Министерство высшего образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет» (ПНИПУ)

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЁТ

по работе

Тема: «Решение задачи Коммивояжера и АРМ Тикера»

Выполнил

Студент группы РИС-22-2б

Радостев И.А.

Проверил доц. Кафедры ИТАС

Полякова О. А.

Пермь 2023

**Постановка задачи**

1. Создание и разработка автоматизированного рабочего места(арм) Тикера
2. В программе должны генерироваться случайные концерты, дата и цена. Отслеживание концертов по дате, покупка и возврат, доступ к купленным билетам.

Используемые инструменты.

## Для реализации АРМ Тикера был использован язык программирования C++, а также фреймворк Qt, с собственной средой разработки Qt Creator. Для записи видео с экрана компьютера была использована встроенная утилита для создания скриншотов и записи скринкастов QuickTime Player, для редактирования видео Final Cut. Для создания иконки программы- Adobe Photoshop 2022.

Код программы

Main.cpp:

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(*argc*, *argv*);

*//* *Создание* *объекта* *QIcon* *с* *указанием* *пути* *к* *иконке*

QIcon avatarIcon(":/images/ticker-logo.png");

*//* *Установка* *иконки* *для* *главного* *окна* *приложения*

a.setWindowIcon(avatarIcon);

MainWindow w;

w.show();

*return* a.exec();

}

Mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "mainwindowbuy.h"

#include "mainwindowreturn.h"

#include <QMessageBox>

#include <QFile>

#include <QTextStream>

#include <QDate>

#include <QCheckBox>

#include "concertsbydatedialog.h"

MainWindow::**MainWindow**(QWidget \*parent)

: QMainWindow(*parent*)

, ui(*new* Ui::MainWindow)

{

ui->setupUi(*this*);

*this*->setWindowTitle("Ticker");

}

MainWindow::~***MainWindow***()

{

*delete* ui;

}

void MainWindow::**on\_pushButton\_clicked**()

{

MainWindowBuy buyDialog;

buyDialog.*exec*();

}

void MainWindow::**on\_pushButton\_2\_clicked**()

{

MainWindowReturn returnDialog;

returnDialog.*exec*();

}

void MainWindow::**on\_pushButton\_3\_clicked**()

{

*//* *Открытие* *файла* *"buyerdata.txt"* *для* *чтения*

QFile file("buyerdata.txt");

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

{

QTextStream in(*&file*);

QString ticketNumbers;

*//* *Чтение* *номеров* *билетов* *из* *файла*

*while* (!in.atEnd())

{

QString line = in.readLine();

ticketNumbers += line + "\n";

}

file.*close*();

*//* *Открытие* *информационного* *окна* *с* *перечислением* *номеров* *билетов*

QMessageBox::information(*this*, "Ваши билеты:", ticketNumbers);

}

*else*

{

QMessageBox::critical(*this*, "Error", "Failed to open buyerdata.txt");

}

}

void MainWindow::**on\_calendarWidget\_clicked**(*const* QDate& date)

{

ConcertsByDateDialog dialog(date);

dialog.*exec*();

}

*//* *Слот* *для* *получения* *информации* *о* *балансе* *и* *вывода* *уведомления*

void MainWindow::**updateBalance**(double balance)

{

QMessageBox::information(*this*, "My Balance", "Your account balance is: " + QString::number(balance));

}

void MainWindow::**on\_actionMyBalance\_clicked**()

{

MainWindowBuy buyDialog;

connect(&buyDialog, &MainWindowBuy::balanceUpdated, *this*, &MainWindow::updateBalance); *//* *Подключение* *сигнала* *и* *слота*

buyDialog.*exec*();

}

void MainWindow::**addTestConcert**(*const* QString& artist, *const* QDate& date, double price)

{

*//* *Запись* *информации* *о* *тестовом* *концерте* *в* *файл* *"concertdata.txt"*

QFile file("concertdata.txt");

*if* (file.*open*(QIODevice::*Append* | QIODevice::*Text*))

{

QTextStream out(*&file*);

out << artist << "," << date.toString("dd.MM.yyyy") << "," << price << endl;

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Error", "Failed to open concertdata.txt");

}

}

void MainWindow::**clearTestConcerts**()

{

*//* *Очистка* *тестовых* *концертов*

QFile file("concertdata.txt");

*if* (file.*open*(QIODevice::*WriteOnly* | QIODevice::*Text*))

{

file.*resize*(0); *//* *Очистка* *содержимого* *файла*

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Error", "Failed to open concertdata.txt");

}

}

void MainWindow::**on\_actionTest\_Mode\_1\_triggered**()

{

*//* *Включение* *тестового* *режима*

QMessageBox::information(*this*, "Test Mode", "Test mode enabled.");

generateTestConcerts(); *//* *Генерация* *тестовых* *концертов*

}

void MainWindow::**on\_actionTest\_Mode\_2\_triggered**()

{

*//* *Выключение* *тестового* *режима*

QMessageBox::information(*this*, "Test Mode", "Test mode disabled.");

clearTestConcerts(); *//* *Очистка* *тестовых* *концертов*

}

void MainWindow::**generateTestConcerts**()

{

*//* *Генерация* *тестовых* *концертов*

QDate currentDate = QDate::currentDate();

qsrand(*static\_cast*<uint>(QTime::currentTime().msec()));

*//* *Генерация* *случайных* *данных* *для* *тестовых* *концертов*

QStringList artists = { "Artist1", "Artist2", "Artist3" };

QStringList firstNames = { "John", "Emma", "Michael", "Sophia" };

QStringList lastNames = { "Smith", "Johnson", "Williams", "Brown" };

*for* (int i = 0; i < 90; ++i) {

*//* *Генерация* *случайной* *даты*

QDate randomDate = currentDate.addDays(qrand() % 30); *//* *Случайная* *дата* *в* *течение* *30* *дней* *от* *текущей* *даты*

*//* *Генерация* *случайной* *цены* *от* *50* *до* *200*

int randomPrice = qrand() % (200 - 50 + 1) + 50;

*//* *Генерация* *случайного* *имени* *и* *фамилии*

QString randomFirstName = firstNames[qrand() % firstNames.size()];

QString randomLastName = lastNames[qrand() % lastNames.size()];

*//* *Добавление* *тестового* *концерта*

addTestConcert(QString("%1 %2").arg(randomFirstName, randomLastName), randomDate, randomPrice);

}

emit testConcertsGenerated();

}

Mainwindow.h  
#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QDate>

#include "mainwindowbuy.h"

#include "mainwindowreturn.h"

QT\_BEGIN\_NAMESPACE

*namespace* **Ui** {

*class* **MainWindow**;

}

QT\_END\_NAMESPACE

*class* **MainWindow** : *public* QMainWindow

{

Q\_OBJECT

*public*:

**MainWindow**(QWidget \*parent = *nullptr*);

~***MainWindow***();

void **updateAccountBalance**(double balance);

void **on\_actionMyBalance\_clicked**();

*public* slots:

void **updateBalance**(double balance);

*private* slots:

void **on\_actionTest\_Mode\_1\_triggered**();

void **on\_actionTest\_Mode\_2\_triggered**();

void **on\_pushButton\_clicked**();

void **on\_pushButton\_2\_clicked**();

void **on\_pushButton\_3\_clicked**();

void **clearTestConcerts**();

void **on\_calendarWidget\_clicked**(*const* QDate &date);

void **generateTestConcerts**();

void **addTestConcert**(*const* QString& artist, *const* QDate& date, double price);

*private*:

Ui::MainWindow \*ui;

double accountBalance;

QDate selectedDate;

signals:

void **testConcertsGenerated**();

};

#endif *//* *MAINWINDOW\_H*

Mainwindowbuy.cpp

#include "mainwindowbuy.h"

#include "ui\_mainwindowbuy.h"

#include <QMessageBox>

#include <QFile>

#include <QTextStream>

#include <QDateTime>

MainWindowBuy::**MainWindowBuy**(QWidget \*parent) :

QDialog(*parent*),

ui(*new* Ui::MainWindowBuy),

accountBalance(10000.0) *//* *Изначальный* *баланс* *счета*

{

ui->setupUi(*this*);

*this*->setWindowTitle("Купить");

*//* *Загрузка* *данных* *о* *концертах*

loadConcertData();

*//* *Подключение* *сигнала* *изменения* *выбора* *элемента* *в* *таблице*

connect(ui->tableWidgetConcerts, &QTableWidget::itemSelectionChanged, *this*, &MainWindowBuy::onTableWidgetSelectionChanged);

