player chooses to break the code with four and checks no duplicates

Generate a puzzle that is a length of 4 icons with no duplicates in the answer

Else if the player chooses generate the code with duplicates

The player will drag the indicated number of icons in to the window. The program will check for duplicates in the players puzzle.

Else

Generate a puzzle of the chosen length and selected option for duplicates.

Begin game loop

If player is breaker – they enter guesses into the main window.

Compare guess to puzzle.

If guess matches puzzle exactly

Show the puzzle to the screen, the number of guesses, and a message that player won.

Else if guess does not match answer and guesses are less than 10

Print out dummy puzzle answer made up of ?s, print out cows and bulls for guesses,

Go next loop iteration for next guess

Else

Show puzzle, print out cows and bulls for guesses, print guesses,

and message for better luck next time.

End game loop and ask user if they want to start a new game.

If user chooses to play again

Start new game

Else

Exit the game loop.

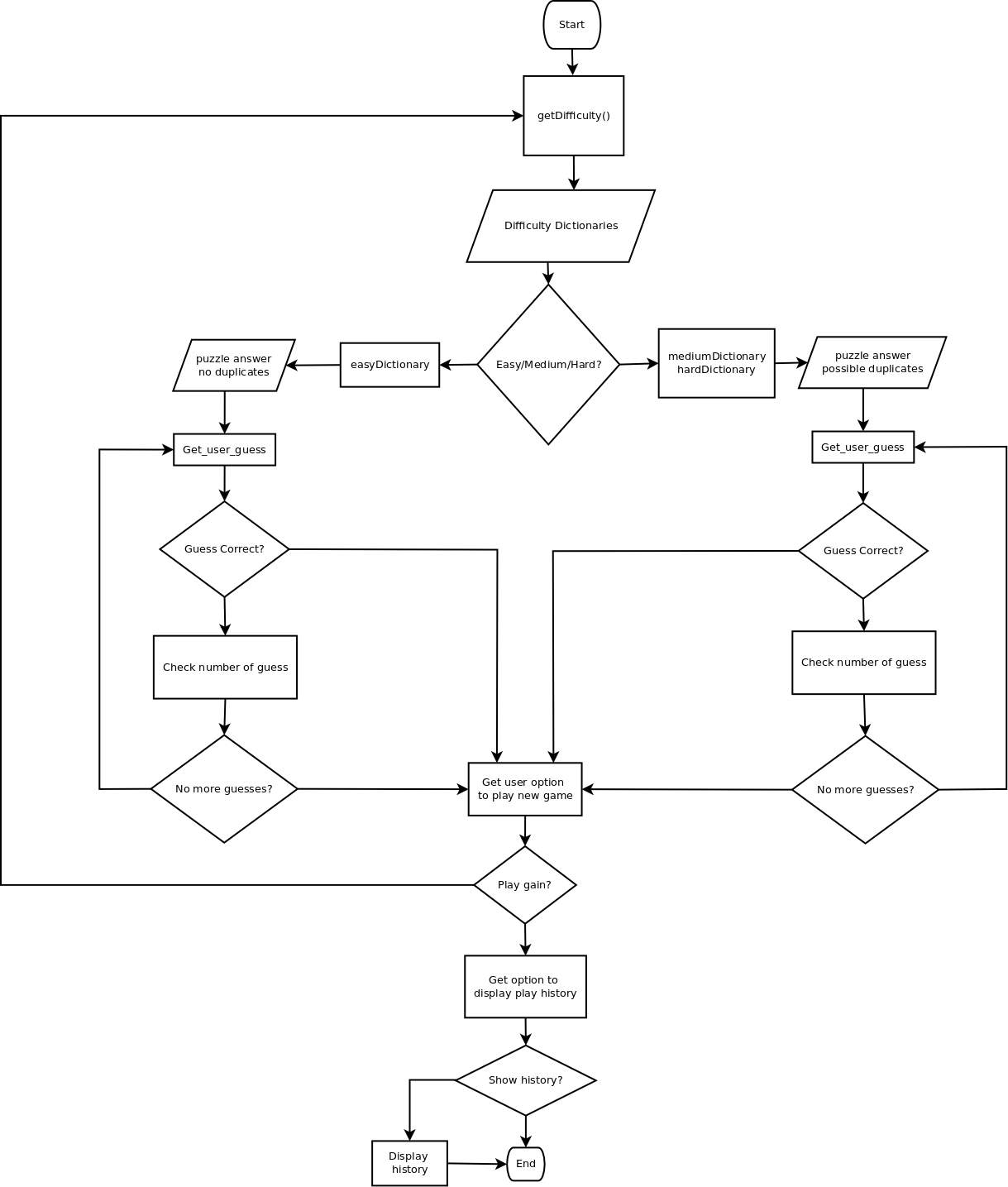
If user chooses to show play history

display play history

else

Exit program

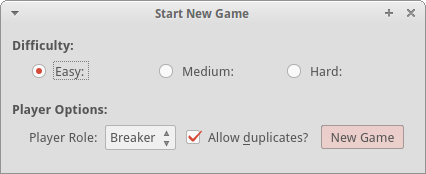
**Flow chart**



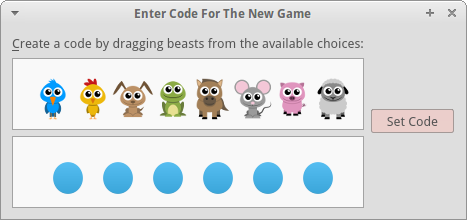
**Screen Shots**

Regular program flow.

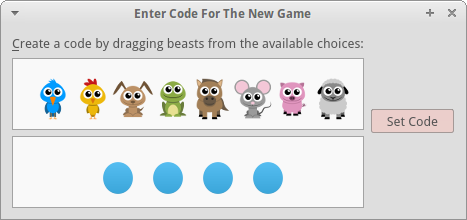
The new game dialog



The actual game screen after choosing a code



User guess generator dialog



After the game is won the code displays



**Major Variables**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Variable Name | Description | Location |
| QMap<Key,Value> | map | Holds unique characters in a puzzle and image paths. Also used to calculate bulls and cows in guess. | playspace.h, guessentry.h, newgamedialog.h. It is used in the constructors |
| QSet<Key> | intersection | Used to get the number of bulls and cows in a guess compared with a code. Also used to determine if there are duplicates in a code. | guessentry.cpp In guessFinished(bool) function  codedialop.cpp containsDuplicates(QString) |
| Tree<User> | rankings | Holds all the words available for the game. | GamePlay.java file under main and in MyFileReader.java under fillDictionary() |
| QStack<T> | guesses | In the Game class holds all the user guesses | In mainwindow.h and playspace.h. |
| Queue<T> | guessQueue | Used to a breadth first traversal in the tree | tree.h |
| List<T> | guessInterface | Hold the individual guessEntry QWidgets | In playspace.h. Has its own class. |
| Game<> | currentGame | Holds the current game contents and helps load a game | In mainwindow.h and in playspace.h load(const Game & game) |
| Guess | newGuess | Holds a new guess and the numbers for bulls and cows | In mainwindow.h and playspace.h It is used to load a game |
| QString | codeWord | Temporarily holds the code | In playspace.h and guessentry.h member variable |
| Int | size | Holds the size of a given datastructure | Most data structure classes use it e.g. MyLinkedList, MySet, MyMap |

**C++ Data Structures**

|  |  |  |
| --- | --- | --- |
| Data Structure | New syntax and keywords | Location |
| Linked Lists | List<T> list; | Underlying structure for MyMap.java and used for guessinterface list in playspace.h |
| Recursion | getGuess(guess, puzzle); | getGuess(String,String) in GamePlay.java |
| Hashing | Used to hash a user password when logging in; | Underlying method for hash password |
| Queue |  |  |
| Map | MyMap<Key,Value> map = new MyMap<Key,Value>(); | puzzleMap, guessMap in GamePlay.java  keySet in MySet.java |
| Templates | List<T>,Queue<T> | Used in all classes |
| Stacks | Stack<Game> playHistory; | playHistory variable in mainwindow.h |
| Trees | Tree<T> tree | Underlying structure for the player rankings |

**Reference**

1. Textbook
2. stackoverflow.com

**Code:**

/\*

\* File: codedialog.h

\* Interface file for CodeDialog class. The class inherits from

\* QDialog. It used to get user generated codes for the game. It makes

\* use of CodeScene and IconAcceptor objects which accept

\* implement mousePressEvent, mouseMoveEvent, and mouseReleaseEvent

\* and transport an icons text value during a drag event.

\* Author: Marlo Zeroth

\*/

#ifndef CODEDIALOG\_H

#define CODEDIALOG\_H

#include <QDialog>

#include <QCheckBox>

#include <QComboBox>

#include <QLineEdit>

#include <QPushButton>

#include <QRadioButton>

#include <QButtonGroup>

#include <QGroupBox>

#include <QLabel>

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QString>

#include "codescene.h"

#include "guessscene.h"

class CodeDialog : public QDialog

{

Q\_OBJECT

public:

CodeDialog(QMap<QString, QString> map, int length, QWidget \*parent = 0);

public slots:

void updateSize(const int &defaultTries);

// Updates the size of the code generator view

void setAllowDuplicates(bool dups);

// Set whether to allow duplicates in the code generator view

signals:

void startGame(QString word);

// Emits the code word entered by the user in codeSetter

void sizeChanged(int defaultTries)const;

// Emits the new size

void duplicatesChanged(bool);

private slots:

void setGame(bool test);

private:

bool duplicates;

int defaultTries;

QLabel \*codeLabel;

CodeScene \*codeOptions;

GuessScene \*codeSetter;

QGraphicsView \*codeSceneView;

QGraphicsView \*guessSceneView;

QPushButton \*newGameButton;

bool containsDuplicates(QString test);

// Test the passed string for duplicate characters

};

#endif // CODEDIALOG\_H

#ifndef CODEDIALOG\_CPP

#define CODEDIALOG\_CPP

#include <QMessageBox>

#include <QRegExp>

#include <QRegExpValidator>

#include "codedialog.h"

CodeDialog::CodeDialog(QMap<QString,QString> map,int length, QWidget \*parent):QDialog(parent){

// If the length is greater than 0 set size equal to it otherwise default it to 2

length>0?defaultTries=length:defaultTries = 2;

// Allow duplicates by default. Should be changed by slot/signal

duplicates = true;

// Set the code label and code options for the code

codeLabel = new QLabel("&Create a code by dragging beasts from the available choices:",parent);

codeOptions = new CodeScene(map,parent);

codeSceneView = new QGraphicsView;

codeSceneView->setScene(codeOptions);

codeLabel->setBuddy(codeSceneView);

// Create an ImageAcceptor scene to create the code

codeSetter = new GuessScene(length,parent);

guessSceneView = new QGraphicsView;

guessSceneView ->setScene(codeSetter);

QVBoxLayout \*codeLayout = new QVBoxLayout;

codeLayout->addWidget(codeLabel);

codeLayout->addWidget(codeSceneView);

codeLayout->addWidget(guessSceneView);

newGameButton = new QPushButton("Set Code",parent);

connect(newGameButton,SIGNAL(clicked(bool)),this,SLOT(setGame(bool)));

QHBoxLayout \*gameLayout = new QHBoxLayout;

gameLayout->addWidget(newGameButton);

QHBoxLayout \*mainLayout = new QHBoxLayout;

mainLayout->addLayout(codeLayout);

mainLayout->addLayout(gameLayout);

setWindowTitle("Enter Code For The New Game");

setLayout(mainLayout);

setModal(true);

}

// Changes the size on the guessAcceptor based on the selected option

// from the difficulty in the PlaySpace class QGroupButton class.

void CodeDialog::updateSize(const int &test){

if(test!=defaultTries && test > 0){

// Resize with the new size

defaultTries = test;

codeSetter->resize(defaultTries);

emit sizeChanged(defaultTries);

}

}

// Starts a new game with the entered code word and difficulty

// signal

void CodeDialog::setGame(bool test){

if(!test){

if(codeSetter->filled()){

QString code = codeSetter->text();

// If duplicates are allowed accept the generated code

if(duplicates){

emit startGame(code);

close();

}

else{

// If duplicates not allowed check the code for duplicates

if(containsDuplicates(code)){

QMessageBox msgBx;

msgBx.setText(QString("The game has been set to not allow duplicates. "+

QString("Please create a code without duplicates.")));

msgBx.*exec*();

}

else{

emit startGame(code);

close();

}

}

}

else{

QMessageBox msgBx;

msgBx.setText("Please create a code");

msgBx.*exec*();

}

}

}

// Sets whether to accept duplicate entries in the code generator view

void CodeDialog::setAllowDuplicates(bool dups){

if(dups!=duplicates){

duplicates=dups;

emit duplicatesChanged(duplicates);

}

}

bool CodeDialog::containsDuplicates(QString test){

// Test how many characters are in the test string

QSet<QChar> testDups;

for(int i = 0; i<test.size();i++)

testDups.insert(test[i]);

// If the size of the set is less than the size of

// the string we have duplicates otherwise no duplicates

return testDups.size()<test.size();

}

#endif // CODEDIALOG\_CPP

/\*

\* File: codescene.h

\* Interface file for CodeScene class. The class inherits from

\* QGraphicsScene. It stores IconHolder objects which accept

\* implement mousePressEvent, mouseMoveEvent, and mouseReleaseEvent

\* and transport an icons text value during a drag event

\* Author: Marlo Zeroth

\*/

#ifndef CODESCENE\_H

#define CODESCENE\_H

#include <QGraphicsItem>

#include <QList>

#include <QImage>

#include <QWidget>

#include <QGraphicsView>

#include "queue.h"

class IconHolder:public QGraphicsObject{

public:

// Main constructor

IconHolder(QString value, QString imagePath, QGraphicsItem \*parent = 0);

// Destructor

//~IconHolder();

QRectF *boundingRect*() const;

void *paint*(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget);

QString getValue() const {return imageValue;}

// Returns the stored value

QPixmap getPixmap() const {return pixmap;}

// Returns the stored pixmap

protected:

void *mousePressEvent*(QGraphicsSceneMouseEvent \*event) Q\_DECL\_OVERRIDE;

void *mouseMoveEvent*(QGraphicsSceneMouseEvent \*event) Q\_DECL\_OVERRIDE;

void *mouseReleaseEvent*(QGraphicsSceneMouseEvent \*event) Q\_DECL\_OVERRIDE;

private:

QPixmap pixmap;

QString imageValue;

};

// Class to hold a queue of IconHolder objects.

class CodeScene:public QGraphicsScene{

Q\_OBJECT

public:

// Main constructor

CodeScene(QMap<QString,QString> map, QObject \*parent=0);

// Destructor

//~ImageChooser();

QString text() const;

// Returns a text string of values IconChoooser objects

void updateImages(QMap<QString,QString> map);

// Updates the internal images of the IconHolder objects

private:

int sceneHeight; // Determines the height of the scene

int iconWidth; // Width of the icons. Determines how far apart icons will be

};

#endif // CODESCENE\_H

/\*

\* File: codescene.h

\* Implementation file for CodeScene class. The class inherits from

\* QGraphicsScene. It stores IconHolder objects which accept

\* implement mousePressEvent, mouseMoveEvent, and mouseReleaseEvent

\* and transport an icons text value during a drag event

\* Author: Marlo Zeroth

\*/

#include <QtWidgets>

#include "codescene.h"

// Main constructor

IconHolder::IconHolder(QString value, QString imagePath, QGraphicsItem \*parent):QGraphicsObject(parent),

pixmap(imagePath),imageValue(value){

// Tool tip to show when the left mouse button is pressed on image

setToolTip(QString("Click and drag this animal onto the slot!"));

// Set the cursor to show an open hand when the left mouse button is pressed

setCursor(Qt::OpenHandCursor);

// Set it so only the left mouse button can be pressed

setAcceptedMouseButtons(Qt::LeftButton);

}

// Used to set the bounding rectangle when the icons are drawn.

