Home

SQLite C tutorial

This is a C programming tutorial for the SQLite database. It covers the basics of SQLite programming with the C language. You might also want to check the, SQLite tutorial, MySQL C tutorial, or PostgreSQL C tutorial on ZetCode.

SQLite database

SQLite is an embedded relational database engine. Its developers call it a self-contained, serverless, zero-configuration, and transactional SQL database engine. It is currently very popular and there are hundreds of millions copies worldwide in use today. SQLite is used in the Solaris 10, Mac OS, Android, or in the iPhone. The Qt4 library has built-in support for SQLite as well as the Python and PHP. Many popular applications use SQLite internally such as Firefox, Google Chrome, or Amarok.

The sqlite3 tool

The *sqlite3* tool is a terminal based frontend to the SQLite library. It evaluates queries interactively and displays the results in multiple formats. It can also be used within scripts. It has its own set of meta commands including `.tables`, `.load`, `.databases`, or `.dump`. To get the list of all instructions, we type the `.help` command.

Now we are going to use the `sqlite3` tool to create a new database.

```
$ sqlite3 test.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

We provide a parameter to the `sqlite3 tool`; `test.db` is a database name. It is a file on our disk. If it is present, it is opened. If not, it is created.

```
sqlite> .tables
sqlite> .exit
$ ls
test.db
```

The `.tables` command gives a list of tables in the `test.db` database. There are currently no tables. The `.exit` command terminates the interactive session of the `sqlite3` command line tool. The `ls` Unix command shows the contents of the current working directory. We can see the `test.db` file. All data will be stored in this single file.

C99

This tutorial uses C99. For GNU C compiler, we need to use the `-std=c99` option. For Windows users, the Pelles C IDE is highly recommended. (MSVC does not support C99.)

```
int rc = sqlite3_open("test.db", &db);
```

In C99, we can mix declarations with code. In older C programs, we would need to separate this line into two lines.

SQLite version

In the first code example, we will get the version of the SQLite database.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    printf("%s\n", sqlite3_libversion());

    return 0;
}
```

The `sqlite3_libversion()` function returns a string indicating the SQLite library.

```
#include <sqlite3.h>
```

This header file defines the interface that the SQLite library presents to the client programs. It contains definitions, function prototypes, and comments. It is an authoritative source for SQLite API.

```
$ gcc -o version version.c -lsqlite3 -std=c99
```

We compile the program with the GNU C compiler.

```
$ ./version
3.8.2
```

This is the output of the example.

In the second example, we again get the version of the SQLite database. This time we will use an SQL query.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    sqlite3_stmt *res;

    int rc = sqlite3_open(":memory:", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    rc = sqlite3_prepare_v2(db, "SELECT SQLITE_VERSION()", -1, &res, 0);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Failed to fetch data: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    rc = sqlite3_step(res);

    if (rc == SQLITE_ROW) {
        printf("%s\n", sqlite3_column_text(res, 0));
    }

    sqlite3_finalize(res);
    sqlite3_close(db);

    return 0;
}
```

The `SQLITE_VERSION()` query is used to get the version of the SQLite library.

```
sqlite3 *db;
```

The `sqlite3` structure defines a database handle. Each open SQLite database is represented by a database handle.

```
sqlite3_stmt *res;
```

The `sqlite3_stmt` structure represents a single SQL statement.

```
int rc = sqlite3_open(":memory:", &db);
```

The `sqlite3_open()` function opens a new database connection. Its parameters are the database name and the database handle. The `:memory:` is a special database name using which results in opening an in-memory database. The function's return code indicates whether the database was successfully opened. The `SQLITE_OK` is returned when the connection was successfully established.

```
if (rc != SQLITE_OK) {

    fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);

    return 1;
}
```

If the return code indicates an error, we print the message to the console, close the database handle, and terminate the program. The `sqlite3_errmsg()` function returns a description of the error. Whether or not an error occurs when it is opened, resources associated with the database connection handle should be released by passing it to `sqlite3_close()` function.

