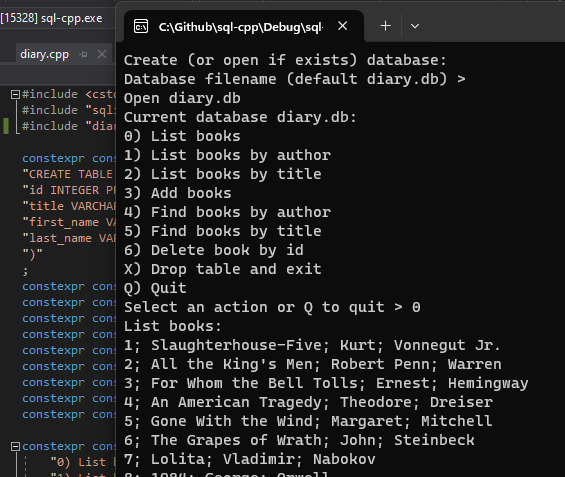
**Приложения Reading Diary**

**Автор:** Елизавета Руденко

<https://github.com/rudenyel/sql-cpp.git>

**Задача:**

Разработаем приложение позволяющее работать с дневником, в котором хранятся названия книг и авторы. Приложение должно уметь вставлять и удалять записи, сортировать записи при выводе на экран, искать книги по названию и автору.



**Инструменты:**

Для хранения данных будем использовать SQLite – компактную, встраиваемую систему управления реляционными базами данных (СУБД). Она отличается от многих других СУБД, таких как MySQL или PostgreSQL, тем, что она встраивается непосредственно в приложение, хранится в одном файле, не требуя отдельного сервера баз данных.

SQLite API version 3.35.5

Microsoft Visual Studio Community 2019  
DBeaver Community 23

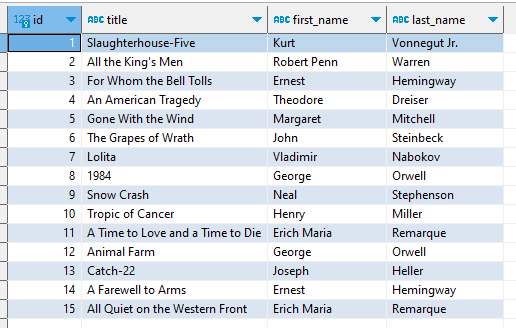
**Приобретенные навыки:**

Just finished the course “Using SQL with C++” by Bill Weinman! Check it out: <https://www.linkedin.com/learning/certificates/90144b68d909fc0773808f26b71f04d9b15ef661c1406f47fa337d24d545be6e>

**Реляционная модель и проектирование базы данных**

Основой современной технологии баз данных является реляционная модель данных – абстрактная теория данных, основная на некоторых положениях математики (в основном теории множеств и предикативной логики). Принципы реляционной модели были изначально заложены доктором Коддом (Codd), сотрудником IBM. В конце 1968 года Кодд, математик по образованию, впервые осознал, что математические дисциплины можно использовать, чтобы привнести в область управления базами данных строгие принципы и точность.

Реляционная база данных хранит информацию в таблицах (в реляционной теории– отношения) со строками (кортежами) и столбцами (атрибутами).



Отношение делится на две части: заголовок и тело. Заголовок – это множество атрибутов, а тело – это множество кортежей. Количество атрибутов называется степенью, а количество кортежей – кардинальным числом. Отношения обладают четырьмя очень важными свойствами:

**1. В них нет одинаковых кортежей.** Это свойство следует из того факта, что тело отношения – это математическое множество (кортежей), а множество в математике по определению не содержит одинаковых элементов. Важным следствием этого факта, является то, что всегда существует первичный ключ – уникальное значение в пределах отношения (в нашем случае это поле id). Первичные ключи обеспечивают основной механизм адресации на уровне кортежей в реляционной системе. Единственный гарантируемый системой способ точно указать на какой-нибудь кортеж – это указать значение некоторого первичного ключа.