*//* *Подключение* *сигнала* *нажатия* *кнопки* *покупки*

connect(ui->pushButton, &QPushButton::clicked, *this*, &MainWindowBuy::on\_pushButtonBuy\_clicked);

*//* *Отключение* *кнопки* *покупки* *при* *запуске*

*//* *Отключение* *кнопки* *покупки* *при* *запуске*

ui->pushButton->setEnabled(*false*);

}

MainWindowBuy::~***MainWindowBuy***()

{

*delete* ui;

}

void MainWindowBuy::**setConcertInfo**(*const* QString& artist, *const* QDate& concertDate, double price)

{

selectedArtist = artist;

selectedConcertDate = concertDate;

concertPrice = price;

}

void MainWindowBuy::**setConcertData**(*const* QString &artist, *const* QDate &concertDate, double price)

{

ui->tableWidgetConcerts->clearContents();

ui->tableWidgetConcerts->setRowCount(0);

int row = ui->tableWidgetConcerts->rowCount();

ui->tableWidgetConcerts->insertRow(row);

ui->tableWidgetConcerts->setItem(row, 0, *new* QTableWidgetItem*(artist)*);

ui->tableWidgetConcerts->setItem(row, 1, *new* QTableWidgetItem*(concertDate.toString(*"dd.MM.yyyy"*))*);

ui->tableWidgetConcerts->setItem(row, 2, *new* QTableWidgetItem*(*QString*::number(price))*);

}

void MainWindowBuy::**on\_pushButtonBuy\_clicked**()

{

*//* *Получение* *имени* *покупателя* *из* *поля* *ввода*

QString buyerName = ui->lineEditBuyerName->text();

*//* *Проверка,* *выбран* *ли* *концерт* *и* *введено* *ли* *имя* *покупателя*

*if* (!selectedArtist.isEmpty() && !buyerName.isEmpty())

{

*//* *Проверка,* *достаточно* *ли* *средств* *на* *счете*

*if* (deductMoneyFromAccount(concertPrice))

{

*//* *Генерация* *номера* *билета*

QString ticketNumber = generateTicketNumber(selectedArtist);

*//* *Формирование* *сообщения* *о* *покупке*

QString message = "Поздравляем, " + buyerName + "!\n";

message += "Вы успешно приобрели билет на концерт " + selectedArtist + ".\n";

message += "Дата концерта: " + selectedConcertDate.toString("dd.MM.yyyy") + "\n";

message += "Цена билета: " + QString::number(concertPrice) + "\n";

message += "Номер билета: " + ticketNumber;

*//* *Вывод* *сообщения* *о* *покупке*

QMessageBox::information(*this*, "Покупка билета", message);

*//* *Сброс* *выбора* *и* *отключение* *кнопки* *покупки*

ui->tableWidgetConcerts->clearSelection();

ui->pushButton->setEnabled(*false*);

*//* *Отправка* *сигнала* *обновления* *баланса*

emit balanceUpdated(accountBalance);

*//* *Запись* *информации* *о* *покупке* *в* *файл*

QFile file("buyerdata.txt");

*if* (file.*open*(QIODevice::*Append* | QIODevice::*Text*))

{

QTextStream out(*&file*);

out << "Name: " << buyerName << endl;

out << "Concert: " << selectedArtist << endl;

out << "Date: " << selectedConcertDate.toString("dd.MM.yyyy") << endl;

out << "Number: " << ticketNumber << endl;

out << "Price: " << QString::number(concertPrice) << endl;

out << "-----------------------------------" << endl;

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Ошибка", "Не удалось открыть файл buyerdata.txt для записи.");

}

}

*else*

{

*//* *Вывод* *сообщения* *об* *отсутствии* *средств* *на* *счете*

QMessageBox::critical(*this*, "Ошибка", "На вашем счете недостаточно средств.");

}

}

*else*

{

*//* *Вывод* *сообщения* *о* *незаполненных* *полях*

QMessageBox::warning(*this*, "Ошибка", "Пожалуйста, выберите концерт и введите ваше имя.");

}

}

void MainWindowBuy::**onTableWidgetSelectionChanged**()

{

*//* *Получение* *выбранной* *строки*

QList<QTableWidgetItem\*> selectedItems = ui->tableWidgetConcerts->selectedItems();

*//* *Проверка,* *выбрана* *ли* *строка*

*if* (!selectedItems.isEmpty())

{

*//* *Получение* *информации* *о* *выбранном* *концерте*

int row = ui->tableWidgetConcerts->row(selectedItems.at(0));

QString artist = ui->tableWidgetConcerts->item(row, 0)->text();

QDate concertDate = QDate::fromString(ui->tableWidgetConcerts->item(row, 1)->text(), "dd.MM.yyyy");

double price = ui->tableWidgetConcerts->item(row, 2)->text().toDouble();

*//* *Установка* *информации* *о* *концерте*

setConcertInfo(artist, concertDate, price);

*//* *Включение* *кнопки* *покупки*

ui->pushButton->setEnabled(*true*);

}

*else*

{

*//* *Отключение* *кнопки* *покупки*

ui->pushButton->setEnabled(*false*);

}

}

bool MainWindowBuy::**deductMoneyFromAccount**(double amount)

{

*//* *Проверка,* *достаточно* *ли* *средств* *на* *счете*

*if* (accountBalance >= amount)

{

*//* *Вычет* *суммы* *покупки* *со* *счета*

accountBalance -= amount;

*return* *true*;

}

*else*

{

*return* *false*;

}

}

QString MainWindowBuy::**generateTicketNumber**(*const* QString& concert)

{

*//* *Получение* *текущей* *даты*

QDate currentDate = QDate::currentDate();

*//* *Преобразование* *даты* *в* *строку* *без* *точек* *(ддммгггг)*

QString dateString = currentDate.toString("ddMMyyyy");

*//* *Преобразование* *имени* *исполнителя* *в* *верхний* *регистр* *и* *взятие* *первых* *трех* *символов*

QString artistCode = concert.left(3).toUpper();

*//* *Получение* *текущего* *времени* *в* *миллисекундах*

qint64 currentMillis = QDateTime::currentDateTime().currentMSecsSinceEpoch();

*//* *Получение* *последних* *трех* *цифр* *из* *текущего* *времени* *в* *миллисекундах*

QString millisString = QString::number(currentMillis % 1000);

*//* *Генерация* *номера* *билета* *на* *основе* *даты,* *кода* *исполнителя* *и* *миллисекунд*

QString ticketNumber = artistCode + dateString + millisString;

*return* ticketNumber;

}

void MainWindowBuy::**loadConcertData**()

{

ui->tableWidgetConcerts->setColumnCount(3);

ui->tableWidgetConcerts->setHorizontalHeaderLabels({"Concert", "Date", "Price"});

QFile file("concertdata.txt");

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

{

QTextStream in(*&file*);

*while* (!in.atEnd())

{

QString line = in.readLine();

QStringList data = line.split(",");

*if* (data.size() >= 3)

{

QString concert = data[0];

QString date = data[1];

QString price = data[2];

*//* *Добавить* *запись* *в* *таблицу*

int row = ui->tableWidgetConcerts->rowCount();

ui->tableWidgetConcerts->insertRow(row);

ui->tableWidgetConcerts->setItem(row, 0, *new* QTableWidgetItem*(concert)*);

ui->tableWidgetConcerts->setItem(row, 1, *new* QTableWidgetItem*(date)*);

ui->tableWidgetConcerts->setItem(row, 2, *new* QTableWidgetItem*(price)*);

}

}

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Error", "Failed to open concertdata.txt");

}

}

Mainwindowbuy.h

#ifndef MAINWINDOWBUY\_H

#define MAINWINDOWBUY\_H

#include <QDialog>

#include <QDate>

*namespace* **Ui** {

*class* **MainWindowBuy**;

}

*class* **MainWindowBuy** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **MainWindowBuy**(QWidget \*parent = *nullptr*);

~***MainWindowBuy***();

void **setConcertInfo**(*const* QString& artist, *const* QDate& concertDate, double price);

void **setConcertData**(*const* QString &artist, *const* QDate &concertDate, double price);

*private* slots:

void **on\_pushButtonBuy\_clicked**();

void **onTableWidgetSelectionChanged**();

*private*:

Ui::MainWindowBuy \*ui;

QString selectedArtist;

QDate selectedConcertDate;

double accountBalance;

double concertPrice;

bool **deductMoneyFromAccount**(double amount);