// Overrides the QGraphicsObject boudingRect() function

QRectF IconHolder::*boundingRect*() const{

return QRectF(QPointF(0,0),QSize(40,40));

}

// Used to paint the icons. Overrides the QGraphicsObject paint function

void IconHolder::*paint*(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget){

// Overrides the paint event for QGraphicsObject

// Set the unused macro for option and widget so wont complain at compile

Q\_UNUSED(option);

Q\_UNUSED(widget);

// If there is no image, display a circle bounded by a rectangle 40 by 40

if(pixmap.isNull()){

painter->setBrush(QColor("red"));

painter->drawEllipse(QRectF(0,0,40,40));

}

else{

painter->scale(1,1);

painter->drawPixmap(QPointF(0,0),pixmap);

}

}

void IconHolder::*mousePressEvent*(QGraphicsSceneMouseEvent \*event){

Q\_UNUSED(event);

// Set the cursor to be closed hand when the left mouse button is pressed

setCursor(Qt::ClosedHandCursor);

}

void IconHolder::*mouseMoveEvent*(QGraphicsSceneMouseEvent \*event){

// Test to make sure the position of the mouseMoveEvent is valid

if (QLineF(event->screenPos(), event->buttonDownScreenPos(Qt::LeftButton))

.length() < QApplication::startDragDistance()) {

return;

}

// Create a drag event and its mime data

QDrag \*drag = new QDrag(event->widget());

QMimeData \*mime = new QMimeData;

mime->setText(imageValue);

mime->setImageData(pixmap);

drag->setMimeData(mime);

// Set image that will be draged

drag->setPixmap(QPixmap(pixmap.scaled(40,40)));

drag->setHotSpot(QPoint(15,30));

// Execute the drag event

drag->exec();

// Set the cursor to be open hand

setCursor(Qt::OpenHandCursor);

}

void IconHolder::*mouseReleaseEvent*(QGraphicsSceneMouseEvent \*event){

Q\_UNUSED(event);

// Set the cursor to show an open hand when it is realeased

setCursor(Qt::OpenHandCursor);

}

// Accepts a map whose key is the first letter of an animal and value is the file path to its image.

CodeScene::CodeScene(QMap<QString, QString> map, QObject \*parent):QGraphicsScene(parent){

// Set the icon width, scene height and create a new scene

iconWidth = 40;

sceneHeight = 40;

// Iterate through the map, create a new IconHolder object and add it to the scene

QMap<QString,QString>::iterator i;

int count = 0;

for(i = map.begin(); i!=map.end();i++){

// Create an IconHolder object with the image value and image path

IconHolder \*chooser = new IconHolder(i.key(),i.value());

chooser->setPos(count\*iconWidth,0);

// Add the new chooser to the scene and icons queue

addItem(chooser);

// Increment the count to adjust the spacing

count++;

}

}

// Updates the ImageChooser to the map of image paths and values

void CodeScene::updateImages(QMap<QString, QString> map){

// Empty the scene

clear();

// Iterate through the map, create a new IconHolder object and add it to the scene and

// internal icons queue

QMap<QString,QString>::iterator i;

int count = 0;

for(i = map.begin(); i!=map.end();i++){

IconHolder \*chooser = new IconHolder(i.key(),i.value());

chooser->setPos(count\*iconWidth,0);

// Add the new chooser to the scene

addItem(chooser);

// Increment the count to adjust the spacing

count++;

}

}

// Returns a concatenanted QString with the value assigned to each of the IconHolder objects

// in the scene

QString CodeScene::text()const{

QString optionString("");

// Iterate through the icons and concatenate the QString

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

foreach (QGraphicsItem \*item, list) {

IconHolder \*holder = qgraphicsitem\_cast<IconHolder\*>(item);

optionString=(optionString+holder->getValue());

}

return optionString;

}

#ifndef COMPUTER\_H

#define COMPUTER\_H

#include <QString>

#include <QList>

class Computer{

public:

void generate(int n, QString options, QList<QString>& mylist);

// Fills a list with all permutations without repetition based on an input string of

// valid characters. Code implemented from:

// https://en.wikipedia.org/wiki/Heap's\_algorithm

void generateDups(QString options, QString output, int last, int index,QList<QString>& mylist);

// Fills a list will all possible permutations with repetitions based on string of valid

// character options. Code implemented from:

//http://www.geeksforgeeks.org/print-all-permutations-with-repetition-of-characters/

private:

void swap(QChar a, QChar b);

};

#endif // COMPUTER\_H

#include <QDebug>

#include "computer.h"

#include <iostream>

#include "list"

void Computer::generate(int n, QString options, QList<QString>& mylist) {

if (n == 1){

mylist.append(options);

}

else {

for (int i = 0; i < n - 1; i++) {

generate(n - 1, options,mylist);

if (n % 2 == 0) {

swap(options[i], options[n - 1]);

} else

swap(options[0], options[n - 1]);

}

generate(n - 1, options,mylist);

}

}

void Computer::swap(QChar a, QChar b) {

QChar temp = a;

a = b;

b = temp;

}

void Computer::generateDups(QString options, QString output, int last, int index, QList<QString>& mylist){

int i, len = options.size();

// One by one fix all characters at the given index and recur for

// the/ subsequent indexes

for ( i=0; i<len; i++ )

{

// Fix the ith character at index and if this is not the last

// index then recursively call for higher indexes

output[index] = options[i] ;

// If this is the last index then print the string stored in

// data[]

if (index == last)

mylist.append(output);

else // Recur for higher indexes

generateDups(options, output, last, index+1,mylist);

}

}

#ifndef CONNECTION

#define CONNECTION

#include <QSqlDatabase>

// Struct for create database connections

struct Connection{

QSqlDatabase db;

bool connected;

};

#endif // CONNECTION

/\*

\* Emtpy.h

\*

\* Created on: Nov 4, 2015

\* Author: super0

\*/

#ifndef EMPTY\_H\_

#define EMPTY\_H\_

#include <iostream>

#include <cstdlib>

class Empty{

public:

Empty(){

std::cout <<"Structure is empty." << std::endl;

exit (EXIT\_FAILURE);

};

};

#endif /\* EMPTY\_H\_ \*/

#ifndef GAME

#define GAME

#include <QString>

#include <QDataStream>

#include <QStack>

// User Libraries

#include "user.h"

#include "guess.h"

class Game{

public:

// Main constructor

Game(QString map, const unsigned int id, const QString &word, const unsigned int &totalTries,

const User &p1, const User &p2);

// Mutators

void setGuess(const QString &guess, const unsigned int &bull, const unsigned int &cow);

// Adds a guess and its associated bull and cow to the stack of

// guess in the current game

void setFinished(const bool &complete);

// Sets the finished state of the game

void setWinner(const int &win);

// Sets the winner of the game if the game is finished.

// Accepts only the digits 1 or 2 as valid options

void setPlayerRole(const int & role);

// Sets the player role for the game.

// Accepts only the digits 1 or 2 as valid options.

// Accesors

QString getCode() const {return codeWord;}

bool isFinished() const {return finished;}

User getPlayer1() const {return player1;}

User getPlayer2() const {return player2;}

unsigned int getWinner() const {return winner;}

unsigned int getCodeBreaker() const {return codeBreaker;}

unsigned int getTotalGuessesMade(){return guesses.size();}

Guess getCurrentGuess() const {return guesses.top();}

QStack<Guess> getGuesses() const;

private:

// Overloaded operators

friend QDataStream &operator << (QDataStream &stream, const Game &game);

friend QDataStream &operator >> (QDataStream &stream, Game &game);

// Available options for the code

QString options;

// GameId, code word, finished state, and number of tries

unsigned int gameId;

// Code word and guesses made

QString codeWord;

unsigned int maximumTries;

// Players for the game

User player1;

User player2;

bool finished; // If the game has been completed

// Player roles and winner

unsigned int codeBreaker;

unsigned int winner;

QStack<Guess> guesses;

};

#endif // GAME

#include "game.h"

Game::Game(QString map, const unsigned int id, const QString &word,

const unsigned int &totalTries, const User &p1, const User &p2):options(map),

gameId(id),codeWord(word),maximumTries(totalTries), player1(p1), player2(p2), finished(false){

}

// Add all member variables into the stream. Could create a toString() function to handle most of this

QDataStream &operator << (QDataStream &stream, const Game &game){

stream << game.gameId;

stream << game.codeWord;

stream << game.maximumTries;

stream << game.player1;

stream << game.player2;

stream << game.finished;

stream << game.codeBreaker;

stream << game.winner;

stream << game.guesses;

return stream;

}

QDataStream & operator >> (QDataStream &stream, Game &game){

stream >> game.gameId;

stream >> game.codeWord;

stream >> game.maximumTries;

stream >> game.player1;

stream >> game.player2;

stream >> game.finished;

stream >> game.codeBreaker;

stream >> game.winner;

stream >> game.guesses;

return stream;

}

void Game::setGuess(const QString &guess, const unsigned int &bull, const unsigned int &cow){

// Add a guess only if the game is not finished

if(!finished){

Guess temp;

temp.bull=bull;

temp.cow=cow;

temp.guess = guess;

guesses.push(temp);

}

}

void Game::setFinished(const bool &complete){

if(complete!=finished){

finished=complete;

}

}

void Game::setWinner(const int &win){

if(finished && (win==1||win==2)){

winner=win;

}

}

void Game::setPlayerRole(const int &role){

if(finished && (role==1||role==2)){

codeBreaker=role;

}

}

QStack<Guess> Game::getGuesses() const {

QStack<Guess> temp(guesses);

return temp;

}

#ifndef GUESS\_H

#define GUESS\_H

#include <QString>

#include <QDataStream>

class Guess{

public:

QString guess;

unsigned int bull;

unsigned int cow;

Guess(){}

Guess(const QString& g, const unsigned int& b, const unsigned int& c):

guess(g),bull(b),cow(c){}

friend QDataStream& operator << (QDataStream &os, const Guess &obj){

os << obj.guess << obj.bull << obj.cow;

return os;

}

friend QDataStream& operator >> (QDataStream& is, Guess &obj){

is >> obj.guess >> obj.bull >> obj.cow;

return is;

}

};

#endif // GUESS\_H

/\*

\* File: guessentry.cpp

\* Interface file for GuessEntry class. The class inherits from

\* QWidget. It displays and accepts user entries for guesses for the

\* game bulls and cows - a version predating mastermind.\*

\* Author: Marlo Zeroth

\*/

#ifndef GUESSENTRY\_H

#define GUESSENTRY\_H

#include <QWidget>

#include <QLabel>

#include <QString>

#include <QPushButton>

#include <QLineEdit>

#include <QPixmap>

#include <QGridLayout>

#include <QRegExp>

#include <QRegExpValidator>

#include <QSet>

#include <QChar>

#include "guessscene.h"

#include "guess.h"

class GuessEntry : public QWidget

{

Q\_OBJECT

public:

explicit GuessEntry(QMap<QString,QString> options, QPixmap bull, QPixmap cow, int defaultTries,

bool current, QWidget \*parent = 0, Qt::WindowFlags f=0);

public slots:

void setCurrent(bool status);

// Sets the current member variable

void setCodeWord(QString code);

// Sets the current code word to test

void setSize(int test);

// Updates the view for guessScene

void setGuess(QString guess, unsigned int bull, unsigned int cow);

// Sets the values for the current guessScene

void reset();

// Resets the displayed widgets

private slots:

void guessFinished(bool);

// Connects to checkButton's clicked(bool) signal and

// emits solved(true) if the codeWord matches the user guess

// or emits finished(true) if the codeWord does not

signals:

void finished(bool);

// Emits if the processing the code and guess words is completed

void solved(bool);

// Emits whether the code word and guess are the same

void current(bool);

// Emits the value of the member variable current

void guessWord(QString,unsigned int, unsigned int);

// Emits the guess word and its bulls and cows

void codeWordChanged(QString);

// Emits the changed code word

private:

QMap<QString,QString> options; // Options available to generate a guess

QString codeWord; // Code to test the guess against

QLabel \*bullLabel;// To hold an image displaying a bull

QLabel \*cowLabel; // To hold an image displaying a cow

QLabel \*bullCount; // Show how many elements are in the set of a code and in the right place

QLabel \*cowCount; // Shows how many elements form a guess are also in the set of a code but wrong place

QGraphicsView \*guessView; // Will hold the guessAcceptor scene

GuessScene \*guessAcceptor; // Accepts icons with text values

QPushButton \*checkButton; // Button to process a guess - checks whether it matches a code

bool currentGuess; // Determines whether this is the current guess in a set of guesses

};

#endif // GUESSENTRY\_H

/\*

\* File: guessentry.cpp

\* Implementation file for GuessEntry class. The class inherits from

\* QWidget. It displays and accepts user entries for guesses for the

\* game bulls and cows - a version predating mastermind.\*

\* Author: Marlo Zeroth

\*/

#include <QMessageBox>

// For error output

#include "guessentry.h"

GuessEntry::GuessEntry(QMap<QString, QString> options, QPixmap bull, QPixmap cow, int size,

bool current, QWidget \*parent, Qt::WindowFlags f):QWidget(parent,f){

this->options = options; // Set the options available

// Sets whether this is the current guess entry

currentGuess = current;

// Set the labels and images for the cow and bull labels

bullLabel = new QLabel(parent);

cowLabel = new QLabel(parent);

bullLabel->setPixmap(bull);

bullLabel->setStyleSheet("background-color: green;");

bullLabel->setFixedSize(50,50);

cowLabel->setPixmap(cow);

cowLabel->setStyleSheet("background-color: green;");

cowLabel->setFixedSize(50,50);

// Set the labels for the bull and cow counts

cowCount = new QLabel(parent);

bullCount = new QLabel(parent);

cowCount->setStyleSheet("color: yellow; background-color: green; font: 18pt");

bullCount->setStyleSheet("color: yellow; background-color: green; font: 18pt");

cowCount->setFixedSize(50,50);

cowCount->setAlignment(Qt::AlignCenter);

bullCount->setAlignment(Qt::AlignCenter);

bullCount->setFixedSize(50,50);

cowCount->setText("");

bullCount->setText("");

// Prepare the guessAcceptor, guessView and push button

guessAcceptor = new GuessScene(size,parent);

guessView = new QGraphicsView;

qreal width =guessAcceptor->sceneRect().width();

qreal height=guessAcceptor->sceneRect().height();

qreal padding = 10.0;

guessView->setScene(guessAcceptor);

guessView->setFixedSize(width+padding,height+padding);

guessView->setSceneRect(0,0,width,height);

guessView->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

guessView->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

checkButton = new QPushButton(tr("Check"));

// By default set the button to hide and not enabled until

// the entry is current

checkButton->hide();

guessView->setEnabled(false);

guessView->setStyleSheet("background-color: green;");

// Set the checkButton to emit the guess when it is clicked

connect(checkButton,SIGNAL(clicked(bool)),this,SLOT(guessFinished(bool)));

