**1. Лабораторная работа 118 База данных (access, SQL server, PostgreSQL,…) – хранилище данных программного средства (чтение, запись и временное хранение данных программного средства**

# **Цель**

Изучить основы работы с различными системами управления базами данных (Microsoft Access, SQL Server, PostgreSQL) как хранилищами данных программного средства, освоить методы чтения, записи и временного хранения данных на языке C++.

# **1.1 Теоретические сведения**

## **1.1.1 Зачем нужны БД ?**

Представьте, что у вас есть куча информации о программах, которые вы используете: их названия, версии, даты установки — всё, что нужно для работы. Теперь подумайте, как это хранить? Можно, конечно, записать всё в блокнот: "Программа 1, версия 1.0", "Программа 2, версия 2.5". Но что, если программ станет сто или тысяча? А если нужно быстро найти программу по имени или изменить версию? Листать блокнот и искать вручную — это долго и неудобно, как искать иголку в стоге сена.

Вот тут и приходят на помощь базы данных (БД) — это как умный шкаф с ящиками, где всё аккуратно разложено по полочкам. Например, у нас есть Microsoft Access, SQL Server и PostgreSQL — это разные виды таких "шкафов". Они помогают организовать данные так, чтобы:

* Быстро находить нужное (например, "покажи все программы с версией выше 1.0").
* Легко добавлять новое (например, "добавь новую программу App1").
* Менять данные (например, "обнови версию программы с ID 1").
* Удалять ненужное (например, "удали программу, которую больше не используем").

Простыми словами, база данных — это место, где данные хранятся в виде таблиц, как в Excel, только гораздо мощнее. Но зачем нам три разные базы — Access, SQL Server и PostgreSQL? Они как разные шкафы для разных задач:

* **Microsoft Access** — это как небольшой шкаф у вас дома: удобно для маленьких проектов, всё хранится в одном файле (например, .accdb), и его легко настроить.
* **SQL Server** — это как огромный шкаф в офисе: он мощный, может хранить данные для целой компании, но требует сервера и сложнее в настройке.
* **PostgreSQL** — это как универсальный шкаф с открытым доступом: он бесплатный, мощный, и его любят разработчики за гибкость.

## **1.1.2** **Основы работы с базами данных?**

Базы данных — это реляционные системы управления базами данных (СУБД), которые хранят данные в виде таблиц. Каждая таблица состоит из строк и столбцов, где строки — это записи, а столбцы — это поля (например, ID, Name, Version).

### **1.1.2.1 Основные СУБД, используемые в работе**

* **Microsoft Access**: Файловая СУБД, хранит данные в файлах .accdb. Подходит для небольших проектов.
* **SQL Server**: Серверная СУБД от Microsoft, предназначена для крупных приложений.
* **PostgreSQL**: Открытая серверная СУБД, популярна среди разработчиков за гибкость и производительность.

### **1.1.2.2 ODBC (Open Database Connectivity)**

ODBC — это стандартный API, который позволяет программам на C++ подключаться к разным базам данных (Access, SQL Server, PostgreSQL) через специальные драйверы. Для Access используется драйвер Microsoft Access Driver (\*.mdb, \*.accdb).

Основные типы данных:

* SQL\_VARCHAR – строка переменной длины
* SQL\_INTEGER – целое число
* SQLDOUBLE – число с двойной точностью
* SQL\_TIMESTAMP – дата и время

Основные функции:

* SQLAllocHandle() – создает дескриптор для среды, соединения или запроса
* SQLSetEnvAttr() – устанавливает атрибуты окружения
* SQLConnect() – подключается к базе данных
* SQLDisconnect() – закрывает соединение с БД

### **1.1.2.3 SQL (Structured Query Language)**

SQL — это язык запросов для работы с данными в базах.

Основные типы данных:

* INT – целое число
* DOUBLE – 8-байтное число с двойной точностью
* VARCHAR – строка переменной длины
* BLOB – большие бинарные объекты

Основные функции:

* **INSERT** - Добавление новой записи.
* **SELECT** - Извлечение данных.
* **UPDATE** - Изменение данных.
* **DELETE** - Удаление данных.

### **1.1.2.4 Временное хранение данных**

Временное хранение данных в программе на C++ можно реализовать с помощью контейнеров, таких как std::vector. Данные сначала сохраняются в память, а затем могут быть записаны в базу данных. Это ускоряет работу приложения, так как доступ к памяти быстрее, чем к базе данных.

**1.1.3** **Дескриптор**

Дескриптор (handle) – объект или указатель, который используется для доступа к ресурсам СУБД (например, соединению, запросу или результату). Он действует как «ключ» для взаимодействия с внутренними структурами базы данных через API (ODBC). Дескриптор, в основном, абстрагирует сложность, управляет ресурсами и обеспечивает безопасность.

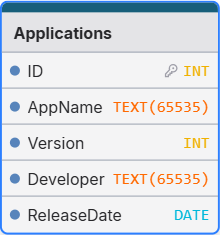
# **1.2 Порядок выполнения работы.**

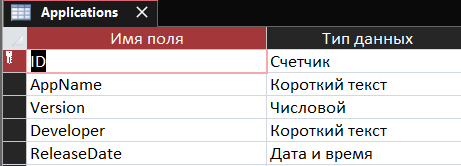
### **1.2.1 Типовой пример 1**

Для начала давайте сделаем БД и свяжем её с нашим проектом в VS. Для начала переходим в Microsoft Access и создаём новый файл

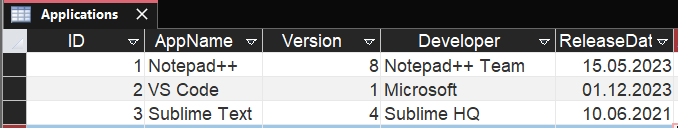


с такой структурой (таблицу называем Applications):

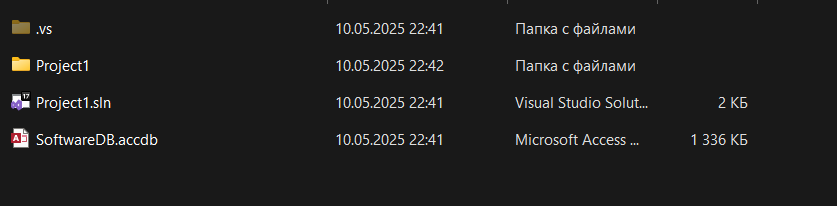




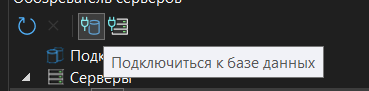
Далее заполняем все поля такой информацией



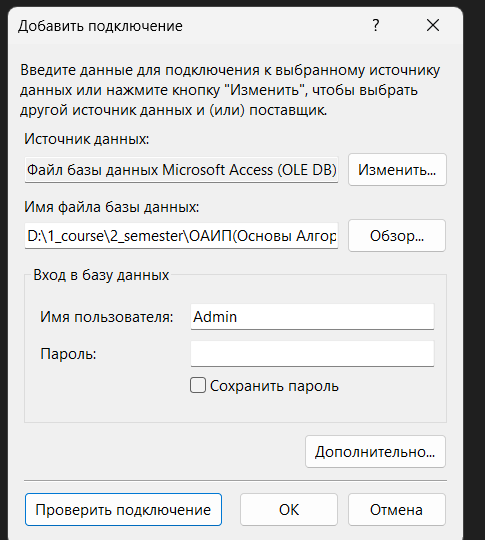
Теперь давайте создадим проект в VS ( я проект назвал Project1 ). После этого переносим нашу БД в папку к проекту и получается такая структура:

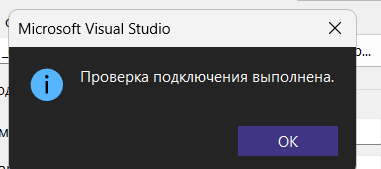


Создаём как обычно пустой проект Нажимаем Вид -> Обозреватель серверов



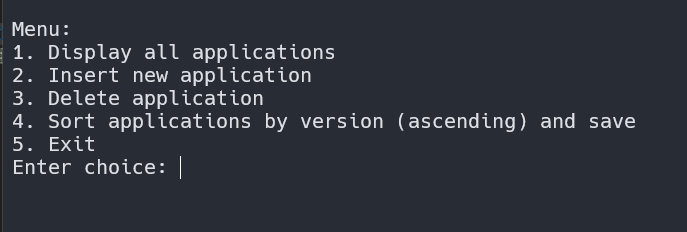
Нажимаем подключиться к базе данных. Дальше в этом окошке в строке имя файла указываем путь к нашей БД.



И теперь нажимаем Проверить подключение. Если отобразилось 

Значит всё хорошо. Теперь напишем main.cpp

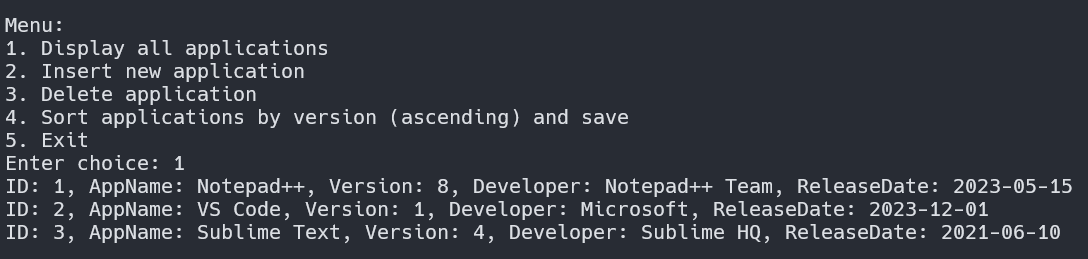
|  |
| --- |
| #include <windows.h>  #include <sqlext.h>  #include <iostream>  #include <vector>  #include <algorithm>  #include <string>  #include <iomanip>  // Функция для проверки формата даты YYYY-MM-DD  // Проверяет, что введённая дата корректна (например, 2023-05-15)  // Возвращает true, если дата правильная, и false, если есть ошибки  bool validateDate(const std::wstring& date) {  // Проверяем длину строки и наличие символов '-' на нужных позициях  if (date.length() != 10 || date[4] != L'-' || date[7] != L'-') {  return false;  }  try {  // Извлекаем год, месяц и день из строки и преобразуем их в числа  int year = std::stoi(date.substr(0, 4)); // Первые 4 символа — год  int month = std::stoi(date.substr(5, 2)); // Символы 5-6 — месяц  int day = std::stoi(date.substr(8, 2)); // Символы 8-9 — день  // Проверяем, что месяц от 1 до 12  if (month < 1 || month > 12) return false;  // Проверяем, что день от 1 до 31  if (day < 1 || day > 31) return false;  // Проверяем, что год в разумных пределах (1900–9999)  if (year < 1900 || year > 9999) return false;  // Проверяем количество дней в месяце (простая проверка, без учёта високосных годов)  if (month == 2 && day > 29) return false; // Февраль не может иметь больше 29 дней  // Апрель, июнь, сентябрь и ноябрь имеют 30 дней  if ((month == 4 || month == 6 || month == 9 || month == 11) && day > 30) return false;  return true; // Дата корректна  }  catch (...) {  return false;  }  }  // Функция для отображения ошибок ODBC  // Если что-то идёт не так при работе с базой данных, эта функция покажет, что именно пошло не так  void showSQLError(SQLSMALLINT handleType, SQLHANDLE handle) {  SQLWCHAR sqlState[1024]; // Буфер для кода состояния ошибки  SQLWCHAR message[2048]; // Буфер для текста ошибки  SQLINTEGER nativeError; // Код ошибки  SQLSMALLINT textLength; // Длина текста ошибки  // Получаем информацию об ошибке  SQLRETURN ret = SQLGetDiagRecW(handleType, handle, 1, sqlState, &nativeError, message, 2048, &textLength);  if (ret == SQL\_SUCCESS || ret == SQL\_SUCCESS\_WITH\_INFO) {  // Выводим текст ошибки и код состояния  std::wcout << L"ODBC Error: " << message << L" [State: " << sqlState << L"]\n";  }  }  // Структура для хранения данных одной записи из таблицы Applications  struct Application {  SQLINTEGER id; // ID записи (автоинкрементное поле в базе)  std::wstring appName; // Название приложения (например, Notepad++)  double version; // Версия приложения (например, 8.5)  std::wstring developer; // Разработчик (например, Notepad++ Team)  std::wstring releaseDate; // Дата выпуска (например, 2023-05-15)  };  // Функция для подключения к базе данных Access  // Устанавливает соединение с файлом SoftwareDB.accdb  bool connectToDB(SQLHENV& hEnv, SQLHDBC& hDbc) {  // Создаём окружение (environment) для работы с ODBC  SQLAllocHandle(SQL\_HANDLE\_ENV, SQL\_NULL\_HANDLE, &hEnv);  // Устанавливаем версию ODBC (используем версию 3)  SQLSetEnvAttr(hEnv, SQL\_ATTR\_ODBC\_VERSION, (void\*)SQL\_OV\_ODBC3, 0);  // Создаём дескриптор соединения (connection handle)  SQLAllocHandle(SQL\_HANDLE\_DBC, hEnv, &hDbc);  // Строка подключения: указываем драйвер и путь к файлу базы данных  SQLWCHAR connStr[] = L"Driver={Microsoft Access Driver (\*.mdb, \*.accdb)};DBQ=D:\\1\_course\\2\_semester\\ОАИП(Основы Алгоритмизации и Программирования)\\лабы\_белодед\\код ко второй\\для бд\\SoftwareDB.accdb;";  SQLWCHAR outConnStr[1024]; // Буфер для строки подключения после выполнения  SQLSMALLINT outConnStrLen; // Длина строки подключения  // Выполняем подключение к базе данных  SQLRETURN ret = SQLDriverConnectW(hDbc, NULL, connStr, SQL\_NTS, outConnStr, 1024, &outConnStrLen, SQL\_DRIVER\_COMPLETE);  if (!SQL\_SUCCEEDED(ret)) {  // Если подключение не удалось, показываем ошибку  showSQLError(SQL\_HANDLE\_DBC, hDbc);  return false;  }  return true; // Подключение успешно  }  // Функция для чтения всех данных из таблицы Applications  // Возвращает вектор записей (Application), которые хранятся в базе  std::vector<Application> readData(SQLHDBC hDbc) {  std::vector<Application> apps; // Вектор для хранения всех записей  SQLHSTMT hStmt; // Дескриптор запроса (statement handle)  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt); // Создаём дескриптор запроса  // Выполняем SQL-запрос: выбираем все столбцы из таблицы Applications  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT ID, AppName, Version, Developer, ReleaseDate FROM Applications;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  // Переменные для хранения данных из столбцов  SQLINTEGER id;  SQLWCHAR appName[101], developer[101], releaseDate[50];  SQLDOUBLE version;  SQLLEN idLen, appNameLen, versionLen, developerLen, releaseDateLen;  // Читаем строки из таблицы по одной  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  Application app = { 0 }; // Создаём пустую структуру для текущей записи  // Читаем ID (целочисленное значение)  ret = SQLGetData(hStmt, 1, SQL\_C\_LONG, &id, 0, &idLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching ID\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.id = id;  // Читаем AppName (строку)  ret = SQLGetData(hStmt, 2, SQL\_C\_WCHAR, appName, sizeof(appName), &appNameLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching AppName\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.appName = appName;  // Читаем Version (число с плавающей точкой)  ret = SQLGetData(hStmt, 3, SQL\_C\_DOUBLE, &version, 0, &versionLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching Version\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.version = version;  // Читаем Developer (строку)  ret = SQLGetData(hStmt, 4, SQL\_C\_WCHAR, developer, sizeof(developer), &developerLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching Developer\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  break;  }  app.developer = developer;  // Читаем ReleaseDate (строку)  ret = SQLGetData(hStmt, 5, SQL\_C\_WCHAR, releaseDate, sizeof(releaseDate), &releaseDateLen);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error fetching ReleaseDate\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  app.releaseDate = L"Invalid Date";  }  else {  // Если дата содержит время (например, 2023-05-15 00:00:00), обрезаем его  std::wstring dateStr(releaseDate);  size\_t spacePos = dateStr.find(L" ");  if (spacePos != std::wstring::npos) {  app.releaseDate = dateStr.substr(0, spacePos);  }  else {  app.releaseDate = dateStr;  }  }  apps.push\_back(app); // Добавляем запись в вектор  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор запроса  return apps; // Возвращаем вектор записей  }  // Функция для вывода данных на экран  // Принимает вектор записей и печатает их в консоль  void displayData(const std::vector<Application>& apps) {  for (const auto& app : apps) {  std::wcout << L"ID: " << app.id << L", AppName: " << app.appName  << L", Version: " << app.version  << L", Developer: " << app.developer  << L", ReleaseDate: " << app.releaseDate << L"\n";  }  }  // Функция для вставки новой записи в таблицу Applications  void insertData(SQLHDBC hDbc, const std::wstring& appName, double version, const std::wstring& developer, const std::wstring& releaseDate) {  // Проверяем, что дата корректна  if (!validateDate(releaseDate)) {  std::wcout << L"Invalid date format. Please use YYYY-MM-DD with valid values (e.g., 2023-05-15).\n";  return;  }  SQLHSTMT hStmt; // Создаём дескриптор для SQL-запроса  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  // Формируем SQL-запрос для вставки новой записи  // Для даты в Access нужно использовать формат #YYYY-MM-DD#  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Applications (AppName, Version, Developer, ReleaseDate) VALUES ('%s', %f, '%s', #%s#);",  appName.c\_str(), version, developer.c\_str(), releaseDate.c\_str());  // Выполняем запрос  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  std::wcout << L"Record inserted successfully.\n"; // Сообщаем об успешной вставке  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc)); // Показываем обновлённую таблицу  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор  }  // Функция для удаления записи из таблицы по ID  void deleteData(SQLHDBC hDbc, int id) {  SQLHSTMT hStmt; // Создаём дескриптор для SQL-запроса  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  // Формируем SQL-запрос для удаления записи с заданным ID  SQLWCHAR query[256];  swprintf\_s(query, 256, L"DELETE FROM Applications WHERE ID=%d;", id);  // Выполняем запрос  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  std::wcout << L"Record deleted successfully.\n"; // Сообщаем об успешном удалении  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc)); // Показываем обновлённую таблицу  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не удался  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем дескриптор  }  // Функция для сортировки записей по версии и сохранения их в базу  void sortByVersionAndSave(SQLHDBC hDbc) {  // 1. Читаем все записи из базы  std::vector<Application> apps = readData(hDbc);  // 2. Сортируем записи в памяти по возрастанию версии  std::sort(apps.begin(), apps.end(), [](const Application& a, const Application& b) {  return a.version < b.version;  });  // 3. Очищаем таблицу Applications перед вставкой отсортированных данных  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"DELETE FROM Applications;", SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error clearing table before sorting\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  // 4. Вставляем отсортированные записи обратно в таблицу  for (const auto& app : apps) {  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  // Формируем запрос для вставки каждой записи  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Applications (AppName, Version, Developer, ReleaseDate) VALUES ('%s', %f, '%s', #%s#);",  app.appName.c\_str(), app.version, app.developer.c\_str(), app.releaseDate.c\_str());  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error inserting sorted record\n";  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  }  // 5. Показываем результат сортировки  std::wcout << L"Applications sorted and saved successfully.\n";  std::wcout << L"\nUpdated database:\n";  displayData(readData(hDbc));  }  // Главная функция программы  int main() {  SetConsoleOutputCP(CP\_UTF8); // Устанавливаем кодировку консоли для поддержки Unicode  SQLHENV hEnv = nullptr; // Переменная для окружения ODBC  SQLHDBC hDbc = nullptr; // Переменная для соединения с базой данных  // Подключаемся к базе данных  if (!connectToDB(hEnv, hDbc)) {  return 1; // Если подключение не удалось, завершаем программу  }  int choice; // Переменная для выбора опции меню  std::wstring appName, developer, releaseDate; // Переменные для ввода данных  double version;  int id;  // Основной цикл программы  while (true) {  // Выводим меню с доступными опциями  std::wcout << L"\nMenu:\n";  std::wcout << L"1. Display all applications\n"; // Показать все записи  std::wcout << L"2. Insert new application\n"; // Добавить новую запись  std::wcout << L"3. Delete application\n"; // Удалить запись  std::wcout << L"4. Sort applications by version (ascending) and save\n"; // Сортировать и сохранить  std::wcout << L"5. Exit\n"; // Выход из программы  std::wcout << L"Enter choice: ";  std::cin >> choice; // Считываем выбор пользователя  std::cin.ignore(); // Очищаем буфер ввода  if (choice == 5) break; // Если выбрана опция 5, выходим из цикла  // Обрабатываем выбор пользователя  switch (choice) {  case 1: // Показать все записи  displayData(readData(hDbc));  break;  case 2: // Добавить новую запись  std::wcout << L"Enter AppName: ";  std::getline(std::wcin, appName); // Считываем название приложения  std::wcout << L"Enter Version: ";  std::cin >> version; // Считываем версию  std::cin.ignore(); // Очищаем буфер  std::wcout << L"Enter Developer: ";  std::getline(std::wcin, developer); // Считываем разработчика  std::wcout << L"Enter ReleaseDate (YYYY-MM-DD): ";  std::getline(std::wcin, releaseDate); // Считываем дату выпуска  insertData(hDbc, appName, version, developer, releaseDate); // Добавляем запись  break;  case 3: // Удалить запись  std::wcout << L"Enter ID to delete: ";  std::cin >> id; // Считываем ID записи для удаления  deleteData(hDbc, id); // Удаляем запись  break;  case 4: // Сортировать и сохранить  sortByVersionAndSave(hDbc);  break;  default:  std::wcout << L"Invalid choice.\n"; // Сообщаем о некорректном выборе  }  }  // Закрываем соединение с базой данных перед завершением  if (hDbc) {  SQLDisconnect(hDbc); // Отключаемся от базы  SQLFreeHandle(SQL\_HANDLE\_DBC, hDbc); // Освобождаем дескриптор соединения  }  if (hEnv) SQLFreeHandle(SQL\_HANDLE\_ENV, hEnv); // Освобождаем окружение  return 0; // Завершаем программу  } |