QString **generateTicketNumber**(*const* QString& concert);

void **loadConcertData**();

signals:

void **balanceUpdated**(double balance);

};

#endif *//* *MAINWINDOWBUY\_H*

Mainwindowreturn.cpp

#include "mainwindowreturn.h"

#include "ui\_mainwindowreturn.h"

#include <QMessageBox>

#include <QFile>

#include <QTextStream>

MainWindowReturn::**MainWindowReturn**(QWidget \*parent) :

QDialog(*parent*),

ui(*new* Ui::MainWindowReturn)

{

ui->setupUi(*this*);

*this*->setWindowTitle("Возврат");

}

MainWindowReturn::~***MainWindowReturn***()

{

*delete* ui;

}

void MainWindowReturn::**on\_pushButton\_clicked**()

{

QString buyerName = ui->lineEdit->text();

QString ticketNumber = ui->lineEdit\_2->text();

QString returnReason = ui->lineEdit\_3->text();

*//* *Проверка* *условий* *для* *возврата* *билета*

QFile file("buyerdata.txt");

*if* (file.*open*(QIODevice::*ReadWrite* | QIODevice::*Text*))

{

bool found = *false*;

QString line;

QStringList currentTicket;

QList<QStringList> validTickets;

*while* (!file.*atEnd*())

{

line = file.readLine().trimmed();

*if* (line == "-----------------------------------")

{

*if* (!currentTicket.isEmpty())

{

QString currentTicketNumber;

QString currentBuyerName;

foreach (*const* QString& field, currentTicket)

{

*if* (field.startsWith("Number: "))

{

currentTicketNumber = field.mid(8);

}

*else* *if* (field.startsWith("Name: "))

{

currentBuyerName = field.mid(6);

}

}

*if* (currentTicketNumber == ticketNumber && currentBuyerName == buyerName)

{

found = *true*;

}

*else*

{

validTickets.append(currentTicket);

}

}

currentTicket.clear();

}

currentTicket.append(line);

}

file.*close*();

*if* (found)

{

*//* *Возврат* *средств* *на* *баланс*

refundMoneyToAccount(ticketNumber);

*//* *Запись* *информации* *о* *возврате* *в* *файл* *или* *базу* *данных*

QFile returnFile("returndata.txt");

*if* (returnFile.*open*(QIODevice::*Append* | QIODevice::*Text*))

{

QTextStream out(*&returnFile*);

foreach (*const* QString& field, currentTicket)

{

out << field << endl;

}

out << "Return Reason: " << returnReason << endl;

out << "-----------------------------------" << endl;

returnFile.*close*();

QMessageBox::information(*this*, "Возврат билета", "Билет успешно возвращен. Возврат средств обработан.");

}

*else*

{

QMessageBox::critical(*this*, "Ошибка", "Не удалось открыть файл returndata.txt");

}

}

*else*

{

QMessageBox::warning(*this*, "Некорректные данные", "Некорректный номер билета или имя покупателя.");

}

*//* *Перезапись* *файла* *buyerdata.txt* *без* *возвратившегося* *билета*

QFile rewriteFile("buyerdata.txt");

*if* (rewriteFile.*open*(QIODevice::*WriteOnly* | QIODevice::*Text*))

{

QTextStream out(*&rewriteFile*);

foreach (*const* QStringList& ticket, validTickets)

{

foreach (*const* QString& field, ticket)

{

out << field << endl;

}

out << "-----------------------------------" << endl;

}

rewriteFile.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Ошибка", "Не удалось открыть файл buyerdata.txt для перезаписи.");

}

}

*else*

{

QMessageBox::critical(*this*, "Ошибка", "Не удалось открыть файл buyerdata.txt");

}

}

void MainWindowReturn::**refundMoneyToAccount**(*const* QString& )

{

QFile file("mainwindowbuy.txt");

*if* (file.*open*(QIODevice::*ReadWrite* | QIODevice::*Text*))

{

QString line;

QStringList fields;

*while* (!file.*atEnd*())

{

line = file.readLine().trimmed();

fields = line.split(",");

*if* (fields.size() == 2 && fields.at(0) == "Account Balance")

{

double accountBalance = fields.at(1).toDouble();

*//* *Найден* *баланс* *счета,* *можно* *начислить* *сумму* *возврата*

*//* *Например,* *если* *refundAmount* *-* *сумма* *возврата*

accountBalance += 1;

*//* *Обновляем* *баланс* *счета*

file.*seek*(file.*pos*() - line.length() - 1);

QTextStream out(*&file*);

out << "Account Balance," << accountBalance << endl;

*break*;

}

}

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Ошибка", "Не удалось открыть файл mainwindowbuy.txt");

}

}

Mainwindowreturn.h

#ifndef MAINWINDOWRETURN\_H

#define MAINWINDOWRETURN\_H

#include <QDialog>

*namespace* **Ui** {

*class* **MainWindowReturn**;

}

*class* **MainWindowReturn** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **MainWindowReturn**(QWidget \*parent = *nullptr*);

~***MainWindowReturn***();

*private* slots:

void **on\_pushButton\_clicked**();

*private*:

Ui::MainWindowReturn \*ui;

void **refundMoneyToAccount**(*const* QString &ticketNumber);

double accountBalance;

};

#endif *//* *MAINWINDOWRETURN\_H*

Concertsbydatedialog.cpp

#include "concertsbydatedialog.h"

#include "ui\_concertsbydatedialog.h"

#include <QFile>

#include <QTextStream>

#include <QMessageBox>

ConcertsByDateDialog::**ConcertsByDateDialog**(*const* QDate& date, QWidget \*parent) :

QDialog(*parent*),

ui(*new* Ui::ConcertsByDateDialog)

{

ui->setupUi(*this*);

*this*->setWindowTitle("Концерты в выбранный день");

ui->tableWidgetConcerts->setColumnCount(3);

ui->tableWidgetConcerts->setHorizontalHeaderLabels({"Исполнитель", "Дата", "Цена"});

*//* *Открытие* *файла* *"concertdata.txt"* *для* *чтения*

QFile file("concertdata.txt");

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

{

QTextStream in(*&file*);

*while* (!in.atEnd())

{

QString line = in.readLine();

QStringList data = line.split(",");

*if* (data.size() >= 3)

{

QString concert = data[0];

QString dateString = data[1];

QString priceString = data[2];

*//* *Преобразование* *строки* *с* *датой* *в* *QDate*

QDate concertDate = QDate::fromString(dateString, "dd.MM.yyyy");

*//* *Проверка* *соответствия* *даты* *концерта* *выбранной* *дате*

*if* (concertDate == date)

{

*//* *Добавление* *концерта* *в* *таблицу*

int row = ui->tableWidgetConcerts->rowCount();

ui->tableWidgetConcerts->insertRow(row);

ui->tableWidgetConcerts->setItem(row, 0, *new* QTableWidgetItem*(concert)*);

ui->tableWidgetConcerts->setItem(row, 1, *new* QTableWidgetItem*(dateString)*);

ui->tableWidgetConcerts->setItem(row, 2, *new* QTableWidgetItem*(priceString)*);

}

}

}

file.*close*();

}

*else*

{

QMessageBox::critical(*this*, "Error", "Failed to open concertdata.txt");

}

}

ConcertsByDateDialog::~***ConcertsByDateDialog***()

{

*delete* ui;

}

Concertsbydatedialog.h

#ifndef CONCERTSBYDATEDIALOG\_H

#define CONCERTSBYDATEDIALOG\_H

#include <QDialog>

#include <QDate>

*namespace* **Ui** {

*class* **ConcertsByDateDialog**;

}

*class* **ConcertsByDateDialog** : *public* QDialog

{

Q\_OBJECT

*public*:

*explicit* **ConcertsByDateDialog**(*const* QDate& date, QWidget \*parent = *nullptr*);

~***ConcertsByDateDialog***();