// Create and set the layout

QGridLayout \*layout = new QGridLayout;

layout->addWidget(guessView,0,0);

layout->addWidget(bullLabel,0,1);

layout->addWidget(bullCount,0,2);

layout->addWidget(cowLabel,0,3);

layout->addWidget(cowCount,0,4);

layout->addWidget(checkButton,0,5);

layout->setMargin(1);

//layout->setSizeConstraint(QLayout::SetFixedSize);

setLayout(layout);

}

// Sets the GuessEntry current variable as true, enables checkButton,

// and guessAcceptor object.

void GuessEntry::setCurrent(bool status){

if(status!=currentGuess){

currentGuess = status;

// Show the checkButton and enable the guessAcceptor

guessView->setEnabled(status);

checkButton->*setVisible*(status);

emit current(currentGuess);

}

}

// Sets the code word for the game.

void GuessEntry::setCodeWord(QString code){

if(codeWord!=code){

codeWord = code;

emit codeWordChanged(code);

}

}

// Resets the current and displayed QLabels

void GuessEntry::reset(){

currentGuess = false;

bullCount->setText("");

cowCount->setText("");

guessAcceptor->reset();

guessView->update();

checkButton->*setVisible*(false);

}

// Checks the guessAcceptor entry with the passed codeWord. It sets the

// cowCount and bullCount QLabels to the current bulls and cows

// in the code word and guess made by the user.

void GuessEntry::guessFinished(bool test){

// If neither the text nor the guess are empty check

if(!codeWord.isEmpty()&&currentGuess){

if(guessAcceptor->filled()){

// Hide the check button and disable the guess acceptor and set this

// guess as not current. This prepares to go to the next guess

checkButton->hide();

guessView->setEnabled(false);

currentGuess = false;

// Get the guess entered and check it

QString guess = guessAcceptor->text();

int cows = 0; // If letters are in the puzzle, but wrong spot

int bulls = 0; // If letters are in the puzzle and right spot

// Maps and sets for the code and guess

QMap<QChar,int> codeMap;

QMap<QChar,int> guessMap;

QSet<QChar> guessSet;

QSet<QChar> codeSet;

for (int i = 0; i < codeWord.length(); i++) {

guessSet.insert(guess[i]);

codeSet.insert(codeWord[i]);

if (guess[i] == codeWord[i])

bulls++;

// Fill the maps of the code and the guess

// if the QChar is in the map update its value

// for both code and guess

if(!codeMap.contains(codeWord[i])){

codeMap.insert(codeWord[i],1);

}

else{

int count = codeMap.value(codeWord[i]);

codeMap.insert(codeWord[i],(count+1));

}

if(!guessMap.contains(guess[i])){

guessMap.insert(guess[i],1);

}

else{

int count = guessMap.value(guess[i]);

guessMap.insert(guess[i],(count+1));

}

}

// Get the intersection between the puzzleSet and guessSet characters.

QSet<QChar> intersection = codeSet.intersect(guessSet);

// Iterate through the intersection and find the total that are

// in the code and how often they appear in the code

foreach(QChar i,intersection){

if(codeMap.value(i)<guessMap.value(i))

cows+=codeMap.value(i);

else

cows+=guessMap.value(i);

}

// The difference between those in the right position and the total

// the show up in the code will be those that are in the wrong position

// and show in the code - the cows

cows = cows - bulls;

bullCount->setText(QString::number(bulls));

cowCount->setText(QString::number(cows));

if(guess==codeWord){

// Send out if the code word is broken

emit solved(true);

}

// The default value from QPushButton clicked() is false. We need to send opposite

// signal to show this guess entry is finished.

emit finished(!test);

// Send out the user entered guess

emit guessWord(guess,bulls, cows);

}

else{

QMessageBox msgBx;

msgBx.setText("Please enter a guess");

msgBx.*exec*();

}

}

}

// Changes the size of the guessAcceptor based on the selected option

// from the difficulty QGroupButton class

void GuessEntry::setSize(int test){

if(test!=guessAcceptor->getSize()){

guessAcceptor->resize(test);

// Update the view

qreal width =guessAcceptor->sceneRect().width();

qreal height=guessAcceptor->sceneRect().height();

qreal padding = 10.0;

guessView->setScene(guessAcceptor);

guessView->setFixedSize(width+padding,height+padding);

guessView->setSceneRect(0,0,width,height);

guessView->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

guessView->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

guessView->update();

}

}

// Updates the guessScene, bull and cow labels to to the passed parameters.

// Also emits if this matches the code word and emits that the current guess

// is finished.

void GuessEntry::setGuess(QString guess, unsigned int bull, unsigned int cow){

// If this is the current guess

if(currentGuess){

// Update the guess scene

guessAcceptor->updateGuessScene(options,guess);

// Update the bull and cow label

bullCount->setText(QString::number(bull));

cowCount->setText(QString::number(cow));

// If this finishes the game emit solved

if(guess==codeWord)

emit solved(true);

// Emit this completes this guess

emit finished(true);

}

}

/\*

\* File: guessscene.h

\* Interface file for the GuessScene class.

\* The class inherits from QGraphicsScene and is a storage

\* for IconAcceptor objects which accept drag and drop events

\* Author: Marlo Zeroth

\*

\*/

#ifndef GUESSSCENE\_H

#define GUESSSCENE\_H

#include <QWidget>

#include <QtWidgets>

#include <QGraphicsItem>

#include <QDragEnterEvent>

#include <QDropEvent>

#include <QPixmap>

#include <QString>

#include "list.h"

class IconAcceptor:public QGraphicsObject{

public:

IconAcceptor(QGraphicsItem \*parent=0);

// Main constructor

QRectF *boundingRect*() const;

void *paint*(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget = 0) Q\_DECL\_OVERRIDE;

// Paints/redraws the image each time

void clear();

// Resets the default image and text.

QString getValue() const {return value;}

// Returns the stored value

QPixmap getPixmap() const {return pixmap;}

// Returns the stored pixmap

QColor getColor(){return color;}

// Returns the stored color

void updateImageValue(QString newValue, QPixmap newPix);

protected:

void *dragEnterEvent*(QGraphicsSceneDragDropEvent \*event) Q\_DECL\_OVERRIDE;

// Executes when an item is dragged over the instantiated object

void *dragLeaveEvent*(QGraphicsSceneDragDropEvent \*event) Q\_DECL\_OVERRIDE;

// Executes when an item being dragged over the instatntiated object leaves

// the areas

void *dropEvent*(QGraphicsSceneDragDropEvent \*event) Q\_DECL\_OVERRIDE;

// Executes when an item is dropped into the instantiated object.

// If the data passed is valid, it overrides the pixmap and the

// value member variables and updates the displayed image

// and value associated with the new image.

private:

const QPixmap DEFAULTIMAGE; // Constant for the default image to display.

QPixmap pixmap; // Image to display

QString value; // Value of the image

QColor color;

bool dragOver;

};

class GuessScene:public QGraphicsScene{

Q\_OBJECT

public:

// Main constructor

GuessScene(unsigned int size, QObject \*parent = 0);

QString text()const;

// Returns a text string of values IconAcceptors

void updateGuessScene(QMap<QString,QString> map, QString guess);

// Updates the scene with new image values and respective paths

int getSize() const;

// Returns the current number of IconAcceptor objects stored

void reset();

// Resets the IconAcceptor object images

bool filled() const;

void resize(const int &newSize);

// Resizes and redraws the scene based on the input size

private:

int iconWidth;

int sceneHeight;

};

#endif // ICONSCENE\_H

/\*

\* file: guessscene.cpp

\* Implementation file for GuessScene class Inherits

\* from QGraphicsScene and holds IconAcceptor class

\* objects which accept drag and drop events

\* Author: Marlo Zeroth

\*/

#include "guessscene.h"

// Sets the default color to lightGray and accepts drop

IconAcceptor::IconAcceptor(QGraphicsItem \*parent):QGraphicsObject(parent),

DEFAULTIMAGE(QString(":/images/circle-icon.png")),

pixmap(QString(":/images/circle-icon.png")),value(""), color(Qt::lightGray), dragOver(false){

setAcceptDrops(true);

}

// Overriden boundingRect from QGraphicsObject. Required for painting object.

QRectF IconAcceptor::*boundingRect*() const {

return QRectF(QPointF(0,0),QSize(40,40));

}

// Overriden paint() function from QGraphicsObject. Required for painting/drawing object.

void IconAcceptor::*paint*(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget){

Q\_UNUSED(option);

Q\_UNUSED(widget);

if(pixmap.isNull()){

painter->setBrush(dragOver?color.light(130):color);

painter->drawEllipse(QRectF(0,0,40,40));

}

else{

painter->scale(1,1);

painter->drawPixmap(QPoint(0,0),pixmap);

}

}

// Resets the values to default. Clears the value string

void IconAcceptor::clear(){

if(!pixmap.isNull()&& !value.isEmpty()){

pixmap = DEFAULTIMAGE;

value = QString("");

}

}

// Updates the pixmap and value of the pixmap

void IconAcceptor::updateImageValue(QString newValue, QPixmap newPix){

value=newValue;

pixmap=newPix;

}

// Sets behaviour when another object is dragged into this object

void IconAcceptor::*dragEnterEvent*(QGraphicsSceneDragDropEvent \*event){

if(event->mimeData()->hasImage()&&event->mimeData()->hasText()){

event->setAccepted(true);

dragOver = true;

update();

}

}

// Sets behaviour for when an object leaves this object during drag

void IconAcceptor::*dragLeaveEvent*(QGraphicsSceneDragDropEvent \*event){

Q\_UNUSED(event);

dragOver = false;

update();

}

// Sets behavious for when another object is dropped into this object

void IconAcceptor::*dropEvent*(QGraphicsSceneDragDropEvent \*event){

if(event->mimeData()->hasImage()&&event->mimeData()->hasText()){

dragOver = false;

// Set the pixmap to the image dropped onto this widget

pixmap = qvariant\_cast<QPixmap>(event->mimeData()->imageData());

// Set the value to text dropped on this widget

value = qvariant\_cast<QString>(event->mimeData()->text());

// Now update the image displayed

update();

}

}

// Main constructor

GuessScene::GuessScene(unsigned int size, QObject \*parent):QGraphicsScene(parent){

// Set the icon width, scene height and create a new scene

iconWidth = 40;

sceneHeight = 40;

// Create new IconAcceptor objects and add them to the icons queue

for(unsigned int i =0; i<size;i++){

IconAcceptor \*acceptor = new IconAcceptor;

acceptor->setPos(i\*(iconWidth+10),0);

addItem(acceptor);

}

}

QString GuessScene::text() const{

QString guess("");

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

// Iterate through each item and concatenate the guess QString

foreach(QGraphicsItem \*item,list){

IconAcceptor \*acceptor = qgraphicsitem\_cast<IconAcceptor\*>(item);

guess=(guess+acceptor->getValue());

}

return guess;

}

// Resets each IconHolder object to default images

void GuessScene::reset(){

// Iterate thorugh the list of IconAcceptor objects in the scene and reset

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

foreach(QGraphicsItem \*item,list){

IconAcceptor \*acceptor = qgraphicsitem\_cast<IconAcceptor\*>(item);

acceptor->clear();

}

update();

}

// Updates the scene with new images and values

void GuessScene::updateGuessScene(QMap<QString, QString> map, QString guess){

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

QString charOptions = guess;

if(list.length()==guess.length()){

// Iterate through the items and update their image paths and values

foreach(QGraphicsItem \*item,list){

QString charToAdd;

if(charOptions.size()>0){

// Get the first letter in the guess

charToAdd=charOptions[0];

// Now set the image path and value in the graphics item

IconAcceptor \*acceptor = qgraphicsitem\_cast<IconAcceptor\*>(item);

acceptor->updateImageValue(charToAdd,map.value(charToAdd));

// Update the r

if(charOptions.size()>1)

charOptions=charOptions.remove(charToAdd,Qt::CaseInsensitive);

}

}

// Now update the scene with the new images

update();

}

}

// Returns true if the IconAcceptor objects in the scene have a value

bool GuessScene::filled()const{

// Assume the IconAcceptor objects in the scene have received a string value

bool complete=true;

// Get a list of address of the items in the scene

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

// Now iterate through each of the addresses

QList<QGraphicsItem \*>::iterator i = list.begin();

QList<QGraphicsItem \*>::iterator finish = list.end();

while(i!=finish && complete){

// Cast to ensure we can use IconAcceptor getValue()

IconAcceptor \*acceptor = qgraphicsitem\_cast<IconAcceptor\*>(\*i);

// If the value is an empty string the set of objects in the scene

// have not all received a string value

if(acceptor->getValue().isEmpty())

complete = false;

i++;

}

return complete;

}

// Returns the total number of objects in the scene

int GuessScene::getSize() const{

QList<QGraphicsItem \*> list = items(Qt::AscendingOrder);

return list.length();

}

// Clears the scene and add new IconHolder objects according to the new size

void GuessScene::resize(const int &newSize){

clear(); // Clear the scene

// Add new IconAcceptor items

for(int i =0;i<newSize;i++){

IconAcceptor \*acceptor = new IconAcceptor;

acceptor->setPos(i\*(iconWidth+10),0);

addItem(acceptor);

}

}

/\*

\* File: invalidargument.h

\* Author: Marlo Zeroth

\*

\* Created on November 9, 2015, 7:37 AM

\*/

#ifndef INVALIDARGUMENT\_H

#define INVALIDARGUMENT\_H

#include <iostream>

#include <cstdlib>

class InvalidArgument{

public:

InvalidArgument(){

std::cout <<"Argument is not valid." << std::endl;

exit (EXIT\_FAILURE);

};

};

#endif /\* INVALIDARGUMENT\_H \*/

#ifndef LIST\_H

#define LIST\_H

/\*

\* Author: Marlo Zeroth

\* List.h

\*

\* Implements a template doubly linked list data structure. Allows duplicates,

\* insertion at front, back, and any point in the list. The list is not ordered.

