Министерство образования Республики Беларусь

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Программирование на языках высокого уровня

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

ИНФОРМАЦИОННАЯ СИСТЕМА АВТОВОКЗАЛА

БГУИР КП 1–40 02 01 517 ПЗ

Студент: Наркевич А.С.

Руководитель: Марзалюк А.В.

МИНСК 2023**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc153871742)

[1 ПОСТАНОВКА ЗАДАЧИ 4](#_Toc153871743)

[2 ОБЗОР ЛИТЕРАТУРЫ 5](#_Toc153871744)

[2.1 Обзор методов и алгоритмов решения поставленной задачи 6](#_Toc153871745)

[3 ФУНКЦИАНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 7](#_Toc153871746)

[3.1 Структура входных и выходных данных 7](#_Toc153871747)

[3.2 Разработка диаграммы классов 8](#_Toc153871748)

[3.3 Описание классов 8](#_Toc153871749)

[4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 19](#_Toc153871750)

[4.1. Разработка схем алгоритмов 19](#_Toc153871751)

[4.2. Разработка схемы алгоритма 19](#_Toc153871752)

[4.2.1 Разработка алгоритма bookTransport(Database& Db, TransportType transportType) 19](#_Toc153871753)

[4.2.2 Разработка алгоритма setRoutePrice(Database &Db, int routeId, double price, int isLogged) 20](#_Toc153871754)

[5 РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ 21](#_Toc153871755)

[ЗАКЛЮЧЕНИЕ 26](#_Toc153871756)

[ЛИТЕРАТУРА 27](#_Toc153871757)

[ПРИЛОЖЕНИЕ А 28](#_Toc153871758)

[ПРИЛОЖЕНИЕ Б 29](#_Toc153871759)

[ПРИЛОЖЕНИЕ В 30](#_Toc153871760)

[ПРИЛОЖЕНИЕ Г 31](#_Toc153871761)

[ПРИЛОЖЕНИЕ Е 32](#_Toc153871762)

[ПРИЛОЖЕНИЕ Д 33](#_Toc153871763)

# ВВЕДЕНИЕ

В эпоху цифровизации и быстрого технологического прогресса информационные системы стали неотъемлемой частью нашей повседневной жизни. Они играют ключевую роль в управлении, контроле и координации деятельности организаций различных масштабов и сфер деятельности.

В рамках данной работы мы планируем разработать информационную систему с использованием языка программирования C++. Эта система будет спроектирована таким образом, чтобы обеспечивать эффективное управление и обработку данных, а также облегчать процесс принятия решений.

Система будет состоять из нескольких модулей, каждый из которых будет выполнять определенные функции. Например, один модуль может быть ответственным за сбор и хранение данных, в то время как другой - за обработку этих данных и предоставление полезной информации. Все эти модули будут взаимодействовать друг с другом, чтобы обеспечить.

Язык программирования C++ был выбран для разработки этой информационной системы из-за его высокой производительности и гибкости. C++ позволяет программистам иметь более тонкий контроль над системными ресурсами и памятью, что может быть критически важно для систем, обрабатывающих большие объемы данных или требующих высокой производительности для быстрой обработки данных.

Кроме того, C++ поддерживает объектно-ориентированное программирование, что облегчает разработку сложных систем. Это позволяет программистам создавать модули, или “объекты”, которые могут быть повторно использованы в различных частях системы, упрощая тем самым процесс разработки и обслуживания системы.

C++ также имеет обширную поддержку библиотек, которые могут быть использованы для ускорения процесса разработки, предоставляя готовые решения для распространенных задач программирования. Это может включать в себя все, от работы с файлами и сетями до сложных математических вычислений.

В целом, использование C++ для разработки информационной системы может обеспечить эффективность, надежность и масштабируемость, что делает его идеальным выбором для этого проекта.

# 1 ПОСТАНОВКА ЗАДАЧИ

Целью данной работы является создание информационной системы для автовокзала, обеспечивающей эффективное управление и обработку данных. Система, разработанная на языке программирования C++ и использующая базу данных PostgreSQL, представляет собой интегрированное решение, которое облегчает процесс принятия решений и управления ресурсами автовокзала.

Система предоставляет информацию о доступном транспорте, маршрутах, расписаниях и остановках, а также позволяет редактировать эти данные. Это обеспечивает точность и актуальность информации, что важно для эффективного функционирования автовокзала.

Использование C++ и PostgreSQL позволяет системе обрабатывать большие объемы данных и обеспечивает высокую производительность, что критически важно для оперативного управления автовокзалом. Кроме того, эта система обеспечивает надежное и безопасное хранение данных, что является важным аспектом управления информацией.

В дополнение к вышеупомянутым функциям, система также обеспечивает гибкость и масштабируемость, что позволяет ей адаптироваться к изменяющимся потребностям автовокзала. Благодаря модульной структуре, новые функции могут быть легко добавлены или существующие функции могут быть модифицированы для удовлетворения новых требований.

Система также предусматривает возможность интеграции с другими системами или приложениями, что может быть полезно для обмена данными или для расширения функциональности системы. Это может включать в себя интеграцию с системами бронирования билетов, системами управления транспортом или другими информационными системами.