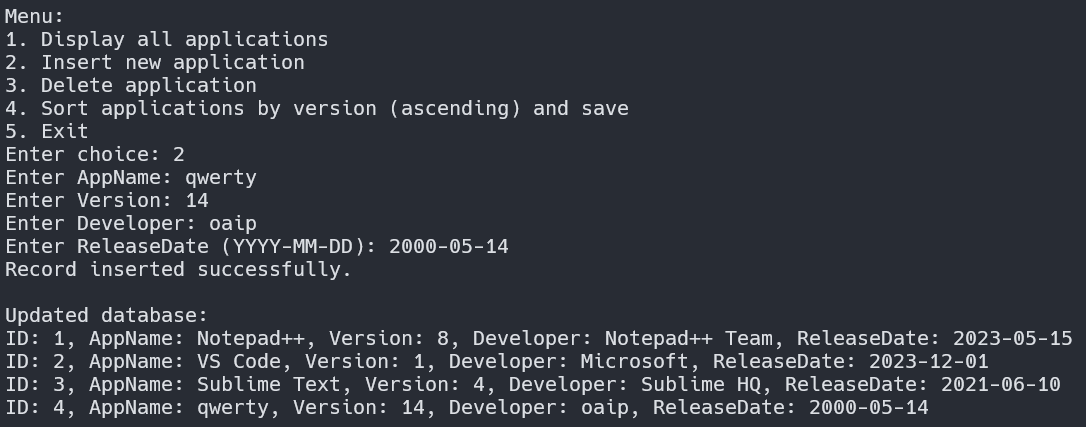
Если всё запустилось , то у вас после запуска выведется меню 

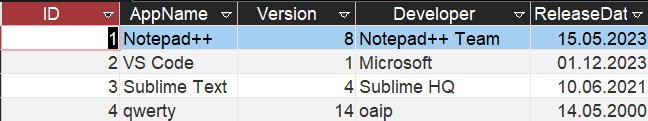
Программа реализует основные задачи управления данными в таблице Applications:

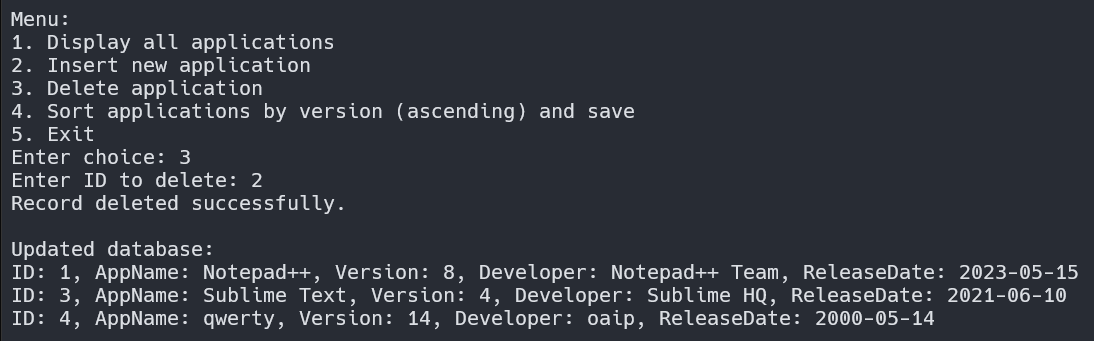
* Отображение всех приложений из базы данных.
* Вставка нового приложения в таблицу.
* Удаление приложения по ID.
* Сортировка приложений по версии (по возрастанию) с сохранением в базу.

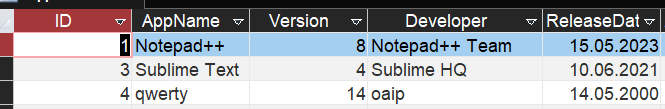
Если мы добавляем/сортируем/удаляем нам нужно закрыть и открыть нащу БД на компьютере, т.к. в онлайне данные не меняются. Пример работы программы:

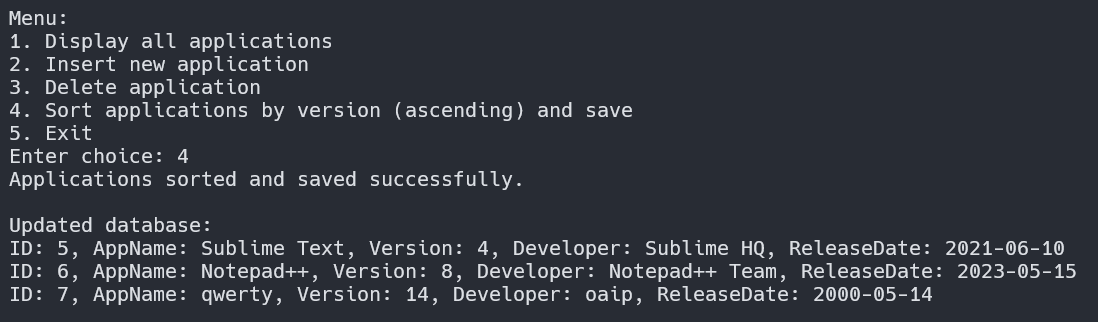














### **1.2.2 Типовой пример 2**

Для начала давайте установим необходимый компонент в Microsoft Visual Studio. Открываем Visual Studio Installer

Изображение выглядит как снимок экрана, Графика, Шрифт, логотип

Контент, сгенерированный ИИ, может содержать ошибки.

Переходим на страницу с компонентами нажатием кнопки «**Изменить»**

Изображение выглядит как текст, Мультимедийное программное обеспечение, программное обеспечение, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Переходим на страницу «**Отдельные компоненты»,** пишем в поиске **SQL Server ODBC Driver,** нажимаем на галочку (если не установлено), нажимаем кнопку «**Изменить**» Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Далее создаем консольное приложение, открываем **Обозреватель серверов** (Вид -> Обозреватель серверов / Ctrl + alt + S). Нажимаем на кнопку **Подключиться к базе данных**.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

В поле источник данных по умолчанию установлено **Microsoft SQL Server (SqlClient),** его не трогаем. В поле имя сервера мы указываем **.\sqlexpress**, где точка заменяет надпись localhost, sqlexpress – базовое название сервера. Устанавливаем галочкусертификат сервера в значение **true.** В поле имя базы данных вводим произвольное название БД, в нашем случаем это будет **OurFirstBD.**

После нажатия на кнопку **OK** нас оповестят, что нашей базы данных не существует, попытаться ли создать ее, соглашаемся с этим.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Контент, сгенерированный ИИ, может содержать ошибки.

Если же все завершилось успешно, то теперь мы подключены к нашей базе данных, теперь давайте создадим нашу первую таблицу. Нажимаем по папке **Таблицы** правой кнопкой мыши, после чего выбираем **Добавить новую таблицу.**

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

У нас создалась наша первая таблица, сразу же дадим ей название, путем изменения в коде, назовем **FirstTable.** Также укажем первому столбцу, что он будет являться уникальным идентификатором, путем его выделения, и в окне свойств, установлением значения **Индентификатор в значение True**. Код изменится автоматически.

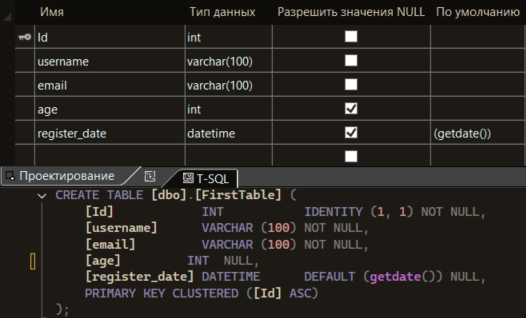
Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

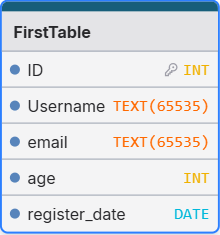
Контент, сгенерированный ИИ, может содержать ошибки.

Добавим значения нашей таблицы, путем нажатия на ячейки. Изображение выглядит как снимок экрана, текст, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Таблица примет следующий вид:





Добавили 4 поля: username, email, phone, register\_date. Значение поля register\_date по умолчанию будет устанавливать текущую дату и время. Нажимаем кнопку **Обновить** в левом верхнем углу над таблицей.

Открывается окно предварительного просмотра, после чего выбираем **Обновить базу данных**.

Изображение выглядит как текст, электроника, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

После чего выбираем нажимаем правой кнопкой мыши на нашу базу данных, в моем случае **msi\sqlexpress.OurFirstBD.dbo** и нажимаем **Обновить**, должна появиться наша таблица **FirstTable** в папке **Таблицы.**

Далее нажимаем правой кнопкой мыши на нашу таблицу, нажимаем на **Показать таблицу данных**, после чего должно открыться окно просмотра таблицы.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Теперь мы видим нашу таблицу, давайте заполним ее тестовыми значениями. Заполняем только поля **username, email, phone.** Поля **Id** и **register\_date**, заполняются автоматически. Таблица примет следующий вид: Изображение выглядит как текст, снимок экрана, Шрифт, линия

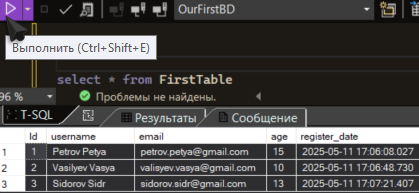
Контент, сгенерированный ИИ, может содержать ошибки.

Для проверки, что все данные успешно сохранились, мы можем создать новый запрос для нашей БД и получить значения. Нажимаем правой кнопкой мыши по нашей БД и выбираем **Новый запрос**.

Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

В открывшемся запросе прописываем следующий код и нажимаем **Выполнить** (ctrl + shift + E). После чего видим, что таблица сохранилась правильно.