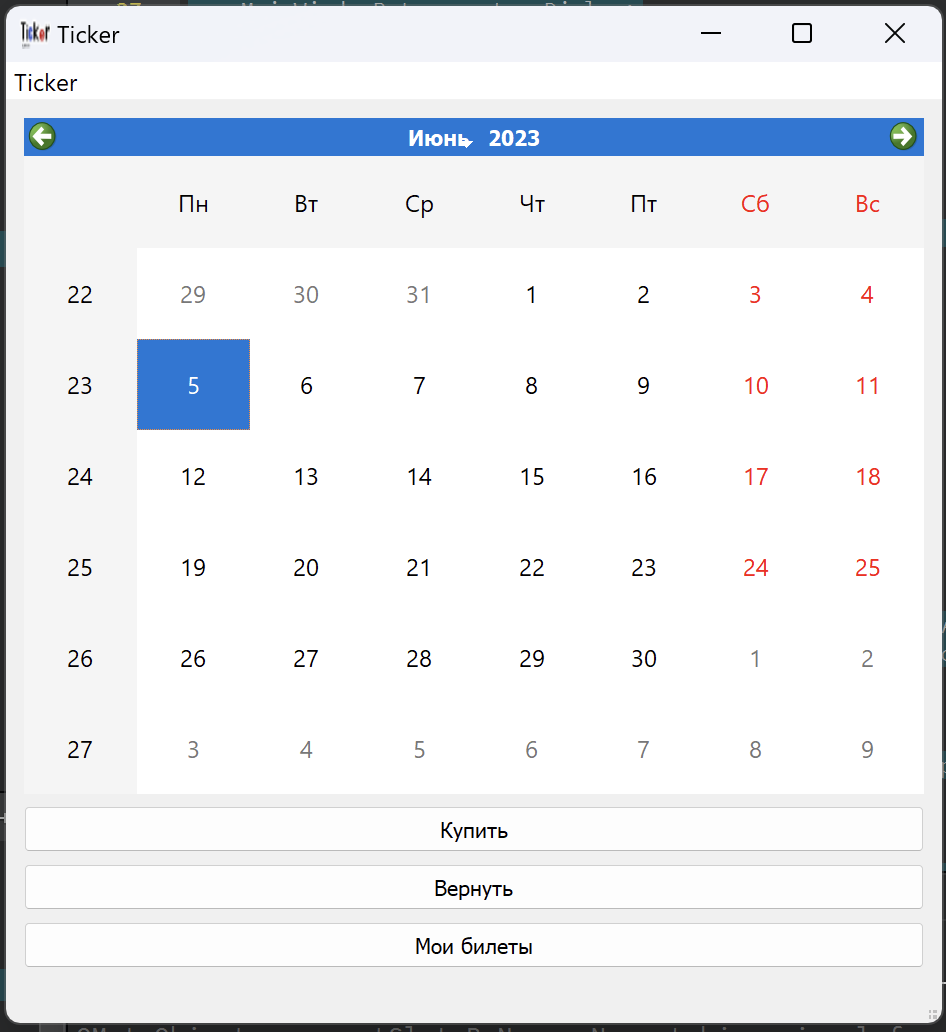
*private*:

Ui::ConcertsByDateDialog \*ui;

};

#endif *//* *CONCERTSBYDATEDIALOG\_H*

**Вывод**



**Решение задачи Коммивояжера.**

**Постановка задачи**

1) Создание и визуализация графа

2) Выполнение задачи Коммивояжера

Принцип работы кода заключается в реализации алгоритма коммивояжера с использованием метода ветвей и границ.

Основные функции в коде:

Этот код представляет собой часть графического представления графа. Он определяет класс `GraphWidget`, который имеет несколько методов:

1. `reshGraph(const Graph& graph, const PathInfo& optimalPath)`: Этот метод принимает объект типа `Graph` и информацию об оптимальном пути в качестве параметров. Он очищает сцену, вычисляет позиции вершин на сцене на основе количества вершин и размеров сцены, а затем вызывает другие функции для отрисовки ребер, вершин и оптимального пути на сцене.

2. `visGraph(const Graph& graph)`: Этот метод принимает объект типа `Graph` в качестве параметра и очищает сцену. Он вычисляет позиции вершин на сцене и затем вызывает другие функции для отрисовки ребер и вершин на сцене.

3. `drawEdges(const Graph& graph, const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку ребер графа. Он принимает объект типа `Graph` и вектор позиций вершин в качестве параметров. Для каждой пары вершин он создает линию, соединяющую их, и добавляет ее на сцену. Также он добавляет текстовую метку с весом ребра в середину каждого ребра.

4. `drawVertices(const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку вершин графа. Он принимает вектор позиций вершин в качестве параметра. Для каждой вершины он создает эллипс и добавляет его на сцену. Также он добавляет текстовую метку с номером вершины рядом с каждой вершиной.

5. `drawPath(const std::vector<int>& path, const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку оптимального пути в графе. Он принимает вектор с последовательностью вершин пути и вектор позиций вершин в качестве параметров. Он создает линии, соединяющие вершины пути, и добавляет их на сцену.

**Код**

**Main.cpp**

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(*argc*, *argv*);

MainWindow w;

w.show();

*return* a.exec();

}

Graphwidgen.cpp

#include "graph.h"

#include "graphwidget.h"

#include <QGraphicsView>

#include <QGraphicsScene>

#include <QGraphicsItem>

#include <QGraphicsLineItem>

#include <QPen>

#include <QBrush>

#include <QColor>

#include <QPointF>

#include <QRectF>

#include <cmath>

#include <QtMath>

void GraphWidget::reshGraph(const Graph& graph, const PathInfo& optimalPath) {

clearScene();

int numVertices = graph.getNumVertices();

std::vector<QPointF> vertexPositions = calculateVertexPositions(numVertices);

drawEdges(graph, vertexPositions);

drawVertices(vertexPositions);

drawPath(optimalPath.path, vertexPositions);

}

void GraphWidget::visGraph(const Graph& graph) {

clearScene();

int numVertices = graph.getNumVertices();

std::vector<QPointF> vertexPositions = calculateVertexPositions(numVertices);

drawEdges(graph, vertexPositions);

drawVertices(vertexPositions);

}

void GraphWidget::drawEdges(const Graph& graph, const std::vector<QPointF>& vertexPositions) {

const int numVertices = graph.getNumVertices();

const QPen edgePen(Qt::black);

const QFont weightFont("Arial", 10, QFont::Bold);

for (int v1 = 0; v1 < numVertices; v1++) {

for (int v2 = 0; v2 < numVertices; v2++) {

int weight = graph.getEdgeWeight(v1, v2);

if (weight > 0) {

QPointF p1 = vertexPositions[v1];

QPointF p2 = vertexPositions[v2];

QGraphicsLineItem\* line = addLine(p1.x(), p1.y(), p2.x(), p2.y(), edgePen);

line->setZValue(-1); // Поместить линию под текст

QString weightText = QString::number(weight);

QGraphicsTextItem\* weightLabel = addText(weightText);

weightLabel->setFont(weightFont);

weightLabel->setDefaultTextColor(Qt::white);

weightLabel->setPos((p1.x() + p2.x()) / 2, (p1.y() + p2.y()) / 2);

}

}

}

}

void GraphWidget::drawVertices(const std::vector<QPointF>& vertexPositions) {

const int numVertices = vertexPositions.size();

const int vertexRadius = 10;

const QPen vertexPen(Qt::white);

const QBrush vertexBrush(Qt::yellow);

for (int v = 0; v < numVertices; v++) {

QPointF position = vertexPositions[v];

QRectF rect(position.x() - vertexRadius, position.y() - vertexRadius,

2 \* vertexRadius, 2 \* vertexRadius);

addEllipse(rect, vertexPen, vertexBrush);

QGraphicsTextItem\* label = addText(QString::number(v));

label->setPos(position.x() - vertexLabelOffset, position.y() - vertexLabelOffset);

}

}

void GraphWidget::drawPath(const std::vector<int>& path, const std::vector<QPointF>& vertexPositions) {

if (path.empty())

return;

const int pathPenWidth = 2;

const QColor pathColor = Qt::red;

const QPen pathPen(pathColor, pathPenWidth);

const int numVertices = path.size();

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

int v1 = path[i];

int v2 = path[i + 1];

QPointF p1 = vertexPositions[v1];

QPointF p2 = vertexPositions[v2];

addLine(p1.x(), p1.y(), p2.x(), p2.y(), pathPen);

}

int v1 = path.back();

int v2 = path.front();

QPointF p1 = vertexPositions[v1];

QPointF p2 = vertexPositions[v2];

addLine(p1.x(), p1.y(), p2.x(), p2.y(), pathPen);

}

void GraphWidget::clearScene() {

scene()->clear();

}

std::vector<QPointF> GraphWidget::calculateVertexPositions(int numVertices) {

std::vector<QPointF> vertexPositions;

const int sceneWidth = 1000;

const int sceneHeight = 1000;

const QPointF center(sceneWidth / 2, sceneHeight / 2);

const qreal radius = qMin(sceneWidth, sceneHeight) \* 0.4;

const qreal angleIncrement = 2 \* M\_PI / numVertices;