**2. Кортежи не упорядочены (сверху вниз).** Это свойство также следует из того, что тело отношения – это математическое множество (кортежей), а простые множества в математике не упорядочены.

**3. Атрибуты не упорядочены (слева направо).** Это свойство следует из того факта, что заголовок отношения также определен как множество (атрибутов).

Первые три свойства отношений служат хорошей иллюстрацией того факта, что отношение и таблица – это не одно и то же. Хотя дальше в тексте мы будем использовать название таблица.

**4. Все значения атрибутов атомарные.** Отношение, удовлетворяющее этому условию, называется нормализованным, или представленным в первой нормальной форме. В нашем случае мы нормализовали таблицу, разделив не только название книги и автора, но и имя и фамилию. Это ускорит поиск по названию и по фамилии автора, позволит, например, сортировать дневник как по названию, так и по автору.

Существуют вторая нормальная форма, третья и т.д. В нашем случае мы могли бы привести наше отношение ко второй нормальной форме, создав таблицу authors (id, first\_name, last\_name), оставив в таблице books поле id\_author и связав эти таблицы отношением один ко многим authors.id -> books.id\_author. Это уменьшило бы размеры базы данных, ведь множество повторяющихся строк с именами и фамилиями авторов были бы заменены числовым идентификатором. Это также уменьшает процент ошибок при наборе и исправлении текста – имя и фамилия автора находится в одном месте, в таблице Authors.

Так же мы могли бы решить проблему хранения нескольких авторов для одной книги, введя промежуточную таблицу с соответствием id\_books - id\_authors. Но мы остановимся в проектировании базы данных на одной таблице.

**Язык запросов SQL**

Для работы с базами данными используется язык структурированных запросов (SQL). Инструкции SQL можно использовать для хранения, обновления, удаления, поиска и извлечения информации из базы данных.

Язык SQL используется также для управления и оптимизации базы данных. Например, после создания базы данных мы можем создать в ней таблицу books с помощью SQL-запроса:

CREATE TABLE IF NOT EXISTS books (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title VARCHAR(127) CHECK(title != ''),

first\_name VARCHAR(127) CHECK(first\_name != ''),

last\_name VARCHAR(127) CHECK(last\_name != ''))

Он создает в ней таблицу books с уникальным первичным ключом id, который автоматически увеличивается на единицу при добавлении новых записей. Также в этой таблице создаются текстовые поля title, first\_name, last\_name размером до 127 символов, которые не могут быть пустыми строками.

Мы можем добавить данные в таблицу с помощью команды INSERT:

INSERT INTO books (title, first\_name, last\_name) VALUES ('1984', 'George', 'Orwell')

Значение первичного ключа при операциях вставки указывать не нужно – его значение контролирует база данных.

Удаление записи:

DELETE FROM books WHERE id = 1

Для выполнения запросов выборки данных используется команда SELECT. Например, для отображения данных, отсортированных по фамилии (по возрастанию):

SELECT \* FROM books ORDER BY last\_name

и в обратном порядке:

SELECT \* FROM books ORDER BY last\_name DESC

Вместо \* можно указать конкретные имена столбцов, которые мы хотим видеть в выборке:

SELECT last\_name, first\_name FROM books ORDER BY last\_name

В этом запросе есть повторяющиеся имена и фамилии авторов, их можно сгруппировать и не выводить повторяющиеся значения:

SELECT last\_name, first\_name

FROM books

GROUP BY last\_name

ORDER BY last\_name

Можно выбрать авторов с конкретным именем:

SELECT last\_name, first\_name

FROM books

WHERE first\_name = 'Kurt'

ORDER BY last\_name

или с именем, начинающимся на конкретную букву:

SELECT last\_name, first\_name

FROM books

WHERE first\_name LIKE 'E%'

ORDER BY last\_name

GROUP BY last\_name

Также в запросы, как и в функции, можно передавать параметры, которые позволяют использовать один и тот же запрос для различных входных данных:

SELECT title FROM books WHERE last\_name = :last\_name

При запуске этого запроса нужно ввести или передать через API параметр last\_name