Также, в этом же запросе, мы можем создать таблицу через код:

create table connections

(

id INT NOT NULL PRIMARY KEY IDENTITY(1,1),

ipaddress VARCHAR(20) NOT NULL,

machine VARCHAR(100) NOT NULL,

connection\_date datetime default CURRENT\_TIMESTAMP

);

После чего опять же выполняем код и в окне вывода, по сообщению **Команды выполнены успешно.** видим, что таблица создалась успешно. Обновляем в **Обозревателе серверов**, и видим, что появилась наша новая таблица:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Теперь напишем main.cpp

#include <windows.h>

#include <sql.h>

#include <sqlext.h>

#include <iostream>

#include <iomanip>

#include <string>

#include <conio.h>

// Вывод ошибок ODBC

void showError(unsigned int handleType, const *SQLHANDLE*& handle) {

*SQLWCHAR* sqlState[1024];

*SQLWCHAR* message[1024];

*SQLSMALLINT* i = 1;

while (*SQLGetDiagRecW*(handleType, handle, i, sqlState, *NULL*, message, sizeof(message) / sizeof(*SQLWCHAR*), *NULL*) == *SQL\_SUCCESS*) {

*std*::*wcerr* << L"SQL Error [" << i << L"]: " << message

<< L"\nSQL State: " << sqlState << *std*::*endl*;

i++;

}

}

// Подключение к базе данных SQL Server через ODBC

*SQLHANDLE* connectToDatabase() {

*SQLHANDLE* sqlEnvHandle = nullptr;

*SQLHANDLE* sqlConnHandle = nullptr;

// Создание и настройка окружения ODBC

if (*SQL\_SUCCESS* != *SQLAllocHandle*(*SQL\_HANDLE\_ENV*, *SQL\_NULL\_HANDLE*, &sqlEnvHandle)) {

*std*::*cerr* << "Error allocating environment handle" << *std*::*endl*;

return nullptr;

}

if (*SQL\_SUCCESS* != *SQLSetEnvAttr*(sqlEnvHandle, *SQL\_ATTR\_ODBC\_VERSION*, (*SQLPOINTER*)*SQL\_OV\_ODBC3*, 0)) {

showError(*SQL\_HANDLE\_ENV*, sqlEnvHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_ENV*, sqlEnvHandle);

return nullptr;

}

if (*SQL\_SUCCESS* != *SQLAllocHandle*(*SQL\_HANDLE\_DBC*, sqlEnvHandle, &sqlConnHandle)) {

showError(*SQL\_HANDLE\_ENV*, sqlEnvHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_ENV*, sqlEnvHandle);

return nullptr;

}

// Строка подключения к серверу SQL Server

*SQLWCHAR*\* connStr = (*SQLWCHAR*\*)L"DRIVER={ODBC Driver 17 for SQL Server};SERVER=localhost\\SQLEXPRESS;DATABASE=OurFirstBD;Trusted\_Connection=yes;";

*SQLWCHAR* outConnStr[1024];

*SQLSMALLINT* outConnStrLen;

*SQLRETURN* retCode = *SQLDriverConnectW*(sqlConnHandle, *NULL*, connStr, *SQL\_NTS*,

outConnStr, sizeof(outConnStr) / sizeof(*SQLWCHAR*), &outConnStrLen,

*SQL\_DRIVER\_COMPLETE*);

if (!*SQL\_SUCCEEDED*(retCode)) {

*std*::*wcerr* << L"Failed to connect to SQL Server. Connection string: " << connStr << *std*::*endl*;

showError(*SQL\_HANDLE\_DBC*, sqlConnHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_DBC*, sqlConnHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_ENV*, sqlEnvHandle);

return nullptr;

}

return sqlConnHandle;

}

// Вывод содержимого таблицы

void printTable(*SQLHANDLE* sqlConnHandle) {

*SQLHANDLE* sqlStmtHandle;

*SQLAllocHandle*(*SQL\_HANDLE\_STMT*, sqlConnHandle, &sqlStmtHandle);

// SQL-запрос на выборку данных

*SQLWCHAR*\* sqlQuery = (*SQLWCHAR*\*)L"SELECT Id, username, email, age, register\_date FROM dbo.FirstTable";

*SQLExecDirectW*(sqlStmtHandle, sqlQuery, *SQL\_NTS*);

// Заголовки таблицы

*std*::*wcout* << *std*::*left*

<< *std*::*setw*(5) << L"ID"

<< *std*::*setw*(20) << L"Username"

<< *std*::*setw*(25) << L"Email"

<< *std*::*setw*(10) << L"Age"

<< *std*::*setw*(20) << L"Register Date"

<< *std*::*endl*;

*std*::*wcout* << *std*::*setfill*(L'-') << *std*::*setw*(80) << L"" << *std*::*setfill*(L' ') << *std*::*endl*;

// Обработка и вывод каждой строки результата

while (*SQLFetch*(sqlStmtHandle) == *SQL\_SUCCESS*) {

*SQLINTEGER* id, age;

*SQLWCHAR* username[101], email[101];

*SQL\_TIMESTAMP\_STRUCT* registerDate;

*SQLLEN* usernameLen, emailLen, ageLen, registerDateLen;

*SQLGetData*(sqlStmtHandle, 1, *SQL\_C\_LONG*, &id, 0, *NULL*);

*SQLGetData*(sqlStmtHandle, 2, *SQL\_C\_WCHAR*, username, sizeof(username), &usernameLen);

*SQLGetData*(sqlStmtHandle, 3, *SQL\_C\_WCHAR*, email, sizeof(email), &emailLen);

*SQLGetData*(sqlStmtHandle, 4, *SQL\_C\_LONG*, &age, 0, &ageLen);

*SQLGetData*(sqlStmtHandle, 5, *SQL\_C\_TYPE\_TIMESTAMP*, &registerDate, sizeof(registerDate), &registerDateLen);

// Форматирование даты

wchar\_t dateBuffer[20];

*swprintf*(dateBuffer, 20, L"%04d-%02d-%02d %02d:%02d:%02d",

registerDate.*year*, registerDate.*month*, registerDate.*day*,

registerDate.*hour*, registerDate.*minute*, registerDate.*second*);

*std*::*wcout* << *std*::*left*

<< *std*::*setw*(5) << id

<< *std*::*setw*(20) << (usernameLen == *SQL\_NULL\_DATA* ? L"NULL" : username)

<< *std*::*setw*(25) << (emailLen == *SQL\_NULL\_DATA* ? L"NULL" : email)

<< *std*::*setw*(10) << (ageLen == *SQL\_NULL\_DATA* ? L"NULL" : *std*::*to\_wstring*(age))

<< *std*::*setw*(20) << (registerDateLen == *SQL\_NULL\_DATA* ? L"NULL" : dateBuffer)

<< *std*::*endl*;

}

*SQLFreeHandle*(*SQL\_HANDLE\_STMT*, sqlStmtHandle);

}

// Добавление новой записи в таблицу

void addItem(*SQLHANDLE* sqlConnHandle) {

*std*::*wstring* username, email;

int age;

*std*::*wcout* << L"Enter username: ";

*std*::*getline*(*std*::*wcin*, username);

*std*::*wcout* << L"Enter email: ";

*std*::*getline*(*std*::*wcin*, email);

*std*::*wcout* << L"Enter age (or 0 for NULL): ";

*std*::*wcin* >> age;

*std*::*wcin*.*ignore*();

*SQLHANDLE* sqlStmtHandle;

*SQLAllocHandle*(*SQL\_HANDLE\_STMT*, sqlConnHandle, &sqlStmtHandle);

// Формирование SQL-запроса с параметрами

*std*::*wstring* query = L"INSERT INTO dbo.FirstTable (username, email, age) VALUES (?, ?, ";

query += (age == 0 ? L"NULL" : *std*::*to\_wstring*(age)) + L")";

*SQLPrepareW*(sqlStmtHandle, (*SQLWCHAR*\*)query.*c\_str*(), *SQL\_NTS*);

// Привязка параметров (username и email)

*SQLBindParameter*(sqlStmtHandle, 1, *SQL\_PARAM\_INPUT*, *SQL\_C\_WCHAR*, *SQL\_VARCHAR*, 100, 0, (*SQLPOINTER*)username.*c\_str*(), 0, *NULL*);

*SQLBindParameter*(sqlStmtHandle, 2, *SQL\_PARAM\_INPUT*, *SQL\_C\_WCHAR*, *SQL\_VARCHAR*, 100, 0, (*SQLPOINTER*)email.*c\_str*(), 0, *NULL*);

*SQLExecute*(sqlStmtHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_STMT*, sqlStmtHandle);

*std*::*wcout* << L"Item added successfully!" << *std*::*endl*;

}

// Удаление записи по ID

void deleteItem(*SQLHANDLE* sqlConnHandle) {

printTable(sqlConnHandle);

int id;

*std*::*wcout* << L"Enter ID to delete: ";

*std*::*wcin* >> id;

*std*::*wcin*.*ignore*();

*SQLHANDLE* sqlStmtHandle;

*SQLAllocHandle*(*SQL\_HANDLE\_STMT*, sqlConnHandle, &sqlStmtHandle);

// SQL-запрос на удаление

*std*::*wstring* query = L"DELETE FROM dbo.FirstTable WHERE Id = " + *std*::*to\_wstring*(id);

*SQLExecDirectW*(sqlStmtHandle, (*SQLWCHAR*\*)query.*c\_str*(), *SQL\_NTS*);

*SQLFreeHandle*(*SQL\_HANDLE\_STMT*, sqlStmtHandle);

*std*::*wcout* << L"Item deleted successfully!" << *std*::*endl*;

}

// Сортировка записей по возрасту

void sortByAge(*SQLHANDLE* sqlConnHandle) {

*SQLHANDLE* sqlStmtHandle;

*SQLAllocHandle*(*SQL\_HANDLE\_STMT*, sqlConnHandle, &sqlStmtHandle);

// SQL-запрос с сортировкой

*SQLWCHAR*\* sqlQuery = (*SQLWCHAR*\*)L"SELECT Id, username, email, age, register\_date FROM dbo.FirstTable ORDER BY age";

*SQLExecDirectW*(sqlStmtHandle, sqlQuery, *SQL\_NTS*);

*std*::*wcout* << L"\nTable sorted by age:\n";

*std*::*wcout* << *std*::*left*

<< *std*::*setw*(5) << L"ID"

<< *std*::*setw*(20) << L"Username"

<< *std*::*setw*(25) << L"Email"

<< *std*::*setw*(10) << L"Age"

<< *std*::*setw*(20) << L"Register Date"

<< *std*::*endl*;

*std*::*wcout* << *std*::*setfill*(L'-') << *std*::*setw*(80) << L"" << *std*::*setfill*(L' ') << *std*::*endl*;

// Вывод отсортированных строк

while (*SQLFetch*(sqlStmtHandle) == *SQL\_SUCCESS*) {

*SQLINTEGER* id, age;

*SQLWCHAR* username[101], email[101];

*SQL\_TIMESTAMP\_STRUCT* registerDate;

*SQLLEN* usernameLen, emailLen, ageLen, registerDateLen;

*SQLGetData*(sqlStmtHandle, 1, *SQL\_C\_LONG*, &id, 0, *NULL*);

*SQLGetData*(sqlStmtHandle, 2, *SQL\_C\_WCHAR*, username, sizeof(username), &usernameLen);

*SQLGetData*(sqlStmtHandle, 3, *SQL\_C\_WCHAR*, email, sizeof(email), &emailLen);

*SQLGetData*(sqlStmtHandle, 4, *SQL\_C\_LONG*, &age, 0, &ageLen);

*SQLGetData*(sqlStmtHandle, 5, *SQL\_C\_TYPE\_TIMESTAMP*, &registerDate, sizeof(registerDate), &registerDateLen);

wchar\_t dateBuffer[20];

*swprintf*(dateBuffer, 20, L"%04d-%02d-%02d %02d:%02d:%02d",

registerDate.*year*, registerDate.*month*, registerDate.*day*,

registerDate.*hour*, registerDate.*minute*, registerDate.*second*);

*std*::*wcout* << *std*::*left*

<< *std*::*setw*(5) << id

<< *std*::*setw*(20) << (usernameLen == *SQL\_NULL\_DATA* ? L"NULL" : username)

<< *std*::*setw*(25) << (emailLen == *SQL\_NULL\_DATA* ? L"NULL" : email)

<< *std*::*setw*(10) << (ageLen == *SQL\_NULL\_DATA* ? L"NULL" : *std*::*to\_wstring*(age))

<< *std*::*setw*(20) << (registerDateLen == *SQL\_NULL\_DATA* ? L"NULL" : dateBuffer)

<< *std*::*endl*;

}

*SQLFreeHandle*(*SQL\_HANDLE\_STMT*, sqlStmtHandle);

}

// Главное текстовое меню

void showMenu(*SQLHANDLE* sqlConnHandle) {

int choice;

do {

*std*::*wcout* << L"\nMenu:\n";

*std*::*wcout* << L"1. Print table\n";

*std*::*wcout* << L"2. Add item\n";

*std*::*wcout* << L"3. Delete item\n";

*std*::*wcout* << L"4. Sort by age\n";

*std*::*wcout* << L"5. Exit\n";

*std*::*wcout* << L"Enter your choice: ";

*std*::*wcin* >> choice;

*std*::*wcin*.*ignore*();

switch (choice) {

case 1:

printTable(sqlConnHandle);

break;

case 2:

addItem(sqlConnHandle);

break;

case 3:

deleteItem(sqlConnHandle);

break;

case 4:

sortByAge(sqlConnHandle);

break;

case 5:

*std*::*wcout* << L"Exiting...\n";

break;

default:

*std*::*wcout* << L"Invalid choice!\n";

}

if (choice != 5) {

*std*::*wcout* << L"\nPress any key to continue...";

*\_getch*();

}

} while (choice != 5);

}

// Точка входа в программу

int *main*() {

*SQLHANDLE* sqlConnHandle = connectToDatabase();

if (!sqlConnHandle) {

*std*::*wcerr* << L"Failed to connect to database. Exiting...\n";

return 1;

}

showMenu(sqlConnHandle);

// Завершение работы и освобождение ресурсов

*SQLDisconnect*(sqlConnHandle);

*SQLFreeHandle*(*SQL\_HANDLE\_DBC*, sqlConnHandle);

return 0;

}

Пример работы программы:

Программа выполняет следующие функции:

1. Выводит всю таблицу

2. Добавляет новый элемент в таблицу

3. Удаляет элемент из таблицы

4. Сортирует таблицу по возрасту

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

### **1.2.3 Типовой пример 3**

Для начала работы с PostgreSQL нам необходимо установить pgAdmin 4 – ведущий инструмент управления с открытым исходным кодом для Postgres. После чего регистрируемся там, после чего создаем нашу новую базу данных. Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.

Выбираем вкладку **Таблицы** и создаем новую таблицуИзображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Переходим в окно запросов и пишем следующий код, который запускаем по блокам.

CREATE TABLE IF NOT EXISTS zxc(

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT *NULL*,

password VARCHAR(100) NOT *NULL*,

email VARCHAR(255) UNIQUE NOT *NULL*,

age INTEGER

);

INSERT INTO zxc(name, password, email, age) VALUES

('abc1', 'ivan123', 'ivan.petrov@example.com', 28),

('abc2', 'maria456', 'maria.sidorova@example.com', 32),

('abc3', 'alex789', 'alex.ivanov@example.com', 25),

('abc4', 'elena101', 'elena.kuznetsova@example.com', 30),

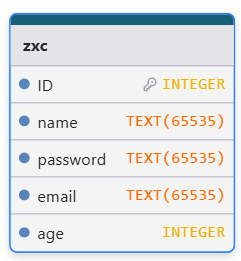
('abc5', 'dmitri202', 'dmitri.smirnov@example.com', 40);

SELECT\* FROM zxc ORDER BY id;

Получается следующая таблица:

Изображение выглядит как текст, Шрифт, число, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.



Пишем следующий код:

#include <iostream>

#include <string>

#include <libpq-fe.h>

#include <iomanip>

// Проверка успешности соединения с базой данных

void checkConnection(*PGconn*\* conn) {

if (PQstatus(conn) != CONNECTION\_OK) {

*std*::*cerr* << "Connection to database failed: " << PQerrorMessage(conn) << *std*::*endl*;

PQfinish(conn);

*exit*(1);

}

}

// Выполнение SQL-запроса без вывода результата (например, INSERT, UPDATE)

void executeSQL(*PGconn*\* conn, const *std*::*string*& sql, const *std*::*string*& successMsg) {

*PGresult*\* res = PQexec(conn, sql.*c\_str*());

if (PQresultStatus(res) != PGRES\_COMMAND\_OK) {

*std*::*cerr* << "Error executing SQL: " << PQerrorMessage(conn) << *std*::*endl*;

PQclear(res);

return;

}

*std*::*cout* << successMsg << *std*::*endl*;

PQclear(res);

}

// Вывод всех записей таблицы zxc с форматированием

void printTable(*PGconn*\* conn) {

*PGresult*\* res = PQexec(conn, "SELECT \* FROM zxc ORDER BY id");

if (PQresultStatus(res) != PGRES\_TUPLES\_OK) {

*std*::*cerr* << "Error retrieving data: " << PQerrorMessage(conn) << *std*::*endl*;

PQclear(res);

return;

}

int rows = PQntuples(res);

// Установим ширину столбцов для форматирования вывода

const int id\_width = 5;

const int name\_width = 15;

const int password\_width = 15;

const int email\_width = 30;

const int age\_width = 5;

// Заголовки таблицы

*std*::*cout* << *std*::*left* << *std*::*setw*(id\_width) << "id";

*std*::*cout* << *std*::*left* << *std*::*setw*(name\_width) << "name";

*std*::*cout* << *std*::*left* << *std*::*setw*(password\_width) << "password";

*std*::*cout* << *std*::*left* << *std*::*setw*(email\_width) << "email";

*std*::*cout* << *std*::*left* << *std*::*setw*(age\_width) << "age" << *std*::*endl*;

// Вывод данных построчно

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

*std*::*cout* << *std*::*left* << *std*::*setw*(id\_width) << PQgetvalue(res, i, 0);

*std*::*cout* << *std*::*left* << *std*::*setw*(name\_width) << PQgetvalue(res, i, 1);

*std*::*cout* << *std*::*left* << *std*::*setw*(password\_width) << PQgetvalue(res, i, 2);

*std*::*cout* << *std*::*left* << *std*::*setw*(email\_width) << PQgetvalue(res, i, 3);

*std*::*cout* << *std*::*left* << *std*::*setw*(age\_width) << PQgetvalue(res, i, 4) << *std*::*endl*;

}

PQclear(res);

}

// Добавление новой записи в таблицу zxc

void addRecord(*PGconn*\* conn) {

*std*::*string* name, password, email;

int age;

// Ввод данных пользователем

*std*::*cout* << "Enter name: ";

*std*::*cin* >> name;

*std*::*cout* << "Enter password: ";

*std*::*cin* >> password;

*std*::*cout* << "Enter email: ";

*std*::*cin* >> email;

*std*::*cout* << "Enter age: ";

*std*::*cin* >> age;

// Формирование SQL-запроса INSERT

*std*::*string* sql = "INSERT INTO zxc (name, password, email, age) VALUES ('" +

name + "', '" + password + "', '" + email + "', " + *std*::*to\_string*(age) + ")";

executeSQL(conn, sql, "Record added successfully!");

}

// Вывод таблицы, отсортированной по возрасту

void sortByAge(*PGconn*\* conn) {

*PGresult*\* res = PQexec(conn, "SELECT \* FROM zxc ORDER BY age");

if (PQresultStatus(res) != PGRES\_TUPLES\_OK) {

*std*::*cerr* << "Error retrieving data: " << PQerrorMessage(conn) << *std*::*endl*;