```
rc = sqlite3_prepare_v2(db, "SELECT SQLITE_VERSION()", -1, &res, 0);
```

Before an SQL statement is executed, it must be first compiled into a byte-code with one of the sqlite3_prepare* functions. (The `sqlite3_prepare()` function is deprecated.)

The `sqlite3_prepare_v2()` function takes five parameters. The first parameter is the database handle obtained from the `sqlite3_open()` function. The second parameter is the SQL statement to be compiled. The third parameter is the maximum length of the SQL statement measured in bytes. Passing -1 causes the SQL string to be read up to the first zero terminator which is the end of the string here. According to the documentation, it is possible to gain some small performance advantage by passing the exact number of bytes of the supplied SQL string. The fourth parameter is the statement handle. It will point to the precompiled statement if the `sqlite3_prepare_v2()` runs successfully. The last parameter is a pointer to the unused portion of the SQL statement. Only the first statement of the SQL string is compiled, so the parameter points to what is left uncompiled. We pass 0 since the parameter is not important for us.

On success, `sqlite3_prepare_v2()` returns `SQLITE_OK`; otherwise an error code is returned.

```
if (rc != SQLITE_OK) {

    fprintf(stderr, "Failed to fetch data: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);

    return 1;
}
```

This is error handling code for the `sqlite3_prepare_v2()` function call.

```
rc = sqlite3_step(res);
```

The `sqlite3_step()` runs the SQL statement. `SQLITE_ROW` return code indicates that there is another row ready. Our SQL statement returns only one row of data, therefore, we call this function only once.

```
sqlite3_finalize(res);
```

The `sqlite3_finalize()` function destroys the prepared statement object.

```
sqlite3_close(db);
```

The `sqlite3_close()` function closes the database connection.

Inserting data

We create a `Cars` table and insert several rows to it.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "DROP TABLE IF EXISTS Cars;"
                "CREATE TABLE Cars(Id INT, Name TEXT, Price INT);"
                "INSERT INTO Cars VALUES(1, 'Audi', 52642);"
                "INSERT INTO Cars VALUES(2, 'Mercedes', 57127);"
                "INSERT INTO Cars VALUES(3, 'Skoda', 9000);"
                "INSERT INTO Cars VALUES(4, 'Volvo', 29000);"
                "INSERT INTO Cars VALUES(5, 'Bentley', 350000);"
                "INSERT INTO Cars VALUES(6, 'Citroen', 21000);"
                "INSERT INTO Cars VALUES(7, 'Hummer', 41400);"
                "INSERT INTO Cars VALUES(8, 'Volkswagen', 21600);";

    rc = sqlite3_exec(db, sql, 0, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "SQL error: %s\n", err_msg);

        sqlite3_free(err_msg);
        sqlite3_close(db);

        return 1;
    }

    sqlite3_close(db);

    return 0;
}
```

We connect to the `test.db` database, create a `Cars` table, and insert 8 rows into the created table.

```
char *err_msg = 0;
```

If an error occurs, this pointer will point a the created error message.

```
int rc = sqlite3_open("test.db", &db);
```

The connection to the `test.db` database is created.

```
char *sql = "DROP TABLE IF EXISTS Cars;"
            "CREATE TABLE Cars(Id INT, Name TEXT, Price INT);"
            "INSERT INTO Cars VALUES(1, 'Audi', 52642);"
```

```
             "INSERT INTO Cars VALUES(2, 'Mercedes', 57127);"
             "INSERT INTO Cars VALUES(3, 'Skoda', 9000);"
             "INSERT INTO Cars VALUES(4, 'Volvo', 29000);"
             "INSERT INTO Cars VALUES(5, 'Bentley', 350000);"
             "INSERT INTO Cars VALUES(6, 'Citroen', 21000);"
             "INSERT INTO Cars VALUES(7, 'Hummer', 41400);"
             "INSERT INTO Cars VALUES(8, 'Volkswagen', 21600);";
```

These SQL statements create a `Cars` table and fill it with data. The statements must be separated by semicolons.

```
rc = sqlite3_exec(db, sql, 0, 0, &err_msg);
```

The `sqlite3_exec()` function is a convenience wrapper around `sqlite3_prepare_v2()`, `sqlite3_step()`, and `sqlite3_finalize()` that allows an application to run multiple statements of SQL without having to use a lot of C code.

