C++ SQLite POS Relational CLI

Contents

Time required: 90 minutes

- Comment each line of code as show in the tutorials and other code examples.
- Follow all directions carefully and accurately.
- Think of the directions as minimum requirements.

SQL Tutorial

- https://www.w3schools.com/sql/sql_intro.asp
- https://www.w3schools.com/sql/sql_syntax.asp
- https://www.w3schools.com/sql/sql_create_db.asp
- https://www.w3schools.com/sql/sql_create_table.asp
- https://www.w3schools.com/sql/sql_drop_table.asp
- https://www.w3schools.com/sql/sql_insert.asp
- https://www.w3schools.com/sql/sql_update.asp
- https://www.w3schools.com/sql/sql_delete.asp
- https://www.w3schools.com/sql/sql_select.asp
- https://www.w3schools.com/sql/sql_in.asp
- https://www.w3schools.com/sql/sql_wildcards.asp
- *https://www.w3schools.com/sql/sql_join_inner.asp*

Entity Relationship Diagram Tutorials

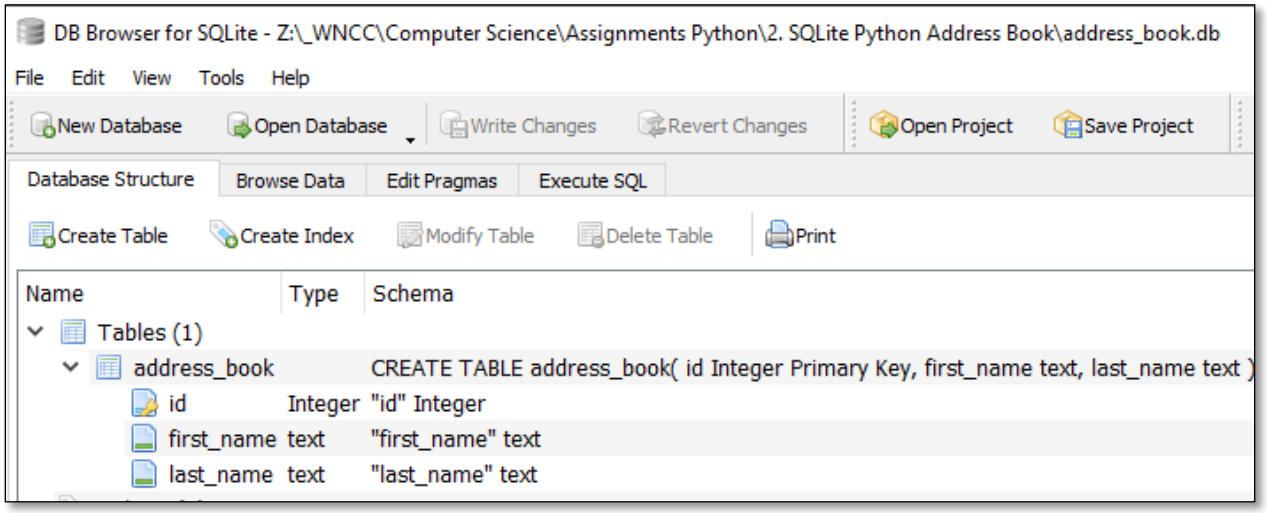
- https://www.tutorialspoint.com/dbms/er_model_basic_concepts.htm
- https://www.tutorialspoint.com/dbms/er_diagram_representation.htm
- https://www.lucidchart.com/pages/videos/entity-relationship-diagram-erd-tutorial-part-1

SQLite Database Browser

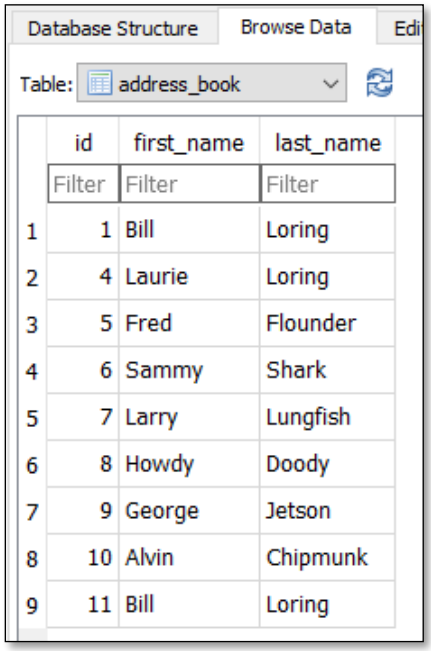
This is a handy tool to look at, troubleshoot, and manipulate your database.