PQclear(res);

return;

}

int rows = PQntuples(res);

// Повтор того же форматирования, что и в printTable

const int id\_width = 5;

const int name\_width = 15;

const int password\_width = 15;

const int email\_width = 30;

const int age\_width = 5;

// Заголовки

*std*::*cout* << *std*::*left* << *std*::*setw*(id\_width) << "id";

*std*::*cout* << *std*::*left* << *std*::*setw*(name\_width) << "name";

*std*::*cout* << *std*::*left* << *std*::*setw*(password\_width) << "password";

*std*::*cout* << *std*::*left* << *std*::*setw*(email\_width) << "email";

*std*::*cout* << *std*::*left* << *std*::*setw*(age\_width) << "age" << *std*::*endl*;

// Данные

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

*std*::*cout* << *std*::*left* << *std*::*setw*(id\_width) << PQgetvalue(res, i, 0);

*std*::*cout* << *std*::*left* << *std*::*setw*(name\_width) << PQgetvalue(res, i, 1);

*std*::*cout* << *std*::*left* << *std*::*setw*(password\_width) << PQgetvalue(res, i, 2);

*std*::*cout* << *std*::*left* << *std*::*setw*(email\_width) << PQgetvalue(res, i, 3);

*std*::*cout* << *std*::*left* << *std*::*setw*(age\_width) << PQgetvalue(res, i, 4) << *std*::*endl*;

}

PQclear(res);

}

int *main*() {

// Установка параметров подключения к PostgreSQL

*PGconn*\* conn = PQconnectdb(

"host=localhost "

"port=5432 "

"dbname=test\_db "

"user=postgres "

"password=your\_password "

"connect\_timeout=5"

);

checkConnection(conn); // Проверка соединения

*std*::*cout* << "Connected to database successfully!" << *std*::*endl*;

int choice;

do {

// Меню

*std*::*cout* << "\nMenu:\n";

*std*::*cout* << "1. Show table\n";

*std*::*cout* << "2. Add new record\n";

*std*::*cout* << "3. Sort by age\n";

*std*::*cout* << "0. Exit\n";

*std*::*cout* << "Enter your choice: ";

*std*::*cin* >> choice;

// Обработка выбора пользователя

switch (choice) {

case 1:

printTable(conn);

break;

case 2:

addRecord(conn);

break;

case 3:

sortByAge(conn);

break;

case 0:

*std*::*cout* << "Exiting...\n";

break;

default:

*std*::*cout* << "Invalid choice. Try again.\n";

}

} while (choice != 0);

// Завершение соединения

PQfinish(conn);

return 0;

}

Результаты работы программы:

Изображение выглядит как текст, снимок экрана, меню

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, меню, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

# **4. Задачи по теме**

|  |  |
| --- | --- |
| **№ варианта** | **Условие задачи** |
| 1 | Создайте базу данных StudentsDB.accdb в Microsoft Access с таблицей Students, имеющей следующую структуру:   * ID — уникальный идентификатор студента. * FullName— полное имя студента. * AverageScore— средний балл (от 0 до 100). * Major— специальность (например, "Computer Science"). * EnrollmentDate — дата поступления.   Добавьте записи по своему усмотрению.  **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Student> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить всех студентов. * Сортировать студентов по среднему баллу (по убыванию) и сохранить в базу. * Выход.  1. После сортировки обновите базу данных и выведите результат. |
| 2 | Создайте базу данных LibraryDB с таблицей Books в Microsoft Visual Studio используя SQL, имеющей следующую структуру:   * ID – уникальный идентификатор книги * Title – название книги * Author – автор книги * YearPublished – год выпуска книги * Genre – жанр книги * Available (Yes/No) – доступность к получению книги   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Book> для временного хранения  3. Реализуйте программу с меню:   * Показать все книги * Поиск по автору * Взять книгу (изменить статус Available) * Выход |
| 3 | Создайте базу данных CarsDB с таблицей Cars используя PostgreSQL, имеющей следующую структуру:   * ID – уникальный идентификатор машины * Brand – марка * Model – модель * Year – год выпуска * Mileage – пробег * Price - цена   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Car> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все автомобили * Сортировать по цене (по убыванию) и сохранить в базу * Изменить информацию о машине * Выход |
| 4 | Создайте базу данных LibraryDB.accdb с таблицей Books, имеющей следующую структуру:   * ID — уникальный идентификатор книги. * Title — название книги. * Author — автор книги. * Genre — жанр (например, "Fiction", "Science"). * YearPublished — год публикации.   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Book> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все книги. * Фильтровать книги по жанру (пользователь вводит жанр, например, "Fantasy"). * Выход.  1. После фильтрации выведите только те книги, которые соответствуют введённому жанру. |
| 5 | Создайте базу данных HRDepartment с таблицей в Microsoft Visual Studio используя SQL, Employees имеющей следующую структуру:   * EmployeeID (PK) – уникальный идентификатор сотрудника * Name – имя сотрудника * Position – должность сотрудника * Department – отдел в котором работает сотрудник * HireDate – дата вступления на должность * Salary – зарплата сотрудника   **Задача:**   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Product> для временного хранения. 3. Реализуйте программу с меню:  * Список всех сотрудников * Фильтр по отделу * Повысить зарплату (на 10% выбранному отделу или сотруднику) * Выход |
| 6 | Создайте базу данных CoursesDB с таблицей Courses используя PostgreSQL, имеющей следующую структуру:   * ID – уникальный идентификатор * CourseName – название курса * Instructor – преподаватель * DurationWeeks – длительность в неделях * Credits – кол-во зачетных единиц * StartDate – дата начала курса   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Course> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все курсы * Сортировать по длительности курса (по убыванию) и сохранить в базу * Изменить длительность курса * Выход |
| 7 | Создайте базу данных StoreDB.accdb с таблицей Products, имеющей следующую структуру:   * ID — уникальный идентификатор продукта. * ProductName — название продукта. * Price — цена продукта (в долларах). * Category — категория (например, "Electronics"). * InStock — наличие на складе (Yes = 1, No = 0).   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Product> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все продукты. * Найти продукты в заданном ценовом диапазоне (пользователь вводит минимальную и максимальную цену). * Выход.  1. После поиска выведите только те продукты, цена которых попадает в указанный диапазон. |
| 8 | Создайте базу данных Inventory с таблицей Products в Microsoft Visual Studio используя SQL, имеющей следующую структуру   * ID – уникальный идентификатор продукта * ProductName – название продукта * Category – категория продукта * Price – цена продукта * QuantityInStock – количество на складе * LastRestockDate – последняя дата переучета   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Product> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все продукты. * Найти продукты в заданном ценовом диапазоне (пользователь вводит минимальную и максимальную цену). * Выход.   После поиска выведите только те продукты, цена которых попадает в указанный диапазон. |
| 9 | Создайте базу данных OrdersDB с таблицей Orders используя PostgreSQL, имеющей следующую структуру:   * ID – уникальный идентификатор * CustomerName – имя клиента * OrderDate – дата заказа * Product – продукт * Quantity – количество * TotalPrice - итоговая цена   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Order> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все заказы * Сортировать по итоговой цене (по убыванию) и сохранить в базу * Добавить заказ * Изменить количество в заказе * Выход |
| 10 | Создайте базу данных EmployeesDB.accdb с таблицей Employees, имеющей следующую структуру:   * ID — уникальный идентификатор сотрудника. * Name — имя сотрудника. * Salary — зарплата (в долларах). * Department — отдел (например, "IT"). * HireDate — дата приёма на работу.   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Employee> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить всех сотрудников. * Сортировать сотрудников сначала по стажу (от большего к меньшему, на основе HireDate), а затем по зарплате (по убыванию), и сохранить в базу. * Выход.  1. После сортировки обновите базу данных и выведите результат. |
| 11 | Создайте базу данных HotelDB с таблицей Rooms в Microsoft Visual Studio используя SQL, имеющей следующую структуру:   * ID – уникальный идентификатор номера * RoomType – тип номера * PricePerNight – цена за ночь * Capacity – количество людей в номере * IsOccupied – занятость номера * GuestName – имя гостя (NULL если свободно)   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Room> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить свободные номера. * Забронировать номер * Освободить номер * Выход. |
| 12 | Создайте базу данных FlightsDB с таблицей Flights используя PostgreSQL, имеющей следующую структуру:   * ID – уникальный идентификатор * FlightNumber – номер рейса * DepartureCity – город вылета * ArrivalCity – город прибытия * DepartureTime – время вылета * DurationMinutes – продолжительность (в минутах)   **Задача:**  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Flight> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все полеты * Сортировать по продолжительности (по убыванию) и сохранить в базу * Добавить рейс * Удалить рейс * Выход |
| 13 | Создайте базу данных MoviesDB.accdb с таблицей Movies, имеющей следующую структуру:   * ID — уникальный идентификатор фильма. * Title — название фильма. * Director — режиссёр. * Genre — жанр (например, "Action"). * Rating — рейтинг (от 0 до 10).   **Задача**:   1. Подключитесь к базе данных через ODBC. 2. Загружайте данные в std::vector<Movie> для временного хранения. 3. Реализуйте программу с меню:  * Отобразить все фильмы. * Подсчитать средний рейтинг фильмов для заданного жанра (пользователь вводит жанр, например, "Sci-Fi"). * Выход.  1. После ввода жанра выведите средний рейтинг фильмов этого жанра. |
| 14 | Создайте базу данных ClinicDB с таблицей Patients в Microsoft Visual Studio используя SQL, имеющей следующую структуру:   * ID – уникальный идентификатор пациента * Name – имя пациента * BirthDate – дата рождения пациента * BloodType – тип крови пациента * LastVisitDate – последнее посещение клиники * DoctorNotes – заметки врача о пациенте   **Задача**:  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Patient> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить всех пациентов * Поиск по группе крови * Добавить запись к врачу * Выход. |
| 15 | Создайте базу данных InvoicesDB с таблицей Invoices используя PostgreSQL, имеющей следующую структуру:   * ID – уникальный идентификатор * InvoiceNumber – номер счета * ClientName – имя клиента * IssueDate – дата выпуска счета * Amount – сумма * Status – Статус («Оплачен», «Ожидает», «Отклонен»   **Задача**:  1. Подключитесь к базе данных через ODBC.  2. Загружайте данные в std::vector<Invoice> для временного хранения.  3. Реализуйте программу с меню:   * Отобразить все счета * Сортировать по сумме (по убыванию) и сохранить в базу * Добавить счет * Удалить счет * Изменить сумму * Изменить статус * Выход |

# **5. Дополнительные задачи**

**5.1 Управление библиотечной системой с анализом выдачи книг и штрафами**  
Создайте базу данных LibrarySystemDB.accdb в Microsoft Access с двумя таблицами: Books (книги) и BorrowRecords (записи о выдаче книг).

* Таблица Books:
  + BookID (AutoNumber) — уникальный идентификатор книги.
  + Title (Text, 150) — название книги.
  + Author (Text, 100) — автор книги.
  + Genre (Text, 50) — жанр (например, "Fiction", "Science").
  + PublicationYear (Number, Long Integer) — год издания.
  + AvailableCopies (Number, Integer) — количество доступных экземпляров.
* Таблица BorrowRecords:
  + RecordID (AutoNumber) — уникальный идентификатор записи.
  + BookID (Number, Long Integer) — ID книги (связь с Books).
  + UserName (Text, 100) — имя пользователя, взявшего книгу.
  + BorrowDate (Date/Time) — дата выдачи книги.
  + ReturnDate (Date/Time) — дата возврата (может быть пустой, если книга не возвращена).
  + Fine (Number, Double) — штраф за просрочку (по умолчанию 0).

**Начальные данные (пример)**:

* Таблица Books:
  + BookID: 1, Title: "The Hobbit", Author: "J.R.R. Tolkien", Genre: "Fantasy", PublicationYear: 1937, AvailableCopies: 3
  + BookID: 2, Title: "1984", Author: "George Orwell", Genre: "Dystopia", PublicationYear: 1949, AvailableCopies: 2
  + BookID: 3, Title: "A Brief History of Time", Author: "Stephen Hawking", Genre: "Science", PublicationYear: 1988, AvailableCopies: 1
* Таблица BorrowRecords:
  + RecordID: 1, BookID: 1, UserName: "Alice Smith", BorrowDate: 2025-04-01, ReturnDate: 2025-04-15, Fine: 0
  + RecordID: 2, BookID: 2, UserName: "Bob Johnson", BorrowDate: 2025-04-10, ReturnDate: NULL, Fine: 0
  + RecordID: 3, BookID: 1, UserName: "Clara Lee", BorrowDate: 2025-03-20, ReturnDate: NULL, Fine: 0

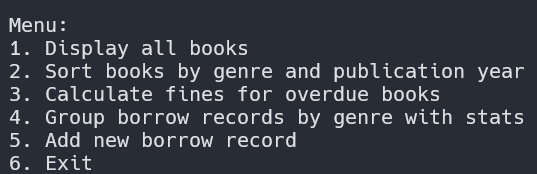
**Задача**:

1. Подключитесь к базе данных LibrarySystemDB.accdb из Visual Studio, используя ODBC.
2. Загружайте данные из обеих таблиц в std::vector для временного хранения (std::vector<Book> для Books и std::vector<BorrowRecord> для BorrowRecords).
3. Реализуйте программу с меню, включающим следующие опции:

* Отобразить все книги с количеством доступных экземпляров.
* Сортировать книги по жанру (по алфавиту) и году издания (по убыванию), с сохранением результата в базу.
* Найти просроченные книги (срок возврата — 14 дней с даты выдачи) и начислить штраф (0.5 у.е. за каждый день просрочки). Обновить Fine в таблице BorrowRecords.
* Группировать записи о выдаче по жанру книги и подсчитать общее количество выданных книг и сумму штрафов для каждого жанра.
* Добавить новую запись о выдаче книги, обновив AvailableCopies в таблице Books.
* Выход.

1. Используйте текущую дату для расчёта просрочек.
2. После выполнения операций сортировки, начисления штрафов или добавления записи синхронизируйте данные с базой.

**Ожидаемый результат**



**5.2 Управление складом электроники с анализом продаж и автоматическим заказом**

**Описание:**

Microsoft Access:

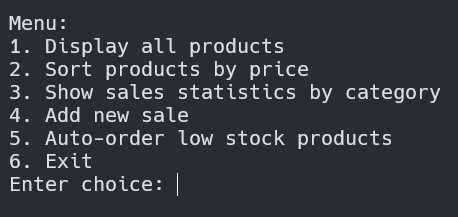
Программа должна:

1. Хранить данные о товарах (ID, название, категория, цена, количество на складе) в таблице Products.
2. Хранить данные о продажах (ID продажи, ID товара, дата продажи, количество, общая стоимость) в таблице Sales.
3. Иметь меню с следующими функциями:
   * Показать все товары.
   * Отсортировать товары по цене и сохранить в базу.
   * Показать статистику продаж по категориям (сколько продано товаров каждой категории и общая выручка).
   * Добавить новую продажу (уменьшает количество товара на складе).
   * Автоматически заказать товары, если их осталось меньше 5 штук.
4. Использовать std::set для хранения уникальных категорий товаров.
5. Проверять корректность даты продажи с помощью функции validateDate.

**Структура базы данных:**

* Таблица Products:
  + ProductID (целое, первичный ключ)
  + Name (текст, до 150 символов)
  + Category (текст, до 50 символов)
  + Price (дробное)
  + Stock (целое)
* Таблица Sales:
  + SaleID (целое, первичный ключ)
  + ProductID (целое)
  + SaleDate (текст в формате YYYY-MM-DD)
  + Quantity (целое)
  + TotalCost (дробное)

**Ожидаемый результат:**

****

**5.3 Система учета пациентов в поликлинике**

Создайте базу данных PatientsDB в pgAdmin4 с двумя таблицами: Patients (пациенты) и Visits (посещения).

Программа должна:

1. Хранить данные о пациентах (ID, ФИО, дата рождения, пол, номер полиса) в таблице Patients

2. Хранить данные о посещениях (ID посещения, ID пациента, дата, врач, диагноз, стоимость) в таблице Visits

3. Иметь меню:

* Показать всех пациентов
* Найти пациента по диагнозу
* Показать статистику посещений по врачам
* Добавить новое посещение
* Назначить повторный прием через 2 недели для хронических заболеваний

**Структура базы данных:**

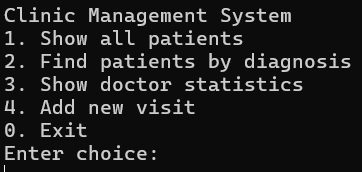
1. Таблица Patiens:

* PatiendID (целое, РК)
* FullName (текст 150)
* BithDate (текст YYYY-MM-DD)
* Gender (текст 10)
* InsuranceNumber (текст 20)

1. Visits:

* VisitID (целое, РК)
* PatientID (целок, FK)
* VisitDate (текст YYYY-MM-DD)
* Doctor (текст 100)
* Diagnosis (текст 200)
* Cost (дробное)

**Ожидаемый результат:**



**5.4 Система учета инвентаря в фитнес-клубе**

Создайте базу данных ClubDB в SQL Microsoft Visual Studio с двумя таблицами: Equipment (оборудование) и Maintenance (обслуживание).