В заключение, данная информационная система представляет собой всеобъемлющее решение, которое не только облегчает управление автовокзалом, но и способствует повышению уровня обслуживания пассажиров, обеспечивая им доступ к актуальной и точной информации. Это делает ее неотъемлемым инструментом в эпоху цифровизации и быстрого технологического прогресса.

# 2 ОБЗОР ЛИТЕРАТУРЫ

В современном мире информационные системы играют важную роль во многих областях, включая транспорт. Они обеспечивают эффективное управление и координацию различных аспектов работы, таких как расписание автобусов, бронирование билетов, информация о пассажирах, данные о водителях и автобусах и многое другое.

Информационная система (ИС) - это совокупность взаимосвязанных элементов, которые собирают, обрабатывают, хранят и предоставляют информацию для достижения определенных целей. Она используется для организации и управления информацией, обеспечивая эффективное решение задач и принятие решений. В широком смысле информационная система есть совокупность технического, программного и организационного обеспечения, а также персонала, предназначенная для того, чтобы своевременно обеспечивать надлежащих людей надлежащей информацией.

PostgreSQL - это объектно-реляционная СУБД с мощным функционалом, работающая со сложными, составными запросами. Система справляется с задачами разбора и выполнения трудоемких операций, которые подразумевают и чтение, и запись, и валидацию одновременно.

C++ и PostgreSQL могут быть использованы вместе для создания мощных и эффективных информационных систем. C++ - это высокоуровневый язык программирования, который обеспечивает большую гибкость и производительность, что делает его идеальным выбором для разработки сложных систем. PostgreSQL, с другой стороны, предлагает надежное и масштабируемое решение для управления данными. Вместе они могут быть использованы для создания информационных систем, которые могут эффективно обрабатывать и управлять большими объемами данных.

Информационная система автовокзала, разработанная на C++ и PostgreSQL, может обеспечивать эффективное управление и координацию различных аспектов работы автовокзала. Это помогает улучшить эффективность операций, улучшить обслуживание клиентов и повысить общую производительность автовокзала.

Важно отметить, что разработка такой системы требует глубоких знаний в области программирования на C++ и работы с PostgreSQL. Разработчик должен быть знаком с различными аспектами этих технологий, чтобы создать эффективную и надежную систему.

В заключение, информационные системы играют важную роль в современном мире, и использование C++ и PostgreSQL для их создания может обеспечить множество преимуществ. Они могут помочь организациям, таким как автовокзалы, эффективно управлять своими операциями и предоставлять высококачественные услуги своим клиентам.

## 2.1 Обзор методов и алгоритмов решения поставленной задачи

В процессе разработки информационной системы автовокзала была задействована база данных (БД), позволяющая хранить информацию о различных моделях, таких как транспорт, покупатели, билеты, маршруты и другие. БД была реализована с использованием PostgreSQL, что позволило обеспечить надежное и эффективное хранение данных.

С помощью БД можно было удобно управлять всей этой информацией. Например, можно было легко добавлять, удалять или изменять данные о транспорте, маршрутах, билетах и ценах. Это обеспечивало гибкость и удобство в управлении данными.

Код, написанный на C++, позволял просматривать актуальную информацию о транспорте, маршрутах, билетах и ценах. Это было особенно полезно для пользователей системы, которые могли получить всю необходимую информацию в удобном и понятном формате.

С ролью администратора можно было редактировать эти данные. Это позволяло обеспечить актуальность и точность информации, предоставляемой пользователям. Администратор мог, например, обновлять информацию о расписании транспорта или изменять цены на билеты.

Таким образом, использование БД и кода на C++ позволило создать эффективную и надежную информационную систему автовокзала. Эта система обеспечивает удобное и эффективное управление данными, что важно для обеспечения качественного обслуживания клиентов автовокзала.

Кроме того, система предоставляла возможность автоматизации некоторых процессов. Например, при поступлении нового транспорта в автопарк, информация о нем автоматически добавлялась в базу данных. Это упрощало процесс учета транспортных средств и позволяло избежать ошибок, связанных с ручным вводом данных.

Система также обеспечивала безопасность данных. Все данные, хранящиеся в базе данных, были защищены от несанкционированного доступа. Это было особенно важно для защиты персональных данных покупателей. Кроме того, регулярное резервное копирование данных позволяло предотвратить потерю информации в случае сбоев в работе системы.

Наконец, использование C++ для написания кода системы обеспечивало высокую производительность и надежность работы. C++ является мощным языком программирования, который позволяет создавать эффективные и оптимизированные приложения. Это обеспечивало быстрый и стабильный доступ к данным, что является критически важным для работы автовокзала.

# 3 ФУНКЦИАНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

## 3.1 Структура входных и выходных данных

Разрабатываемая информационная система автовокзала предназначена для обработки и управления большим объемом данных, хранящихся в базе данных PostgreSQL. База данных может содержать различные типы данных, такие как информация о транспорте, покупателях, билетах, маршрутах и другие. Основной функционал системы включает в себя управление этими данными, включая добавление, удаление и изменение информации.

Пользователи системы имеют возможность просматривать актуальную информацию о транспорте, маршрутах, билетах и ценах в удобном и понятном формате. Это особенно полезно для клиентов автовокзала, которые могут получить всю необходимую информацию для планирования своего путешествия.