The function's third parameter is a callback function invoked for each result row coming out of the evaluated SQL statement. The fourth parameter is the first parameter to the callback function. If we do not need them, we can pass 0 to these parameters.

If an error occurs then the last parameter points to the allocated error message.

```
sqlite3_free(err_msg);
```

The allocated message string must be freed with the `sqlite3_free()` function call.

```
sqlite> .mode column
sqlite> .headers on
```

We verify the written data with the `sqlite3` tool. First we modify the way the data is displayed in the console. We use the column mode and turn on the headers.

```
sqlite> SELECT * FROM Cars;
Id          Name        Price
----------  ----------  ----------
1           Audi        52642
2           Mercedes    57127
3           Skoda       9000
4           Volvo       29000
5           Bentley     350000
6           Citroen     21000
7           Hummer      41400
8           Volkswagen  21600
```

This is the data that we have written to the `Cars` table.

The last inserted row id

Sometimes, we need to determine the id of the last inserted row. For this, we have the `sqlite3_last_insert_rowid()` function.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open(":memory:", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
    "INSERT INTO Friends(Name) VALUES ('Tom');"
    "INSERT INTO Friends(Name) VALUES ('Rebecca');"
    "INSERT INTO Friends(Name) VALUES ('Jim');"
    "INSERT INTO Friends(Name) VALUES ('Roger');"
    "INSERT INTO Friends(Name) VALUES ('Robert');";


    rc = sqlite3_exec(db, sql, 0, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "Failed to create table\n");
        fprintf(stderr, "SQL error: %s\n", err_msg);
        sqlite3_free(err_msg);

    } else {

        fprintf(stdout, "Table Friends created successfully\n");
    }

    int last_id = sqlite3_last_insert_rowid(db);
    printf("The last Id of the inserted row is %d\n", last_id);

    sqlite3_close(db);

    return 0;
}
```

A `Friends` table is created in memory. Its `Id` column is automatically incremented.

```
char *sql = "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
"INSERT INTO Friends(Name) VALUES ('Tom');"
"INSERT INTO Friends(Name) VALUES ('Rebecca');"
"INSERT INTO Friends(Name) VALUES ('Jim');"
"INSERT INTO Friends(Name) VALUES ('Roger');"
"INSERT INTO Friends(Name) VALUES ('Robert');";
```

In SQLite, `INTEGER PRIMARY KEY` column is auto-incremented. There is also an `AUTOINCREMENT` keyword. When applied in `INTEGER PRIMARY KEY AUTOINCREMENT` a slightly different algorithm for Id creation is used.

When using auto-incremented columns, we need to explicitly state the column names except for the auto-incremented column, which is omitted.

```
int last_id = sqlite3_last_insert_rowid(db);
printf("The last Id of the inserted row is %d\n", last_id);
```

The `sqlite3_last_insert_rowid()` returns the row Id of the most recent successfull insert into the table.

```
$ ./last_row_id
Table Friends created successfully
The last Id of the inserted row is 5
```

We see the output of the program.

Retrieving data

We have inserted some data into the `test.db` database. In the following example, we retrieve the data from the database.

```
#include <sqlite3.h>
#include <stdio.h>
```

```
int callback(void *, int, char **, char **);


int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n",
                sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "SELECT * FROM Cars";

    rc = sqlite3_exec(db, sql, callback, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "Failed to select data\n");
        fprintf(stderr, "SQL error: %s\n", err_msg);

        sqlite3_free(err_msg);
        sqlite3_close(db);

        return 1;
    }

    sqlite3_close(db);

    return 0;
}

int callback(void *NotUsed, int argc, char **argv,
                    char **azColName) {

    NotUsed = 0;

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

        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }

    printf("\n");

    return 0;
}
```

We use the `SELECT * FROM Cars` SQL statement to retrieve all rows from the `Cars` table.

```
int callback(void *, int, char **, char **);
```

This is a function prototype for the callback function that is used in conjunction with the `sqlite3_exec()` function.

```
int rc = sqlite3_open("test.db", &db);
```

We connect to the `test.db` database.