\* Created on: Nov 4, 2015

\*

\*/

#include <new>

#include "invalidargument.h"

#include "empty.h"

template<class T>

class List {

private:

// Internal struct class to hold data and point to other nodes

struct Node {

T data;

Node \*next;

Node \*previous;

};

/\*Utility functions\*/

void memError() const; // Handles memory allocation errors

void subError() const; // Handles subscripts out of range

// Returns the position of an entry in the list

typename List<T>::iterator position(const T& entry) const;

/\* DoublyList Member variables \*/

Node \*head; // Reference to first/head node

Node \*tail; // Reference to last/tail node

int size;

public:

// iterator class for traversing the list

class iterator {

public:

friend class List;

// Default constructor. Allows .

iterator();

// Copy constructor

//iterator(const iterator & obj);

// Destructor

//~iterator();

/\*Overloaded operators\*/

// Assignment operator

//iterator& operator = (const iterator& obj);

// Prefix increment operator

iterator& operator++();

// Postfix increment operator

iterator operator++(int);

// Dereference operator

T& operator\*();

// Returns the addres of the value referred to

T\* operator->();

// Comparison operator

bool operator==(const iterator &obj) const;

// Comparison operator

bool operator!=(const iterator &obj) const;

private:

iterator(Node \*pter);

Node \*position;

};

//Constructors and Destructor

List();

// Default constructor

List(const T &entry);

// Main constructor

List(const List<T> &obj);

// Copy constructor

~List();

// Destructor

// Overloaded Operators

List<T>& operator=(const List<T> &obj);

// Oveloaded = operator

/\* Mutators and Accessors \*/

void prepend(const T& entry);

// Adds an entry to the front of the list

void append(const T& entry);

// Adds an entry to the back of the list

void insertBefore(List<T>::iterator \*position, const T& entry);

// Inserts an element before the passed parameter

void insertAfter(List<T>::iterator \*position, const T& entry);

// Inserts an element after the passed parameter

T& first() throw (Empty);

// Returns the first element in the list

T& last() throw (Empty);

// Returns the last element in the list

T &removeFirst() throw (Empty);

// Removes and returns the first element in the list

T &removeLast() throw (Empty);

// Removes and returns the last element in the list

iterator begin();

// Returns a pointer to the beginning of the list.

iterator begin() const;

// Returns a const iterator to the beginning of the list.

iterator end();

// Returns a null iterator to signify the end of the list.

iterator end() const;

// Returns a const null iterator to signify the end of the list.

bool contains(const T& entry);

// Returns whether the list contains the passed entry.

int length() const { return size;}

// Returns the size of the list

void remove(const T& entry);

// Removes the passed entry from the list if it exits. Throws

// an invalid argument if it does not exist.

void clear();

// Removes all items for the list

};

/\*DoublyList Constructor, Destructor, and Function Definitions \*/

// Default constructor

template<class T>

List<T>::List() {

tail = NULL;

head = NULL;

size = 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Main constructor. Adds an entry and increments the size by 1. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

List<T>::List(const T &entry) {

// Create new node with entry parameter

Node \*link;

// Attempt to allocate memory

try {

link = new Node;

} catch (const std::bad\_alloc&) {

memError();

}

// Set its data member and pointers

link->data = entry;

link->next = NULL;

link->previous = NULL;

// Now point the head and tail to this new link in the list

head = tail = link;

size++;

}

// Copy constructor. This is order O(N).

template<class T>

List<T>::List(const List<T> &obj) {

std::cout << "copy constructor" << std::endl;

// Set the head and tail pointers to null.

tail = head = NULL;

// Set the size of the list

size = 0;

// Now set the cursor to the object's head and

// traverse the links in obj if there are any.

Node \*cursor; // For navigating through the obj list

for (cursor = obj.head; cursor != 0; cursor = cursor->next) {

append(cursor->data);

}

}

// Destructor

template<class T>

List<T>::~List() {

// Traverse the list deleting each link

while (head) {

// Set the cursor to the current head

Node \*cursor = head;

// Set the head pointer to the next link

head = head->next;

// Deallocate the current link

delete cursor;

}

tail = NULL;

}

// Overloaded operators

// Overload = operator declaration

template<class T>

List<T>& List<T>::operator=(const List<T> &obj) {

std::cout << "= operator " << std::endl;

// Test to make sure the passed object is not the same

// as calling object e.g. input = input

if (this != &obj) {

clear();

// Set the size of the list

size = 0;

// Now set the cursor to the object's head and

// traverse the links in obj if there are any.

Node \*cursor; // For navigating through the obj list

for (cursor = obj.head; cursor != 0; cursor = cursor->next) {

append(cursor->data);

}

}

// Just return a reference

return \*this;

}

/\*Mutators and accessors\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// prepend Function. Adds a element at the beginning of the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void List<T>::prepend(const T& input) {

// Create a new node for the input.

Node \*link;

// Allocate memory

try {

link = new Node;

} catch (const std::bad\_alloc&) {

memError();

}

// Set its data element.

link->data = input;

// If the list is empty, add the new link and point head and tail to it

if (!head) {

// Set the link's next and previous to null as this is the only link

link->next = NULL;

link->previous = NULL;

// Now point the head and tail to the new link

tail = head = link;

size++;

} else {

// Since it will be at the beginning, it should

// point to the current first link in the list.

link->next = head;

link->previous = NULL;

// Now set the previous pointer in the existing first link

// to the new link.

head->previous = link;

// Set the new head.

head = link;

size++;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// append Function. Adds a element at the end of the list. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void List<T>::append(const T& input) {

// Create a new node for the input.

Node \*link;

// Allocate memory

try {

link = new Node;

} catch (const std::bad\_alloc&) {

memError();

}

// Set its data element.

link->data = input;

// If the list is empty, add the new link and point head and tail to it

if (!tail) {

// Set the link's next and previous to null as this is the only link

link->next = NULL;

link->previous = NULL;

// Now point the head and tail to the new link

tail = head = link;

size++;

} else {

// Since it will be at the end, it should

// point to the current last link in the list.

link->previous = tail;

link->next = NULL;

// Now set the next pointer in the existing last link

// to the new link.

tail->next = link;

// Set the new tail.

tail = link;

size++;

}

}

// Inserts an element before the passed parameter

template<class T>

void List<T>::insertBefore(List<T>::iterator \*location, const T& entry) {

Node \*cursor = const\_cast<Node\*> (location->position);

// Check to be sure this is a valid position

if (cursor == 0) {

throw InvalidArgument("invalid position");

}

Node \* const link;

try {

link = new Node;

} catch (const std::bad\_alloc&) {

memError();

}

link->data = entry;

// If the position is at the tail, change the tail

if (cursor == head) {

// Since it will be at the beginning, it should

// point to the current first link in the list.

link->next = head;

link->previous = NULL;

// Now set the previous pointer in the existing first link

// to the new link.

head->previous = link;

// Set the new head.

head = link;

size++;

} else {

Node \*prevPtr = head;

while (prevPtr != 0 && prevPtr->next != cursor) {

prevPtr = prevPtr->next;

}

if (prevPtr == 0) {

throw InvalidArgument("invalid position");

}

prevPtr->next = link;

}

}

// Inserts an element after the passed parameter

template<class T>

void List<T>::insertAfter(List<T>::iterator \*location, const T& entry) {

// Check to be sure this is a valid position

if (location == 0) {

throw InvalidArgument("invalid position");

}

Node \*link;

try {

link = new Node;

} catch (const std::bad\_alloc&) {

memError();

}

link->data = entry;

location->position->next = link;

link->previous = location.position;

// If the position is at the tail, change the tail

if (location->position == tail)

tail = link;

}

// Returns the first element in the list

template<class T>

T& List<T>::first() throw (Empty) {

if (head) {

return head->data;

} else {

throw Empty();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// last function. Returns the last element in the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

T& List<T>::last() throw (Empty) {

if (tail) {

return tail->data;

} else {

throw Empty();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// removeFirst function. Removes and returns the last element in the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

T& List<T>::removeFirst() throw (Empty){

if(head){

//Temporary pointer to head

Node \*temp = head;

T \*data = new T;

\*data = temp->data;

// Delete the temp and set tail to the

// previous link and reduce the size

delete temp;

head = head->next;

--size;

return \*data;

}

else{

throw Empty();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// removeLast function. Removes and returns the last element in the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

T& List<T>::removeLast() throw (Empty){

if(tail){

// Temporary pointer to tail

Node \*temp = tail;

T \*data = new T;

\*data = temp->data;

// Delete the temp and set tail to the

// previous link and reduce the size

delete temp;

tail = tail->previous;

--size;

return \*data;

}

else{

throw Empty();

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// begin function. Returns an iterator to the beginning of the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

typename List<T>::iterator List<T>::begin() {

// Create an iterator and set its

// position to the head.

return iterator(head);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// begin function. Returns an iterator to the beginning of the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

typename List<T>::iterator List<T>::begin() const {

// Create an iterator and set its

// position to the head.

return iterator(head);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// end function. Returns an iterator to the end of the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

typename List<T>::iterator List<T>::end() {

// Create an iterator and set its position to the

// tail.

return iterator(NULL);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// end function. Returns an iterator to the end of the list \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

typename List<T>::iterator List<T>::end() const {

// Create an iterator and set its position to the

// tail.

return iterator(NULL);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// contains function. Returns true if entry is in the list, false \*

// otherwise \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

bool List<T>::contains(const T& entry) {

Node \*cursor = head;

bool found = false;

// Go through the list until we reach the end

// or find the entry

while (!found && cursor != 0) {

if (cursor->data == entry)

found = true;

cursor = cursor->next;

}

return found;

}

template<class T>

void List<T>::remove(const T& entry){

// For traversing the list

Node \*cursor = head;

// For changing the link before the entry to delete

Node \*prevPtr = NULL;

bool found = false;

// Traverse the list until we find the node or

// we reach the end

while (cursor->data != entry && cursor != NULL) {

if (cursor->data == entry)

found = true;

// Set the previous link to the current link

prevPtr = cursor;

// Go to the next link

cursor = cursor->next;

}

// If it is not in the list throw an exception

if (cursor == 0)

throw InvalidArgument("Entry not in the list");

// If we found it delete it

if (cursor == head) {

// If it is at the front of the list, there is no previous

// link. Just set the head to the next link

head = cursor->next;

found = true;

} else {

// Link the previous node and the node after the

// one to delete

prevPtr->next = cursor->next;

found = true;

}

// Deallocate it

delete cursor;

--size;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// clear function. Deallocates each link in the list and sets the size to 0 \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void List<T>::clear() {

Node \*cursor;

// Clear out the data in the existing list

while (head) {

// Set the cursor to the current head

cursor = head;

// Set the head pointer to the next link

head = head->next;

// Deallocate the current link

delete cursor;

}

tail = NULL;

size = 0;

}

/\*iterator definitions\*/

// Default constructor

template<class T>

List<T>::iterator::iterator() {

}

// Main constructor

template<class T>

List<T>::iterator::iterator(Node \*ptr) {

position = ptr;

}

/\*Overloaded operators\*/

// != operator

template<class T>

bool List<T>::iterator::operator!=(const iterator &obj) const {

return position != obj.position;

}

// == operator

template<class T>

bool List<T>::iterator::operator==(const iterator &obj) const {

return position == obj.position;

}

// Dereferences the pointer and returns value pointed by position

template<class T>

T& List<T>::iterator::operator\*() {

return position->data;

}

template<class T>

T\* List<T>::iterator::operator->() {

return (&\*(List<T>::iterator) \* this);

}

// Prefix increments the iterator to the next element

template<class T>

typename List<T>::iterator& List<T>::iterator::operator++() {

position = position->next;

return \*this;

}

// Postfix increments the iterator to the next element

template<class T>

typename List<T>::iterator List<T>::iterator::operator++(int) {

iterator temp = \*this;

++(\*this);

return temp;

}

/\*Utility Functions\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// memError function. Displays an error message and \*

// terminates the program when memory allocation fails. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void List<T>::memError() const {

std::cout << "ERROR:Cannot allocate memory.\n";

exit(EXIT\_FAILURE);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// subError function. Displays an error message and \*

// terminates the program when a subscript is out of range. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void List<T>::subError() const {

std::cout << "ERROR: Subscript out of range.\n";

exit(EXIT\_FAILURE);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Returns the position of an entry in the list. \*

// If the entry is not in the list, it returns null. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

typename List<T>::iterator List<T>::position(const T& entry) const {

iterator cursor(head);

iterator finish(NULL);

while (\*cursor != entry && cursor != finish) {

cursor++;

}

return cursor;

}

#endif /\* LIST\_H \*/

#ifndef LOGINDIALOG\_H

#define LOGINDIALOG\_H

#include <QDialog>

#include <QRegularExpression> // For testing regular expressions

#include <QRegularExpressionValidator>

#include <QLabel>

#include <QLineEdit>

#include <QCheckBox>

#include <QDialogButtonBox>

#include <QGridLayout>

#include <QSql>

#include <QSqlQuery>

#include <QSqlDatabase>

#include "user.h"

#include "connection.h"

class LoginDialog : public QDialog

{

Q\_OBJECT

public:

explicit LoginDialog(const Connection& connection,QWidget \*parent = 0);

//~LoginDialog();

private slots:

//void checkUserPassword();

signals:

void acceptLogin(User);

void loginSuccess(bool);

public slots:

void slotAcceptLogin(bool);

private:

QLabel \*passwordLabel;

QLabel \*userLabel;

QLineEdit \*userNameLine;

QLineEdit \*passwordLine;

QCheckBox \*rememberCheckBox;

QDialogButtonBox \*buttonBox;

Connection currentConnection;

};

#endif // LOGINDIALOG\_H

#include <QMessageBox>

#include <QPushButton>

#include <QGridLayout>

#include "logindialog.h"

LoginDialog::LoginDialog(const Connection& connection, QWidget \*parent) :

QDialog(parent)

{

// Set the connection to test

currentConnection = connection;

// Regular expressions to check valid user name and input

QRegularExpression userExp("^[a-zA-Z0-9\_@.-]{3,15}$");

QRegularExpression passwordExp("^((?=.\*[0-9])(?=.\*[a-zA-Z])[a-zA-Z0-9!@#$%&\*]{6,20})$");

// Initialize the username and password line edits

userNameLine = new QLineEdit(this);

passwordLine = new QLineEdit(this);

// Set up up the validators for the user and password lines

userNameLine->setValidator(new QRegularExpressionValidator(userExp,this));

passwordLine->setValidator(new QRegularExpressionValidator(passwordExp,this));

// Set the password line so that it does not show characters

// passwordLine->setEchoMode(QLineEdit::Password);

// Initialize the labels and set their buddys

userLabel = new QLabel(this);

passwordLabel = new QLabel(this);

userLabel->setText((tr("Username")));

passwordLabel->setText((tr("Password")));

userLabel->setBuddy(userNameLine);

passwordLabel->setBuddy(passwordLine);

// Initialize and set up the buttons

buttonBox = new QDialogButtonBox(this);

buttonBox->addButton(QDialogButtonBox::Ok);

buttonBox->addButton(QDialogButtonBox::Cancel);

buttonBox->button(QDialogButtonBox::Ok)->setText(tr("Login"));

buttonBox->button(QDialogButtonBox::Cancel)->setText(tr("Cancel"));

// Connect the buttons slots and signals

connect(buttonBox->button(QDialogButtonBox::Cancel),SIGNAL(clicked()),this,SLOT(close()));

connect(buttonBox->button(QDialogButtonBox::Ok),SIGNAL(clicked(bool)),this,SLOT(slotAcceptLogin(bool)));

// Now create and set the layout

QGridLayout \*formLayout = new QGridLayout(this);

formLayout->addWidget(userLabel,0,0);

formLayout->addWidget(userNameLine,0,1);

formLayout->addWidget(passwordLabel,1,0);

formLayout->addWidget(passwordLine,1,1);

formLayout->addWidget(buttonBox,2,0,1,2);

setLayout(formLayout);

// Set the window title and whether it is modal>

setWindowTitle(tr("User Login"));

setModal(true);

}

void LoginDialog::slotAcceptLogin(bool){

QMessageBox msgBx;

if(currentConnection.connected){

QSqlQuery query;

unsigned int userId ;

QString name;

QString passP;

QString first;

QString last;

QString middle;

QString emailAddress;

QDate birth;

query.exec("SELECT user\_id, user\_name, password, first\_name, last\_name, middle\_name, email, birthdate FROM users");

while (query.next()) {

userId = query.value(0).toUInt();

name = query.value(1).toString();

passP = query.value(2).toString();

first = query.value(3).toString();

last = query.value(4).toString();

middle = query.value(5).toString();

emailAddress = query.value(6).toString();

birth = query.value(7).toDate();

}

if(name!=userNameLine->text()|| passP!=passwordLine->text()){

msgBx.setText("Error: Please enter a valid username and password");

msgBx.*exec*();

emit loginSuccess(false);

}

else{

User currentUser(userId,first,middle,last,name,passP,emailAddress,birth);

emit acceptLogin(currentUser);

emit loginSuccess(true);

}

currentConnection.db.close();

}

else{

msgBx.setText("Not connected to server");

msgBx.*exec*();

}

}

/\*

\* File: mainwindow.h

\* Interface file for MainWindow class. The class inherits from

\* QWindow. It is the main class for displaying game.