Для сохранения настроек системы используется файл конфигурации, в котором администратор может задавать параметры системы, такие как время работы автовокзала, количество рейсов в день и другие. Это обеспечивает гибкость и удобство в управлении данными.

Приведен пример входных данных в форме таблицы, где содержатся сущности таких объектов как такси и автобус.



Рисунок 3.1 – пример входных данных(такси)

**

Рисунок 3.2 – пример входных данных(автобус)

## 3.2 Разработка диаграммы классов

Диаграмма классов программы приведена в приложении А.

## 3.3 Описание классов

При разработке базы данных использовались SQL таблицы, код которых будет приведен в приложении Д. Диаграмма SQL таблиц будет приведена в приложении Ж.

1. Класс транспорта.

class Transport

{

private:

std::string brand; - марка

std::string model – модель.

std::string color; - цвет.

EngineType engineType; - перечисление тип двигателя.

public:

Transport(); - конструктор по умолчанию.

Transport(std::string brand, std::string model, std::string color, EngineType engineType); - конструктор.

~Transport(); - деструктор.

std::string getBrand(); - геттер для доступа к приватному полю.

void setBrand(std::string brand); - сеттер для изменения приватного поля.

std::string getModel(); - геттер для доступа к приватному полю.

void setModel(std::string model); - сеттер для изменения приватного поля.

std::string getColor(); - геттер для доступа к приватному полю.

void setColor(std::string color); - сеттер для изменения приватного поля.

EngineType getEngineType(); - геттер для доступа к приватному полю.

void setEngineType(EngineType engineType); - сеттер для изменения приватного поля.

std::string getEngineTypeString(); - геттер для доступа к приватному полю.

static EngineType stringToEngineType(const std::string &engineTypeStr); - функция для преобразования строки в значение из перечисления.

void isIdValid(int &id, const std::string &transportName, Database \*Db); - функция для проверки наличия транспорта с определенным уникальным идентификатором в базе данных.

2. Класс общественного транспорта.

class PublicTransport : public Transport {

private:

int transportId; – уникальный идентификатор общественного транспорта.

int capacity; - вместимость.

public:

enum TransportType {

BUS,

TROLLEYBUS,

}; - перечисление, описывающие тип транспорта.

PublicTransport(); - конструктор по умолчанию.

PublicTransport(int transportId, std::string brand, std::string model, std::string color, EngineType engineType, int capacity); - первый параметризированный конструктор.

PublicTransport(std::string brand, std::string model, std::string color, EngineType engineType, int capacity); - второй параметризированный конструктор.

int getTransportId(); - геттер для доступа к приватному полю.

void setTransportId(int id); - сеттер для изменения приватного поля.

int getCapacity(); - геттер для доступа к приватному полю.

void setCapacity(int capacity); - сеттер для изменения приватного поля.

std::string getCurrentTimestampAsString(); - геттер для доступа к приватному полю.

int insertTicket(Database& db, const TransportTicket& ticket); - функция для обработки билета.

void bookTransport(Database& Db, PublicTransport::TransportType transportType); - функция для бронирования транспорта.

~PublicTransport(); - деструктор.

3. Класс автобус.

class Bus : public PublicTransport {

private:

bool hasContactlessPayment; - наличие бесконтактной оплаты.

public:

Bus(); - конструктор по умолчанию.

Bus(const std::string& brand, const std::string& model, const std::string& color, EngineType engineType, int capacity, bool hasContactlessPayment); - первый параметризированный конструктор.

Bus(int transportId, std::string brand, std::string model, std::string color, EngineType engineType, int capacity, bool hasContactlessPayment); - второй параметризированный конструктор.

bool isHasContactlessPayment() const; - геттер для доступа к приватному полю.

void setHasContactlessPayment(bool hasContactlessPayment); - сеттер для изменения значения приватного поля.

void displayBusDetails(const pqxx::result::const\_iterator& row); - функция преобразующая данные из SQL запроса в текст.

void displayAllBuses(Database& Db); - функция для вывода автобусов.

4. Класс троллейбус.

class TrolleyBus : public PublicTransport {

private:

bool hasSockets; - наличие розеток.

public:

TrolleyBus(); - конструктор по умолчанию.

TrolleyBus(int transportId, std::string brand, std::string model, std::string color, EngineType engineType, int capacity, bool hasSockets); - первый параметризированный конструктор.

TrolleyBus(const std::string& brand, const std::string& model, const std::string& color, EngineType engineType, int capacity, bool hasSockets) - параметризированный конструктор.

bool getHasSockets(); - геттер для доступа к приватному полю.

void setHasSockets(bool hasSockets); - сеттер для изменения приватного поля.

void displayTrolleyBusDetails(const pqxx::result::const\_iterator& row); - функция для преобразования данных из SQL запроса в текст.

void displayAllTrolleyBuses(Database& db); - функция для отображения троллейбусов.

5. Класс такси.

class Taxi : public Transport {

private:

int carId; - уникальный идентификатор такси.

double pricePerKil; - цена за километр.

bool hasDriver; - наличие водителя.

bool hasWiFi; - наличие сети.

bool hasChildSeat; - наличие детского кресла.

RentCarTypes rentCarTypes; - тип такси.

public:

Taxi(std::string brand, std::string model, std::string color, EngineType engineType, int carId, double pricePerKil, bool hasDriver, bool hasWiFi, bool hasChildSeat, RentCarTypes rentCarTypes); - параметризированный конструктор.