1. Go to https://sqlitebrowser.org
2. Go to the **Download** tab.
3. Download the **Windows PortableApp → DB Browser for SQLite - PortableApp**
4. Double Click the installation file. Click **Next**.
5. Click **Install**. Click **Finish**.

6. You will find a new folder: **SQLiteDatabaseBrowserPortable**
7. This folder can be moved anywhere, the program will work just ifne.
8. In the folder you will find **SQLiteDatabaseBrowserPortable.exe**
9. Double Click the file. Click **OK** on the warning.
10. Use the **Open Database** button to open your database.



Click the **Browse Data** tab to see your records.



Click the **Close Database** button when you are done.

SQLite Relational Database

SQLite is a relational database. We create tables related by primary keys. We will design our databases using an ERD (Entity Relationship Diagram). www.lucidchart.com is free web-based diagram site used in these SQLite tutorials.

In this tutorial, we will create two related tables, then 3 related tables with a bridge/junction entity.

What is an ERD?

ERD: An Entity Relationship Diagram, also known as ERD, is a diagram that displays the relationship of entity sets stored in a database. ER diagrams help to explain the logical structure of databases. ER diagrams are created based on three basic concepts: entities (tables), attributes (fields), and relationships.

ER Diagrams contain different symbols that use rectangles to represent entities, ovals to define attributes and diamond shapes to represent relationships.

At first look, an ER diagram looks very similar to the flowchart. However, ER Diagram includes many specialized symbols, and its meanings make this model unique. The purpose of ER Diagram is to represent the entity framework infrastructure.