Программа должна:

1. Хранить данные об оборудовании (ID, название, тип, дата покупки, срок службы) в таблице Equipment

2. Хранить данные об обслуживании (ID записи, ID оборудования, дата, тип обслуживания) в таблице Maintenance

3. Иметь меню:

* Показать все оборудование с остатком срока службы
* Сортировать по дате покупки
* Показать статистику обслуживания по типам оборудования
* Добавить запись об обслуживании
* Автоматически помечать оборудование для списания

**Структура базы данных:**

1. Таблица Equipment:

* EquipmentID (целое, РК)
* Name (текст 150)
* Type (текст 50)
* PurchaseDate (текст YYYY-MM-DD)
* Lifespan (целое, лет)

2. Visits:

* MaintenanceID (целое, РК)
* EquipmentID(целок, FK)
* Date (текст YYYY-MM-DD)
* Type (текст 100)

**Ожидаемый результат:**

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

**5.5 Система учета задач в IT-компании**

Создайте базу данных TasksDB в SQL Microsoft Visual Studio с двумя таблицами: Projects (проекты) и Tasks (задачи).

Программа должна:

1. Хранить данные об оборудовании (ID, название, клиент, срок сдачи) в таблице Projects

2. Хранить данные об обслуживании (ID задачи, ID проекта, описание, статус, приоритет) в таблице Tasks

3. Иметь меню:

* Показать все проекты с количеством задач
* Сортировать задачи по приоритету
* Показать статистику по статусам задач
* Добавить новую задачу
* Автоматически помечать просроченные проекты

**Структура базы данных:**

1. Таблица Projects:

* ProjectID (целое, РК)
* Name (текст 150)
* Client (текст 100)
* Deadline (текст YYYY-MM-DD)

2. Visits:

* TaskID (целое, РК)
* ProjectID (целок, FK)
* Description (текст 300)
* Status (текст 20)
* Priority (целое)

**Ожидаемый результат:**

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

# **6. Контрольные вопросы**

1. Зачем нужны базы данных?

2. Что такое ODBC и зачем он нужен?

3. Какой SQL-запрос используется для добавления данных?

4. Чем базы данных удобнее обычных текстовых файлов или блокнотов?

5. Какую роль играет драйвер ODBC при работе с базой данных?

6. Какой SQL-запрос нужен, чтобы извлечь данные из таблицы?

7. Какую структуру имеют базы данных?

8. Можно ли через ODBC подключиться к любой базе данных?

9. Какой SQL-запрос используется для изменения существующей записи?

10. Что такое таблица в базе данных?

11. Какую библиотеку или технологию используют программы на C++ для работы с ODBC?

12. Какой SQL-запрос удаляет запись из таблицы?

13. Что такое запись (строка) в таблице базы данных?

14. Для чего используется std::vector в программе, работающей с БД?

15. Почему иногда данные сначала загружаются в оперативную память, а не сразу обрабатываются в базе?