Taxi(); - конструктор по умолчанию.

int getCarId(); - геттер для доступа к приватному полю.

void setCarId(int carId); - сеттер для изменения приватного поля.

double getPricePerKil(); - геттер для доступа к приватному полю.

void setPricePerKil(double pricePerKil); - сеттер для изменения приватного поля.

bool getHasDriver(); - геттер для доступа к приватному полю.

void setHasDriver(bool hasDriver); - сеттер для изменения приватного поля.

bool getHasWiFi(); - геттер для доступа к приватному полю.

void setHasWiFi(bool hasWiFi); - сеттер для изменения приватного поля.

bool getHasChildSeat(); - геттер для доступа к приватному полю.

void setHasChildSeat(bool hasChildSeat); - сеттер для изменения приватного поля.

RentCarTypes getRentCarTypes(); - геттер для доступа к приватному полю.

void setRentCarTypes(RentCarTypes rentCarTypes); - сеттер для изменения приватного поля.

~Taxi(); - деструктор.

static RentCarTypes stringToRentCarTypes(const std::string& rentCarTypesStr); - функция для преобразования строки в перечисление.

static void displayTaxiDetails(const pqxx::result::const\_iterator& row); - функция для преобразования SQL запроса в текст.

void displayAllTaxis(Database& Db); - функция для отображения такси.

static void displayTaxiById(Database& Db, int taxiId); - функция для поиска такси по уникальному идентификатору.

static void displayTaxisByBrand(Database& Db, const std::string& brand); - функция для поиска такси по марке.

static void displayTaxisByRentCarType(Database& Db, RentCarTypes rentCarType); - функция для поиска такси по типу.

std::string to\_string(const int value); - функция для преобразования в строку.

void orderTaxi(Database& Db); - функция для заказа такси.

1. Класс остановочного пункта.

class Stop {

private:

int stopId; - уникальный идентификатор остановочного пункта.

std::string stopName; - название.

std::string address; - адресс.

public:

Stop(int stopId, const std::string& stopName, const std::string& stopAddress); - параметризированный конструктор.

Stop(); - конструктор по умолчанию.

int getStopId() const; - геттер для доступа к приватному полю.

void setStopId(int stopId); - сеттер для изменения приватного поля.

std::string getStopName(); - геттер для доступа к приватному полю.

void setStopName(std::string stopName);

std::string getAddress() const; - геттер для доступа к приватному полю.

void setAddress(std::string address); - сеттер для изменения приватного поля.

static std::vector<Stop> findAllStops(Database& db); - функция для вывода всех остановок.

static std::optional<Stop> findStopById(Database& db, int stopId); - функция для поиска остановок по уникальному идентификатору.

1. Класс маршрута.

class Route {

private:

int routeId; - уникальный идентификатор маршрута.

std::string routeName; - название маршрута.

std::vector<Stop> stops; - остановочные пункты, привязанные к маршруту.

public:

Route(); - конструктор по умолчанию.

Route(int routeId, std::string& routeName, std::vector<Stop> routeStops); - - параметризированный конструктор.

int getRouteId(); - геттер для доступа к приватному полю.

std::string getRouteName(); - геттер для доступа к приватному полю.

std::vector<Stop> getStops(); - геттер для доступа к приватному полю.

void setRouteId(int newRouteId); - сеттер для изменения приватного поля.

void setRouteName(std::string newRouteName); - сеттер для изменения приватного поля.

void setStops(std::vector<Stop> newRouteStops); - сеттер для изменения приватного поля.

void getStopsForRoute(Database& db, int routeId); - геттер для доступа к приватному полю.

void getRoutesForTransport(Database& db, int transportId, PublicTransport::TransportType transportType); - геттер для доступа к приватному полю.

void displayAllRoutes(Database& db); - функция для вывода всех маршрутов.

1. Класс заказа.

class Order {

private:

int orderId; - уникальный идентификатор заказа.

int customerId; - уникальный идентификатор покупателя.

int carId; - уникальный идентификатор автомобиля.

std::string orderTime; - время заказа.

public:

Order(); - конструктор по умолчанию.

Order(int orderId, int customerId, int carId, const std::string& orderTime); - параметризированный конструктор.

int getOrderId(); - геттер для доступа к приватному полю.

void setOrderId(int orderId); - сеттер для изменения приватного поля.

int getCustomerId(); - геттер для доступа к приватному полю.

void setCustomerId(int customerId); - сеттер для изменения приватного поля.

int getCarId() const; - геттер для доступа к приватному полю.

void setCarId(int id); - сеттер для изменения приватного поля.

std::string getOrderTime(); - геттер для доступа к приватному полю.

void setOrderTime(std::string orderTime); - сеттер для изменения приватного поля.

std::string getCurrentTime(); - геттер для доступа к приватному полю.

static void printAllOrders(Database &db, int isLogged); - функция для вывода всех закзазов.