```
char *sql = "SELECT * FROM Cars";
```

Here we define the SQL statement to select all data from the `Cars` table.

```
rc = sqlite3_exec(db, sql, callback, 0, &err_msg);
```

The `sqlite3_exec()` function evalues the SQL statement. Its callback function is invoked for each result row coming out of the evaluated SQL statement.

```
int callback(void *NotUsed, int argc, char **argv,
                    char **azColName) {

    NotUsed = 0;

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

        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }

    printf("\n");

    return 0;
}
```

The first parameter of the callback function is data provided in the 4th argument of `sqlite3_exec()`; it is often not used. The second parameter is the number of columns in the result. The third parameter is an array of strings representing fields in the row. The last parameter is array of strings representing column names.

In the function body, we go through all columns and print their names and content.

```
$ ./select_all
Id = 1
Name = Audi
Price = 52642

Id = 2
Name = Mercedes
Price = 57127

Id = 3
Name = Skoda
Price = 9000
...
```

This is a partial output of the example.

Parameterized queries

Now we will mention parameterized queries. Parameterized queries, also called prepared statements, increase security and performance. When we use parameterized queries, we use placeholders instead of directly writing the values into the statements.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;
    sqlite3_stmt *res;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "SELECT Id, Name FROM Cars WHERE Id = ?";
    rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

    if (rc == SQLITE_OK) {

        sqlite3_bind_int(res, 1, 3);
    } else {

        fprintf(stderr, "Failed to execute statement: %s\n", sqlite3_errmsg(db));
    }

    int step = sqlite3_step(res);

    if (step == SQLITE_ROW) {

        printf("%s: ", sqlite3_column_text(res, 0));
        printf("%s\n", sqlite3_column_text(res, 1));

    }

    sqlite3_finalize(res);
    sqlite3_close(db);

    return 0;
}
```

In the example, a question mark (?) is used as a placeholder which is later replaced with an actual value.

```
char *sql = "SELECT Id, Name FROM Cars WHERE Id = ?";
```

The question mark is used to provide an Id to the SQL query.

```
rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);
```

The `sqlite3_prepare_v2()` function compiles the SQL query.

```
sqlite3_bind_int(res, 1, 3);
```

The `sqlite3_bind_int()` binds an integer value to the prepared statement. The placeholder is replaced with integer value 3. The function's second parameter is the index of the SQL parameter to be set and the third parameter is the value to bind to the parameter.

```
int step = sqlite3_step(res);
```

The `sqlite3_step()` function evaluates the SQL statement.

```
if (step == SQLITE_ROW) {

    printf("%s: ", sqlite3_column_text(res, 0));
    printf("%s\n", sqlite3_column_text(res, 1));

}
```

If there is some row of data available, we get the values of two columns with the `sqlite3_column_text()` function.

```
$ ./parameterized
3: Skoda
```

The example returns the Id and the car's name.

The second example uses parameterized statements with named placeholders.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;
    sqlite3_stmt *res;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "SELECT Id, Name FROM Cars WHERE Id = @id";

    rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

    if (rc == SQLITE_OK) {

        int idx = sqlite3_bind_parameter_index(res, "@id");
        int value = 4;
        sqlite3_bind_int(res, idx, value);

    } else {

        fprintf(stderr, "Failed to execute statement: %s\n", sqlite3_errmsg(db));
    }

    int step = sqlite3_step(res);

    if (step == SQLITE_ROW) {

        printf("%s: ", sqlite3_column_text(res, 0));
        printf("%s\n", sqlite3_column_text(res, 1));

    }

    sqlite3_finalize(res);
    sqlite3_close(db);

    return 0;
}
```

We select the name and the price of a car using named placeholders.

```
char *sql = "SELECT Id, Name FROM Cars WHERE Id = @id";
```

Named placeholders are prefixed with the colon (:) character or the at-sign (@) character.

```
int idx = sqlite3_bind_parameter_index(res, "@id");
```

The `sqlite3_bind_parameter_index()` function returns the index of an SQL parameter given its name.

Inserting images

In this section, we are going to insert an image to the SQLite database. Note that some people argue against putting images into databases. Here we only show how to do it. We do not dwell on the technical issues of whether to save images in databases or not.