\* Author: Marlo Zeroth

\*/

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QAction>

#include <QMenu>

#include <QSql>

#include <QSqlQuery>

#include <QSqlDatabase>

#include "playspace.h"

#include "logindialog.h"

#include "newuserdialog.h"

#include "newgamedialog.h"

#include "user.h"

#include "stack.h"

#include "connection.h"

class MainWindow : public QMainWindow{

Q\_OBJECT

public:

MainWindow();

protected:

void *closeEvent*(QCloseEvent \*event);

private slots:

void newGame();

void open();

bool save();

bool saveAs();

void openRecentFile();

void updateStatusBar();

void gameModified();

void about();

void login(); // Opens the login dialog

private:

void createActions();

void createMenus();

void createContextMenu();

void createToolBars();

void createStatusBar();

void readSettings();

void writeSettings();

bool okToContinue();

Connection createConnection();

bool loadFile(const QString &fileName); // For loading an existing game stored locally

QMap<QString,QString> loadImages(QFile &file); // For loading images to central widget

bool saveFile(const QString &fileName); // Saving a game locally

void setCurrentFile(const QString &fileName); // Sets the most current game at start up, if none it creates a new one

void updateRecentFileActions();

QString strippedName(const QString &fileName);

const unsigned int TRIES;

const unsigned int DEFAULTDIFFICULTY;

QFile \*imageFiles;

LoginDialog \*loginDialog;

NewUserDialog \*newUserDialog;

NewGameDialog \*newGameDialog;

PlaySpace \*playSpace;

User \*loginUser;

Connection currentConnection;

QLabel \*loginLabel;

QLabel \*currentMoveLabel;

QStringList recentFiles;

QString curFile;

enum { MaxRecentFiles = 5 };

QAction \*recentFileActions[MaxRecentFiles];

QAction \*separatorAction;

QMenu \*fileMenu;

QMenu \*helpMenu;

QToolBar \*fileToolBar;

QAction \*newAction;

QAction \*openAction;

QAction \*saveAction;

QAction \*saveAsAction;

QAction \*loginAction;

QAction \*exitAction;

QAction \*aboutAction;

QAction \*aboutQtAction;

};

#endif // MAINWINDOW\_H

#include "mainwindow.h"

MainWindow::MainWindow():TRIES(10), DEFAULTDIFFICULTY(4){

imageFiles = new QFile(":/text/code\_options.txt");

QMap<QString,QString> map = loadImages(\*imageFiles);

playSpace = new PlaySpace(map,DEFAULTDIFFICULTY,TRIES);

playSpace->setStyleSheet("background-color: green;");

setCentralWidget(playSpace);

createActions();

createMenus();

createContextMenu();

createToolBars();

createStatusBar();

readSettings();

currentConnection = createConnection();

loginDialog = new LoginDialog(currentConnection, this);

newUserDialog = new NewUserDialog;

newGameDialog = new NewGameDialog(map,this);

connect(newGameDialog,SIGNAL(codeChanged(QString)),playSpace,SLOT(updatePlayspace(QString)));

connect(playSpace,SIGNAL(gameFinished(bool)),newGameDialog,SLOT(setVisible(bool)));

loginUser = NULL;

//setAttribute(Qt::WA\_DeleteOnClose);

// Release memory when the application closes

setWindowIcon(QIcon(":/images/bull-icon.png"));

setCurrentFile("");

newGameDialog->show();

}

void MainWindow::login(){

loginDialog->show();

}

Connection MainWindow::createConnection(){

QSqlDatabase::drivers();

QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL","QMYSQLCSC17B");

// Create database object to access remote database

// Set the login information

db.setHostName("209.129.8.2");

db.setDatabaseName("mastermind");

db.setUserName("48941");

db.setPassword("48941cis17b");

Connection connection;

connection.db = db;

if (!db.open()) {

QMessageBox msgBx;

msgBx.setText("Error connecting to remote server.");

msgBx.*exec*();

connection.connected = false;

return connection;

}

connection.connected = true;

return connection;

}

void MainWindow::*closeEvent*(QCloseEvent \*event){

if (okToContinue()) {

writeSettings();

event->accept();

} else {

event->ignore();

}

}

void MainWindow::newGame(){

// Open multiple instances of the application

newGameDialog->show();

}

void MainWindow::open(){

if (okToContinue()) {

QString fileName = QFileDialog::getOpenFileName(this,

tr("Open Bulls and Cows"), ".",

tr("Bulls and Cows files (\*.bc)"));

if (!fileName.isEmpty())

loadFile(fileName);

}

}

bool MainWindow::save(){

if (curFile.isEmpty()) {

return saveAs();

} else {

return saveFile(curFile);

}

}

bool MainWindow::saveAs(){

QString fileName = QFileDialog::getSaveFileName(this,

tr("Save game"), ".",

tr("Game files (\*.bc)"));

if (fileName.isEmpty())

return false;

return saveFile(fileName);

}

void MainWindow::about()

{

QMessageBox::about(this, tr("About Bulls and Cows"),

tr("<h2>Bulls and Cows 1.1</h2>"

"<p>Copyright &copy; 2015 Marlo Zeroth."

"<p>Bulls and Cows is a small application that "

"demonstrates QNetwork, QMySQL, QAction, QMainWindow, QMenuBar, "

"QStatusBar, QTableWidget, QToolBar, and many other "

"Qt classes."));

}

void MainWindow::openRecentFile(){

if (okToContinue()) {

QAction \*action = qobject\_cast<QAction \*>(sender());

if (action)

loadFile(action->data().toString());

}

}

void MainWindow::updateStatusBar(){

}

void MainWindow::gameModified(){

}

void MainWindow::createActions(){

newAction = new QAction(tr("&New"), this);

newAction->setIcon(QIcon(":/images/new.png"));

newAction->setShortcut(QKeySequence::New);

newAction->setStatusTip(tr("Create a new bulls and cows game file"));

connect(newAction, SIGNAL(triggered()), this, SLOT(newGame()));

openAction = new QAction(tr("&Open..."), this);

openAction->setIcon(QIcon(":/images/open.png"));

openAction->setShortcut(QKeySequence::Open);

openAction->setStatusTip(tr("Open an existing bulls and cows game file"));

connect(openAction, SIGNAL(triggered()), this, SLOT(open()));

saveAction = new QAction(tr("&Save"), this);

saveAction->setIcon(QIcon(":/images/save.png"));

saveAction->setShortcut(QKeySequence::Save);

saveAction->setStatusTip(tr("Save the game to disk"));

connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));

saveAsAction = new QAction(tr("Save &As..."), this);

saveAsAction->setStatusTip(tr("Save the bulls and cows under a new "

"name"));

connect(saveAsAction, SIGNAL(triggered()), this, SLOT(saveAs()));

for (int i = 0; i < MaxRecentFiles; ++i) {

recentFileActions[i] = new QAction(this);

recentFileActions[i]->setVisible(false);

connect(recentFileActions[i], SIGNAL(triggered()),

this, SLOT(openRecentFile()));

}

loginAction = new QAction(tr("L&ogin"),this);

loginAction->setIcon(QIcon(":/images/login-icon.png"));

loginAction->setShortcut(tr("Ctrl+L"));

loginAction->setStatusTip(tr("Login to the remote server"));

connect(loginAction,SIGNAL(triggered()),this,SLOT(login()));

exitAction = new QAction(tr("E&xit"), this);

exitAction->setShortcut(tr("Ctrl+Q"));

exitAction->setStatusTip(tr("Exit the application"));

connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));

aboutAction = new QAction(tr("&About"), this);

aboutAction->setStatusTip(tr("Show the application's About box"));

connect(aboutAction, SIGNAL(triggered()), this, SLOT(about()));

aboutQtAction = new QAction(tr("About &Qt"), this);

aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));

connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));

}

void MainWindow::createMenus(){

fileMenu = menuBar()->addMenu(tr("&File"));

fileMenu->addAction(newAction);

fileMenu->addAction(openAction);

fileMenu->addAction(saveAction);

fileMenu->addAction(saveAsAction);

fileMenu->addAction(loginAction);

separatorAction = fileMenu->addSeparator();

for (int i = 0; i < MaxRecentFiles; ++i)

fileMenu->addAction(recentFileActions[i]);

fileMenu->addSeparator();

fileMenu->addAction(exitAction);

menuBar()->addSeparator();

helpMenu = menuBar()->addMenu(tr("&Help"));

helpMenu->addAction(aboutAction);

helpMenu->addAction(aboutQtAction);

}

void MainWindow::createContextMenu(){

}

void MainWindow::createToolBars(){

fileToolBar = addToolBar(tr("&File"));

fileToolBar->addAction(newAction);

fileToolBar->addAction(openAction);

fileToolBar->addAction(saveAction);

fileToolBar->addAction(loginAction);

}

void MainWindow::createStatusBar(){

loginLabel = new QLabel(" Local Player - Not Logged in ");

loginLabel->setAlignment(Qt::AlignHCenter);

loginLabel->setMinimumSize(loginLabel->*sizeHint*());

currentMoveLabel = new QLabel;

currentMoveLabel->setIndent(3);

statusBar()->addWidget(loginLabel);

statusBar()->addWidget(currentMoveLabel, 1);

//connect(playSpace, SIGNAL(currentCellChanged(int, int, int, int)),

//this, SLOT(updateStatusBar()));

//connect(spreadsheet, SIGNAL(modified()),

//this, SLOT(spreadsheetModified()));

updateStatusBar();

}

void MainWindow::readSettings(){

}

void MainWindow::writeSettings(){

}

bool MainWindow::okToContinue(){

return true;

}

bool MainWindow::loadFile(const QString &fileName){

return true;

}

QMap<QString, QString> MainWindow::loadImages(QFile &file){

// Creates a value pair image key (a string) and image value path (path to icon)

QMap<QString,QString> map;

// Test the file to ensure it opened

if(!file.*open*(QIODevice::ReadOnly|QIODevice::Text))

std::cerr<<"Cannot open file for reading "

<<qPrintable(file.errorString())<<std::endl;

// Iterate through the text stream and load into the map

QTextStream in(&file);

while(!in.atEnd()){

QString line = in.readLine();

QStringList iconList;

iconList=line.split("\t");

std::cerr<<"Size of list "<<iconList.size()<<std::endl;

QString iconValue = iconList.takeFirst();

QString iconImage = QString(":/images/"+iconList.takeFirst());

map.insert(iconValue,iconImage);

}

return map;

}

bool MainWindow::saveFile(const QString &fileName){

return true;

}

void MainWindow::setCurrentFile(const QString &fileName){

}

void MainWindow::updateRecentFileActions(){

}

QString MainWindow::strippedName(const QString &fileName){

QString a("a");

return a;

}

#ifndef NEWGAMEDIALOG\_H

#define NEWGAMEDIALOG\_H

#include <QWidget>

#include <QDialog>

// User Libraries

#include "codedialog.h"

class NewGameDialog : public QDialog

{

Q\_OBJECT

public:

explicit NewGameDialog(QMap<QString, QString> map, QWidget \*parent = 0);

public slots:

void newGame(bool start); // Connects the newGameButton with this object

void setCode(const QString& entry); // Updates the code word

signals:

void startGame(bool);

void playerOneRole(int);

void duplicatesAllowed(bool);

void codeChanged(QString);

void codeLength(int);

private:

QLabel \*codeLabel;

QString codeOptions; // Available icons to choose from

QMap<QString,QString> codeComponents; // Valid Components that can be used to generate a code and output images

CodeDialog \*codeEntry; // Used to get an user generated code

QButtonGroup \*difficulty; // Determines the size of the code

QCheckBox \*duplicates; // Determines whether duplicates are allowed in the code

QComboBox \*playerRole; // Determines whether a player generates or breaks the code

QPushButton \*newGameButton; // Starts/resets the game

QPushButton \*quitButton;

/\*Utility functions\*/

QString generateCode();

// Generates a code if player is breaker

QGroupBox\* createDifficultyBox();

// Creates a radion button group box with difficulty levels

QGroupBox\* createOptionsBox();

// Creates a button group box with the player role and new game button

};

#endif // NEWGAMEDIALOG\_H

#include "newgamedialog.h"

NewGameDialog::NewGameDialog(QMap<QString,QString> map, QWidget \*parent) : QDialog(parent){

codeComponents = map;

// Iterate through the map to get the available code options

QMap<QString,QString>::iterator i;

QMap<QString,QString>::iterator end = codeComponents.end();

for(i=codeComponents.begin();i!=end;i++){

codeOptions = codeOptions+i.key();

}

// Create the difficulty radio button group

QGroupBox \*difficultyRadioBox = createDifficultyBox();

QGroupBox \*groupOptionsBox = createOptionsBox();

// Set the bottom layout

QVBoxLayout \*bottomLayout = new QVBoxLayout;

bottomLayout->addWidget(difficultyRadioBox);

bottomLayout->addWidget(groupOptionsBox);

// Create a dialog for entering the code

codeEntry = new CodeDialog(map,difficulty->checkedId(),parent);

// Connect the slots for several of the objects such as when the newGameButton is pressed

// a new game starts

connect(newGameButton,SIGNAL(clicked(bool)),this,SLOT(newGame(bool)));

connect(difficulty,SIGNAL(buttonPressed(int)),codeEntry,SLOT(updateSize(int)));

connect(codeEntry,SIGNAL(startGame(QString)),this,SLOT(setCode(QString)));

connect(duplicates,SIGNAL(toggled(bool)),codeEntry,SLOT(setAllowDuplicates(bool)));

connect(codeEntry,SIGNAL(startGame(QString)),this,SLOT(setCode(QString)));

setLayout(bottomLayout);

setWindowTitle("Start New Game");

setModal(true);

}

void NewGameDialog::setCode(const QString &entry){

emit codeChanged(entry);

}

void NewGameDialog::newGame(bool start){

if(!start){

emit playerOneRole(playerRole->currentIndex());

emit duplicatesAllowed(duplicates->isChecked());

emit codeLength(difficulty->checkedId());

// Get the code word for the game

if(playerRole->currentIndex()==1){

codeEntry->show();

}

else{

emit codeChanged(generateCode());

}

*setVisible*(false);

}

}

// Generates a QString code based on available code components

QString NewGameDialog::generateCode(){

QString randomCode("");

int optionSize = codeComponents.size();

QString optionString = codeOptions;

int codeSize = difficulty->checkedId();

if(!duplicates->isChecked()){

QString charOptions = optionString;

// Iterate through the available options and concatenate a string based

// on a pseudo-random choice of its letters until the requested sized

// string is generated.

for(int i = 0; i<codeSize;i++){

QString charToAdd;

if(charOptions.size()>1)

charToAdd=QString(charOptions[qrand()% (charOptions.size()-1)]);

else

charToAdd=QString(charOptions[0]);

randomCode = (randomCode+charToAdd);

if(charOptions.size()>1)

charOptions=charOptions.remove(charToAdd,Qt::CaseInsensitive);