1. Класс покупатель.

class Customer {

private:

int customerId; - уникальный идентификатор покупателя.

std::string name; - имя.

std::string surname; - фамилия.

std::string contactInformation; - контактная информация.

public:

Customer(); - конструктор по умолчанию.

int getCustomerId(); - геттер для доступа к приватному полю.

void setCustomerId(int customerId); - сеттер для изменения приватного поля.

std::string getName(); - геттер для доступа к приватному полю.

void setName(std::string name); - сеттер для изменения приватного поля.

std::string getSurname(); - геттер для доступа к приватному полю.

void setSurname(std::string surname); - сеттер для изменения приватного поля.

std::string getContactInformation(); - геттер для доступа к приватному полю.

void setContactInformation(std::string info); - сеттер для изменения приватного поля.

1. Класс билет.

class TransportTicket {

private:

int ticketId; - уникальный идентификатор билета.

int transportId; - уникальный идентификатор транспорта.

int customerId; - уникальный идентификатор покупателя.

double price; - цена.

std::string purchaseTime; - время покупки.

TransportType transportType; - тип транспортного средства, на который будет действителен билет.

Database db; - база данных.

public:

TransportTicket(); - конструктор по умолчанию.

TransportTicket(int ticketId, int transportId, int customerId, double price, std::string purchaseTime, TransportType transportType); - параметризированный конструктор.

int getTicketId(); - геттер для доступа к приватному полю.

int getTransportId(); - геттер для доступа к приватному полю.

int getCustomerId(); - геттер для доступа к приватному полю.

double getPrice(); - геттер для доступа к приватному полю.

std::string getPurchaseTime(); - геттер для доступа к приватному полю.

TransportType getTransportType(); - геттер для доступа к приватному полю.

void setCustomerId(int customerId); - сеттер для изменения приватного поля.

void setTransportId(int transportId); - сеттер для изменения приватного поля.

void setPrice(double price); - сеттер для изменения приватного поля.

void displayTicketInfo(); - функция для вывода составляющих билета на экран.

1. Класс расписание.

class Schedule {

private:

int scheduleId; - уникальный идентификатор расписания.

int routeId; - уникальный идентификатор маршрута.

int stopId; - уникальный идентификатор маршрута.

TransportType transportType; - тип транспортного средства, на которое распространяется расписание.

int transportId; - уникальный идентификатор транспортного средства.

std::string arrivalTime; - время прибытия на определенный остановочный пункт.

public:

Schedule();- конструктор по умолчанию.

Schedule(int scheduleId, int routeId, int stopId, TransportType transportType, int transportId, const std::string& arrivalTime); - параметризированный конструктор.

int getScheduleId(); - геттер для доступа к приватному полю.

void setScheduleId(int scheduleId); - сеттер для изменения приватного поля.

int getRouteId(); - геттер для доступа к приватному полю.

void setRouteId(int routeId); - сеттер для изменения приватного поля.

int getStopId(); - геттер для доступа к приватному полю.

void setStopId(int stopId); - сеттер для изменения приватного поля.

TransportType getTransportType(); - геттер для доступа к приватному полю.

void setTransportType(TransportType transportType); - сеттер для изменения приватного поля.

int getTransportId(); - геттер для доступа к приватному полю.

void setTransportId(int transportId); - сеттер для изменения приватного поля.

std::string getArrivalTime(); - геттер для доступа к приватному полю.

void setArrivalTime(std::string arrivalTime); - сеттер для изменения приватного поля.

static std::vector<Schedule> getScheduleForRoute(Database& db, int routeId); - функция для получения расписания для определенного маршрута.

void printStopsForRoute(Database& db, int routeId);- функция для вывода на экран остановок для маршрута с заданным уникальным идентификатором.

1. Класс цена билета.

class RoutePrice {

private:

int routeId; - уникальный идентификатор маршрута.

double price; - цена.

public:

RoutePrice(); - конструктор по умолчанию.

RoutePrice(int id, double price); - параметризированный конструктор.

int getRouteId(); - геттер для доступа к приватному полю.

double getPrice(); - геттер для доступа к приватному полю.

static RoutePrice fromSQLResult(const pqxx::row& row); - функция для преобразования SQL запроса в текст.

RoutePrice getTicketPrice(Database& db, int routeId); - функция для получения цены билета для маршрута с заданным уникальным идентификатором.

void displayTicketPrice(Database& db, int routeId); -

функция для вывода цены билета.

1. Класс база данных.

class Database {

private:

pqxx::connection C; - переменная, устанавливающая соединение с базой данных.

public:

Database() : C("dbname=CourseWorkDb user=postgres password=aboba hostaddr=127.0.0.1 port=5432") {

if (C.is\_open()) {

std::cout << "Opened database successfully: " << C.dbname() << std::endl;

} else {

std::cout << "Can't open database" << std::endl;

throw std::runtime\_error("Failed to open database");

}

} – параметризированный конструктор, который выполняет подключение к базе данных, в случае ошибки будет выброшено исключение.

~Database() {

} - деструктор.

pqxx::result executeQuery(const std::string& query) {

pqxx::nontransaction N(C);

return N.exec(query);

}- функция для выполнения SQL запроса.

1. Класс администратор.

class Admin {

private:

int adminId; - уникальный идентификатор администратора.

std::string username; - имя.

std::string hashedPassword; - хэшированный пароль.