```
sqlite> CREATE TABLE Images(Id INTEGER PRIMARY KEY, Data BLOB);
```

For this example, we create a new table called Images. For the images, we use the `BLOB` data type, which stands for Binary Large Objects.

```
#include <sqlite3.h>
#include <stdio.h>


int main(int argc, char **argv) {

    FILE *fp = fopen("woman.jpg", "rb");
    if (fp == NULL) {

        fprintf(stderr, "Cannot open image file\n");

        return 1;
    }

    fseek(fp, 0, SEEK_END);

    if (ferror(fp)) {

        fprintf(stderr, "fseek() failed\n");
        int r = fclose(fp);

        if (r == EOF) {
            fprintf(stderr, "Cannot close file handler\n");
        }

        return 1;
    }

    int flen = ftell(fp);

    if (flen == -1) {

        perror("error occurred");
        int r = fclose(fp);

        if (r == EOF) {
            fprintf(stderr, "Cannot close file handler\n");
        }

        return 1;
    }

    fseek(fp, 0, SEEK_SET);

    if (ferror(fp)) {

        fprintf(stderr, "fseek() failed\n");
        int r = fclose(fp);

        if (r == EOF) {
            fprintf(stderr, "Cannot close file handler\n");
        }

        return 1;
    }

    char data[flen+1];

    int size = fread(data, 1, flen, fp);

    if (ferror(fp)) {

        fprintf(stderr, "fread() failed\n");
        int r = fclose(fp);

        if (r == EOF) {
            fprintf(stderr, "Cannot close file handler\n");
        }

        return 1;
    }

    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "Cannot close file handler\n");
    }


    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }


    sqlite3_stmt *pStmt;

    char *sql = "INSERT INTO Images(Data) VALUES(?)";

    rc = sqlite3_prepare(db, sql, -1, &pStmt, 0);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot prepare statement: %s\n", sqlite3_errmsg(db));

        return 1;
    }

    sqlite3_bind_blob(pStmt, 1, data, size, SQLITE_STATIC);

    rc = sqlite3_step(pStmt);

    if (rc != SQLITE_DONE) {

        printf("execution failed: %s", sqlite3_errmsg(db));
    }

    sqlite3_finalize(pStmt);

    sqlite3_close(db);

    return 0;
}
```

In this program, we read an image from the current working directory and write it into the `Images` table of the SQLite `test.db` database.

```
FILE *fp = fopen("woman.jpg", "rb");

if (fp == NULL) {

    fprintf(stderr, "Cannot open image file\n");

    return 1;
}
```

We read binary data from the filesystem. We have a JPG image called `woman.jpg`. The `fopen()` function opens the specified file for for reading. It returns a pointer to a FILE object or NULL if the operation fails.

```
fseek(fp, 0, SEEK_END);

if (ferror(fp)) {

    fprintf(stderr, "fseek() failed\n");
    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "Cannot close file handler\n");
    }

    return 1;
}
```

We move the file pointer to the end of the file using the `fseek()` function. We need to determine the size of the image. If an error occurs, the error indicator is set. We check the indicator using the `fseek()` function. In case of an error, the opened file handler is closed.

```
int flen = ftell(fp);
```

```
if (flen == -1) {

    perror("error occurred");
    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "Cannot close file handler\n");
    }

    return 1;
}
```

For binary streams, the `ftell()` function returns the number of bytes from the beginning of the file, e.g. the size of the image file. In case of an error, the function returns `-1` and the `errno` is set. The `perror()` function interprets the value of errno as an error message, and prints it to the standard error output stream.

```
char data[flen+1];
```

This array will store the image data.

```
int size = fread(data, 1, flen, fp);
```

The `fread()` function reads the data from the file pointer and stores it in the data array. The function returns the number of elements successfully read.

```
int r = fclose(fp);

if (r == EOF) {
    fprintf(stderr, "Cannot close file handler\n");
}
```

After the data is read, we can close the file handler.

```
char *sql = "INSERT INTO Images(Data) VALUES(?)";
```

This SQL statement is used to insert the image into the database.