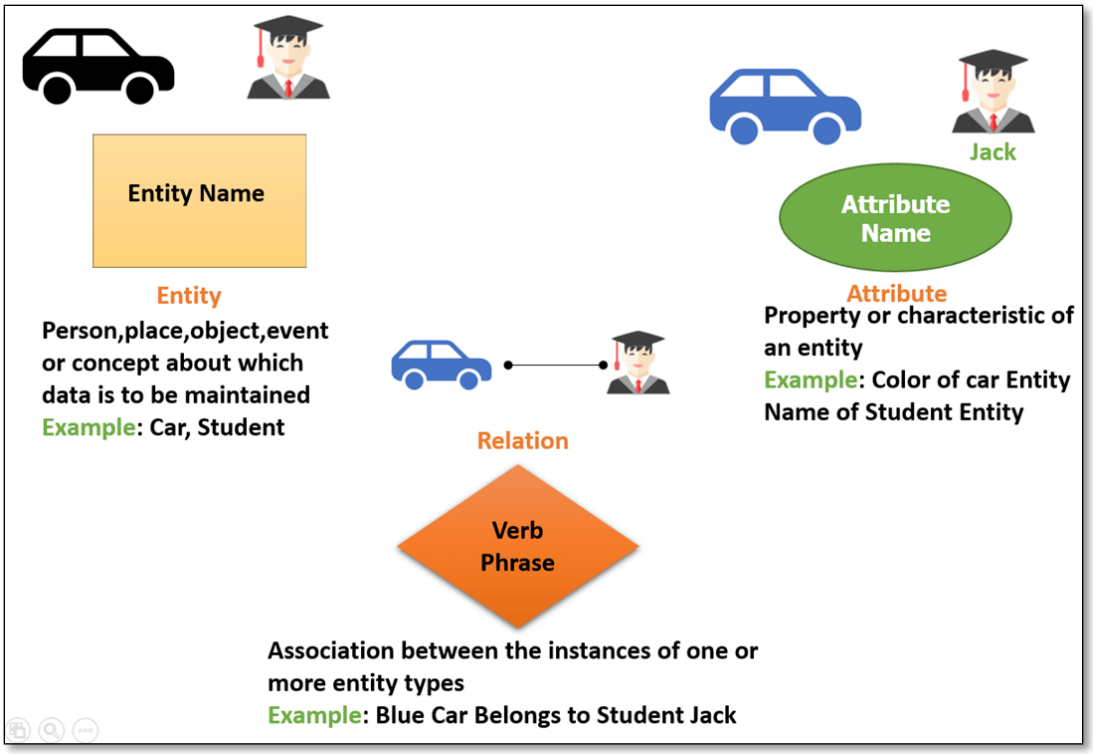
Components of the ER Diagram

This model is based on three basic concepts:

- Entities (Objects)
- Attributes (Properties)
- Relationships

ER Diagram Examples

For example, in a University database, we might have entities for Students, Courses, and Professors. The Student entity can have attributes like StudentID, Name, and DeptID. They might have relationships with Courses and Professors.



2-Table ERD with Bridge Entity

This is where database planning starts.

We have our entities. Like OOP, entities represent something in the real world we want to keep track of.

- Customer
- Product

Business Rules

Business rules are how the entities interact. A functional real-life customer sales tracking database would have these business rules.

- A product can be sold to many customers.
- A customer can purchase many products.

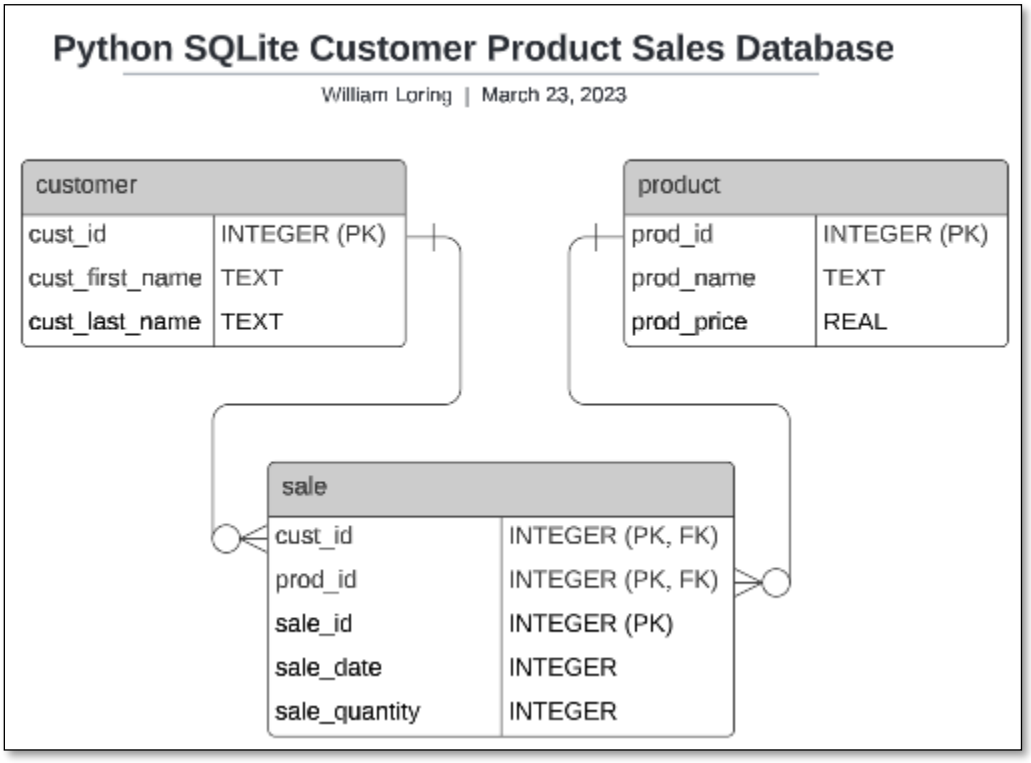
This is an example of many to many relationships. You can't have many to many relationships in SQL. You can have two tables with a bridge or junction table connecting them as shown below to implement the many to many business rules.