Database &db; - база данных.

std::string hashPassword(const std::string &password); - функция для хэширования пароля.

public:

Admin(Database &db, const std::string &username, const std::string &password); - параметризированный конструктор.

bool login(); - функция выполняющая запрос к базе данных, для проверки корректности введенных данных администратора.

bool isValidRouteID(Database &db, int routeId); – функция для проверки наличия маршрута в таблице.

std::string engineTypeToString(EngineType engineType); – функция для представления enum в виде строки.

std::string rentCarTypesToString(RentCarTypes rentCarTypes); – функция для представления enum в виде строки.

bool registerAdmin(Database &db, const std::string &username, const std::string &password); – функция для регистрации нового администратора.

void addBus(const std::string &brand, const std::string &model, const std::string &color, EngineType engineType,

int capacity, bool hasContactlessPayment, int isLogged); – функция для добавления автобуса.

void addTrolleyBus(const std::string &brand, const std::string &model, const std::string &color, EngineType engineType, int capacity, bool hasSockets, int isLogged); – функция для добавления троллейбуса.

void addTaxi(const std::string &brand, const std::string &model, const std::string &color, EngineType engineType, double pricePerKilometer, bool hasDriver, bool hasWiFi, bool hasChildSeat, RentCarTypes rentCarTypes, int isLogged); – функция для добавления такси.

void addStop(const std::string &stopName, const std::string &address, int isLogged); – функция для добавления остановки.

void addRoute(const std::string &routeName, int isLogged); – функция для добавления маршрута.

void addSchedule(Database& db, int routeId, TransportType transportType, int transportId, int isLogged); – функция для добавления расписания.

void setRoutePrice(Database &Db, int routeId, double price, int isLogged); – функция для установки цены билета для маршрута с заданным уникальным идентификатором.

void linkTransportToRoute(Database &Db, int route\_id, TransportType transport\_type, int transport\_id, int isLogged); – функция для привязки транспорта с заданным уникальным идентификатором к маршруту с заданным уникальным идентификатором.

void linkStopToRoute(Database &db, int routeId, int stopId, int isLoggedIn); – функция для добавления остановок с заданным определенным идентификатором к маршруту с заданным уникальным идентификатором.

bool adminLogin(Database &db, const std::string &username, const std::string &password); – функция для авторизации администратора.

# 4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## 4.1. Разработка схем алгоритмов

Схемы алгоритмов orderTaxi(Database &db) и getStopsForRoute(Database& db, int routeId). (см. приложения В, Г)

## 4.2. Разработка схемы алгоритма

### 4**.2.1** Разработка алгоритма bookTransport(Database& Db, TransportType transportType)

1. Начало.
2. Функция bookTransport принимает два аргумента: объект базы данных Db и тип транспорта transportType.
3. Определяется тип транспорта в виде строки transportTypeName, который может быть либо “bus”, либо “trolleybus”.
4. Создается объект customer класса Customer.
5. Запрашивается ввод имени, фамилии и контактной информации пользователя.
6. Выполняется SQL-запрос к базе данных для добавления нового клиента в таблицу customer и возвращается идентификатор клиента.
7. Запрашивается ввод идентификатора маршрута routeId.
8. Пытается получить цену билета для данного маршрута.
9. Если происходит исключение при получении цены билета, выводится сообщение об ошибке и функция завершает работу.
10. Запрашивается ввод идентификатора транспорта transportId.
11. Выполняется SQL-запрос к базе данных для проверки существования транспорта с данным идентификатором.
12. Если транспорт с данным идентификатором не существует, выводится сообщение об ошибке и функция завершает работу.
13. Создается объект ticket класса TransportTicket и устанавливаются его свойства.
14. Выполняется функция insertTicket для добавления билета в базу данных и возвращается идентификатор билета.
15. Выводится сообщение о успешном бронировании транспорта и идентификатор билета.
16. Если в процессе выполнения функции происходит исключение, оно перехватывается, и выводится сообщение об ошибке.
17. Конец.

### 4.2.2 Разработка алгоритма setRoutePrice(Database &Db, int routeId, double price, int isLogged)

1. Начало.
2. Функция setRoutePrice принимает четыре аргумента: объект базы данных Db, идентификатор маршрута routeId, цену price и флаг isLogged, указывающий, вошел ли пользователь в систему.
3. Проверяется, вошел ли пользователь в систему. Если isLogged равно 0, это означает, что пользователь не вошел в систему.
4. Если пользователь не вошел в систему, выводится сообщение об ошибке “Error: You must be logged in as an admin to set the route price.” и функция завершает работу.
5. Если пользователь вошел в систему, проверяется, действителен ли идентификатор маршрута routeId с помощью функции isValidRouteID.
6. Если функция isValidRouteID возвращает false, это означает, что предоставленный идентификатор маршрута недействителен.
7. Если идентификатор маршрута недействителен, выводится сообщение об ошибке “Invalid route ID provided.” и функция завершает работу.
8. Если идентификатор маршрута действителен, выполняется SQL-запрос к базе данных для проверки существования цены для данного маршрута в таблице RoutePrice.
9. Если цена для данного маршрута уже существует (то есть результат запроса не пуст), выполняется SQL-запрос для обновления цены этого маршрута в таблице RoutePrice.
10. После успешного обновления цены маршрута выводится сообщение “Route price updated successfully.”
11. Если цена для данного маршрута не существует (то есть результат запроса пуст), выполняется SQL-запрос для добавления новой цены для этого маршрута в таблицу RoutePrice.
12. После успешной установки цены маршрута выводится сообщение “Route price set successfully.”
13. Конец.