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

qreal angle = i \* angleIncrement;

qreal x = center.x() + radius \* qCos(angle);

qreal y = center.y() + radius \* qSin(angle);

vertexPositions.emplace\_back(x, y);

}

return vertexPositions;

}

Graphwidget.h

#ifndef GRAPHWIDGET\_H

#define GRAPHWIDGET\_H

#include <QMainWindow>

#include <QObject>

#include <QWidget>

#include <QGraphicsView>

#include <QGraphicsScene>

#include <QGraphicsItem>

#include <QGraphicsLineItem>

#include <QPen>

#include <QBrush>

#include <QColor>

#include <QPointF>

#include <QRectF>

#include <QGraphicsTextItem>

#include "graph.h"

*using* *namespace* std;

*struct* **PathInfo** {

std::vector<int> path;

int cost;

**PathInfo**(*const* std::vector<int>& \_path, int \_cost) : path(\_path), cost(\_cost) {}

};

*class* **GraphWidget** : *public* QGraphicsView

{

Q\_OBJECT

*public*:

**GraphWidget**(QWidget\* parent = *nullptr*) : QGraphicsView(*parent*) {

setRenderHint(QPainter::*Antialiasing*);

setScene(*new* QGraphicsScene*(this)*);

void **setGraph**(*const* Graph& graph); *//* *Добавьте* *эту* *функцию* *для* *установки* *графа*

void **setOptimalPath**(*const* PathInfo& optimalPath); *//* *Добавьте* *эту* *функцию* *для* *установки* *оптимального* *пути*

void **reshGraph**(); *//* *Обновленная* *функция* *reshGraph*

void **visGraph**(); *//* *Обновленная* *функция* *visGrap*

}

void **reshGraph**(*const* Graph& graph, *const* PathInfo& optimalPath);

void **visGraph**(*const* Graph& graph);

*private*:

void **drawEdges**(*const* Graph& graph, *const* vector<QPointF>& vertexPositions);

void **drawVertices**(*const* vector<QPointF>& vertexPositions);

void **drawPath**(*const* vector<int>& path, *const* vector<QPointF>& vertexPositions);

*const* int vertexLabelOffset = 5;

};

#endif *//* *GRAPHWIDGET\_H*

Graph.cpp

#include "graph.h"

Graph::Graph(int numVertices) {

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

addVertex();

}

}

void Graph::addVertex() {

int index = vertices.size();

vertices.emplace\_back(index);

for (auto& row : adjacencyMatrix) {

row.push\_back(std::numeric\_limits<int>::max());

}

adjacencyMatrix.emplace\_back(vertices.size(), std::numeric\_limits<int>::max());

}

void Graph::removeVertex(int index) {

vertices.erase(vertices.begin() + index);

for (auto& row : adjacencyMatrix) {

row.erase(row.begin() + index);

}

adjacencyMatrix.erase(adjacencyMatrix.begin() + index);

}

void Graph::addEdge(int from, int to, int weight) {

edges.emplace\_back(from, to, weight);

adjacencyMatrix[from][to] = weight;

}

void Graph::removeEdge(int from, int to) {

for (auto it = edges.begin(); it != edges.end(); ++it) {

if (it->from == from && it->to == to) {

edges.erase(it);

break;

}

}

adjacencyMatrix[from][to] = std::numeric\_limits<int>::max();

}

void Graph::setWeight(int from, int to, int weight) {

for (auto& edge : edges) {

if (edge.from == from && edge.to == to) {

edge.weight = weight;

break;

}

}

adjacencyMatrix[from][to] = weight;

}

const std::vector<Graph::Edge>& Graph::getEdges() const {

return edges;

}

const std::vector<std::vector<int>>& Graph::getAdjacencyMatrix() const {

return adjacencyMatrix;

}

const std::vector<Graph::Vertex>& Graph::getVertices() const {

return vertices;

}

Graph.h

#ifndef GRAPH\_H

#define GRAPH\_H

#include <vector>

*class* **Graph** {

*private*:

int numVertices;

std::vector<std::vector<int>> adjacencyMatrix;

*public*:

**Graph**(int vertices);

void **addVertex**();

void **removeVertex**(int vertex);

void **removeEdge**(int v1, int v2);

void **editEdgeWeight**(int v1, int v2, int weight);

int **getNumVertices**() *const*;

int **getEdgeWeight**(int v1, int v2) *const*;

void **addEdge**(int v1, int v2, int weight);

std::vector<int> **getVertices**() *const*;

std::vector<std::vector<int>> **getAdjacencyMatrix**() *const*;

};

#endif *//* *GRAPH\_H*

Mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <iostream>

#include <queue>

#include <QMessageBox>

#include <QLineEdit>

#include <QIntValidator>

#include <QInputDialog>

#include <QVBoxLayout>

using namespace std;

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent), ui(new Ui::MainWindow), graph(0)

{

ui->setupUi(this);

setStyleSheet("background-color: #333333; color: #ffffff;");

graphWidget = new GraphWidget(this);

reshButton = new QPushButton("Решение", this);

connect(reshButton, &QPushButton::clicked, this, &MainWindow::resh);

vertexCountLineEdit = new QLineEdit(this);

vertexCountLineEdit->setPlaceholderText("Число вершин");

vertexCountLineEdit->setValidator(new QIntValidator(2, 100, this));

connect(vertexCountLineEdit, &QLineEdit::textChanged, this, &MainWindow::onVertexCountChanged);

adjacencyMatrixButton = new QPushButton("Матрица смежности", this);

connect(adjacencyMatrixButton, &QPushButton::clicked, this, &MainWindow::showAdjacencyMatrix);

addVertexButton = new QPushButton("Добавить вершину", this);

connect(addVertexButton, &QPushButton::clicked, this, &MainWindow::addVertex);

addEdgeButton = new QPushButton("Добавить ребро", this);

connect(addEdgeButton, &QPushButton::clicked, this, &MainWindow::addEdge);

removeEdgeButton = new QPushButton("Удалить ребро", this);

connect(removeEdgeButton, &QPushButton::clicked, this, &MainWindow::removeEdge);

removeVertexButton = new QPushButton("Удалить вершину", this);

connect(removeVertexButton, &QPushButton::clicked, this, &MainWindow::removeVertex);

editEdgeWeightButton = new QPushButton("Изменить вес ребра", this);

connect(editEdgeWeightButton, &QPushButton::clicked, this, &MainWindow::editWeight);

QGridLayout \*layout = new QGridLayout;

layout->addWidget(vertexCountLineEdit, 0, 0, 1, 2);

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

layout->addWidget(removeVertexButton, 1, 1);

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

layout->addWidget(removeEdgeButton, 2, 1);

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

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

layout->addWidget(adjacencyMatrixButton, 4, 1);

QVBoxLayout \*rightLayout = new QVBoxLayout;

rightLayout->addWidget(graphWidget);

rightLayout->setAlignment(Qt::AlignTop);

QHBoxLayout \*mainLayout = new QHBoxLayout;

mainLayout->addLayout(layout, 1);

mainLayout->addLayout(rightLayout, 3);

QWidget \*widget = new QWidget(this);

widget->setLayout(mainLayout);

setCentralWidget(widget);

}

MainWindow::~MainWindow()

{

delete ui;

}

const int INF = numeric\_limits<int>::max();

int findMin(const vector<int> &arr)

{

return \*min\_element(arr.begin(), arr.end());

}

int findMinIndex(const vector<int> &arr)

{

auto minElement = min\_element(arr.begin(), arr.end());

return minElement != arr.end() ? distance(arr.begin(), minElement) : -1;

}

struct Edge

{

int source;

int destination;

int weight;

};

class Graph

{

public:

int vertexCount;

vector<vector<int>> adjacencyMatrix;

Graph(int count)

{

vertexCount = count;

adjacencyMatrix.resize(vertexCount, vector<int>(vertexCount, INF));

}

void addEdge(int source, int destination, int weight)

{

adjacencyMatrix[source][destination] = weight;

adjacencyMatrix[destination][source] = weight;

}

void removeEdge(int source, int destination)

{

adjacencyMatrix[source][destination] = INF;

adjacencyMatrix[destination][source] = INF;

}

vector<Edge> getEdges()

{

vector<Edge> edges;

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

{

for (int j = i + 1; j < vertexCount; ++j)