Primary Key: A primary key is a column or a set of columns in a table whose values uniquely identify a row in the table.

Foreign Key: A foreign key is a column or a set of columns in a table whose values correspond to the values of the primary key in another table.

Composite Key: A composite key is made by the combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness of a row is guaranteed, but when it is taken individually it does not guarantee uniqueness, or it can also be understood as a primary key made by the combination of two or more attributes to uniquely identify every row in a table.

These two tables are related through a composite primary key in the sale table. This composite primary key connects the two tables.



SQLite and C++

SQLite3 is the most-used SQL implementation in the world. It is a self-contained C library included in everything from portable devices to web browsers.

SQLite is a self-contained, file-based SQL database. SQLite does not come with C++. We will install a C library to work with SQLite.

Setup the Project: SQLite with C++

We need a C library to work with SQLite database. Download the sqlite-amalgamation(versionnumber).zip file from:

https://www.sqlite.org/download.html

Extract the contents of the ZIP file into your project folder. Only store this program in this folder. The compile batch will attempt to compile every file in the folder.

Tutorial 1: Connect to an SQLite Database

Please keep copies of each tutorial as you go in case something goes wrong. Use the name shown in the tutorial screenshot.

1. In your project folder, create a C++ file named: **Connect.cpp**
2. Insert the following code to create and/or connect to a database.

```cpp
/**
 * Filename: 1.Connect.cpp
 * Written by:
 * Written on:
 * Description: Connect to or create an SQLite3 database
 */

#include <iostream>
#include "sqlite3.h"

int main()
{
    // Pointer (memory reference) to SQLite connection
    sqlite3 *db;

    // Save the connection result
    int dbOpen = 0;

    // Save the result of opening the file
    // If there is a database file, open it,
    // otherwise create a new database file
    dbOpen = sqlite3_open("data.db", &db);

    // Was there an error opening the database file?
    if (dbOpen)
    {
        std::cout << " DB Open Error: " << sqlite3_errmsg(db) << std::endl;
    }
    else
    {
        std::cout << " Database Opened Successfully!" << std::endl;
    }
    // Close the connection
    sqlite3_close(db);
    return (0);
}
```

SQLite3 is an external library. To compile it correctly, it must also be linked to the project.

There are multiple files in this project, including the sqlite3 library files.

3. Create a batch file named **ConnectCompile.bat** with the following code.

```
rem Compile the sqlite3 c library with gcc. This only needs to be done once.
rem Place rem in front of the line to not compile it each time.
gcc sqlite3.c -c
rem This line compiles the sqlite3 library and the project file with g++
g++ Connect.cpp -o Connect.exe sqlite3.o -Isqlite3
pause
```

Run the batch file to compile the program.

Example run:

```
Opened Database Successfully!
```

4. You should see **data.db** in your project folder.

SQLite DataTypes

- **NULL:** The value is a NULL value.
- **INTEGER:** Store a whole number.
- **REAL:** Floating-point value, for example, 3.14, the value of PI.
- **TEXT:** A text string. TEXT value stored using UTF-8, UTF-16BE or UTF-16LE encoding.
- **BLOB:** The value is a blob of data, i.e., binary data. It is used to store images and files.

The following Python types convert to SQLite types.

C++ Types	SQLite Types
None	NULL
int	INTEGER
float	REAL
std::string	TEXT
bytes	BLOB

Creating Tables

In an SQL database, data is stored in tables. Tables define a set of columns and fields, much like a spreadsheet. They contain 0 or more rows with data for each of the defined fields.

Let’s create a table named **products** that tracks the following data. This is a data dictionary.