# **7. Решения к задачам повышенной сложности**

|  |  |
| --- | --- |
| **№** | **Номер задачи** |
| 1 | **5.1 Управление библиотечной системой с анализом выдачи книг и штрафами**  Создайте базу данных LibrarySystemDB.accdb в Microsoft Access с двумя таблицами: Books (книги) и BorrowRecords (записи о выдаче книг).   |  | | --- | | #include <windows.h> // Подключаем всё нужное, чтобы работать с компьютером и базой данных  #include <sqlext.h>  #include <iostream>  #include <vector>  #include <algorithm>  #include <string>  #include <iomanip>  #include <map>  // Проверяем дату (например, 2025-05-12), чтобы она была правильной  bool validateDate(const std::wstring& date) {  if (date.length() != 10 || date[4] != L'-' || date[7] != L'-') { // Смотрим, есть ли 10 символов и дефисы  return false;  }  try {  int year = std::stoi(date.substr(0, 4));  int month = std::stoi(date.substr(5, 2));  int day = std::stoi(date.substr(8, 2));  if (month < 1 || month > 12 || day < 1 || day > 31 || year < 1900 || year > 9999) return false;  if (month == 2 && day > 29) return false;  if ((month == 4 || month == 6 || month == 9 || month == 11) && day > 30) return false;  return true;  }  catch (...) {  return false;  }  }  // Показываем, если база данных сломалась  void showSQLError(SQLSMALLINT handleType, SQLHANDLE handle) {  SQLWCHAR sqlState[1024];  SQLWCHAR message[2048];  SQLINTEGER nativeError;  SQLSMALLINT textLength;  SQLRETURN ret = SQLGetDiagRecW(handleType, handle, 1, sqlState, &nativeError, message, 2048, &textLength);  if (ret == SQL\_SUCCESS || ret == SQL\_SUCCESS\_WITH\_INFO) {  std::wcout << L"ODBC Error: " << message << L" [State: " << sqlState << L"]\n";  }  }  // Это как карточка книги в библиотеке  struct Book {  SQLINTEGER bookID;  std::wstring title;  std::wstring author;  std::wstring genre;  SQLINTEGER publicationYear;  SQLINTEGER availableCopies;  };  // Это как запись, кто взял книгу  struct BorrowRecord {  SQLINTEGER recordID;  SQLINTEGER bookID;  std::wstring userName;  std::wstring borrowDate;  std::wstring returnDate;  SQLDOUBLE fine;  };  // Подключаемся к базе, где хранятся все книги  bool connectToDB(SQLHENV& hEnv, SQLHDBC& hDbc) {  SQLAllocHandle(SQL\_HANDLE\_ENV, SQL\_NULL\_HANDLE, &hEnv);  SQLSetEnvAttr(hEnv, SQL\_ATTR\_ODBC\_VERSION, (void\*)SQL\_OV\_ODBC3, 0);  SQLAllocHandle(SQL\_HANDLE\_DBC, hEnv, &hDbc);  SQLWCHAR connStr[] = L"Driver={Microsoft Access Driver (\*.mdb, \*.accdb)};DBQ=D:\\1\_course\\2\_semester\\ОАИП(Основы Алгоритмизации и Программирования)\\лабы\_белодед\\код ко второй\\для бд\\SoftwareDB.accdb;";  SQLWCHAR outConnStr[1024];  SQLSMALLINT outConnStrLen;  SQLRETURN ret = SQLDriverConnectW(hDbc, NULL, connStr, SQL\_NTS, outConnStr, 1024, &outConnStrLen, SQL\_DRIVER\_COMPLETE);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_DBC, hDbc);  return false;  }  return true;  }  // Читаем все книги из базы в список  std::vector<Book> readBooks(SQLHDBC hDbc) {  std::vector<Book> books;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT BookID, Title, Author, Genre, PublicationYear, AvailableCopies FROM Books;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  SQLINTEGER bookID, publicationYear, availableCopies;  SQLWCHAR title[151], author[101], genre[51];  SQLLEN bookIDLen, titleLen, authorLen, genreLen, pubYearLen, copiesLen;  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  Book book = { 0 };  SQLGetData(hStmt, 1, SQL\_C\_LONG, &bookID, 0, &bookIDLen);  SQLGetData(hStmt, 2, SQL\_C\_WCHAR, title, sizeof(title), &titleLen);  SQLGetData(hStmt, 3, SQL\_C\_WCHAR, author, sizeof(author), &authorLen);  SQLGetData(hStmt, 4, SQL\_C\_WCHAR, genre, sizeof(genre), &genreLen);  SQLGetData(hStmt, 5, SQL\_C\_LONG, &publicationYear, 0, &pubYearLen);  SQLGetData(hStmt, 6, SQL\_C\_LONG, &availableCopies, 0, &copiesLen);  book.bookID = bookID;  book.title = title;  book.author = author;  book.genre = genre;  book.publicationYear = publicationYear;  book.availableCopies = availableCopies;  books.push\_back(book);  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return books;  }  // Читаем все записи о том, кто взял книги  std::vector<BorrowRecord> readBorrowRecords(SQLHDBC hDbc) {  std::vector<BorrowRecord> records;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT RecordID, BookID, UserName, BorrowDate, ReturnDate, Fine FROM BorrowRecords;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  SQLINTEGER recordID, bookID;  SQLDOUBLE fine;  SQLWCHAR userName[101], borrowDate[50], returnDate[50];  SQLLEN recordIDLen, bookIDLen, userNameLen, borrowDateLen, returnDateLen, fineLen;  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  BorrowRecord record = { 0 };  SQLGetData(hStmt, 1, SQL\_C\_LONG, &recordID, 0, &recordIDLen);  SQLGetData(hStmt, 2, SQL\_C\_LONG, &bookID, 0, &bookIDLen);  SQLGetData(hStmt, 3, SQL\_C\_WCHAR, userName, sizeof(userName), &userNameLen);  SQLGetData(hStmt, 4, SQL\_C\_WCHAR, borrowDate, sizeof(borrowDate), &borrowDateLen);  SQLGetData(hStmt, 5, SQL\_C\_WCHAR, returnDate, sizeof(returnDate), &returnDateLen);  SQLGetData(hStmt, 6, SQL\_C\_DOUBLE, &fine, 0, &fineLen);  record.recordID = recordID;  record.bookID = bookID;  record.userName = userName;  record.borrowDate = borrowDate;  if (returnDateLen == SQL\_NULL\_DATA) {  record.returnDate = L"NULL";  }  else {  record.returnDate = returnDate;  }  record.fine = fine;  records.push\_back(record);  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return records;  }  // Показываем все книги на экране  void displayBooks(const std::vector<Book>& books) {  for (const auto& book : books) {  std::wcout << L"BookID: " << book.bookID << L", Title: " << book.title  << L", Author: " << book.author << L", Genre: " << book.genre  << L", PublicationYear: " << book.publicationYear  << L", AvailableCopies: " << book.availableCopies << L"\n";  }  }  // Показываем все записи о выдаче книг  void displayBorrowRecords(const std::vector<BorrowRecord>& records) {  for (const auto& record : records) {  std::wcout << L"RecordID: " << record.recordID << L", BookID: " << record.bookID  << L", UserName: " << record.userName << L", BorrowDate: " << record.borrowDate  << L", ReturnDate: " << record.returnDate << L", Fine: " << record.fine << L"\n";  }  }  // Сортируем книги по жанру и году, чтобы было удобнее искать  void sortBooksByGenreAndYear(SQLHDBC hDbc, std::vector<Book>& books) {  std::sort(books.begin(), books.end(), [](const Book& a, const Book& b) {  if (a.genre != b.genre) return a.genre < b.genre;  return a.publicationYear > b.publicationYear;  });  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"DELETE FROM Books;", SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error clearing Books table\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  for (const auto& book : books) {  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Books (BookID, Title, Author, Genre, PublicationYear, AvailableCopies) VALUES (%d, '%s', '%s', '%s', %d, %d);",  book.bookID, book.title.c\_str(), book.author.c\_str(), book.genre.c\_str(), book.publicationYear, book.availableCopies);  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error inserting book\n";  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  }  std::wcout << L"Books sorted and saved successfully.\n";  }  // Считаем, кто опоздал вернуть книгу, и добавляем штраф  void calculateFines(SQLHDBC hDbc, std::vector<BorrowRecord>& records) {  const double finePerDay = 0.5; // 50 копеек за день просрочки  const std::wstring currentDate = L"2025-05-12"; // Сегодняшняя дата  std::wcout << L"Overdue books found and fines updated:\n";  for (auto& record : records) {  if (record.returnDate != L"NULL") continue;  int borrowYear = std::stoi(record.borrowDate.substr(0, 4));  int borrowMonth = std::stoi(record.borrowDate.substr(5, 2));  int borrowDay = std::stoi(record.borrowDate.substr(8, 2));  int currentYear = std::stoi(currentDate.substr(0, 4));  int currentMonth = std::stoi(currentDate.substr(5, 2));  int currentDay = std::stoi(currentDate.substr(8, 2));  int daysBorrowed = (currentYear - borrowYear) \* 365 + (currentMonth - borrowMonth) \* 30 + (currentDay - borrowDay);  int overdueDays = daysBorrowed - 14;  if (overdueDays > 0) {  double fine = overdueDays \* finePerDay;  record.fine = fine;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLWCHAR query[256];  swprintf\_s(query, 256, L"UPDATE BorrowRecords SET Fine = %f WHERE RecordID = %d;", fine, record.recordID);  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  std::wcout << L"RecordID: " << record.recordID << L", BookID: " << record.bookID  << L", UserName: " << record.userName << L", BorrowDate: " << record.borrowDate  << L", ReturnDate: " << record.returnDate << L", Fine: " << fine  << L" (" << overdueDays << L" days overdue)\n";  }  }  }  // Считаем, сколько книг каждого жанра взяли и какие штрафы  void groupByGenre(SQLHDBC hDbc, const std::vector<BorrowRecord>& records, const std::vector<Book>& books) {  std::map<std::wstring, std::pair<int, double>> stats;  for (const auto& record : records) {  auto it = std::find\_if(books.begin(), books.end(), [&record](const Book& book) {  return book.bookID == record.bookID;  });  if (it != books.end()) {  stats[it->genre].first += 1;  stats[it->genre].second += record.fine;  }  }  std::wcout << L"Borrow records grouped by genre:\n";  for (const auto& pair : stats) {  std::wcout << pair.first << L": " << pair.second.first << L" books borrowed, Total Fine: " << pair.second.second << L"\n";  }  }  // Добавляем запись, что кто-то взял книгу  void addBorrowRecord(SQLHDBC hDbc, std::vector<BorrowRecord>& records, std::vector<Book>& books) {  int bookID;  std::wstring userName, borrowDate;  std::wcout << L"Enter BookID to borrow: ";  std::cin >> bookID;  std::cin.ignore();  std::wcout << L"Enter UserName: ";  std::getline(std::wcin, userName);  std::wcout << L"Enter BorrowDate (YYYY-MM-DD): ";  std::getline(std::wcin, borrowDate);  if (!validateDate(borrowDate)) {  std::wcout << L"Invalid date format. Please use YYYY-MM-DD with valid values (e.g., 2025-05-01).\n";  return;  }  auto bookIt = std::find\_if(books.begin(), books.end(), [bookID](const Book& book) {  return book.bookID == bookID;  });  if (bookIt == books.end() || bookIt->availableCopies <= 0) {  std::wcout << L"Book not available or not found.\n";  return;  }  bookIt->availableCopies -= 1;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLWCHAR query[256];  swprintf\_s(query, 256, L"UPDATE Books SET AvailableCopies = %d WHERE BookID = %d;", bookIt->availableCopies, bookID);  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  swprintf\_s(query, 256, L"INSERT INTO BorrowRecords (BookID, UserName, BorrowDate, ReturnDate, Fine) VALUES (%d, '%s', #%s#, NULL, 0);",  bookID, userName.c\_str(), borrowDate.c\_str());  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  std::wcout << L"Book borrowed successfully. Available copies updated.\n";  std::wcout << L"Updated Books:\n";  displayBooks(readBooks(hDbc));  }  // Главная часть: показываем меню и делаем, что выберет пользователь  int main() {  SetConsoleOutputCP(CP\_UTF8); // Делаем, чтобы русские буквы работали  SQLHENV hEnv = nullptr;  SQLHDBC hDbc = nullptr;  if (!connectToDB(hEnv, hDbc)) {  return 1;  }  int choice;  while (true) { // Показываем меню, пока не выберут "выход"  std::wcout << L"\nMenu:\n";  std::wcout << L"1. Display all books\n"; // 1: Показать книги  std::wcout << L"2. Sort books by genre and publication year\n"; // 2: Сортировать  std::wcout << L"3. Calculate fines for overdue books\n"; // 3: Считать штрафы  std::wcout << L"4. Group borrow records by genre with stats\n"; // 4: Посчитать по жанрам  std::wcout << L"5. Add new borrow record\n"; // 5: Взять книгу  std::wcout << L"6. Exit\n"; // 6: Выход  std::wcout << L"Enter choice: ";  std::cin >> choice;  std::cin.ignore();  if (choice == 6) break;  std::vector<Book> books = readBooks(hDbc);  std::vector<BorrowRecord> records = readBorrowRecords(hDbc);  switch (choice) {  case 1:  displayBooks(books);  break;  case 2:  sortBooksByGenreAndYear(hDbc, books);  std::wcout << L"Updated database:\n";  displayBooks(readBooks(hDbc));  break;  case 3:  calculateFines(hDbc, records);  break;  case 4:  groupByGenre(hDbc, records, books);  break;  case 5:  addBorrowRecord(hDbc, records, books);  break;  default:  std::wcout << L"Invalid choice.\n";  }  }  if (hDbc) { // Закрываем базу, когда закончили  SQLDisconnect(hDbc);  SQLFreeHandle(SQL\_HANDLE\_DBC, hDbc);  }  if (hEnv) SQLFreeHandle(SQL\_HANDLE\_ENV, hEnv);  return 0;  } | |
| 2 | **5.2 Управление складом электроники с анализом продаж и автоматическим заказом**  Microsoft Access:  Программа должна:   1. Хранить данные о товарах (ID, название, категория, цена, количество на складе) в таблице Products. 2. Хранить данные о продажах (ID продажи, ID товара, дата продажи, количество, общая стоимость) в таблице Sales.  |  | | --- | | #include <windows.h> // Подключаем библиотеку для работы с Windows  #include <sqlext.h> // Подключаем библиотеку для работы с базой данных через ODBC  #include <iostream> // Подключаем для вывода текста в консоль  #include <vector> // Подключаем для хранения списков товаров и продаж  #include <set> // Подключаем для уникальных категорий товаров  #include <algorithm> // Подключаем для сортировки  #include <string> // Подключаем для работы с текстом  // Проверяем, правильно ли введена дата (например, 2025-05-12)  bool validateDate(const std::wstring& date) {  if (date.length() != 10 || date[4] != L'-' || date[7] != L'-') {  return false; // Если длина не 10 или дефисы на неправильных местах, возвращаем false  }  try {  int year = std::stoi(date.substr(0, 4)); // Берем год (первые 4 символа)  int month = std::stoi(date.substr(5, 2)); // Берем месяц (5-6 символы)  int day = std::stoi(date.substr(8, 2)); // Берем день (8-9 символы)  if (month < 1 || month > 12 || day < 1 || day > 31 || year < 1900 || year > 9999) return false;  if (month == 2 && day > 29) return false; // Февраль не больше 29 дней  if ((month == 4 || month == 6 || month == 9 || month == 11) && day > 30) return false;  return true; // Если всё ок, возвращаем true  }  catch (...) {  return false; // Если ошибка при преобразовании, возвращаем false  }  }  // Показываем ошибки, если база данных не работает  void showSQLError(SQLSMALLINT handleType, SQLHANDLE handle) {  SQLWCHAR sqlState[1024];  SQLWCHAR message[2048];  SQLINTEGER nativeError;  SQLSMALLINT textLength;  SQLRETURN ret = SQLGetDiagRecW(handleType, handle, 1, sqlState, &nativeError, message, 2048, &textLength);  if (ret == SQL\_SUCCESS || ret == SQL\_SUCCESS\_WITH\_INFO) {  std::wcout << L"ODBC Error: " << message << L" [State: " << sqlState << L"]\n";  }  }  // Структура для хранения информации о товаре  struct Product {  SQLINTEGER productID; // Уникальный номер товара  std::wstring name; // Название товара  std::wstring category; // Категория (например, "Electronics")  SQLDOUBLE price; // Цена товара  SQLINTEGER stock; // Количество на складе  };  // Структура для хранения информации о продаже  struct Sale {  SQLINTEGER saleID; // Уникальный номер продажи  SQLINTEGER productID; // Номер товара, который продали  std::wstring saleDate; // Дата продажи  SQLINTEGER quantity; // Количество проданных единиц  SQLDOUBLE totalCost; // Общая стоимость продажи  };  // Подключаемся к базе данных (путь как в прошлой лабе, но для WarehouseDB)  bool connectToDB(SQLHENV& hEnv, SQLHDBC& hDbc) {  SQLAllocHandle(SQL\_HANDLE\_ENV, SQL\_NULL\_HANDLE, &hEnv); // Создаем окружение для базы  SQLSetEnvAttr(hEnv, SQL\_ATTR\_ODBC\_VERSION, (void\*)SQL\_OV\_ODBC3, 0); // Устанавливаем версию ODBC  SQLAllocHandle(SQL\_HANDLE\_DBC, hEnv, &hDbc); // Создаем соединение  // Путь как в прошлой лабе, заменяем SoftwareDB на WarehouseDB  SQLWCHAR connStr[] = L"Driver={Microsoft Access Driver (\*.mdb, \*.accdb)};DBQ=D:\\1\_course\\2\_semester\\ОАИП(Основы Алгоритмизации и Программирования)\\лабы\_белодед\\код ко второй\\для бд\\SoftwareDB.accdb;";  SQLWCHAR outConnStr[1024];  SQLSMALLINT outConnStrLen;  SQLRETURN ret = SQLDriverConnectW(hDbc, NULL, connStr, SQL\_NTS, outConnStr, 1024, &outConnStrLen, SQL\_DRIVER\_COMPLETE);  if (!SQL\_SUCCEEDED(ret)) { // Если подключение не удалось  showSQLError(SQL\_HANDLE\_DBC, hDbc); // Показываем ошибку  return false; // Возвращаем false  }  return true; // Успешное подключение  }  // Читаем все товары из базы данных  std::vector<Product> readProducts(SQLHDBC hDbc) {  std::vector<Product> products;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt); // Готовим запрос  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT ProductID, Name, Category, Price, Stock FROM Products;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  SQLINTEGER productID, stock;  SQLDOUBLE price;  SQLWCHAR name[151], category[51];  SQLLEN productIDLen, nameLen, categoryLen, priceLen, stockLen;  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  Product product = { 0 };  SQLGetData(hStmt, 1, SQL\_C\_LONG, &productID, 0, &productIDLen);  SQLGetData(hStmt, 2, SQL\_C\_WCHAR, name, sizeof(name), &nameLen);  SQLGetData(hStmt, 3, SQL\_C\_WCHAR, category, sizeof(category), &categoryLen);  SQLGetData(hStmt, 4, SQL\_C\_DOUBLE, &price, 0, &priceLen);  SQLGetData(hStmt, 5, SQL\_C\_LONG, &stock, 0, &stockLen);  product.productID = productID;  product.name = name;  product.category = category;  product.price = price;  product.stock = stock;  products.push\_back(product);  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt); // Показываем ошибку, если запрос не работает  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt); // Освобождаем ресурсы  return products;  }  // Читаем все продажи из базы данных  std::vector<Sale> readSales(SQLHDBC hDbc) {  std::vector<Sale> sales;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"SELECT SaleID, ProductID, SaleDate, Quantity, TotalCost FROM Sales;", SQL\_NTS);  if (SQL\_SUCCEEDED(ret)) {  SQLINTEGER saleID, productID, quantity;  SQLDOUBLE totalCost;  SQLWCHAR saleDate[50];  SQLLEN saleIDLen, productIDLen, saleDateLen, quantityLen, totalCostLen;  while (SQLFetch(hStmt) == SQL\_SUCCESS) {  Sale sale = { 0 };  SQLGetData(hStmt, 1, SQL\_C\_LONG, &saleID, 0, &saleIDLen);  SQLGetData(hStmt, 2, SQL\_C\_LONG, &productID, 0, &productIDLen);  SQLGetData(hStmt, 3, SQL\_C\_WCHAR, saleDate, sizeof(saleDate), &saleDateLen);  SQLGetData(hStmt, 4, SQL\_C\_LONG, &quantity, 0, &quantityLen);  SQLGetData(hStmt, 5, SQL\_C\_DOUBLE, &totalCost, 0, &totalCostLen);  sale.saleID = saleID;  sale.productID = productID;  sale.saleDate = saleDate;  sale.quantity = quantity;  sale.totalCost = totalCost;  sales.push\_back(sale);  }  }  else {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return sales;  }  // Показываем все товары на экране  void displayProducts(const std::vector<Product>& products) {  for (const auto& product : products) {  std::wcout << L"ProductID: " << product.productID << L", Name: " << product.name  << L", Category: " << product.category << L", Price: " << product.price  << L", Stock: " << product.stock << L"\n";  }  }  // Сортируем товары по цене и обновляем базу  void sortProductsByPrice(SQLHDBC hDbc, std::vector<Product>& products) {  std::sort(products.begin(), products.end(), [](const Product& a, const Product& b) {  return a.price < b.price; // Сортируем по возрастанию цены  });  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLRETURN ret = SQLExecDirectW(hStmt, (SQLWCHAR\*)L"DELETE FROM Products;", SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error clearing Products table\n";  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  for (const auto& product : products) {  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  SQLWCHAR query[512];  swprintf\_s(query, 512, L"INSERT INTO Products (ProductID, Name, Category, Price, Stock) VALUES (%d, '%s', '%s', %f, %d);",  product.productID, product.name.c\_str(), product.category.c\_str(), product.price, product.stock);  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  std::wcout << L"Error inserting product\n";  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  }  std::wcout << L"Products sorted by price and saved successfully.\n";  }  // Считаем продажи по категориям  void salesStatsByCategory(const std::vector<Sale>& sales, const std::vector<Product>& products) {  std::set<std::wstring> categories; // Храним уникальные категории  for (const auto& product : products) {  categories.insert(product.category); // Добавляем каждую категорию  }  std::wcout << L"Sales statistics by category:\n";  for (const auto& category : categories) { // Проходим по всем категориям  int totalSold = 0;  double totalRevenue = 0.0;  for (const auto& sale : sales) {  auto it = std::find\_if(products.begin(), products.end(), [&sale, &category](const Product& product) {  return product.productID == sale.productID && product.category == category; // Сравниваем по ID и категории  });  if (it != products.end()) {  totalSold += sale.quantity; // Суммируем количество  totalRevenue += sale.totalCost; // Суммируем выручку  }  }  std::wcout << category << L": " << totalSold << L" items sold, Total Revenue: " << totalRevenue << L"\n";  }  }  // Добавляем новую продажу  void addSale(SQLHDBC hDbc, std::vector<Sale>& sales, std::vector<Product>& products) {  int productID, quantity;  std::wstring saleDate;  std::wcout << L"Enter ProductID to sell: ";  std::cin >> productID;  std::cin.ignore();  std::wcout << L"Enter Quantity: ";  std::cin >> quantity;  std::cin.ignore();  std::wcout << L"Enter SaleDate (YYYY-MM-DD): ";  std::getline(std::wcin, saleDate);  if (!validateDate(saleDate)) {  std::wcout << L"Invalid date format. Please use YYYY-MM-DD (e.g., 2025-05-01).\n";  return;  }  auto productIt = std::find\_if(products.begin(), products.end(), [productID](const Product& product) {  return product.productID == productID; // Ищем товар по ID  });  if (productIt == products.end() || productIt->stock < quantity) {  std::wcout << L"Product not found or insufficient stock.\n";  return;  }  double totalCost = productIt->price \* quantity; // Рассчитываем стоимость  productIt->stock -= quantity; // Уменьшаем запас  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLWCHAR query[256];  swprintf\_s(query, 256, L"UPDATE Products SET Stock = %d WHERE ProductID = %d;", productIt->stock, productID);  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  SQLHSTMT insertStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &insertStmt);  swprintf\_s(query, 256, L"INSERT INTO Sales (ProductID, SaleDate, Quantity, TotalCost) VALUES (%d, #%s#, %d, %f);",  productID, saleDate.c\_str(), quantity, totalCost);  ret = SQLExecDirectW(insertStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, insertStmt);  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  return;  }  SQLFreeHandle(SQL\_HANDLE\_STMT, insertStmt);  std::wcout << L"Sale recorded successfully. Stock updated.\n";  std::wcout << L"Updated Products:\n";  displayProducts(readProducts(hDbc));  }  // Автоматически заказываем товары, если их меньше 5  void autoOrderLowStock(SQLHDBC hDbc, std::vector<Product>& products) {  std::wcout << L"Checking for low stock (less than 5 units):\n";  for (auto& product : products) {  if (product.stock < 5) {  int orderQuantity = 20 - product.stock; // Заказываем до 20 единиц  product.stock += orderQuantity;  SQLHSTMT hStmt;  SQLAllocHandle(SQL\_HANDLE\_STMT, hDbc, &hStmt);  SQLWCHAR query[256];  swprintf\_s(query, 256, L"UPDATE Products SET Stock = %d WHERE ProductID = %d;", product.stock, product.productID);  SQLRETURN ret = SQLExecDirectW(hStmt, query, SQL\_NTS);  if (!SQL\_SUCCEEDED(ret)) {  showSQLError(SQL\_HANDLE\_STMT, hStmt);  }  SQLFreeHandle(SQL\_HANDLE\_STMT, hStmt);  std::wcout << L"Ordered " << orderQuantity << L" units for ProductID: " << product.productID  << L", Name: " << product.name << L". New stock: " << product.stock << L"\n";  }  }  }  // Главная функция с меню  int main() {  SetConsoleOutputCP(CP\_UTF8); // Настраиваем консоль для русских букв  SQLHENV hEnv = nullptr;  SQLHDBC hDbc = nullptr;  if (!connectToDB(hEnv, hDbc)) { // Пробуем подключиться к базе  return 1; // Если не получилось, выходим  }  int choice;  while (true) {  std::wcout << L"\nMenu:\n"; // Показываем меню  std::wcout << L"1. Display all products\n"; // 1: Показать товары  std::wcout << L"2. Sort products by price\n"; // 2: Сортировать по цене  std::wcout << L"3. Show sales statistics by category\n"; // 3: Статистика продаж  std::wcout << L"4. Add new sale\n"; // 4: Добавить продажу  std::wcout << L"5. Auto-order low stock products\n"; // 5: Автозаказ товаров  std::wcout << L"6. Exit\n"; // 6: Выход  std::wcout << L"Enter choice: ";  std::cin >> choice;  std::cin.ignore();  if (choice == 6) break; // Выходим, если выбрали 6  std::vector<Product> products = readProducts(hDbc); // Читаем список товаров  std::vector<Sale> sales = readSales(hDbc); // Читаем список продаж  switch (choice) {  case 1:  displayProducts(products); // Показываем товары  break;  case 2:  sortProductsByPrice(hDbc, products); // Сортируем товары  std::wcout << L"Updated database:\n";  displayProducts(readProducts(hDbc));  break;  case 3:  salesStatsByCategory(sales, products); // Показываем статистику  break;  case 4:  addSale(hDbc, sales, products); // Добавляем продажу  break;  case 5:  autoOrderLowStock(hDbc, products); // Проверяем и заказываем товары  break;  default:  std::wcout << L"Invalid choice.\n"; // Если выбор неверный  }  }  if (hDbc) { // Закрываем соединение с базой  SQLDisconnect(hDbc);  SQLFreeHandle(SQL\_HANDLE\_DBC, hDbc);  }  if (hEnv) SQLFreeHandle(SQL\_HANDLE\_ENV, hEnv);  return 0; // Завершаем программу  } | |