```
rc = sqlite3_prepare(db, sql, -1, &pStmt, 0);
```

The SQL statement is compiled.

```
sqlite3_bind_blob(pStmt, 1, data, size, SQLITE_STATIC);
```

The `sqlite3_bind_blob()` function binds the binary data to the compiled statement. The `SQLITE_STATIC` parameter means that the pointer to the content information is static and does not need to be freed.

```
rc = sqlite3_step(pStmt);
```

The statement is executed and the image is written to the table.

Reading images

In this section, we are going to perform the reverse operation. We will read an image from the database table.

```
#include <sqlite3.h>
#include <stdio.h>


int main(void) {

    FILE *fp = fopen("woman2.jpg", "wb");

    if (fp == NULL) {

        fprintf(stderr, "Cannot open image file\n");

        return 1;
    }

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "SELECT Data FROM Images WHERE Id = 1";

    sqlite3_stmt *pStmt;
    rc = sqlite3_prepare_v2(db, sql, -1, &pStmt, 0);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "Failed to prepare statement\n");
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));

        sqlite3_close(db);

        return 1;
    }

    rc = sqlite3_step(pStmt);

    int bytes = 0;

    if (rc == SQLITE_ROW) {

        bytes = sqlite3_column_bytes(pStmt, 0);
    }

    fwrite(sqlite3_column_blob(pStmt, 0), bytes, 1, fp);

    if (ferror(fp)) {

        fprintf(stderr, "fwrite() failed\n");

        return 1;
    }

    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "Cannot close file handler\n");
    }

    rc = sqlite3_finalize(pStmt);

    sqlite3_close(db);

    return 0;
}
```

We read image data from the `Images` table and write it to another file, which we call `woman2.jpg`.

```
FILE *fp = fopen("woman2.jpg", "wb");

if (fp == NULL) {

    fprintf(stderr, "Cannot open image file\n");

    return 1;
}
```

We open a binary file in a writing mode. The data from the database is written to the file.

```
char *sql = "SELECT Data FROM Images WHERE Id = 1";
```

This SQL statement selects data from the Images table. We obtain the binary data from the first row.

```
if (rc == SQLITE_ROW) {

    bytes = sqlite3_column_bytes(pStmt, 0);
}
```

The `sqlite3_column_bytes()` function returns the number of bytes in the BLOB.

```
fwrite(sqlite3_column_blob(pStmt, 0), bytes, 1, fp);
```

The binary data is written to the file with the `fwrite()` function. The pointer to the selected binary data is returned by the `sqlite3_column_blob()` function.

```
if (ferror(fp)) {

    fprintf(stderr, "fwrite() failed\n");

    return 1;
}
```

The `ferror()` function checks if the error indicator associated with the stream is set.

Metadata

Metadata is information about the data in the database. Metadata in a SQLite contains information about the tables and columns, in which we store data. Number of rows affected by an SQL statement is a metadata. Number of rows and columns returned in a result set belong to metadata as well.

Metadata in SQLite can be obtained using the `PRAGMA` command. SQLite objects may have attributes, which are metadata. Finally, we can also obtain specific metatada from querying the SQLite system `sqlite_master` table.

```
#include <sqlite3.h>
#include <stdio.h>

int callback(void *, int, char **, char **);

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n",
                sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "PRAGMA table_info(Cars)";

    rc = sqlite3_exec(db, sql, callback, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "Failed to select data\n");
        fprintf(stderr, "SQL error: %s\n", err_msg);
        sqlite3_free(err_msg);
        sqlite3_close(db);

        return 1;
    }

    sqlite3_close(db);

    return 0;
}

int callback(void *NotUsed, int argc, char **argv,
                    char **azColName) {

    NotUsed = 0;

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

        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }

    printf("\n");

    return 0;
}
```

In this example, we issue the `PRAGMA table_info(tableName)` command, to get some metadata info about our `Cars` table.

```
char *sql = "PRAGMA table_info(Cars)";
```

The `PRAGMA table_info(tableName)` command returns one row for each column in the `Cars` table. Columns in the result set include the column order number, column name, data type, whether or not the column can be `NULL`, and the default value for the column.