}

}

else{

// Iterate through the available options and concatenate a string based

// on a pseudo-random choice of its letters.

for(int i = 0; i<codeSize;i++){

randomCode = (randomCode+QString(optionString[qrand()% (optionSize-1)]));

}

}

return randomCode;

}

// Creates a difficulty QGroupBox of radio buttons

QGroupBox\* NewGameDialog::createDifficultyBox(){

// Length of code and guess entries accepted

int easyDiff = 4;

int midDiff = 6;

int hardDiff = 8;

// Set the difficulty

difficulty = new QButtonGroup;

QGroupBox \*difficultyRadioBox = new QGroupBox(tr("Difficulty:"));

QRadioButton \*easy = new QRadioButton(tr("Easy: "));

QRadioButton \*medium = new QRadioButton(tr("Medium:"));

QRadioButton \*hard = new QRadioButton(tr("Hard:"));

// Make it be an exclusive group box

easy->setChecked(true);

// Add the buttons to the difficulty button group

difficulty->addButton(easy);

difficulty->addButton(medium);

difficulty->addButton(hard);

// Set the values for the difficulty

difficulty->setId(easy,easyDiff);

difficulty->setId(medium,midDiff);

difficulty->setId(hard,hardDiff);

// Create a layout for the game options difficulty

QHBoxLayout \*diffLayout = new QHBoxLayout;

diffLayout->addWidget(easy);

diffLayout->addWidget(medium);

diffLayout->addWidget(hard);

difficultyRadioBox->setLayout(diffLayout);

return difficultyRadioBox;

}

// Creates a QGroupBox of a QComboBox and QPushButton

QGroupBox\* NewGameDialog::createOptionsBox(){

// Prepare the playerRole combo box

playerRole = new QComboBox;

QStringList options;

options<< "Breaker" <<"Creator";

playerRole->addItems(options);

codeLabel = new QLabel;

codeLabel->setText(tr("Player Role:"));

codeLabel->setBuddy(playerRole);

duplicates = new QCheckBox(tr("Allow &duplicates?"));

duplicates->setChecked(true);

// Create a new game button

newGameButton = new QPushButton("New Game");

// Create a layout for the role and new game button

QHBoxLayout \*roleLayout = new QHBoxLayout;

roleLayout->addWidget(codeLabel);

roleLayout->addWidget(playerRole);

roleLayout->addWidget(duplicates);

roleLayout->addWidget(newGameButton);

// Create the button group

QGroupBox \*groupBox = new QGroupBox(tr("Player Options:"));

groupBox->setLayout(roleLayout);

return groupBox;

}

#ifndef NEWUSERDIALOG\_H

#define NEWUSERDIALOG\_H

#include <QDialog>

#include "ui\_newuserdialog.h"

class NewUserDialog : public QDialog, public Ui::NewUserDialog

{

Q\_OBJECT

public:

explicit NewUserDialog(QWidget \*parent = 0);

};

#endif // NEWUSERDIALOG\_H

#include <QRegExp>

#include <QRegExpValidator>

#include <QPushButton>

#include "newuserdialog.h"

NewUserDialog::NewUserDialog(QWidget \*parent) :

QDialog(parent)

{

setupUi(this);

buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);

// Set validation for the QLineEdit objects for the fields

QRegExp nameExp("[A-Za-z]{40}"); // Regular expression for names

firstNameEdit->setValidator(new QRegExpValidator(nameExp,this));

middleNameEdit->setValidator(new QRegExpValidator(nameExp,this));

lastNameEdit->setValidator(new QRegExpValidator(nameExp,this));

// Regular expressions to check valid user name, password and email

QRegExp userExp("^[a-z0-9\_@.]{4,40}$");

QRegExp passwordExp("\\A((?=.\*\\d)(?=.\*[a-zA-Z])[a-zA-Z0-9!@#$%^&\*]{6,20})\\z");

QRegExp emailExp("^[a-z0-9\_@.]{4,40}$"); // Regular expression for email

// Set the validators

userNameEdit->setValidator(new QRegExpValidator(userExp,this));

passwordEdit->setValidator(new QRegExpValidator(passwordExp,this));

emailEdit->setValidator(new QRegExpValidator(emailExp,this));

connect(buttonBox,SIGNAL(accepted()),this, SLOT(accept()));

connect(buttonBox,SIGNAL(rejected()),this,SLOT(reject()));

}

/\*

\* File: playspace.h

\* Interface file for PlaySpace class. The class inherits from

\* QWidget. It displays and accepts user created code and guess entries for the

\* game bulls and cows - a version predating mastermind.

\* Author: Marlo Zeroth

\*/

#ifndef PLAYSPACE\_H

#define PLAYSPACE\_H

#include <QWidget>

#include <QString>

#include <QList>

#include <QMap>

#include <QPixmap>

#include <QStringList>

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QGraphicsView>

// User libraries

#include "guessentry.h"

#include "list.h" // User created templated doubly-linked list

#include "codescene.h"

#include "game.h"

class PlaySpace:public QWidget{

Q\_OBJECT

public:

// Main constructor

PlaySpace(QMap<QString, QString> map, unsigned int defaultDifficulty,

unsigned int defaultTries, QWidget \*parent=0, Qt::WindowFlags f=0);

public slots:

void loadGame(const Game& game);

// Loads a game

void updatePlayspace(const QString &code);

// Sets the new code and updates the game interface

private slots:

void showCode(bool complete);

// Updates the dummy view to display the code if the game is finished

void setGameFinished(bool complete);

// Sets the game to complete

void guessUpdate(QString guess,unsigned int bull,unsigned int cow);

// Receives a guess and its bulls and cows

void congratulate(bool win);

// Pops up a window wishing the player better luck next time

signals:

void codeChanged(QString code);

void gameFinished(bool );

void guessChanged(QString, unsigned int,unsigned int); // Emits the current guess bull and cow

private:

QPixmap cow; // To hold images for cows

QPixmap bull; // To hold images for bulls

QString codeWord; // Holds the current code to break

QString currentGuess; // Holds the current guess

unsigned int totalGuesses; // Holds how many guess entries to generate

bool finished; // If the current game is complete

QMap<QString,QString> codeComponents; // Valid Components that can be used to generate a code and output images

QString options; // Holds the keys of codeComponents to easily generate code

List<GuessEntry \*> guessInterface; // Holds the guessing interface.

QGraphicsScene \*dummy; // Shows an graphics scene filled with question marks

QGraphicsView \*dummyView; // Holds the dummy scene

QLabel \*codeLabel; // Buddy label for iconView

CodeScene \*codeScene; // Holds icons elements to drop onto GuessEntry objects

QGraphicsView \*codeView; // Displays the iconScene

/\*Utility functions\*/

QGraphicsView\* createDummyView(int length);

// Creates a dummy code QGraphicsView while playing the game

void updateDummyView(int newCodeLength);

// Updates the size of the shown dummy code whenever the difficulty changes

};

#endif // PLAYSPACE\_H

/\*

\* File: playspace.cpp

\* Implementation file for the PlaySpace class. The class inherits from

\* QWidget. It displays and accepts user created code and guess entries for the

\* game bulls and cows - a version predating mastermind.

\* Author: Marlo Zeroth

\*/

#ifndef PLAYSPACE\_CPP

#define PLAYSPACE\_CPP

#include <QMessageBox>

#include "playspace.h"

PlaySpace::PlaySpace(QMap<QString, QString> map, unsigned int defaultDifficulty, unsigned int defaultTries,

QWidget \*parent, Qt::WindowFlags f):QWidget(parent,f){

// Total number of tries possible for one round

totalGuesses = defaultTries;

// This will be used when loading a saved game that is not finished

// and for displaying the code after the game is finished

codeComponents = map;

// Set the bull and cow images

QString image1(":/images/bull-icon.png");

QString image2(":/images/cow-icon.png");

QPixmap b1(image1);

QPixmap c1 (image2);

bull = b1;

cow = c1;

// Create an CodeScene object with images to drag and choose from

codeScene = new CodeScene(map,parent);

// Creates its view

codeView = new QGraphicsView;

codeView->setScene(codeScene);

// Create the dummy view to display while game is not finished

dummyView = createDummyView(defaultDifficulty);

// Set the top layout

QLabel \*codeDummy = new QLabel(parent);

codeDummy->setText("Code to break: ");

codeDummy->setStyleSheet("background-color: green; color: yellow; font: 18pt");

QHBoxLayout \*topLayout = new QHBoxLayout;

topLayout->addWidget(codeDummy);

topLayout->addWidget(dummyView);

QVBoxLayout \*guessLayout = new QVBoxLayout;

// Create GuessEntry objects and connect the bullsCows signal to their setBullsCows slots

for(unsigned int i = 0; i<totalGuesses;i++){

GuessEntry \*entry = new GuessEntry(map,bull, cow, defaultDifficulty,false,parent);

guessLayout->addWidget(entry);

connect(this,SIGNAL(codeChanged(QString)),entry,SLOT(setCodeWord(QString)));

connect(entry,SIGNAL(guessWord(QString,uint,uint)),this,SLOT(guessUpdate(QString,unsigned int,unsigned int)));

guessInterface.prepend(entry);

}

// Iterators for current and its next GuessEntry object in the list

List<GuessEntry\*>::iterator next = guessInterface.begin();

List<GuessEntry\*>::iterator curr = next++;

List<GuessEntry\*>::iterator finish = guessInterface.end();

// Go through each guessInterface object in the list and connect it to its adjacent

// object so it knows when it is finished. Also connect each guessInterface object

// with this so it knows when the code is cracked and it displays it.

while(next!=finish){

connect(\*curr,SIGNAL(finished(bool)),\*next,SLOT(setCurrent(bool)));

connect(\*curr,SIGNAL(solved(bool)),this,SLOT(showCode(bool)));

curr++;

next++;

}

// Add the codeView right below the guessEntry objects

guessLayout->addWidget(codeView);

// Also connect the last guessInterface object with this, so it knows when the

// game is finished and shows the code

connect(guessInterface.last(),SIGNAL(finished(bool)),this,SLOT(showCode(bool)));

// Put all the layouts together

QVBoxLayout \*mainLayout = new QVBoxLayout;

mainLayout->addLayout(topLayout);

mainLayout->addLayout(guessLayout);

// Set the main layout

setLayout(mainLayout);

}

// Sets the game code and updates the individual guess entry interface

void PlaySpace::updatePlayspace(const QString &code){

if(code!=codeWord){

codeWord = code;

// Reset the current status of the individual GuessEntry objects

// Iterators for current and its next GuessEntry object in the list

List<GuessEntry\*>::iterator curr = guessInterface.begin();

List<GuessEntry\*>::iterator finish = guessInterface.end();

// Go through each guessInterface object in the list and reset current to false

while(curr!=finish){

(\*curr)->setSize(codeWord.length());// Set the size

(\*curr)->setCurrent(false);

(\*curr)->reset();

curr++;

}

// Update the dummy code view

updateDummyView(codeWord.length());

// Set the first guessEntry as current

guessInterface.first()->setCurrent(true);

emit codeChanged(codeWord);

}

}

// Creates a dummy to show while game is not finished

QGraphicsView\* PlaySpace::createDummyView(int length){

int total = length;

dummy = new QGraphicsScene;

QPixmap pixmap(":/images/question.png");

int iconWidth = 40;

for(int i =0;i<total;i++){

QGraphicsPixmapItem \*item = new QGraphicsPixmapItem(pixmap);

item->setPos(i\*iconWidth,0);

dummy->addItem(item);

}

QGraphicsView \*view = new QGraphicsView;

view->setScene(dummy);

//view->setMinimumSize(400,55);

view->setMaximumSize(480,58);

view->setStyleSheet("background-color: green;");

return view;

}

// Updates the dummy view once the game is finished

void PlaySpace::updateDummyView(int newCodeLength){

// Clear the scene - automatically deletes all items in the scene

dummy->clear();

QPixmap pixmap(":/images/question.png");// Default question mark image

int iconWidth = 40; // Width of each image/icon

// Iterate through the number provided in the difficulty radio

// button box and create new items for the scene

for(int i =0;i<newCodeLength;i++){

QGraphicsPixmapItem \*item = new QGraphicsPixmapItem(pixmap);

item->setPos(i\*iconWidth,0); // Set this item's position in the scene

dummy->addItem(item);

}

// Update the view to show the new scene

dummyView->update();

}

void PlaySpace::setGameFinished(bool complete){

if(finished!=complete){

finished=complete;

emit gameFinished(finished);

}

}

void PlaySpace::congratulate(bool win){

QMessageBox msgBx;

if(win){

msgBx.setText("Congratulations you won!");

msgBx.*exec*();

}

else{

msgBx.setText("Better luck next time.");

msgBx.*exec*();

}

}

void PlaySpace::showCode(bool complete){

if(complete){

// Clear the scene to prepare to show the code

dummy->clear();

int total = codeWord.length();

int iconWidth = 40; // Width of each image/icon

// Iterate through the number provided in the difficulty radio

// button box and create new items for the scene

for(int i =0;i<total;i++){

QString imagePath = codeComponents.value(QString(QChar(codeWord[i])));

QPixmap pixmap(imagePath);// Show the image related to the code

QGraphicsPixmapItem \*item = new QGraphicsPixmapItem(pixmap);

item->setPos(i\*iconWidth,0); // Set this item's position in the scene

dummy->addItem(item);

}

// Update the view to show the new scene

dummyView->update();

}

}

void PlaySpace::guessUpdate(QString guess,unsigned int bull, unsigned int cow){

if(guess!=currentGuess){

emit guessChanged(guess,bull,cow);

}

}

void PlaySpace::loadGame(const Game &game){

finished = game.isFinished();

codeWord = game.getCode();

QStack<Guess> guesses = game.getGuesses();

// Reset the current status of the individual GuessEntry objects

// Iterators for current and its next GuessEntry object in the list

List<GuessEntry\*>::iterator curr = guessInterface.begin();

List<GuessEntry\*>::iterator finish = guessInterface.end();

int count = 0;

// Go through each guessInterface object in the list and set its state

while(curr!=finish){

(\*curr)->setSize(codeWord.length());// Set the size

(\*curr)->setCodeWord(codeWord);

if(!guesses.isEmpty()){

(\*curr)->setGuess(guesses.top().guess,guesses.top().bull,guesses.top().cow);

(\*curr)->setCurrent(false);

guesses.pop();

}

else{

if(count<1){

(\*curr)->setCurrent(true);

++count;

}else{

(\*curr)->setCurrent(false);

(\*curr)->reset();

}

}

curr++;

}

// Update the dummy code view

updateDummyView(codeWord.length());

// Show the code if game is finished

showCode(finished);

emit codeChanged(codeWord);

emit gameFinished(finished);

}

#endif // PLAYSPACE\_CPP

/\*

\* Author: Marlo Zeroth

\* doublyList.h

\*

\* Implements a template doubly linked list data structure. Allows duplicates,

\* insertion at front, back, and any point in the list. The list is not ordered.