| 3 | **5.3 Система учета пациентов в поликлинике**  Создайте базу данных PatientsDB в pgAdmin4 с двумя таблицами: Patients (пациенты) и Visits (посещения).  #define *\_CRT\_SECURE\_NO\_WARNINGS*  #include <iostream>  #include <vector>  #include <string>  #include <iomanip>  #include <algorithm>  #include <ctime>  #include <libpq-fe.h>  using namespace *std*;  // Структуры данных  struct Patient {  int id;  *string* fullName;  *string* birthDate;  *string* gender;  *string* insuranceNumber;  };  struct Visit {  int id;  int patientId;  *string* visitDate;  *string* doctor;  *string* diagnosis;  double cost;  };  // Глобальные переменные  *vector*<Patient> patients;  *vector*<Visit> visits;  // Функции работы с базой данных  *PGconn*\* connectToDB() {  *PGconn*\* conn = PQconnectdb("host=localhost dbname=ClinicDB user=postgres password=TemaPostgre2007");  if (PQstatus(conn) != CONNECTION\_OK) {  *cerr* << "Connection to database failed: " << PQerrorMessage(conn) << *endl*;  PQfinish(conn);  return nullptr;  }  *cout* << "Connected to database successfully!" << *endl*;  return conn;  }  void loadPatients(*PGconn*\* conn) {  *PGresult*\* res = PQexec(conn, "SELECT \* FROM Patients");  if (PQresultStatus(res) != PGRES\_TUPLES\_OK) {  *cerr* << "Error loading patients: " << PQerrorMessage(conn) << *endl*;  PQclear(res);  return;  }  patients.*clear*();  int rows = PQntuples(res);  for (int i = 0; i < rows; ++i) {  Patient p;  p.id = *atoi*(PQgetvalue(res, i, 0));  p.fullName = PQgetvalue(res, i, 1);  p.birthDate = PQgetvalue(res, i, 2);  p.gender = PQgetvalue(res, i, 3);  p.insuranceNumber = PQgetvalue(res, i, 4);  patients.*push\_back*(p);  }  PQclear(res);  }  void loadVisits(*PGconn*\* conn) {  *PGresult*\* res = PQexec(conn, "SELECT \* FROM Visits");  if (PQresultStatus(res) != PGRES\_TUPLES\_OK) {  *cerr* << "Error loading visits: " << PQerrorMessage(conn) << *endl*;  PQclear(res);  return;  }  visits.*clear*();  int rows = PQntuples(res);  for (int i = 0; i < rows; ++i) {  Visit v;  v.id = *atoi*(PQgetvalue(res, i, 0));  v.patientId = *atoi*(PQgetvalue(res, i, 1));  v.visitDate = PQgetvalue(res, i, 2);  v.doctor = PQgetvalue(res, i, 3);  v.diagnosis = PQgetvalue(res, i, 4);  v.cost = *atof*(PQgetvalue(res, i, 5));  visits.*push\_back*(v);  }  PQclear(res);  }  // Основные функции  void showAllPatients() {  *cout* << "\nAll Patients:\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(30) << "Name"  << *setw*(12) << "Birth Date" << *setw*(10) << "Gender"  << *setw*(20) << "Insurance #" << *endl*;  for (const auto& p : patients) {  *cout* << *left* << *setw*(5) << p.id << *setw*(30) << p.fullName  << *setw*(12) << p.birthDate << *setw*(10) << p.gender  << *setw*(20) << p.insuranceNumber << *endl*;  }  }  void findPatientsByDiagnosis(const *string*& diagnosis) {  *cout* << "\nPatients with diagnosis '" << diagnosis << "':\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(30) << "Name"  << *setw*(15) << "Doctor" << *setw*(12) << "Visit Date" << *endl*;  for (const auto& v : visits) {  if (v.diagnosis.*find*(diagnosis) != *string*::*npos*) {  auto patient = *find\_if*(patients.*begin*(), patients.*end*(),  [&v](const Patient& p) { return p.id == v.patientId; });  if (patient != patients.*end*()) {  *cout* << *left* << *setw*(5) << patient->id << *setw*(30) << patient->fullName  << *setw*(15) << v.doctor << *setw*(12) << v.visitDate << *endl*;  }  }  }  }  void showDoctorStatistics() {  *cout* << "\nDoctor Statistics:\n";  *cout* << *left* << *setw*(20) << "Doctor" << *setw*(15) << "Visits Count"  << *setw*(15) << "Total Income" << *endl*;  *vector*<*pair*<*string*, *pair*<int, double>>> stats;  for (const auto& v : visits) {  bool found = false;  for (auto& s : stats) {  if (s.*first* == v.doctor) {  s.*second*.*first*++;  s.*second*.*second* += v.cost;  found = true;  break;  }  }  if (!found) {  stats.*emplace\_back*(v.doctor, *make\_pair*(1, v.cost));  }  }  for (const auto& s : stats) {  *cout* << *left* << *setw*(20) << s.*first* << *setw*(15) << s.*second*.*first*  << *setw*(15) << s.*second*.*second* << *endl*;  }  }  void addNewVisit(*PGconn*\* conn) {  int patientId;  *string* doctor, diagnosis;  double cost;  *cout* << "Enter patient ID: ";  *cin* >> patientId;  *cin*.*ignore*();  *cout* << "Enter doctor name: ";  *getline*(*cin*, doctor);  *cout* << "Enter diagnosis: ";  *getline*(*cin*, diagnosis);  *cout* << "Enter cost: ";  *cin* >> cost;  // Получаем текущую дату  *time\_t* now = *time*(0);  *tm*\* ltm = *localtime*(&now);  char date[11];  *strftime*(date, sizeof(date), "%Y-%m-%d", ltm);  // Добавляем в базу  *string* sql = "INSERT INTO Visits (PatientID, VisitDate, Doctor, Diagnosis, Cost) "  "VALUES (" + *to\_string*(patientId) + ", '" + date + "', '" +  doctor + "', '" + diagnosis + "', " + *to\_string*(cost) + ")";  *PGresult*\* res = PQexec(conn, sql.*c\_str*());  if (PQresultStatus(res) != PGRES\_COMMAND\_OK) {  *cerr* << "Error adding visit: " << PQerrorMessage(conn) << *endl*;  }  else {  *cout* << "Visit added successfully\n";  // Если хроническое заболевание, назначаем повторный прием  if (diagnosis.*find*("chronic") != *string*::*npos*) {  *tm* nextVisit = \*ltm;  nextVisit.*tm\_mday* += 14; // Через 2 недели  *mktime*(&nextVisit);  char nextDate[11];  *strftime*(nextDate, sizeof(nextDate), "%Y-%m-%d", &nextVisit);  *cout* << "Chronic disease detected. Next visit scheduled for " << nextDate << *endl*;  }  }  PQclear(res);  loadVisits(conn);  }  int *main*() {  *setlocale*(*LC\_ALL*, "rus");  *PGconn*\* conn = connectToDB();  if (!conn) return 1;  loadPatients(conn);  loadVisits(conn);  int choice;  do {  *cout* << "\nClinic Management System\n";  *cout* << "1. Show all patients\n";  *cout* << "2. Find patients by diagnosis\n";  *cout* << "3. Show doctor statistics\n";  *cout* << "4. Add new visit\n";  *cout* << "0. Exit\n";  *cout* << "Enter choice: ";  *cin* >> choice;  switch (choice) {  case 1:  showAllPatients();  break;  case 2: {  *string* diagnosis;  *cout* << "Enter diagnosis to search: ";  *cin*.*ignore*();  *getline*(*cin*, diagnosis);  findPatientsByDiagnosis(diagnosis);  break;  }  case 3:  showDoctorStatistics();  break;  case 4:  addNewVisit(conn);  break;  case 0:  *cout* << "Exiting...\n";  break;  default:  *cout* << "Invalid choice\n";  }  } while (choice != 0);  PQfinish(conn);  return 0;  } |
| 4 | **5.4 Система учета инвентаря в фитнес-клубе**  Создайте базу данных ClubDB в SQL Microsoft Visual Studio с двумя таблицами: Equipment (оборудование) и Maintenance (обслуживание).  #include <windows.h>  #include <sql.h>  #include <sqlext.h>  #include <iostream>  #include <iomanip>  #include <vector>  #include <string>  #include <ctime>  #include <map>  using namespace *std*;  // Структуры данных  struct Equipment {  int id;  *string* name;  *string* type;  *string* purchaseDate;  int lifespan;  bool needsReplacement;  };  struct Maintenance {  int id;  int equipmentId;  *string* date;  *string* type;  };  // Глобальные переменные  *vector*<Equipment> equipmentList;  *vector*<Maintenance> maintenanceRecords;  // Функции работы с базой данных  void checkSQLError(*SQLHANDLE* handle, *SQLSMALLINT* type, const *string*& message) {  *SQLCHAR* sqlState[1024];  *SQLCHAR* messageText[1024];  *SQLSMALLINT* textLength;  *SQLINTEGER* nativeError;  *SQLRETURN* ret;  ret = *SQLGetDiagRec*(type, handle, 1, sqlState, &nativeError, messageText, 1024, &textLength);  if (ret == *SQL\_SUCCESS* || ret == *SQL\_SUCCESS\_WITH\_INFO*) {  *cerr* << message << ": " << messageText << " (SQL State: " << sqlState << ")" << *endl*;  }  }  *SQLHENV* initDatabase(*SQLHDBC*& hdbc) {  *SQLHENV* henv = *NULL*;  *SQLRETURN* ret;  // Выделяем handle окружения  ret = *SQLAllocHandle*(*SQL\_HANDLE\_ENV*, *SQL\_NULL\_HANDLE*, &henv);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  *cerr* << "Error allocating environment handle" << *endl*;  return *NULL*;  }  // Устанавливаем версию ODBC  ret = *SQLSetEnvAttr*(henv, *SQL\_ATTR\_ODBC\_VERSION*, (*SQLPOINTER*)*SQL\_OV\_ODBC3*, 0);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(henv, *SQL\_HANDLE\_ENV*, "Error setting ODBC version");  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  // Выделяем handle соединения  ret = *SQLAllocHandle*(*SQL\_HANDLE\_DBC*, henv, &hdbc);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(henv, *SQL\_HANDLE\_ENV*, "Error allocating connection handle");  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  // Подключаемся к базе данных  ret = *SQLConnect*(hdbc,  (*SQLCHAR*\*)"FitnessClubDB", *SQL\_NTS*,  (*SQLCHAR*\*)"", *SQL\_NTS*,  (*SQLCHAR*\*)"", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error connecting to database");  *SQLFreeHandle*(*SQL\_HANDLE\_DBC*, hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  *cout* << "Connected to database successfully!" << *endl*;  return henv;  }  void loadEquipment(*SQLHDBC* hdbc) {  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  ret = *SQLExecDirect*(hstmt, (*SQLCHAR*\*)"SELECT \* FROM Equipment", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL query");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  equipmentList.*clear*();  while (*SQLFetch*(hstmt) == *SQL\_SUCCESS*) {  Equipment item;  *SQLGetData*(hstmt, 1, *SQL\_C\_LONG*, &item.id, 0, *NULL*);  char name[151], type[51], date[11];  *SQLGetData*(hstmt, 2, *SQL\_C\_CHAR*, name, 150, *NULL*);  *SQLGetData*(hstmt, 3, *SQL\_C\_CHAR*, type, 50, *NULL*);  *SQLGetData*(hstmt, 4, *SQL\_C\_CHAR*, date, 10, *NULL*);  *SQLGetData*(hstmt, 5, *SQL\_C\_LONG*, &item.lifespan, 0, *NULL*);  item.name = name;  item.type = type;  item.purchaseDate = date;  // Проверка срока службы  *time\_t* now = *time*(0);  *tm* purchaseTm = { 0 };  strptime(item.purchaseDate.*c\_str*(), "%Y-%m-%d", &purchaseTm);  *time\_t* purchaseTime = *mktime*(&purchaseTm);  double yearsInService = *difftime*(now, purchaseTime) / (60 \* 60 \* 24 \* 365.25);  item.needsReplacement = (yearsInService >= item.lifespan);  equipmentList.*push\_back*(item);  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  }  void loadMaintenance(*SQLHDBC* hdbc) {  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  ret = *SQLExecDirect*(hstmt, (*SQLCHAR*\*)"SELECT \* FROM Maintenance", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL query");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  maintenanceRecords.*clear*();  while (*SQLFetch*(hstmt) == *SQL\_SUCCESS*) {  Maintenance record;  *SQLGetData*(hstmt, 1, *SQL\_C\_LONG*, &record.id, 0, *NULL*);  *SQLGetData*(hstmt, 2, *SQL\_C\_LONG*, &record.equipmentId, 0, *NULL*);  char date[11], type[101];  *SQLGetData*(hstmt, 3, *SQL\_C\_CHAR*, date, 10, *NULL*);  *SQLGetData*(hstmt, 4, *SQL\_C\_CHAR*, type, 100, *NULL*);  record.date = date;  record.type = type;  maintenanceRecords.*push\_back*(record);  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  }  // Основные функции  void showAllEquipment() {  *cout* << "\nAll Equipment:\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(25) << "Name"  << *setw*(15) << "Type" << *setw*(12) << "Purchase"  << *setw*(10) << "Lifespan" << *setw*(15) << "Status" << *endl*;  for (const auto& item : equipmentList) {  *cout* << *left* << *setw*(5) << item.id << *setw*(25) << item.name  << *setw*(15) << item.type << *setw*(12) << item.purchaseDate  << *setw*(10) << item.lifespan;  if (item.needsReplacement) {  *cout* << *setw*(15) << "REPLACE";  }  else {  *time\_t* now = *time*(0);  *tm* purchaseTm = { 0 };  strptime(item.purchaseDate.*c\_str*(), "%Y-%m-%d", &purchaseTm);  *time\_t* purchaseTime = *mktime*(&purchaseTm);  double yearsLeft = item.lifespan - (*difftime*(now, purchaseTime) / (60 \* 60 \* 24 \* 365.25);  *cout* << *setw*(15) << *fixed* << *setprecision*(1) << yearsLeft << "y left";  }  *cout* << *endl*;  }  }  void sortByPurchaseDate() {  *sort*(equipmentList.*begin*(), equipmentList.*end*(), [](const Equipment& a, const Equipment& b) {  return a.purchaseDate < b.purchaseDate;  });  *cout* << "\nEquipment sorted by purchase date:\n";  showAllEquipment();  }  void showMaintenanceStats() {  *map*<*string*, *pair*<int, int>> stats; // type -> (maintenance count, equipment count)  for (const auto& item : equipmentList) {  stats[item.type].*second*++;  }  for (const auto& record : maintenanceRecords) {  auto it = *find\_if*(equipmentList.*begin*(), equipmentList.*end*(),  [&record](const Equipment& e) { return e.id == record.equipmentId; });  if (it != equipmentList.*end*()) {  stats[it->type].*first*++;  }  }  *cout* << "\nMaintenance Statistics by Equipment Type:\n";  *cout* << *left* << *setw*(20) << "Type" << *setw*(20) << "Equipment Count"  << *setw*(20) << "Maintenance Count" << *setw*(20) << "Maintenance Ratio" << *endl*;  for (const auto& stat : stats) {  double ratio = (double)stat.*second*.*first* / stat.*second*.*second*;  *cout* << *left* << *setw*(20) << stat.*first*  << *setw*(20) << stat.*second*.*second*  << *setw*(20) << stat.*second*.*first*  << *setw*(20) << *fixed* << *setprecision*(2) << ratio  << *endl*;  }  }  void addMaintenanceRecord(*SQLHDBC* hdbc) {  int equipmentId;  *string* maintenanceType;  *cout* << "Enter equipment ID: ";  *cin* >> equipmentId;  *cin*.*ignore*();  *cout* << "Enter maintenance type: ";  *getline*(*cin*, maintenanceType);  // Получаем текущую дату  *time\_t* now = *time*(0);  *tm*\* ltm = *localtime*(&now);  char date[11];  *strftime*(date, sizeof(date), "%Y-%m-%d", ltm);  // Добавляем в базу  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  *string* sql = "INSERT INTO Maintenance (EquipmentID, Date, Type) VALUES (?, ?, ?)";  ret = *SQLPrepare*(hstmt, (*SQLCHAR*\*)sql.*c\_str*(), *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error preparing SQL statement");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  // Привязываем параметры  ret = *SQLBindParameter*(hstmt, 1, *SQL\_PARAM\_INPUT*, *SQL\_C\_LONG*, *SQL\_INTEGER*, 0, 0, &equipmentId, 0, *NULL*);  ret = *SQLBindParameter*(hstmt, 2, *SQL\_PARAM\_INPUT*, *SQL\_C\_CHAR*, *SQL\_VARCHAR*, 10, 0, (*SQLCHAR*\*)date, 11, *NULL*);  ret = *SQLBindParameter*(hstmt, 3, *SQL\_PARAM\_INPUT*, *SQL\_C\_CHAR*, *SQL\_VARCHAR*, 100, 0, (*SQLCHAR*\*)maintenanceType.*c\_str*(), maintenanceType.*length*(), *NULL*);  ret = *SQLExecute*(hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL statement");  }  else {  *cout* << "Maintenance record added successfully\n";  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  loadMaintenance(hdbc);  }  void markForReplacement(*SQLHDBC* hdbc) {  *vector*<int> toReplace;  for (const auto& item : equipmentList) {  if (item.needsReplacement) {  toReplace.*push\_back*(item.id);  }  }  if (toReplace.*empty*()) {  *cout* << "No equipment needs replacement at this time.\n";  return;  }  *cout* << "\nEquipment marked for replacement:\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(25) << "Name"  << *setw*(15) << "Type" << *setw*(12) << "Purchase"  << *setw*(10) << "Lifespan" << *endl*;  for (int id : toReplace) {  auto it = *find\_if*(equipmentList.*begin*(), equipmentList.*end*(),  [id](const Equipment& e) { return e.id == id; });  if (it != equipmentList.*end*()) {  *cout* << *left* << *setw*(5) << it->id << *setw*(25) << it->name  << *setw*(15) << it->type << *setw*(12) << it->purchaseDate  << *setw*(10) << it->lifespan << *endl*;  }  }  *cout* << "\nWould you like to generate replacement orders? (y/n): ";  char choice;  *cin* >> choice;  if (choice == 'y' || choice == 'Y') {  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  for (int id : toReplace) {  *string* sql = "UPDATE Equipment SET NeedsReplacement = 1 WHERE EquipmentID = " + *to\_string*(id);  ret = *SQLExecDirect*(hstmt, (*SQLCHAR*\*)sql.*c\_str*(), *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error updating equipment");  }  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  *cout* << "Replacement orders generated for " << toReplace.*size*() << " items.\n";  loadEquipment(hdbc);  }  }  int *main*() {  *SQLHDBC* hdbc;  *SQLHENV* henv = initDatabase(hdbc);  if (!henv) return 1;  loadEquipment(hdbc);  loadMaintenance(hdbc);  int choice;  do {  *cout* << "\nFitness Club Inventory System\n";  *cout* << "1. Show all equipment\n";  *cout* << "2. Sort equipment by purchase date\n";  *cout* << "3. Show maintenance statistics\n";  *cout* << "4. Add maintenance record\n";  *cout* << "5. Mark equipment for replacement\n";  *cout* << "0. Exit\n";  *cout* << "Enter choice: ";  *cin* >> choice;  switch (choice) {  case 1:  showAllEquipment();  break;  case 2:  sortByPurchaseDate();  break;  case 3:  showMaintenanceStats();  break;  case 4:  addMaintenanceRecord(hdbc);  break;  case 5:  markForReplacement(hdbc);  break;  case 0:  *cout* << "Exiting...\n";  break;  default:  *cout* << "Invalid choice\n";  }  } while (choice != 0);  *SQLDisconnect*(hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_DBC*, hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return 0;  } |
| 5 | **5.5 Система учета задач в IT-компании**  Создайте базу данных TasksDB в SQL Microsoft Visual Studio с двумя таблицами: Projects (проекты) и Tasks (задачи).  #include <windows.h>  #include <sql.h>  #include <sqlext.h>  #include <iostream>  #include <iomanip>  #include <vector>  #include <string>  #include <ctime>  #include <map>  using namespace *std*;  // Структуры данных  struct Project {  int id;  *string* name;  *string* client;  *string* deadline;  bool isOverdue;  int taskCount;  };  struct Task {  int id;  int projectId;  *string* description;  *string* status;  int priority;  };  // Глобальные переменные  *vector*<Project> projects;  *vector*<Task> tasks;  // Функции работы с базой данных  void checkSQLError(*SQLHANDLE* handle, *SQLSMALLINT* type, const *string*& message) {  *SQLCHAR* sqlState[1024];  *SQLCHAR* messageText[1024];  *SQLSMALLINT* textLength;  *SQLINTEGER* nativeError;  *SQLRETURN* ret;  ret = *SQLGetDiagRec*(type, handle, 1, sqlState, &nativeError, messageText, 1024, &textLength);  if (ret == *SQL\_SUCCESS* || ret == *SQL\_SUCCESS\_WITH\_INFO*) {  *cerr* << message << ": " << messageText << " (SQL State: " << sqlState << ")" << *endl*;  }  }  *SQLHENV* initDatabase(*SQLHDBC*& hdbc) {  *SQLHENV* henv = *NULL*;  *SQLRETURN* ret;  // Выделяем handle окружения  ret = *SQLAllocHandle*(*SQL\_HANDLE\_ENV*, *SQL\_NULL\_HANDLE*, &henv);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  *cerr* << "Error allocating environment handle" << *endl*;  return *NULL*;  }  // Устанавливаем версию ODBC  ret = *SQLSetEnvAttr*(henv, *SQL\_ATTR\_ODBC\_VERSION*, (*SQLPOINTER*)*SQL\_OV\_ODBC3*, 0);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(henv, *SQL\_HANDLE\_ENV*, "Error setting ODBC version");  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  // Выделяем handle соединения  ret = *SQLAllocHandle*(*SQL\_HANDLE\_DBC*, henv, &hdbc);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(henv, *SQL\_HANDLE\_ENV*, "Error allocating connection handle");  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  // Подключаемся к базе данных  ret = *SQLConnect*(hdbc,  (*SQLCHAR*\*)"TasksDB", *SQL\_NTS*,  (*SQLCHAR*\*)"", *SQL\_NTS*,  (*SQLCHAR*\*)"", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error connecting to database");  *SQLFreeHandle*(*SQL\_HANDLE\_DBC*, hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return *NULL*;  }  *cout* << "Connected to database successfully!" << *endl*;  return henv;  }  void loadProjects(*SQLHDBC* hdbc) {  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  // Загружаем проекты и количество задач для каждого  ret = *SQLExecDirect*(hstmt, (*SQLCHAR*\*)"SELECT p.\*, COUNT(t.TaskID) AS TaskCount FROM Projects p LEFT JOIN Tasks t ON p.ProjectID = t.ProjectID GROUP BY p.ProjectID, p.Name, p.Client, p.Deadline, p.IsOverdue", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL query");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  projects.*clear*();  while (*SQLFetch*(hstmt) == *SQL\_SUCCESS*) {  Project project;  *SQLGetData*(hstmt, 1, *SQL\_C\_LONG*, &project.id, 0, *NULL*);  char name[151], client[101], deadline[11];  *SQLGetData*(hstmt, 2, *SQL\_C\_CHAR*, name, 150, *NULL*);  *SQLGetData*(hstmt, 3, *SQL\_C\_CHAR*, client, 100, *NULL*);  *SQLGetData*(hstmt, 4, *SQL\_C\_CHAR*, deadline, 10, *NULL*);  *SQLGetData*(hstmt, 5, *SQL\_C\_BIT*, &project.isOverdue, 0, *NULL*);  *SQLGetData*(hstmt, 6, *SQL\_C\_LONG*, &project.taskCount, 0, *NULL*);  project.name = name;  project.client = client;  project.deadline = deadline;  projects.*push\_back*(project);  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  }  void loadTasks(*SQLHDBC* hdbc) {  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  ret = *SQLExecDirect*(hstmt, (*SQLCHAR*\*)"SELECT \* FROM Tasks", *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL query");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  tasks.*clear*();  while (*SQLFetch*(hstmt) == *SQL\_SUCCESS*) {  Task task;  *SQLGetData*(hstmt, 1, *SQL\_C\_LONG*, &task.id, 0, *NULL*);  *SQLGetData*(hstmt, 2, *SQL\_C\_LONG*, &task.projectId, 0, *NULL*);  char description[301], status[21];  *SQLGetData*(hstmt, 3, *SQL\_C\_CHAR*, description, 300, *NULL*);  *SQLGetData*(hstmt, 4, *SQL\_C\_CHAR*, status, 20, *NULL*);  *SQLGetData*(hstmt, 5, *SQL\_C\_LONG*, &task.priority, 0, *NULL*);  task.description = description;  task.status = status;  tasks.*push\_back*(task);  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  }  // Основные функции  void showAllProjects() {  *cout* << "\nAll Projects:\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(25) << "Name"  << *setw*(20) << "Client" << *setw*(12) << "Deadline"  << *setw*(10) << "Overdue" << *setw*(10) << "Tasks" << *endl*;  for (const auto& project : projects) {  *cout* << *left* << *setw*(5) << project.id << *setw*(25) << project.name  << *setw*(20) << project.client << *setw*(12) << project.deadline  << *setw*(10) << (project.isOverdue ? "Yes" : "No")  << *setw*(10) << project.taskCount << *endl*;  }  }  void sortTasksByPriority() {  *sort*(tasks.*begin*(), tasks.*end*(), [](const Task& a, const Task& b) {  return a.priority < b.priority;  });  *cout* << "\nTasks sorted by priority:\n";  *cout* << *left* << *setw*(5) << "ID" << *setw*(10) << "Project"  << *setw*(30) << "Description" << *setw*(15) << "Status"  << *setw*(10) << "Priority" << *endl*;  for (const auto& task : tasks) {  *cout* << *left* << *setw*(5) << task.id << *setw*(10) << task.projectId  << *setw*(30) << (task.description.*length*() > 27 ? task.description.*substr*(0, 27) + "..." : task.description)  << *setw*(15) << task.status << *setw*(10) << task.priority << *endl*;  }  }  void showTaskStatusStats() {  *map*<*string*, int> statusStats;  for (const auto& task : tasks) {  statusStats[task.status]++;  }  *cout* << "\nTask Status Statistics:\n";  *cout* << *left* << *setw*(20) << "Status" << *setw*(15) << "Count" << *endl*;  for (const auto& stat : statusStats) {  *cout* << *left* << *setw*(20) << stat.*first* << *setw*(15) << stat.*second* << *endl*;  }  }  void addNewTask(*SQLHDBC* hdbc) {  int projectId, priority;  *string* description, status;  *cout* << "Enter project ID: ";  *cin* >> projectId;  *cin*.*ignore*();  *cout* << "Enter task description: ";  *getline*(*cin*, description);  *cout* << "Enter status (New/In Progress/Completed): ";  *getline*(*cin*, status);  *cout* << "Enter priority (1-5, where 1 is highest): ";  *cin* >> priority;  // Добавляем в базу  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  *string* sql = "INSERT INTO Tasks (ProjectID, Description, Status, Priority) VALUES (?, ?, ?, ?)";  ret = *SQLPrepare*(hstmt, (*SQLCHAR*\*)sql.*c\_str*(), *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error preparing SQL statement");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  // Привязываем параметры  ret = *SQLBindParameter*(hstmt, 1, *SQL\_PARAM\_INPUT*, *SQL\_C\_LONG*, *SQL\_INTEGER*, 0, 0, &projectId, 0, *NULL*);  ret = *SQLBindParameter*(hstmt, 2, *SQL\_PARAM\_INPUT*, *SQL\_C\_CHAR*, *SQL\_VARCHAR*, 300, 0, (*SQLCHAR*\*)description.*c\_str*(), description.*length*(), *NULL*);  ret = *SQLBindParameter*(hstmt, 3, *SQL\_PARAM\_INPUT*, *SQL\_C\_CHAR*, *SQL\_VARCHAR*, 20, 0, (*SQLCHAR*\*)status.*c\_str*(), status.*length*(), *NULL*);  ret = *SQLBindParameter*(hstmt, 4, *SQL\_PARAM\_INPUT*, *SQL\_C\_LONG*, *SQL\_INTEGER*, 0, 0, &priority, 0, *NULL*);  ret = *SQLExecute*(hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL statement");  }  else {  *cout* << "Task added successfully\n";  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  loadProjects(hdbc);  loadTasks(hdbc);  }  void markOverdueProjects(*SQLHDBC* hdbc) {  // Получаем текущую дату  *time\_t* now = *time*(0);  *tm*\* ltm = *localtime*(&now);  char currentDate[11];  *strftime*(currentDate, sizeof(currentDate), "%Y-%m-%d", ltm);  *SQLHSTMT* hstmt = *NULL*;  *SQLRETURN* ret = *SQLAllocHandle*(*SQL\_HANDLE\_STMT*, hdbc, &hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hdbc, *SQL\_HANDLE\_DBC*, "Error allocating statement handle");  return;  }  // Помечаем просроченные проекты  *string* sql = "UPDATE Projects SET IsOverdue = 1 WHERE Deadline < ? AND IsOverdue = 0";  ret = *SQLPrepare*(hstmt, (*SQLCHAR*\*)sql.*c\_str*(), *SQL\_NTS*);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error preparing SQL statement");  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  return;  }  ret = *SQLBindParameter*(hstmt, 1, *SQL\_PARAM\_INPUT*, *SQL\_C\_CHAR*, *SQL\_DATE*, 10, 0, (*SQLCHAR*\*)currentDate, 11, *NULL*);  ret = *SQLExecute*(hstmt);  if (ret != *SQL\_SUCCESS* && ret != *SQL\_SUCCESS\_WITH\_INFO*) {  checkSQLError(hstmt, *SQL\_HANDLE\_STMT*, "Error executing SQL statement");  }  else {  *SQLINTEGER* rowsAffected = 0;  *SQLRowCount*(hstmt, &rowsAffected);  *cout* << "Marked " << rowsAffected << " projects as overdue\n";  }  *SQLFreeHandle*(*SQL\_HANDLE\_STMT*, hstmt);  loadProjects(hdbc);  }  int *main*() {  *SQLHDBC* hdbc;  *SQLHENV* henv = initDatabase(hdbc);  if (!henv) return 1;  loadProjects(hdbc);  loadTasks(hdbc);  int choice;  do {  *cout* << "\nProject Management System\n";  *cout* << "1. Show all projects with task count\n";  *cout* << "2. Sort tasks by priority\n";  *cout* << "3. Show task status statistics\n";  *cout* << "4. Add new task\n";  *cout* << "5. Mark overdue projects\n";  *cout* << "0. Exit\n";  *cout* << "Enter choice: ";  *cin* >> choice;  switch (choice) {  case 1:  showAllProjects();  break;  case 2:  sortTasksByPriority();  break;  case 3:  showTaskStatusStats();  break;  case 4:  addNewTask(hdbc);  break;  case 5:  markOverdueProjects(hdbc);  break;  case 0:  *cout* << "Exiting...\n";  break;  default:  *cout* << "Invalid choice\n";  }  } while (choice != 0);  *SQLDisconnect*(hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_DBC*, hdbc);  *SQLFreeHandle*(*SQL\_HANDLE\_ENV*, henv);  return 0;  } |