PROD_ID (primary key)	INTEGER
PROD_NAME	TEXT
PROD_PRICE	REAL
PROD_QTY	INTEGER

We will create a **products** table using SQLite3 in C++.

SQL is a scripting language like Python. You can assign SQL code to a string variable.

Tutorial 2: Create Tables

Working with SQLite3 includes three main functions: sqlite3_open(), sqlite3_exec(), and sqlite3_close().

In the previous part, the functions sqlite3open() and sqlite3_close() were used to open and close the connection.

To execute SQL, the function sqlite3_exec() is used.

---

sqlite3_exec()

Calls to sqlite3_exec() use five parameters:

- Database connection (SQLite3 pointer)
- SQL to run
- Callback function
- First argument to callback function
- Address of where to write error messages

Because SQL commands like SELECT can return multiple results, the callback function is used to act on them. The fourth parameter is optional when working with functions that may not need it.

To create a table within the existing database, the CREATE TABLE keywords are used in SQL.

Let’s create our tables. Add the following code to your program.

2.CreateTable.cpp > main()

```cpp
/**
 * Filename: 2.CreateTable.cpp
 * Written by:
 * Written on:
 * Description: Create a SQLite3 table
 */

#include <iostream>
#include <string>
#include "sqlite3.h"
// Function prototype for callback function
int callback(void *NotUsed, int argc, char **argv, char **azColName);

int main()
{
    // Pointer to SQLite connection
    sqlite3 *db;
    // Save any error messages
    char *zErrMsg = 0;
    // Save the connection result
    int dbOpen = 0;
    // Variable to store SQL commands
    std::string sql;
    // Save the result of opening the database file
    // If there is a database file, open it,
    // otherwise create a new database file
    dbOpen = sqlite3_open("data.db", &db);

    // Was there an error opening the database file?
    if (dbOpen)
    {
        // Show an error message
        std::cout << "DB Error: " << sqlite3_errmsg(db) << std::endl;
        // Close the connection
        sqlite3_close(db);
        // Return an error
        return (1);
    }
    std::cout << " Database Opened Successfully!" << std::endl;

    // ----------------- DROP TABLES -----------------------------------//
    // Drop tables for testing and development
    sql = "DROP TABLE IF EXISTS PRODUCT;";
    dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
    sql = "DROP TABLE IF EXISTS CUSTOMER;";
    dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
    sql = "DROP TABLE IF EXISTS SALE;";
    dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
```

```
50      // ---------------------- CREATE TABLES ----------------------------//
51      // SQL string to create a table
52      sql = "CREATE TABLE PRODUCT ("
53            "PROD_ID INTEGER PRIMARY KEY,"
54            "PROD_NAME      TEXT,"
55            "PROD_PRICE     REAL);";
56      // Create Product table
57      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
58      std::cout << " Product table created!" << std::endl;
59
60      sql = "CREATE TABLE CUSTOMER ("
61            "CUST_ID   INTEGER    PRIMARY KEY,"
62            "CUST_FNAME      TEXT,"
63            "CUST_LNAME      TEXT);";
64      // Create Customer table
65      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
66      std::cout << " Customer table Created!" << std::endl;
67
68      // To create a reference to cust_id and prod_id as foreign keys
69      // they must first exist in the table.
70      // All columns must exist before you can reference foreign keys
71      sql = "CREATE TABLE SALE ("
72            "SALE_ID   INTEGER  PRIMARY KEY, "
73            "CUST_ID   INTEGER, "
74            "PROD_ID   INTEGER, "
75            "SALE_QUANTITY  INTEGER, "
76            "SALE_PRICE  REAL); "
77            "FOREIGN KEY (CUST_ID) REFERENCES CUSTOMER(CUST_ID), "
78            "FOREIGN KEY (PROD_ID) REFERENCES PRODUCT(PROD_ID);";
79
80      // Run the SQL (convert the string to a C-String with c_str() )
81      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
82      std::cout << " Sale table created!" << std::endl;
83
84      // Close the SQL connection
85      sqlite3_close(db);
86      return 0;
87  }
88
89  // callback function for SQL operations
90  int callback(void *NotUsed, int argc, char **argv, char **azColName)
91  {
92      // Return successful
93      return 0;
94  }
```