\* Created on: Nov 4, 2015

\*

\*/

#ifndef QUEUE\_H

#define QUEUE\_H

#include <new>

#include "empty.h"

template<class T>

class Queue {

private:

// Internal struct to hold data and point to other nodes

struct Node {

// Default constructor

Node() : next(NULL), previous(NULL) {

}

// Main constructor

Node(const T& input, Node\* nextNode, Node\* previousNode) :

data(input), next(nextNode), previous(previousNode) {

}

// Node member variables

T data;

Node \*next;

Node \*previous;

};

/\*Utility functions\*/

void memError() const;

// Handles memory allocation errors

/\* queue Member variables \*/

Node \*head; // Reference to first/head node

Node \*tail; // Reference to last/tail node

int size;

public:

Queue();

// Default constructor

Queue(const T &entry);

// Main constructor

Queue(const Queue<T> &obj);

// Copy constructor

~Queue();

// Destructor

// Overloaded Operators

Queue<T>& operator=(const Queue<T> &obj);

// Overloaded = operator

/\* Mutators and Accessors \*/

void push(const T& entry);

// Adds an entry to the front of the queue

void pop() throw (Empty);

// Removes and returns the back entry from the queue

void clear();

// Removes all items for the queue

T& back() throw (Empty);

// Returns the first element in the queue

T& front() throw (Empty);

// Returns the last element in the queue

bool contains(const T& entry);

// Returns whether the queue contains the passed entry.

int length() const {

return size;

}

// Returns the size of the queue

bool isEmpty() const {

return (head == NULL && tail == NULL);

}

// Returns whether the queue is empty

};

/\*Constructors and Destructor Definitions \*/

// Default constructor

template<class T>

Queue<T>::Queue() {

tail = NULL;

head = NULL;

size = 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Main constructor. Adds an entry and increments the size by 1. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

Queue<T>::Queue(const T &entry) {

// Create new node with entry parameter

Node \*link;

// Attempt to allocate memory

try {

link = new Node(entry, NULL, NULL);

} catch (const std::bad\_alloc&) {

memError();

}

// Now point the head and tail to this new link in the list

head = tail = link;

size++;

}

// Copy constructor. This is order O(N).

template<class T>

Queue<T>::Queue(const Queue<T> &obj) {

// Set the head and tail pointers to null.

tail = head = NULL;

// Set the size of the list

size = 0;

// Now set the cursor to the object's head and

// traverse the links in obj if there are any.

Node \*cursor; // For navigating through the obj list

for (cursor = obj.head; cursor != 0; cursor = cursor->next) {

push(cursor->data);

}

}

// Destructor

template<class T>

Queue<T>::~Queue() {

// Traverse the list deleting each link

while (head) {

// Set the cursor to the current head

Node \*cursor = head;

// Set the head pointer to the next link

head = head->next;

// Deallocate the current link

delete cursor;

}

tail = NULL;

}

/\*Overloaded operators\*/

// Overload = operator declaration

template<class T>

Queue<T>& Queue<T>::operator=(const Queue<T> &obj) {

// Test to make sure the passed object is not the same

// as calling object e.g. input = input

if (this != &obj) {

// Remove the current contents of the queue

clear();

// Now set the cursor to the object's head and

// traverse the links in obj if there are any and copy.

Node \*cursor; // For navigating through the obj list

for (cursor = obj.head; cursor != 0; cursor = cursor->next) {

push(cursor->data);

}

}

// Just return a reference

return \*this;

}

/\*Mutators and accessors\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// enqueue Function. Adds an element at the back of the of the queue \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void Queue<T>::push(const T& input) {

// Create a new node for the input.

Node \*link;

// Allocate memory

try {

link = new Node(input, NULL, NULL);

} catch (const std::bad\_alloc&) {

memError();

}

// If the queue is empty, add the new link at the front. Otherwise

// just add it to the back

if (isEmpty()) {

// Set the link's next and previous to null as this is the only link

// Now point the head and tail to the new link

head = link;

} else {

// Since it will be at the end, the new link's previous pointer should

// point to the current tail link in the list.

link->previous = tail;

// Now set the next pointer in the existing tail link

// to the new link.

tail->next = link;

}

// Link it to the queue and increment the size

tail = link;

size++;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// pop function. Removes the front element in the queue if \*

// the queue is not empty. Throws anEmpty exception if \*

// queue is empty. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void Queue<T>::pop() throw (Empty) {

if (isEmpty()) {

throw Empty();

} else {

Node \*cursor = head; // To deallocate later

// If the queue has only one item

if (head == tail) {

// We set both the head and tail to be NULL

head = tail = NULL;

} else {

// Set the front to be the next node.

head = head->next;

}

// Deallocate memory and decrease the size

delete cursor;

cursor = NULL;

--size;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// back function. Returns the back element in the queue \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

T& Queue<T>::back() throw (Empty) {

if (isEmpty()) {

throw Empty();

} else {

return tail->data;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// front function. Returns the front element in the queue \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

T& Queue<T>::front() throw (Empty) {

if (isEmpty()) {

throw Empty();

} else {

return head->data;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// contains function. Returns true if entry is in the list and \*

// moves the found link to the head/front of the list. Otherwise \*

// returns false. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

bool Queue<T>::contains(const T& entry) {

Node \*cursor = head;

bool found = false;

// Go through the list until we reach the end

// or find the entry

while (!found && cursor != 0) {

if (cursor->data == entry) {

if (cursor->next == NULL && cursor->previous != NULL) {

// The previous link before the cursor is now

// the last link

cursor->previous = cursor->next;

// Now we set the entry at the beginning

cursor->next = head;

head->previous = cursor;

head = cursor;

} else if (cursor->next != NULL && cursor->previous != NULL) {

// Link the nodes next to cursor together

Node before = cursor->previous;

Node after = cursor->next;

before->next = cursor->next;

after->previous = cursor->previous;

// Now set the cursor at the front

cursor->previous = NULL;

cursor->next = head;

head->previous = cursor;

head = cursor;

}

found = true;

}

cursor = cursor->next;

}

return found;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// clear function. Deallocates each link in the list and sets the size to 0 \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void Queue<T>::clear() {

Node \*cursor;

// Clear out the data in the existing list

while (head) {

// Set the cursor to the current head

cursor = head;

// Set the head pointer to the next link

head = head->next;

// Deallocate the current link

delete cursor;

}

tail = NULL;

size = 0;

}

/\*Utility Functions\*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// memError function. Displays an error message and \*

// terminates the program when memory allocation fails. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template<class T>

void Queue<T>::memError() const {

std::cout << "ERROR:Cannot allocate memory.\n";

exit(EXIT\_FAILURE);

}

#endif /\* QUEUE\_H \*/

/\*

\* File: user.cpp

\* Interface file for User class. The class inherits from QObject and

\* implements signals and slots.

\* Author: Marlo Zeroth

\*/

#ifndef USER\_H

#define USER\_H

#include <QString>

#include <QDate>

#include <QDataStream> // For reading and writing to files

class User {

public:

// Main constructor

User(const unsigned int &newId, const QString &first, const QString &middle, const QString &last,

const QString &userN, const QString &passW, const QString &email, const QDate &birth);

// Accessors

unsigned int getUserID() const {return userID;}

QString getFirstName() const {return firstName;}

QString getMiddleName() const {return middleName;}

QString getLastName() const {return lastName;}

QString getUserName() const {return userName;}

QString getPassword() const {return password;}

QString getEmail() const {return email;}

QDate getBirthDate() const {return birthDate;}

// Mutators

// Updates all of the user info except for the userID

void updatePlayer(unsigned int newId, QString first, QString middle, QString last, QString userN,

QString passW, QString email, QDate birth);

// Updates various player parameters

void updateUserID(const unsigned int &newId);

void updateFirstName(const QString &first);

void updateMiddleName(const QString &middle);

void updateLastName(const QString &last);

void updateBirthDate(const QDate & date);

void updateEmail(const QString &eMail);

// Updates the player's login information

void updateUserName(const QString &user);

void updatePassword(const QString &passW);

private:

friend QDataStream & operator << (QDataStream &stream, const User &user);

friend QDataStream & operator >> (QDataStream &stream, User &user);

unsigned int userID;

QString firstName;

QString middleName;

QString lastName;

QString userName;

QString password;

QString email;

QDate birthDate;

};

#endif // USER\_H

/\*

\* File: user.cpp

\* Implementation file for User class. The class inherits from QObject and

\* implements signals and slots.

\* Author: Marlo Zeroth

\*/

#include "user.h"

// Main constructor

User::User(const unsigned int &newId, const QString &first, const QString &middle, const QString &last,

const QString &userN, const QString &passW, const QString &email, const QDate &birth){

// Used initialization list. Left empty on purpose.

userID = newId;

firstName = first;

middleName = middle;

lastName = last;

userName = userN;

password = passW;

this->email = email;

birthDate = birth;

}

// Updates the player information

void User::updatePlayer(unsigned int newId, QString first, QString middle, QString last,

QString userN, QString passW, QString email, QDate birth){

userID = newId;

firstName = first;

middleName = middle;

lastName = last;

userName = userN;

password = passW;

this->email = email;

birthDate = birth;

}

void User::updateUserID(const unsigned int &newId){

if(userID!=newId){

userID=newId;

}

}

// Following functions update individual member variables

void User::updateFirstName(const QString & first){

if(first!=firstName){

firstName=first;

}

}

void User::updateMiddleName(const QString & middle){

if(middle!=middleName){

middleName=middle;

}

}

void User::updateLastName(const QString & last){

if(last!=lastName){

lastName=last;

}

}

void User::updateUserName(const QString & user){

if(user!=userName){

userName=user;

}

}

void User::updatePassword(const QString &passW){

if(passW!=password){

password=passW;

}

}

void User::updateEmail(const QString & eMail){

if(email!=eMail){

email=eMail;

}

}

void User::updateBirthDate(const QDate &date){

if(birthDate!=date){

birthDate=date;

}

}

QDataStream & operator << (QDataStream &stream, const User &user){

stream<<user.userID;

stream<<user.firstName;

stream<<user.middleName;

stream<<user.lastName;

stream<<user.userName;

stream<<user.password;

stream<<user.email;

stream<<user.birthDate;

return stream;

}

QDataStream & operator >> (QDataStream &stream, User &user){

stream>>user.userID;

stream>>user.firstName;

stream>>user.middleName;

stream>>user.lastName;

stream>>user.userName;

stream>>user.password;

stream>>user.email;

stream>>user.birthDate;

return stream;

}

/\*

\* File: tree.h

\* Author: Marlo Zeroth

\* Implements a template AVL tree

\* Created on December 15, 2015, 4:30 PM

\*/

#ifndef TREE\_H

#define TREE\_H

#include <sstream>

#include <string>

#include "empty.h"

#include "stack.h"

#include "queue.h"

template <class T>

class Tree {

private:

// Tree node struct

struct Node {

Node(T input, Node\* leftChild, Node\* rightChild) : data(input), left(leftChild), right(rightChild) {

}

~Node() {

delete left;

delete right;

left = NULL;

right = NULL;

}

bool isLeaf() {

return (left == NULL && right == NULL) ? true : false;

}

// Returns true if the node is a leaf and false otherwise

T data;

Node \*left;

Node \*right;

};

// Utility struct to aid in balancing a tree. Keeps the parent and child node addresses

struct NodePair {

Node \*currentNode;

Node \*parentNode;

};

// Member node variables

Node \*root;

unsigned int size; // Total number of nodes in the tree

// Utility functions for balancing.

bool isBalanced();

// Returns true if the tree is balanced.

Node\* rotateLeft(Node\* node);

// Returns a reference to the new root after rotating left.

Node\* rotateRight(Node\* node);

// Returns a reference to the new root after rotating right.

Node\* rotateLeftRight(Node\* node);

// Returns a reference to the new root after a double rotation of left then right.

Node\* rotateRightLeft(Node\* node);

// Returns a reference to the new root after a double rotation of right then left.

int heightDifference(Node\* node);

// Returns the difference between the left and right subtrees.

int getHeight(Node \*node);

// Returns the height of the tree/subtree

Node\* rebalance(Node\* node);

// Balances the tree.

// Utility functions for removing, inserting, copy constructor, and assignment operator

void init();

// Initializes an empty tree

NodePair findNode(const T& entry);

// Finds the parent and the node where the entry is found. If not found the node will be NULL.

// Helps the contains and remove functions.

NodePair predecessor(Node \*node);

// Returns a node pair of the right child node and parent node. Helps the remove function.

void removeNode(Node \*nodeToRemove, Node \*parentNode);

// Removes a node and exchanges it with its parent

void copyTree(const Tree<T>& obj);

// Helper function for copy constructor and assignment operator Recursively copies a tree in order.

// Utility functions for errors and finding an entry

Node\* location(Node \*node, const T& entry) const;

// Returns node where the passed entry is found. Returns NULL if it is not found.

void memError() const;

// Helper function for issuing memory allocation error.

void inOrder(Node \* node) const;

// Traverses the tree in order

void preOrder(Node \* node) const;

// Traverses the tree in pre order

void postOrder(Node \* node) const;

// Traverses the tree in post order

void toStringInOrder(Node\* node, std::stringstream& stream);

// Returns a string of the tree contents in in-order

void toStringPreOrder(Node\* node, std::stringstream& stream);

// Returns a string of the tree contents in in-order

void toStringPostOrder(Node\* node, std::stringstream& stream);

// Returns a string of the tree contents in in-order

public:

Tree();

// Default constructor

Tree(const T& entry);

// Main constructor

Tree(const Tree<T>& obj);

// Copy constructor

~Tree();

// Destructor

Tree<T>& operator=(const Tree<T>& obj);

// Overloaded assignment operator

void insert(const T& entry);

// Inserts the passed entry into the tree

void remove(const T& entry) throw (Empty);

// deletes an entry from the tree. Throws an empty exception if the tree is empty.

bool contains(const T& entry) const;

// Returns true if an entry is in the tree, false otherwise.

void inOrder() const;

// Traverses the tree in order

void preOrder() const;

// Traverses the tree in pre order

void postOrder() const;

// Traverses the tree in post order

void levelOrder()const;

// Traverses the tree breadth first e.g. level-order

T& getRoot() const throw (Empty);

// Returns a reference to the root entry

unsigned int length() {

return size;

}

// Returns the size of the tree

unsigned int height();

// Returns the height of the tree

void clear() throw (Empty);

// Removes all elements in the tree

bool isEmpty() const {

return root == NULL;

}

// Returns true if the tree is empty

std::string toStringInOrder();

// Returns a string of the tree contents in in-order

std::string toStringPreOrder();

// Returns a string of the tree contents in pre-order

std::string toStringPostOrder();

// Returns a string of the tree contents in post-order

std::string toStringLevelOrder();

// Returns a string of the tree contents in level-order

};

// Default constructor

template<class T>

Tree<T>::Tree() {

init();

}

// Main constructor

template<class T>

Tree<T>::Tree(const T& entry) {

init();

insert(entry);

}

// Copy constructor

template<class T>

Tree<T>::Tree(const Tree<T>& obj) {

// Initialize root to NULL and size to 0

init();

// Copy all of obj's data

copyTree(obj);

}

// Destructor

template<class T>

Tree<T>::~Tree() {

if (root != NULL) {

delete root;

root = NULL; // Point it to NULL so we don't get garbage accessing it later

size = 0;

}

}

// Overloaded assignment operator

template<class T>

Tree<T>& Tree<T>::operator=(const Tree<T>& obj) {

// Clear the current tree of all data

if (!isEmpty())

clear();

// Initialize root to NULL and size to 0

init();

// Copy all of obj's data

copyTree(obj);