{

if (adjacencyMatrix[i][j] != INF)

{

Edge edge;

edge.source = i;

edge.destination = j;

edge.weight = adjacencyMatrix[i][j];

edges.push\_back(edge);

}

}

}

return edges;

}

};

void MainWindow::onVertexCountChanged()

{

QString text = vertexCountLineEdit->text();

bool ok;

int vertexCount = text.toInt(&ok);

if (ok)

{

graph = new Graph(vertexCount);

graphWidget->setGraph(graph);

graphWidget->update();

}

}

void MainWindow::addVertex()

{

if (graph == nullptr)

{

QMessageBox::information(this, "Ошибка", "Сначала установите количество вершин");

return;

}

int newVertexCount = graph->vertexCount + 1;

graph->adjacencyMatrix.resize(newVertexCount, vector<int>(newVertexCount, INF));

graph->vertexCount = newVertexCount;

graphWidget->setGraph(graph);

graphWidget->update();

}

void MainWindow::removeVertex()

{

if (graph == nullptr)

{

QMessageBox::information(this, "Ошибка", "Сначала установите количество вершин");

return;

}

int vertexIndex = QInputDialog::getInt(this, "Удаление вершины", "Введите индекс вершины для удаления:");

if (vertexIndex < 0 || vertexIndex >= graph->vertexCount)

{

QMessageBox::information(this, "Ошибка", "Некорректный индекс вершины");

return;

}

graph->adjacencyMatrix.erase(graph->adjacencyMatrix.begin() + vertexIndex);

for (int i = 0; i < graph->vertexCount; ++i)

{

graph->adjacencyMatrix[i].erase(graph->adjacencyMatrix[i].begin() + vertexIndex);

}

graph->vertexCount--;

graphWidget->setGraph(graph);

graphWidget->update();

}

void MainWindow::addEdge()

{

if (graph == nullptr)

{

QMessageBox::information(this, "Ошибка", "Сначала установите количество вершин");

return;

}

bool ok;

int source = QInputDialog::getInt(this, "Добавление ребра", "Введите начальную вершину:", 0, 0, graph->vertexCount - 1, 1, &ok);

if (!ok)

return;

int destination = QInputDialog::getInt(this, "Добавление ребра", "Введите конечную вершину:", 0, 0, graph->vertexCount - 1, 1, &ok);

if (!ok)

return;

int weight = QInputDialog::getInt(this, "Добавление ребра", "Введите вес ребра:", 0, 0, INT\_MAX, 1, &ok);

if (!ok)

return;

graph->addEdge(source, destination, weight);

graphWidget->update();

}

void MainWindow::removeEdge()

{

if (graph == nullptr)

{

QMessageBox::information(this, "Ошибка", "Сначала установите количество вершин");

return;

}

bool ok;

int source = QInputDialog::getInt(this, "Удаление ребра", "Введите начальную вершину:", 0, 0, graph->vertexCount - 1, 1, &ok);

if (!ok)

return;

int destination = QInputDialog::getInt(this, "Удаление ребра", "Введите конечную вершину:", 0, 0, graph->vertexCount - 1, 1, &ok);

if (!ok)

return;

graph->removeEdge(source, destination);

graphWidget->update();

}

void MainWindow::runDijkstra()

{

if (graph == nullptr)

{

QMessageBox::information(this, "Ошибка", "Сначала установите количество вершин");

return;

}

bool ok;

int source = QInputDialog::getInt(this, "Алгоритм Дейкстры", "Введите начальную вершину:", 0, 0, graph->vertexCount - 1, 1, &ok);

if (!ok)

return;

vector<int> distances(graph->vertexCount, INF);

distances[source] = 0;

vector<bool> visited(graph->vertexCount, false);

for (int i = 0; i < graph->vertexCount - 1; ++i)

{

int minDistance = INF;

int minVertex = -1;

for (int j = 0; j < graph->vertexCount; ++j)

{

if (!visited[j] && distances[j] < minDistance)

{

minDistance = distances[j];

minVertex = j;

}

}

if (minVertex == -1)

break;

visited[minVertex] = true;

for (int j = 0; j < graph->vertexCount; ++j)

{

int weight = graph->adjacencyMatrix[minVertex][j];

if (!visited[j] && weight != INF && distances[minVertex] + weight < distances[j])

distances[j] = distances[minVertex] + weight;

}

}

QString result;

for (int i = 0; i < graph->vertexCount; ++i)

{

if (distances[i] == INF)

result += QString::number(i) + ": INF\n";

else

result += QString::number(i) + ": " + QString::number(distances[i]) + "\n";

}

QMessageBox::information(this, "Алгоритм Дейкстры", result);

}

Mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include "graphwidget.h"

#include "graph.h"

#include "qpushbutton.h"

#include <QLineEdit>

#include <QMessageBox>

#include <QString>

QT\_BEGIN\_NAMESPACE

*namespace* **Ui** { *class* **MainWindow**; }

QT\_END\_NAMESPACE

*class* **MainWindow** : *public* QMainWindow

{

Q\_OBJECT

*public*:

**MainWindow**(QWidget \*parent = *nullptr*);

~***MainWindow***();

*private* slots:

void **onVertexCountChanged**(*const* QString& text);

void **updateGraph**(int vertexCount);

void **showAdjacencyMatrix**();

void **addVertex**();

void **resh**();

void **addEdge**();

void **removeEdge**();

void **removeVertex**();

void **editWeight**();

*private*:

Ui::MainWindow \*ui;

GraphWidget \*graphWidget;

Graph graph;

int startVertex;

QPushButton\* reshButton;

QLineEdit\* vertexCountLineEdit;

QPushButton\* adjacencyMatrixButton;

QPushButton\* addVertexButton;

QPushButton\* addEdgeButton;

QPushButton\* removeEdgeButton;

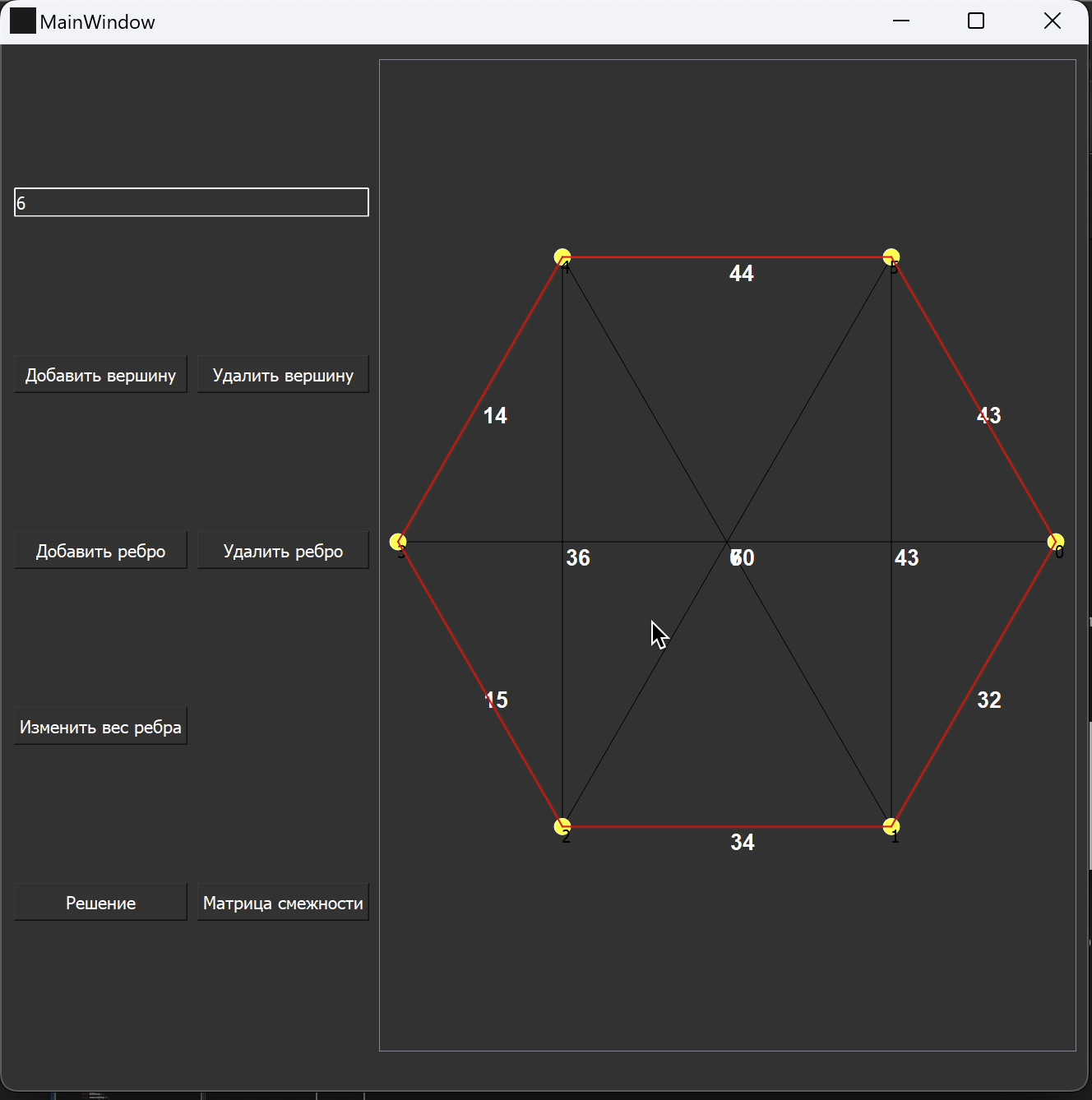
QPushButton\* removeVertexButton;