Example run:

```
Database Opened Successfully!
Product table created!
Customer table Created!
Sale table created!
```

Tutorial 3: Insert Customer Record

Add the following to your program. Move the close connection to the end as shown.

```
77      // Run the SQL (convert the string to a C-String with c_str() )
78      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
79      std::cout << " Sale table created!" << std::endl;
80
81      // ----------------------- INSERT -----------------------------------//
82      // Save SQL insert data
83      sql = "INSERT INTO CUSTOMER (CUST_ID, CUST_FNAME, CUST_LNAME) "
84          "VALUES (0, 'William', 'Loring');";
85      // Run the SQL (convert the string to a C-String with c_str() )
86      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
87      std::cout << " Customer added. " << std::endl;
88
89      // Save SQL insert data
90      sql = "INSERT INTO CUSTOMER (CUST_ID, CUST_FNAME, CUST_LNAME) "
91          "VALUES (1, 'Wyatt', 'Earp');";
92      // Run the SQL (convert the string to a C-String with c_str() )
93      dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
94      std::cout << " Customer added. \n" << std::endl;
95
96      // ----------------------- SELECT -----------------------------------//
97      // Save SQL insert data
98      sql = "SELECT * FROM CUSTOMER;";
99      // Run the SQL (convert the string to a C-String with c_str() )
100     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
101
102     // Close the SQL connection
103     sqlite3_close(db);
104     return 0;
105 }
```

```
107  // callback function for SQL operations
108  int callback(void *NotUsed, int argc, char **argv, char **azColName)
109  {
110      // int argc: holds the number of results
111      // (array) azColName: holds each column returned
112      // (array) argv: holds each value
113
114      for (int i = 0; i < argc; i++)
115      {
116          // Show column name, value, and newline
117          std::cout << " " << azColName[i] << ": "
118                    << argv[i] << std::endl;
119      }
120      // Insert a newline
121      std::cout << std::endl;
122      // Return successful
123      return 0;
124  }
```

Example run:

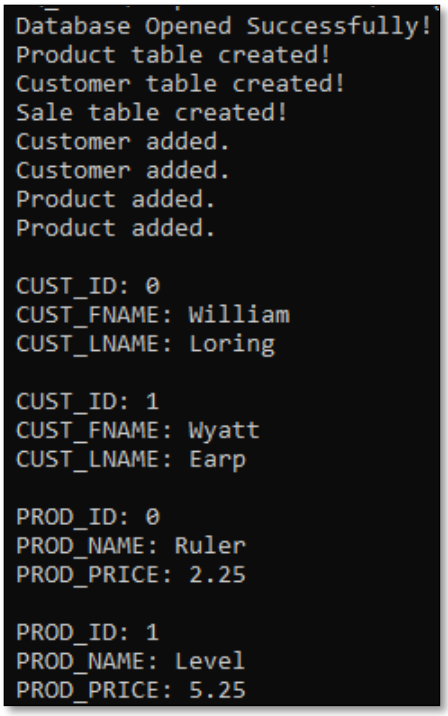
```
Database Opened Successfully!
Product table created!
Customer table created!
Sale table created!
Customer added.
Customer added.

CUST_ID: 0
CUST_FNAME: William
CUST_LNAME: Loring

CUST_ID: 1
CUST_FNAME: Wyatt
CUST_LNAME: Earp
```