# 5 РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

После запуска программы выводится меню для взаимодействия с реализованным функционалом:

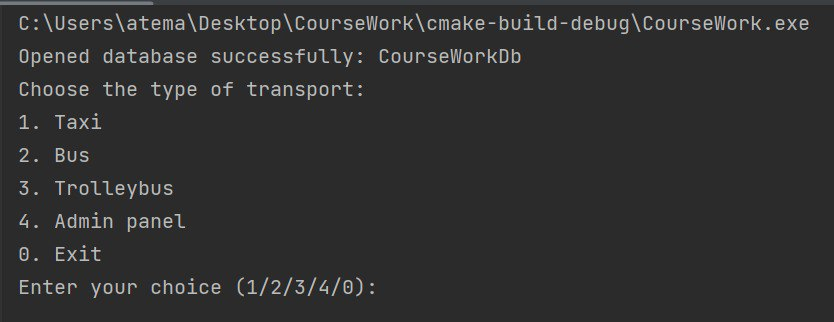


Рисунок 5.1 – Основное меню

Продемонстрируем работу с такси, при выборе первого пункта выведутся следующие кнопки:

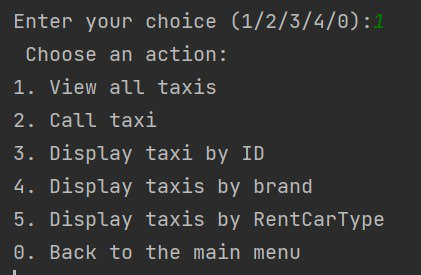


Рисунок 5.2 – Меню такси

Далее можно произвести поиск такси по уникальному идентификатору:

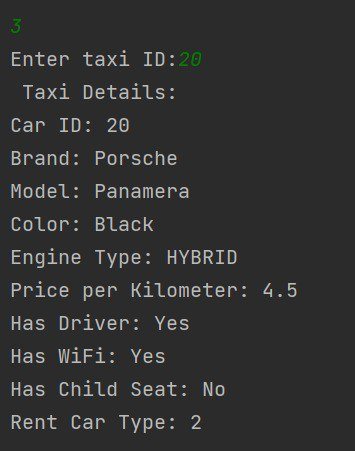


Рисунок 5.3 – Результат поиска

Рассмотрим возможность создавать остановки и маршруты, связывать их, а также ставить на них общественный транспорт:

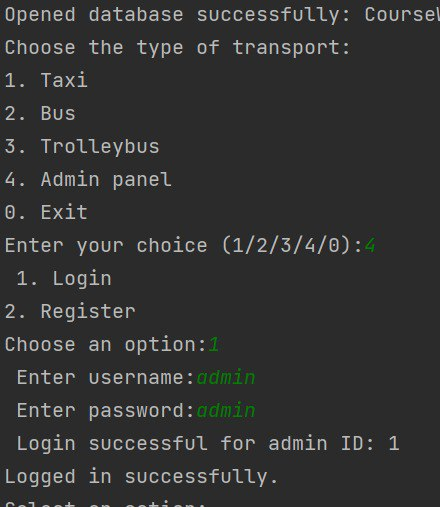


Рисунок 5.4 – Вход в аккаунт администратора

Соответственно пройдя аутентификацию появляется возможность создавать рабочую транспортную систему. Начнем с создания остановочного пункта:

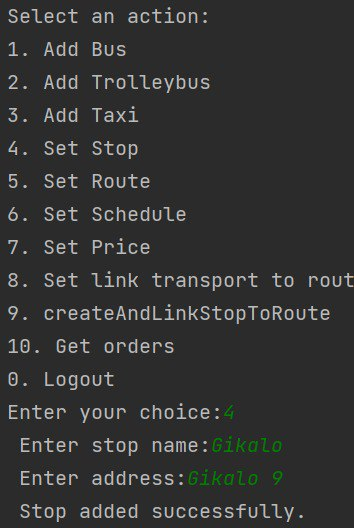


Рисунок 5.5 – Создание остановки

Соответственно далее для создания корректной системы нам необходимо включить остановочный пункт в один из маршрутов. Создадим маршрут:

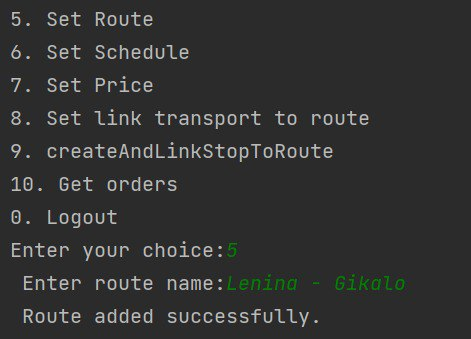


Рисунок 5.6 – Создание маршрута

Далее после создания маршрута появляется возможность привязать остановочные пункты к определенному маршруту:

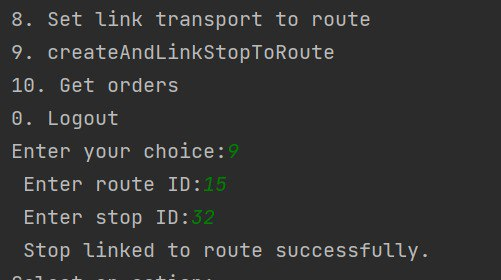


Рисунок 5.7 – Связывание остановки и маршрута

В заключение для полной работоспособности транспортной системы остается поставить транспорт на данный маршрут:

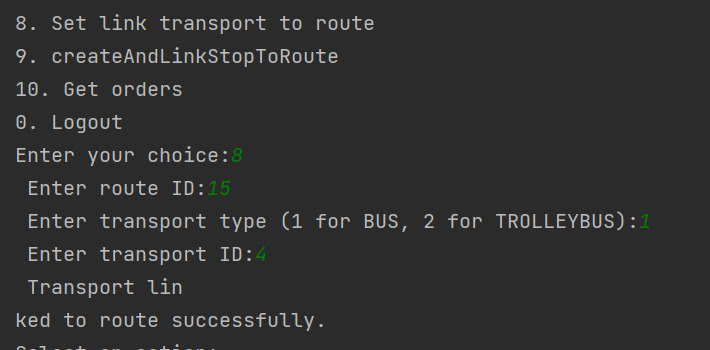


Рисунок 5.8 – Установка автобуса на маршрут

В конечном итоге проведя все необходимые операции с настройкой маршрутов и транспорта, при попытке узнать расписание автобуса под определенным уникальный идентификатором, получим следующий результат:

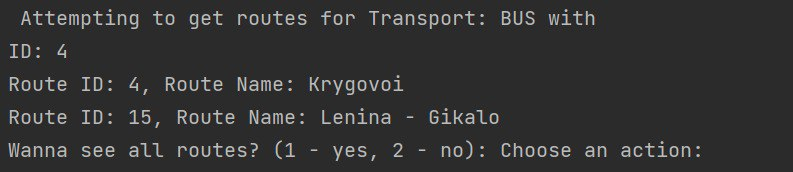


Рисунок 5.9 – Поиск расписания

# ЗАКЛЮЧЕНИЕ

В заключение, разработанная информационная система для автовокзала на языке программирования C++ и с использованием базы данных PostgreSQL представляет собой интегрированное решение, которое обеспечивает эффективное управление и обработку данных. Система предоставляет актуальную информацию о транспорте, маршрутах, расписаниях и остановках, а также обеспечивает возможность редактирования этих данных.

Система обладает высокой производительностью, способна обрабатывать большие объемы данных и обеспечивает надежное и безопасное хранение данных. Благодаря своей гибкости и масштабируемости, система способна адаптироваться к изменяющимся потребностям автовокзала. Модульная структура позволяет легко добавлять новые функции или модифицировать существующие для удовлетворения новых требований.

Возможность интеграции с другими системами или приложениями обеспечивает дополнительную функциональность и упрощает обмен данными. Это может включать в себя интеграцию с системами бронирования билетов, системами управления транспортом или другими информационными системами.

В целом, данная информационная система представляет собой всеобъемлющее решение, которое не только облегчает управление автовокзалом, но и способствует повышению уровня обслуживания пассажиров, обеспечивая им доступ к актуальной и точной информации. В эпоху цифровизации и быстрого технологического прогресса, такая система становится неотъемлемым инструментом для эффективного функционирования автовокзала.

Язык программирования C++ был выбран для разработки этой информационной системы, и это не случайно. C++ - это мощный и гибкий язык, который предлагает широкий спектр возможностей для разработчиков.

C++ позволяет создавать высокопроизводительные приложения и обеспечивает прямой контроль над аппаратными ресурсами. Это особенно важно для систем, которые обрабатывают большие объемы данных, таких как информационная система автовокзала.

Кроме того, C++ поддерживает объектно-ориентированное программирование, что облегчает организацию и структурирование кода. Это делает систему более управляемой, расширяемой и поддерживаемой.

В общем, использование C++ в данной информационной системе обеспечивает высокую производительность, гибкость и надежность, что делает его отличным выбором для такого рода задач. Это подчеркивает его ценность и важность в сфере программирования.

# ЛИТЕРАТУРА

[1] Луцик Ю. А. Объектно-ориентированное программирование на языке С++: учеб. пособие / Ю. А. Луцик, В. Н. Комличенко.– Минск: БГУИР, 2008.– 266 с.

[2] Документация PostgreSQL – Режим доступа: https://www.postgresql.org/ – Дата доступа: 19.10.2023.

[3] Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. Приёмы объектно-ориентированного проектирования. Па́ттерны проектирования. – 1994.

[4] Бьёрн Страуструп. Язык программирования C++. – 1983.

# ПРИЛОЖЕНИЕ А

(Обязательное)

Диаграмма классов

# ПРИЛОЖЕНИЕ Б

(Обязательное)

Схема алгоритма getStopsForRoute(Database& db, int routeId)

# ПРИЛОЖЕНИЕ В

(Обязательное)

Схема алгоритма orderTaxi(Database& Db)

# ПРИЛОЖЕНИЕ Г

(Обязательное)

Исходный текст программы

# ПРИЛОЖЕНИЕ Е

(Обязательное)

Ведомость документов

# ПРИЛОЖЕНИЕ Д

(Обязательное)

Листинг SQL кода.

# ПРИЛОЖЕНИЕ Ж

(Обязательное)

Диаграмма SQL таблиц.