QPushButton\* editEdgeWeightButton;

};

#endif *//* *MAINWINDOW\_H*

Вывод



**Текст выступления.**

Вашем вниманию представляется творческая работа, которая состоит из двух частей. Первая часть-разработка автоматизированного рабочего места ТИКЕРА (билетчика в театре, концертной площадке). вторая часть, решение задачи Коммивояжера . Работу выполнил студент 1 курса кафедры ИТАС Радостев Илья.

Начну с демонстрации первой части творческой работы а именно арм ТИКЕРА. При запуске показывается календарь с датой, три кнопки(Купить, вернуть и мои билеты), а так же функция генерации новых мероприятий(концертов) с датой и ценой.

Итак, перейдем к демонстрации работы программы.

Дата в календаре кликабельна, можно нажать на любую дату и получить список концертов в данный день. Далее покажу окно покупки билетов.   
Чтобы купить билет нужно указать имя покупателя, выбрать нужный концерт в списке и нажать “купить”. Обращаю внимание, что действие купить невозможно выполнить пока не задействованно два фактора: введено имя покупателя и выбран концерт. Вводим имя и выбираем концерт. Я сразу поставил значение баланса в 10.000 потом что сейчас только один пользователь-это я. Покупаем билет, смотрим что он купился и отобразился в списке купленных билетов. Далее вернем этот билет вводим имя и номер билета. Я поставил проверку по имени и номеру билета, поэтому при введении несуществующего имени или номера билета возврат не будет осуществлен.  
после покупки билета надо проверить что он действительно появился в списке купленных билетов.  
Далее продемонстрирую возврат билета.Для возврата билета нужно указать имя покупателя, номер билета и причину сдачи (необязательно) причина сдачи записывается в файл returndata.txt, если имя покупателя или номер билета введены некорректно программа это проверяет и выдаст ошибку. Введем любое имя и номер билета.Теперь укажем правильное имя. Возврат билета успешен! Проверим что билет сдан. Действительно сдан, в окне с билетами пусто.

Программа реализована с помощью таких классов как: concertsbydatedialogs, mainwindow, mainwindowbuy, mainwindowreturn,

Начнем с класса mainwindow

1. `MainWindow::MainWindow(QWidget \*parent)`: Конструктор класса MainWindow. Устанавливает заголовок окна "Ticker".

2. `MainWindow::~MainWindow()`: Деструктор класса MainWindow.

3. `MainWindow::on\_pushButton\_clicked()`: Слот для обработки нажатия кнопки "Buy" (купить). Создает и открывает диалоговое окно MainWindowBuy для покупки билетов.

4. `MainWindow::on\_pushButton\_2\_clicked()`: Слот для обработки нажатия кнопки "Return" (вернуть). Создает и открывает диалоговое окно MainWindowReturn для возврата билетов.

5. `MainWindow::on\_pushButton\_3\_clicked()`: Слот для обработки нажатия кнопки "My Tickets" (мои билеты). Открывает файл "buyerdata.txt" и отображает информацию о всех билетах, хранящихся в файле, в диалоговом окне.

6. `MainWindow::on\_calendarWidget\_clicked(const QDate& date)`: Слот для обработки выбора даты в календаре. Создает и открывает диалоговое окно ConcertsByDateDialog для отображения концертов, запланированных на выбранную дату.

7. `MainWindow::updateBalance(double balance)`: Слот для обновления баланса счета и вывода уведомления о текущем балансе. Этот слот используется для связи с диалоговым окном MainWindowBuy, чтобы обновить баланс после покупки билетов.

8. `MainWindow::on\_actionMyBalance\_clicked()`: Слот для обработки нажатия пункта меню "My Balance" (мой баланс). Создает и открывает диалоговое окно MainWindowBuy и устанавливает соединение между сигналом `balanceUpdated` из диалогового окна и слотом `updateBalance` текущего окна.

9. `MainWindow::addTestConcert(const QString& artist, const QDate& date, double price)`: Функция для добавления информации о тестовом концерте в файл "concertdata.txt". Записывает информацию об артисте, дате и цене концерта в файл.

10. `MainWindow::clearTestConcerts()`: Функция для очистки тестовых концертов из файла "concertdata.txt". Удаляет содержимое файла.

11. `MainWindow::on\_actionTest\_Mode\_1\_triggered()`: Слот для обработки нажатия пункта меню "Test Mode 1". Включает тестовый режим и вызывает функцию `generateTestConcerts` для генерации тестовых концертов.

12. `MainWindow::on\_actionTest\_Mode\_2\_triggered()`: Слот для обработки нажатия пункта меню

"Test Mode 2". Выключает тестовый режим и вызывает функцию `clearTestConcerts` для очистки тестовых концертов.

Функция `MainWindow::generateTestConcerts()` выполняет генерацию тестовых концертов и добавление их в файл "concertdata.txt". Давайте рассмотрим ее подробнее:

1. `QDate currentDate = QDate::currentDate();`: Создается объект `currentDate`, который содержит текущую дату.

2. `qsrand(static\_cast<uint>(QTime::currentTime().msec()));`: Устанавливается начальное значение генератора случайных чисел `qrand()` на основе текущего времени, чтобы получить различные случайные числа при каждом запуске.

3. `QStringList artists = { "Artist1", "Artist2", "Artist3" };`: Создается список `artists`, содержащий имена артистов для тестовых концертов.

4. `QStringList firstNames = { "John", "Emma", "Michael", "Sophia" };`: Создается список `firstNames`, содержащий первые имена для генерации случайных имен и фамилий.

5. `QStringList lastNames = { "Smith", "Johnson", "Williams", "Brown" };`: Создается список `lastNames`, содержащий фамилии для генерации случайных имен и фамилий.

6. `for (int i = 0; i < 90; ++i) {`: Цикл выполняется 90 раз для генерации 90 тестовых концертов.

7. `QDate randomDate = currentDate.addDays(qrand() % 30);`: Генерируется случайная дата, добавляя к текущей дате случайное количество дней в пределах 30 дней. Таким образом, получается случайная дата в течение следующих 30 дней от текущей даты.

8. `int randomPrice = qrand() % (200 - 50 + 1) + 50;`: Генерируется случайная цена в диапазоне от 50 до 200.

9. `QString randomFirstName = firstNames[qrand() % firstNames.size()];`: Генерируется случайное имя, выбирая случайный элемент из списка `firstNames`.

10. `QString randomLastName = lastNames[qrand() % lastNames.size()];`: Генерируется случайная фамилия, выбирая случайный элемент из списка `lastNames`.

11. `addTestConcert(QString("%1 %2").arg(randomFirstName, randomLastName), randomDate, randomPrice);`: Вызывается функция `addTestConcert` для добавления тестового концерта. Передается строка, содержащая случайное имя и фамилию, сформированную с помощью метода `arg()` для подстановки значений `randomFirstName` и `randomLastName`. Также передаются случайная дата и цена.

12. `emit testConcertsGenerated();`: Генерируется сигнал `testConcertsGenerated()`, чтобы оповестить о том, что тестовые концерты были сг

енерированы.