Tutorial 4: Insert Product

Add the following to your program. Move the close statements to the end as shown. Leave the callback function at the end of the program.

```
96      // ------------------------ INSERT PRODUCT ---------------------------//
97      // Save SQL insert data
98      sql = "INSERT INTO PRODUCT (PROD_ID, PROD_NAME, PROD_PRICE) "
99            "VALUES (0, 'Ruler', 2.25);";
100     // Run the SQL (convert the string to a C-String with c_str() )
101     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
102     std::cout << " Product added. " << std::endl;
103
104     // Save SQL insert data
105     sql = "INSERT INTO PRODUCT (PROD_ID, PROD_NAME, PROD_PRICE) "
106           "VALUES (1, 'Level', 5.25);";
107     // Run the SQL (convert the string to a C-String with c_str() )
108     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
109     std::cout << " Product added. \n" << std::endl;
110
111     // ------------------------ SELECT CUSTOMER --------------------------//
112     // Save SQL insert data
113     sql = "SELECT * FROM CUSTOMER;";
114     // Run the SQL (convert the string to a C-String with c_str() )
115     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
116
117     // Save SQL insert data
118     sql = "SELECT * FROM PRODUCT;";
119     // Run the SQL (convert the string to a C-String with c_str() )
120     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
121
122     // Close the SQL connection
123     sqlite3_close(db);
124     return 0;
125 }
```

Example run:



Tutorial 5: Insert Sale Fetch Records

Add the following as before.

```
114     // ------------------------ INSERT SALE -------------------------------//
115     // Save SQL insert data
116     sql = "INSERT INTO SALE (SALE_ID, CUST_ID, PROD_ID, SALE_QUANTITY) "
117         "VALUES (0, 0, 0, 4);";
118     // Run the SQL (convert the string to a C-String with c_str() )
119     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
120     std::cout << " Sale added."
121             << std::endl;
122     // Save SQL insert data
123     sql = "INSERT INTO SALE (SALE_ID, CUST_ID, PROD_ID, SALE_QUANTITY) "
124         "VALUES (1, 1, 1, 5);";
125     // Run the SQL (convert the string to a C-String with c_str() )
126     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
127     std::cout << " Sale added. \n"
128             << std::endl;
129
130     // ------------------------ SELECT CUSTOMER --------------------------//
131     // Save SQL insert data
132     sql = "SELECT * FROM CUSTOMER;";
133     // Run the SQL (convert the string to a C-String with c_str() )
134     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
135
136     // ------------------------ SELECT PRODUCT ---------------------------//
137     // Save SQL insert data
138     sql = "SELECT * FROM TBL_PRODUCT;";
139     // Run the SQL (convert the string to a C-String with c_str() )
140     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
141
142     sql = "SELECT CUST.CUST_ID, CUST.CUST_FNAME, CUST.CUST_LNAME, "
143         "PROD.PROD_NAME, SALE.SALE_QUANTITY, PROD.PROD_PRICE, SALE.SALE_ID "
144         "FROM CUSTOMER CUST "
145         "INNER JOIN SALE "
146         "ON CUST.CUST_ID = SALE.CUST_ID "
147         "INNER JOIN PRODUCT PROD "
148         "ON PROD.PROD_ID = SALE.SALE_ID "
149         "ORDER BY CUST.CUST_LNAME ASC;";
150     // Run the SQL (convert the string to a C-String with c_str() )
151     dbOpen = sqlite3_exec(db, sql.c_str(), callback, 0, &zErrMsg);
152     //     std::cout << zErrMsg << std::endl;
153
154     // Close the SQL connection
155     sqlite3_close(db);
156     return 0;
157 }
```

Example run:

```
Database Opened Successfully
Product table created!
Customer table Created!
Sale table created!
Customer added.
Customer added.
Product added.
Product added.

Sale added.
Sale added.

CUST_ID: 0
CUST_FNAME: William
CUST_LNAME: Loring

CUST_ID: 1
CUST_FNAME: Wyatt
CUST_LNAME: Earp

CUST_ID: 1
CUST_FNAME: Wyatt
CUST_LNAME: Earp
PROD_NAME: Level
SALE_QUANTITY: 5
PROD_PRICE: 5.25
SALE_ID: 1

CUST_ID: 0
CUST_FNAME: William
CUST_LNAME: Loring
PROD_NAME: Ruler
SALE_QUANTITY: 4
PROD_PRICE: 2.25
SALE_ID: 0
```

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.