**SQLite API**

SQLite API содержит более 200 функций, однако большинство из них являются необязательными и очень специализированными. Основной API небольшой, простой и легкий в освоении. Рассмотрим его на примере:

<https://github.com/rudenyel/sql-cpp/blob/main/sqlite3-example.cpp>

В начале программы мы должны установить соединение с базой данных с помощью sqlite3\_open, передав ей название файла базы данных. Если файл не существует, то он будет создан. Функция вернет указатель на объект соединения с базой данных (database connection) – sqlite3\*, который требуют многие функции SQLite API в качестве первого параметра. В конце работы программы не забываем закрывать соединение с базой данных с помощью sqlite3\_close.

Основной задачей ядра базы данных SQL является выполнения SQL-запросов. Для этого разработчику нужен еще один указатель на объект – sqlite3\_stmt\* (prepared statement). Экземпляр этого объекта представляет собой SQL-запрос, скомпилированный в двоичную форму и готовый к выполнению. Рассматривайте каждый SQL-запрос как отдельную компьютерную программу. Текст SQL-запроса является исходным кодом. Prepared statement – это скомпилированный объектный код.

Весь текст SQL-запроса должен быть преобразован в prepared statement, прежде чем его можно будет запустить. Это делает функция sqlite3\_prepare\_v2(). Например, мы передали в нее SELECT \* FROM books WHERE last\_name = ?. На данном этапе происходит анализ текста, который показывает, что нужно вернуть все столбцы таблицы books и передать в запрос один параметр. С помощью функций sqlite3\_column\_count() и sqlite3\_column\_name() можно получить количество и имена столбцов в запросе. С помощью sqlite3\_bind\_parameter\_count() и sqlite3\_bind\_text() можно узнать количество параметров и передать текстовый параметр в запрос.

Далее с помощью функции sqlite3\_step() SQL-запрос запускается на выполнение. Вызывая sqlite3\_step() несколько раз мы получаем следующую строку результирующего набора данных, что похоже на перебор строк в двумерном массиве. На каждом этапе этого цикла с помощью sqlite3\_column\_text() (количество столбцов мы знаем) можно получить отдельное поле в текущем строке данных. В конце можно вернуться к началу набора данных с помощью sqlite3\_reset(). Или не забыть уничтожить объект sqlite3\_stmt\* после выполнения запроса с помощью sqlite3\_finalize().

Нужно понимать, что в базе могут храниться значения различных типов. Для работы с ними нужно использовать соответствующие функции, например вместо sqlite3\_column\_text() – sqlite3\_column\_int(), sqlite3\_column\_double() и т.д.

**SQLite API wrapper и основная программа**

Для еще большего упрощения работы с SQLite API напишем класс-обертку (wrapper):

sqlite3-wrapper.h: <https://github.com/rudenyel/sql-cpp/blob/main/include/sqlite3-wrapper.h>

sqlite3-wrapper.cpp: <https://github.com/rudenyel/sql-cpp/blob/main/sqlite3-wrapper.cpp>

В основной приватный метод класса \_prepare() передаются параметры запроса (va\_list, переменное количество аргументов). В нем вызывается sqlite3\_prepare\_v2(), параметры запроса связываются с запросом.

Далее для выполнения запроса можно вызвать следующие методы, использующие приватный метод \_prepare():

1) **select()** – для запросов, которые возвращают данные. Для последующего перебора данных нужно использовать методы fetch\_row() и column\_count();

2) **execute()** – для управляющих запросов и запросов вставки-удаления, которые не возвращают данные;

3) **value()** – для запросов которые возвращают одно единственное значение и нам не нужен перебор данных;

diary.cpp: <https://github.com/rudenyel/sql-cpp/blob/main/diary.cpp>

В начале работы программы вводится имя файла с базой данных:

Если такой базы не сущнествует в текущей папке, то она создается и в ней появляется пустая таблица books. Если просто нажать ввод, то откроется база данных diary.db с тестовым набором данных. Далее отображается текстовое меню с набором команд, которые можно производить с дневником чтения.