}

// Removes all the entries from the tree

template<class T>

void Tree<T>::clear() throw (Empty) {

if (root != NULL) {

delete root;

root = NULL; // Point it to NULL so we don't get garbage accessing it later

size = 0;

} else {

std::string message = "Tree ";

throw Empty(message);

}

}

// Insert an entry into the tree. Iterate through the nodes until a

// place is found for the new node. Tree is re-balanced after insertion.

template<class T>

void Tree<T>::insert(const T& entry) {

// If we have an empty tree just add a root

if (root == NULL) {

Node \* node = new Node(entry, NULL, NULL);

root = node;

size++;

} else {

// If it is not empty go through the tree to find a place to put the

// entry

Node \*node;

node = root;

bool added = false;

while (!added) {

// If the node is less than, go through the left tree and its

// nodes

if (entry < node->data) {

// Find if the left node is empty

if (node->left == NULL) {

// Create a new node and set its data member

Node \*leaf = new Node(entry, NULL, NULL);

// Now set it as the left subtree of the current node

node->left = leaf;

size++;

added = true;

} else {

// If not go to the next left node

node = node->left;

}

}// Otherwise go through the right tree and its nodes to find a

// place to add it to the tree

else {

if (node->right == NULL) {

// Create a new node and set its data member

Node \*leaf = new Node(entry, NULL, NULL);

// Now link it as the right sub node

node->right = leaf;

size++;

added = true;

} else {

// If not go to the next right node

node = node->right;

}

}

} // end while

root = rebalance(root);

} // end else

}

template<class T>

void Tree<T>::remove(const T& entry) throw (Empty) {

if (root != NULL) {

// Find the entry and its parent that matches the entry

NodePair pair = findNode(entry);

Node \*nodeWithEntry = pair.currentNode;

Node \*parentEntryNode = pair.parentNode;

// If the entry was found remove it

if (nodeWithEntry != NULL) {

// If this node has two children find one child that itself has

// only one child or is a leaf. The replace the contents of that

// node with nodeWithEntry. We can delete this inorder predecessor node.

if (nodeWithEntry->left != NULL && nodeWithEntry->right != NULL) {

pair = predecessor(nodeWithEntry);

Node \*nodeToRemove = pair.currentNode;

parentEntryNode = pair.parentNode;

nodeWithEntry->data = nodeToRemove->data;

nodeWithEntry = nodeToRemove;

}

// Remove the in-order predecessor

removeNode(nodeWithEntry, parentEntryNode);

// Balance the tree

root = rebalance(root);

}

} else {

std::string message = "Tree";

throw Empty(message);

}

}

// Returns true if the tree contains the passed entry, false otherwise.

template<class T>

bool Tree<T>::contains(const T& entry) const {

Node \*node = location(root, entry);

return (node != NULL ? true : false);

}

// Tree traversal - at moment only prints

// Traverses the tree in order

template<class T>

void Tree<T>::inOrder() const {

inOrder(root);

}

// Traverses the tree in pre order

template<class T>

void Tree<T>::preOrder() const {

preOrder(root);

}

// Traverses the tree in post order

template<class T>

void Tree<T>::postOrder() const {

postOrder(root);

}

// Traverse the tree breadth first

template<class T>

void Tree<T>::levelOrder() const {

if (root != NULL) {

// To keep track of the node by level

Queue<Node\*> traverse;

// Prepare the queue

Node \*cursor = root;

traverse.push(cursor);

// Iterate through the queue adding and removing nodes by level

while (!traverse.isEmpty()) {

cursor = traverse.front();

traverse.pop();

std::cout << cursor->data << "\n";

// Store the left child

if (cursor->left != NULL) {

// Save the left children for later use

traverse.push(cursor->left);

}

// Store the right child

if (cursor->right != NULL) {

// Save the left children for later use

traverse.push(cursor->right);

}

}

}

}

template<class T>

T& Tree<T>::getRoot() const throw (Empty) {

if (root != NULL)

return root->data;

else {

std::string message = "Tree ";

throw Empty(message);

}

}

// Returns the actual height of the whole tree

template<class T>

unsigned int Tree<T>::height() {

if (!isEmpty())

return getHeight(root);

else return 0;

}

// Returns a string of the tree contents in in-order

template<class T>

std::string Tree<T>::toStringInOrder() {

std::stringstream s;

std::string contents;

toStringInOrder(root, s);

// Now get the contents of the stream

contents = s.str();

return contents;

}

// Returns a string of the tree contents in pre-order

template<class T>

std::string Tree<T>::toStringPreOrder() {

std::stringstream s;

std::string contents;

// Fill the string stream using the recursive function

toStringPreOrder(root, s);

// Now get the contents of the stream

contents = s.str();

return contents;

}

// Returns a string of the tree contents in post-order

template<class T>

std::string Tree<T>::toStringPostOrder() {

std::stringstream s;

std::string contents;

// Fill the string stream using the recursive function

toStringPostOrder(root, s);

// Now get the contents of the stream

contents = s.str();

return contents;

}

// Returns a string of the tree contents in level-order

template<class T>

std::string Tree<T>::toStringLevelOrder() {

std::stringstream s;

std::string contents = "in order";

if (root != NULL) {

// To keep track of the node by level

Queue<Node\*> traverse;

// Prepare the queue

Node \*cursor = root;

traverse.push(cursor);

// Iterate through the queue adding and removing nodes by level

while (!traverse.isEmpty()) {

cursor = traverse.front();

traverse.pop();

s << cursor->data << " ";

// Store the left child

if (cursor->left != NULL) {

// Save the left children for later use

traverse.push(cursor->left);

}

// Store the right child

if (cursor->right != NULL) {

// Save the left children for later use

traverse.push(cursor->right);

}

}

}

contents = s.str();

return contents;

}

// Utility functions

// Fills a string stream with the node data in in-order

template<class T>

void Tree<T>::toStringInOrder(Node\* node, std::stringstream& stream) {

if (node != NULL) {

toStringInOrder(node->left, stream);

stream << node->data << " ";

toStringInOrder(node->right, stream);

}

}

// Fills a string stream with the node data in pre-order

template<class T>

void Tree<T>::toStringPreOrder(Node\* node, std::stringstream& stream) {

if (node != NULL) {

stream << node->data << " ";

toStringPreOrder(node->left, stream);

toStringPreOrder(node->right, stream);

}

}

// Fills a string stream with the node data in pre-order

template<class T>

void Tree<T>::toStringPostOrder(Node\* node, std::stringstream& stream) {

if (node != NULL) {

toStringPostOrder(node->left, stream);

toStringPostOrder(node->right, stream);

stream << node->data << " ";

}

}

// Helper functions for returning strings depending on traversal

// Helper functions for traversing and printing the list

template<class T>

void Tree<T>::inOrder(Node\* node) const {

// If the node is null it will print the parent node

if (node != NULL) {

// First get the left node

inOrder(node->left);

std::cout << node->data << std::endl;

// Get the right node

inOrder(node->right);

}

}

template<class T>

void Tree<T>::preOrder(Node\* node) const {

// If the node is null it will print the parent node

if (node != NULL) {

// Print

std::cout << node->data << std::endl;

// First get the left node

preOrder(node->left);

// Get the right node

preOrder(node->right);

}

}

template<class T>

void Tree<T>::postOrder(Node\* node) const {

// If the node is null it will print the parent node

if (node != NULL) {

// First get the left node

postOrder(node->left);

// Get the right node

postOrder(node->right);

// Print

std::cout << node->data << std::endl;

}

}

// Helper function to copy constructor and assignment operator. Initializes root to NULL and size to 0.

template<class T>

void Tree<T>::init() {

root = NULL;

size = 0;

}

// Helper function for the copy constructor and assignment operator. Copies the passed tree object

// iteratively in breath first traversal

template<class T>

void Tree<T>::copyTree(const Tree<T>& obj) {

if (!obj.isEmpty()) {

Queue<Node\*> originQueue;

Queue<Node\*> copyQueue;

// Node to traverse the original tree

Node \*cursor;

cursor = obj.root;

// create a root node

Node \*newNode;

try {

newNode = new Node(cursor->data, NULL, NULL);

} catch (const::std::bad\_alloc&) {

memError();

}

size++; // Increment the total size of the tree

root = newNode; // Set the root to be the new node

originQueue.push(cursor);

copyQueue.push(root);

// Iterate through the nodes creating and copying tree nodes by level

while (!originQueue.isEmpty() && !copyQueue.isEmpty()) {

cursor = originQueue.front();

originQueue.pop();

Node \*temp = copyQueue.front();

copyQueue.pop();

// Process and store the left children

if (cursor->left != NULL) {

// Create a new left child node

// and allocate memory for it

Node \*leftChild;

try {

leftChild = new Node(cursor->left->data, NULL, NULL);

} catch (const::std::bad\_alloc&) {

memError();

}

// Link it with its parent node

temp->left = leftChild;

++size;

// Save the left children for later use

originQueue.push(cursor->left);

copyQueue.push(leftChild);

}

// Process and store the right children

if (cursor->right != NULL) {

// Create a new left child node

// and allocate memory for it

Node \*rightChild;

try {

rightChild = new Node(cursor->right->data, NULL, NULL);

} catch (const::std::bad\_alloc&) {

memError();

}

// Link it with its parent node

temp->right = rightChild;

++size;

// Save the left children for later use

originQueue.push(cursor->right);

copyQueue.push(rightChild);

}

}

}

}

// Returns true if the tree is balanced.

template<class T>

bool Tree<T>::isBalanced() {

// Get left and tree subtree heights

int left = getHeight(root->left);

int right = getHeight(root->right);

// Return true if the difference between

return ((left - right) > 1 || (left - right < -1) ? false : true);

}

// Returns a reference to the new root after rotating left.

template<class T>

typename Tree<T>::Node\* Tree<T>::rotateLeft(Node \*node) {

// Code based on Carrano's book algorithm

Node \*newRoot = node->right;

node->right = newRoot->left;

newRoot->left = node;

return newRoot;

}

// Returns a reference to the new root after rotating right.

template<class T>

typename Tree<T>::Node\* Tree<T>::rotateRight(Node \*node) {

// Code based on Carrano's algorithm in book

Node \*newRoot = node->left;

node->left = newRoot->right;

newRoot->right = node;

return newRoot;

}

// Returns a reference to the new root after a double rotation of left then right.

template<class T>

typename Tree<T>::Node\* Tree<T>::rotateLeftRight(Node \*node) {

// Code based on Carrano's algorithm

Node \*newRoot = node->left;

node->left = rotateLeft(newRoot);

return rotateRight(node);

}

template<class T>

typename Tree<T>::Node\* Tree<T>::rotateRightLeft(Node \*node) {

// Code based on Carrano's algorithm

Node \*newRoot = node->right;

node->right = rotateRight(newRoot);

return rotateLeft(node);

}

// Returns a reference to the new root after a double rotation of right then left.

// Returns the difference between the left and right subtrees.

template<class T>

int Tree<T>::heightDifference(Node \*node) {

return (getHeight(node->left) - getHeight(node->right));

}

// Returns the height of the subtree

template<class T>

int Tree<T>::getHeight(Node\* node) {

// If the tree is empty return 0

if (node == NULL) {

return 0;

}

// Get the height of the left and right subtrees

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

// Now return the maximum between the left tree and right tree heights.

return (leftHeight > rightHeight ? (1 + leftHeight) : (1 + rightHeight));

}

// Balances the tree according to the height difference between its subtrees

template<class T>

typename Tree<T>::Node\* Tree<T>::rebalance(Node \*node) {

if (node != NULL) {

// Get the height difference between left and right subtrees

int heightDiff = heightDifference(node);

if (heightDiff > 1) {

// Now go through the subtrees and get difference with its

// subtrees

// Rotate as necessary

if (heightDifference(node->left) > 0) {

node = rotateRight(node);

} else {

node = rotateLeftRight(node);

}

} else if (heightDiff < -1) {

if (heightDifference(node->right) < 0) {

node = rotateLeft(node);

} else {

node = rotateRightLeft(node);

}

}

}

return node;

}

// Helper function to remove function. If found returns a NodePair where the

// child node of the NodePair is where where the entry occurs and the parentNode

// is the parent of that node. It returns a NodePair of nulls if not found.

template<class T>

typename Tree<T>::NodePair Tree<T>::findNode(const T& entry) {

// Now we need to go as far to the right as possible

// Initialize the node pair to return

NodePair result;

result.currentNode = NULL;

result.parentNode = NULL;

Node \*cursor = root;

Node \*cursorParent = NULL;

// Keep going right until we reach null

while (cursor != NULL) {

// If we found it return the pair of nodes

if (cursor->data == entry) {

result.currentNode = cursor;

result.parentNode = cursorParent;

return result;

}// Search the left subtree if entry is less than current node data

else if (entry < cursor->data) {

cursorParent = cursor;

cursor = cursor->left;

}// Otherwise continue searching in the right subtree

else {

cursorParent = cursor;

cursor = cursor->right;

}

}

// Return the right subtree and its parent node

return result;

}

// Helper function to remove function. Returns the rightmost node of the left

// subtree and its parent node. The rightmost node is the inorder predecessor

// of the node to remove in the remove function.

template<class T>

typename Tree<T>::NodePair Tree<T>::predecessor(Node\* node) {

// First go left

Node \*leftSubTree = node->left;

// Now we need to go as far to the right as possible

Node \*rightChild = leftSubTree;

Node \*parentNode = node;

// Keep going right until we reach null

while (rightChild->right != NULL) {

parentNode = rightChild;

rightChild = rightChild->right;

}

// Return the right subtree and its parent node

NodePair pair;

pair.currentNode = rightChild;

pair.parentNode = parentNode;

return pair;

}

// Helper function to remove function. It connects the parentNode with nodeToRemove

// left or right child if any then deallocates nodeToRemove.

template<class T>

void Tree<T>::removeNode(Node \*nodeToRemove, Node \*parentNode) {

Node \*childNode;

// Find if nodeToRemove has either a left or right child

if (nodeToRemove->left != NULL) {

childNode = nodeToRemove->left;

} else {

childNode = nodeToRemove->right;

}

// If the node to delete is the root, just change it with its child

// and delete the nodeToRemove

if (nodeToRemove == root) {

root = childNode; // Set it to NULL

}// If the node to remove is the left child of parent, link nodeToRemove's

// childNode to parentNode and delete nodeToRemove

else if (parentNode->left == nodeToRemove) {

parentNode->left = childNode;

}// If the node to remove is the right child of parent, link nodeToRemove's

// childNode to parentNode and delete nodeToRemove

else {

parentNode->right = childNode;

}

// Delete the node and decrease the size

delete nodeToRemove;

--size;

}

// Function provided by Dr. Mark Lehr. It issues an error and terminates the program if memory fails to allocate.

template<class T>

void Tree<T>::memError() const {

std::cout << "ERROR:Cannot allocate memory.\n";

exit(EXIT\_FAILURE);

}

// Finds the node where an entry is located. If it is not found returns NULL

template<class T>

typename Tree<T>::Node\* Tree<T>::location(Node\* node, const T& entry) const {

if (node != NULL) {

// If we find it we exit the recursive call

if (node->data == entry) {

return node;

}

// If we don't find it in current node go to the subtrees

else if (node->data > entry) {

return location(node->left, entry);

}// TODO remove if condition from else if

// Can remove the if condition. Not necessary

else {

return location(node->right, entry);

}

}

// We never found it

return NULL;

}

#endif /\* TREE\_H \*/