Таким образом, эта функция генерирует случайные данные для тестовых концертов (артисты, даты, цены) и добавляет их в файл "concertdata.txt".

mainwindowbuy

Основные функции в классе `MainWindowBuy`:

1. `setConcertInfo(const QString& artist, const QDate& concertDate, double price)`: Устанавливает информацию о выбранном концерте (исполнитель, дата концерта, цена).

2. `setConcertData(const QString& artist, const QDate& concertDate, double price)`: Устанавливает данные о концерте в таблицу `tableWidgetConcerts`.

3. `on\_pushButtonBuy\_clicked()`: Обработчик нажатия на кнопку покупки. Проверяет выбран ли концерт и введено ли имя покупателя, проверяет достаточно ли средств на счете, генерирует номер билета, выводит сообщение о покупке, обновляет баланс, записывает информацию о покупке в файл.

4. `onTableWidgetSelectionChanged()`: Обработчик изменения выбора элемента в таблице. Получает информацию о выбранном концерте, устанавливает информацию о концерте и включает/отключает кнопку покупки.

5. `deductMoneyFromAccount(double amount)`: Вычитает указанную сумму из баланса счета. Возвращает `true`, если средств достаточно и вычет произведен успешно, и `false` в противном случае.

6. Функция `generateTicketNumber` используется для генерации номера билета на основе переданного имени исполнителя концерта. Она выполняет следующие действия:

1. Получение текущей даты:

- Используется функция `QDate::currentDate()` для получения текущей даты.

2. Преобразование даты в строку без точек (ддммгггг):

- Используется метод `toString()` с форматом "ddMMyyyy" для преобразования текущей даты в строку без точек.

3. Преобразование имени исполнителя в верхний регистр и взятие первых трех символов:

- Метод `toUpper()` преобразует имя исполнителя в верхний регистр.

- Метод `left(3)` возвращает первые три символа имени исполнителя.

4. Получение текущего времени в миллисекундах:

- Используется метод `QDateTime::currentDateTime().currentMSecsSinceEpoch()` для получения текущего времени в миллисекундах.

5. Получение последних трех цифр из текущего времени в миллисекундах:

- Оператор `%` вычисляет остаток от деления текущего времени в миллисекундах на 1000.

- Функция `QString::number()` преобразует полученный остаток в строку.

6. Генерация номера билета на основе даты, кода исполнителя и миллисекунд:

- Номер билета формируется путем объединения кода исполнителя, строки даты и строки миллисекунд.

7. Возврат сгенерированного номера билета.

7. `loadConcertData()`: Загружает данные о концертах из файла "concertdata.txt" и отображает их в таблице `tableWidgetConcerts`.

Эти функции реализуют основную логику покупки билетов на концерты, включая отображение информации о концертах, обработку выбора концерта, вычет средств, генерацию номера билета и сохранение информации о покупке.  
  
mainwindowReturn

`MainWindowReturn::on\_pushButton\_clicked()`, который вызывается при нажатии на кнопку "Возврат". Функция выполняет следующие действия:

1. Получение данных из полей ввода:

- Извлекаются значения из трех полей ввода: `buyerName`, `ticketNumber` и `returnReason`.

2. Проверка условий для возврата билета:

- Открывается файл "buyerdata.txt" в режиме чтения и записи.

- В цикле проходятся все строки файла.

- Если встречается разделитель "-----------------------------------", выполняется обработка данных для текущего билета:

- Извлекается номер билета (`currentTicketNumber`) и имя покупателя (`currentBuyerName`).

- Если найденный номер билета и имя покупателя совпадают с введенными данными (`ticketNumber` и `buyerName`), устанавливается флаг `found` в значение `true`. Иначе данные текущего билета добавляются в список `validTickets`.

- После завершения цикла файл закрывается.

3. Обработка возврата билета:

- Если билет найден (флаг `found` установлен в `true`), выполняются следующие действия:

- Вызывается функция `refundMoneyToAccount(ticketNumber)` для возврата средств на баланс.

- Открывается файл "returndata.txt" в режиме добавления и записи.

- Данные текущего билета и причина возврата записываются в файл.

- Файл "returndata.txt" закрывается.

- Выводится информационное окно с сообщением об успешном возврате билета и обработке возврата средств.

- Если билет не найден, выводится предупреждающее окно о некорректных данных (некорректный номер билета или имя покупателя).

4. Перезапись файла "buyerdata.txt" без возвратившегося билета:

- Открывается файл "buyerdata.txt" в режиме записи.

- В цикле проходятся все билеты из списка `validTickets`.

- Данные каждого билета записываются в файл.

- После завершения цикла файл "buyerdata.txt" закрывается.

Concertsbydatedialog

Данный код относится к классу `ConcertsByDateDialog` и выполняет следующие действия:

1. Импортирует необходимые заголовочные файлы для класса и используемых библиотек.

2. Определяет конструктор класса `ConcertsByDateDialog`, который принимает параметр `date` типа `QDate` и указатель на `QWidget` в качестве родительского виджета.

3. В конструкторе инициализируется пользовательский интерфейс (`ui`) и задается заголовок окна.

4. Устанавливается количество столбцов и метки заголовка горизонтального заголовка таблицы `ui->tableWidgetConcerts`.

5. Открывается файл "concertdata.txt" в режиме чтения (`QIODevice::ReadOnly | QIODevice::Text`).

6. Если файл успешно открыт, выполняется чтение данных из файла.

7. Читает файл построчно с помощью `QTextStream` и цикла `while`.

8. Для каждой строки данных:

- Разделяет строку на части, используя разделитель "," с помощью `split(",")`, и сохраняет результат в `QStringList` `data`.

- Проверяет, что количество частей данных равно или больше 3.

- Извлекает значения исполнителя (`concert`), даты (`dateString`) и цены (`priceString`) из `data`.

- Преобразует строку даты в объект `QDate` с использованием формата "dd.MM.yyyy".

- Проверяет, что дата концерта (`concertDate`) соответствует выбранной дате (`date`).

- Если даты совпадают, добавляет информацию о концерте в таблицу `ui->tableWidgetConcerts`.

9. Закрывает файл "concertdata.txt".

10. Если файл не удалось открыть, выводит сообщение об ошибке с помощью `QMessageBox::critical()`.

11. Определяет деструктор класса `ConcertsByDateDialog`, который освобождает память, выделенную для пользовательского интерфейса (`ui`).

Таким образом, код выполняет чтение информации о концертах из файла "concertdata.txt" и отображает концерты, соответствующие выбранной дате, в таблице `ui->tableWidgetConcerts`.

Перейдем ко второй части творческой работы а именно – решению задачи коммивояжера.в чем вообще заключается решение задачи коммивояжера? Одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута и соответствующие матрицы расстояний, стоимости и тому подобного. Перейдем к обзору программы. на главном экране отображаются поле для ввода количества вершин, кнопки позваляющие добавить вершину, удалить вершину, добавить ребро, удалить ребро, изменить вес ребра, кнопка получить решение и кнопка вызова матрицы смежности для графа.  
предлагаю создать граф. Ввожу в поле количество вершин , теперь программа просит вводить номера вершин и вес ребер, давайте введем .

Основные функции.

1. `reshGraph(const Graph& graph, const PathInfo& optimalPath)`: Этот метод принимает объект типа `Graph` и информацию об оптимальном пути в качестве параметров. Он очищает сцену, вычисляет позиции вершин на сцене на основе количества вершин и размеров сцены, а затем вызывает другие функции для отрисовки ребер, вершин и оптимального пути на сцене.

2. `visGraph(const Graph& graph)`: Этот метод принимает объект типа `Graph` в качестве параметра и очищает сцену. Он вычисляет позиции вершин на сцене и затем вызывает другие функции для отрисовки ребер и вершин на сцене.

3. `drawEdges(const Graph& graph, const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку ребер графа. Он принимает объект типа `Graph` и вектор позиций вершин в качестве параметров. Для каждой пары вершин он создает линию, соединяющую их, и добавляет ее на сцену. Также он добавляет текстовую метку с весом ребра в середину каждого ребра.

4. `drawVertices(const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку вершин графа. Он принимает вектор позиций вершин в качестве параметра. Для каждой вершины он создает эллипс и добавляет его на сцену. Также он добавляет текстовую метку с номером вершины рядом с каждой вершиной.

5. `drawPath(const std::vector<int>& path, const std::vector<QPointF>& vertexPositions)`: Этот метод отвечает за отрисовку оптимального пути в графе. Он принимает вектор с последовательностью вершин пути и вектор позиций вершин в качестве параметров. Он создает линии, соединяющие вершины пути, и добавляет их на сцену.