**8. Ответы к контрольным вопросам**

**1. Зачем нужны базы данных?**

Базы данных нужны для организованного хранения, управления и извлечения больших объёмов данных. Они обеспечивают:

* Быстрый доступ к данным через запросы.
* Структурированное хранение (таблицы, связи).
* Безопасность (управление доступом).
* Масштабируемость и возможность одновременной работы множества пользователей. Пример: база данных магазина хранит информацию о товарах, клиентах и заказах, позволяя быстро находить нужные данные.

**2. Что такое ODBC и зачем он нужен?**

ODBC (Open Database Connectivity) — это стандартный интерфейс, который позволяет приложениям (например, на C++) подключаться к различным базам данных (Access, MySQL, PostgreSQL и др.) через единый API.  
**Зачем нужен**:

* Унифицирует доступ к разным базам данных, не требуя писать отдельный код для каждой СУБД.
* Позволяет использовать SQL для работы с данными.
* Обеспечивает независимость приложения от конкретной базы данных.

**3. Какой SQL-запрос используется для добавления данных?**

Для добавления данных используется запрос INSERT INTO.

**4. Чем базы данных удобнее обычных текстовых файлов или блокнотов?**

* **Структурированность**: данные в БД организованы в таблицы с фиксированными полями, а не в свободной форме, как в текстовых файлах.
* **Быстрый поиск и фильтрация**: SQL-запросы (например, SELECT) позволяют мгновенно находить данные, в отличие от ручного поиска в файле.
* **Обновление и целостность**: базы данных поддерживают транзакции и ограничения (например, уникальность), что предотвращает ошибки.
* **Масштабируемость**: БД легко обрабатывают миллионы записей, тогда как текстовые файлы становятся медленными.
* **Одновременный доступ**: несколько пользователей могут работать с БД одновременно, чего нельзя сделать с текстовым файлом.

**5. Какую роль играет драйвер ODBC при работе с базой данных?**

Драйвер ODBC выступает посредником между приложением и базой данных.  
**Роль**:

* Переводит запросы приложения (например, SQL-запросы через ODBC API) в формат, понятный конкретной СУБД.
* Обеспечивает связь с базой данных (например, Microsoft Access, MySQL).
* Обрабатывает ответы от базы данных и возвращает их приложению.  
  Пример: для Access нужен драйвер "Microsoft Access Driver (\*.mdb, \*.accdb)", который указывается в строке подключения.

**6. Какой SQL-запрос нужен, чтобы извлечь данные из таблицы?**

Для извлечения данных используется запрос SELECT.

**7. Какую структуру имеют базы данных?**

Базы данных обычно имеют **реляционную структуру**:

* **Таблицы**: основная единица хранения данных. Каждая таблица содержит данные одного типа (например, таблица Students для студентов).
* **Столбцы (поля)**: определяют типы данных в таблице (например, FullName, AverageScore).
* **Строки (записи)**: конкретные данные для каждого объекта (например, одна строка = один студент).
* **Связи**: таблицы могут быть связаны через ключи (например, первичный ключ ID связывает таблицы Students и Grades).
* **Схема**: описание структуры базы данных (набор таблиц, их полей и связей).

**8. Можно ли через ODBC подключиться к любой базе данных?**

Не совсем. ODBC позволяет подключиться к большинству реляционных баз данных (Access, MySQL, PostgreSQL, Oracle, SQL Server и др.), но:

* Для каждой базы данных нужен соответствующий ODBC-драйвер. Если драйвера нет, подключиться не получится.
* Некоторые нереляционные базы данных (например, NoSQL, такие как MongoDB) могут не поддерживать ODBC напрямую, хотя существуют сторонние драйверы.
* Настройки подключения (строка подключения) должны быть корректно настроены для конкретной СУБД.

**9. Какой SQL-запрос используется для изменения существующей записи?**

Для изменения записи используется запрос UPDATE.

**10. Что такое таблица в базе данных?**

Таблица — это основная структура в реляционной базе данных, которая хранит данные в виде строк и столбцов:

* **Столбцы** (или поля) определяют типы данных (например, FullName, AverageScore).
* **Строки** (или записи) содержат конкретные данные для каждого объекта (например, данные одного студента).

**11. Какую библиотеку или технологию используют программы на C++ для работы с ODBC?**

В C++ для работы с ODBC используется библиотека **ODBC API**, которая предоставляется через заголовочные файлы:

* <windows.h> (для Windows).
* <sql.h>, <sqlext.h>, <sqltypes.h> — основные заголовочные файлы для ODBC.

**12. Какой SQL-запрос удаляет запись из таблицы?**

Для удаления записи используется запрос DELETE.

**13. Что такое запись (строка) в таблице базы данных?**

Запись (или строка) — это единица данных в таблице, представляющая один объект или сущность.

* Каждая строка содержит значения для всех столбцов таблицы.

**14. Для чего используется std::vector (к примеру) в программе, работающей с БД?**

std::vector используется для временного хранения данных, извлечённых из базы данных, в оперативной памяти:

* Позволяет быстро манипулировать данными (сортировка, фильтрация, поиск) без обращения к базе.
* Удобен для хранения списка записей (например, std::vector<Student> для списка студентов).
* Упрощает итерацию по данным (например, для вывода или обработки). Пример: данные из таблицы Students загружаются в std::vector<Student>, где каждая запись — объект структуры Student.

**15. Почему иногда данные сначала загружаются в оперативную память, а не сразу обрабатываются в базе?**

* **Скорость**: операции в оперативной памяти (например, сортировка в std::vector) быстрее, чем запросы к базе данных, особенно если база хранится на диске.
* **Гибкость**: в памяти можно выполнять сложные операции (фильтрация, преобразования), которые трудно или медленно реализовать через SQL.
* **Снижение нагрузки на БД**: частые запросы к базе данных нагружают её, особенно при многопользовательском доступе. Загрузка данных в память позволяет работать с копией.
* **Кэширование**: данные можно сохранить в памяти для повторного использования без новых запросов. Пример: сортировка студентов по среднему баллу в std::vector быстрее, чем выполнение ORDER BY в базе данных для каждой операции.