```
$ ./column_names
cid = 0
name = Id
type = INT
notnull = 0
dflt_value = NULL
pk = 0
...
```

This is output of the example.

In the next example related to the metadata, we will list all tables in the `test.db` database.

```
#include <sqlite3.h>
#include <stdio.h>


int callback(void *, int, char **, char **);


int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n",
                sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }
    char *sql = "SELECT name FROM sqlite_master WHERE type='table'";

    rc = sqlite3_exec(db, sql, callback, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "Failed to select data\n");
        fprintf(stderr, "SQL error: %s\n", err_msg);

        sqlite3_free(err_msg);
        sqlite3_close(db);

        return 1;
    }

    sqlite3_close(db);

    return 0;
}

int callback(void *NotUsed, int argc, char **argv,
                    char **azColName) {

    NotUsed = 0;

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

        printf("%s\n", argv[i] ? argv[i] : "NULL");
    }

    return 0;
}
```

The code example prints all available tables in the current database to the terminal.

```
char *sql = "SELECT name FROM sqlite_master WHERE type='table'";
```

The table names are stored inside the system `sqlite_master` table.

```
$ ./list_tables
Cars
Images
```

This is a sample output.

Transactions

A transaction is an atomic unit of database operations against the data in one or more databases. The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

In SQLite, any command other than the `SELECT` will start an implicit transaction. Also, within a transaction a command like `CREATE TABLE` ..., `VACUUM`, `PRAGMA`, will commit previous changes before executing.

Manual transactions are started with the `BEGIN TRANSACTION` statement and finished with the `COMMIT` or `ROLLBACK` statements.

SQLite supports three non-standard transaction levels: `DEFERRED`, `IMMEDIATE` and `EXCLUSIVE`.

Autocommit

By default, SQLite version 3 operates in *autocommit mode*. In autocommit mode, all changes to the database are committed as soon as all operations associated with the current database connection complete. Autocommit mode is disabled by a `BEGIN` statement and re-enabled by a `COMMIT` or `ROLLBACK`.

```
#include <sqlite3.h>
```

```
#include <stdio.h>

int main() {

    sqlite3 *db;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n",
                sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    printf("Autocommit: %d\n", sqlite3_get_autocommit(db));

    sqlite3_close(db);

    return 0;
}
```

This example check whether the database is in the autocommit mode.

```
printf("Autocommit: %d\n", sqlite3_get_autocommit(db));
```

The `sqlite3_get_autocommit()` function returns zero if the database is not in the autocommit mode. It returns non-zero if it is in the autocommit mode.

```
$ ./get_ac_mode
Autocommit: 1
```

The example confirms that SQLite is in the autocommit mode by default.

The next example further clarifies the autocommit mode. In the autocommit mode, each non-SELECT statement is a small transaction that is immediately committed.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "DROP TABLE IF EXISTS Friends;"
                "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
                "INSERT INTO Friends(Name) VALUES ('Tom');"
                "INSERT INTO Friends(Name) VALUES ('Rebecca');"
                "INSERT INTO Friends(Name) VALUES ('Jim');"
                "INSERT INTO Friend(Name) VALUES ('Robert');";

    rc = sqlite3_exec(db, sql, 0, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "SQL error: %s\n", err_msg);

        sqlite3_free(err_msg);
        sqlite3_close(db);

        return 1;
    }

    sqlite3_close(db);

    return 0;
}
```

We create the `Friends` table and try to fill it with data.

```
char *sql = "DROP TABLE IF EXISTS Friends;"
            "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
            "INSERT INTO Friends(Name) VALUES ('Tom');"
            "INSERT INTO Friends(Name) VALUES ('Rebecca');"
            "INSERT INTO Friends(Name) VALUES ('Jim');"
            "INSERT INTO Friend(Name) VALUES ('Robert');";
```

The last SQL statement has an error; there is no Friend table.

```
$ ./autocommit
SQL error: no such table: Friend
$ sqlite3 test.db
sqlite> .tables
Cars     Friends  Images
sqlite> SELECT * FROM Friends;
1|Tom
2|Rebecca
3|Jim
```

The table is created and three rows are inserted.

Transaction

In the next example, we put some SQL statements within a transaction.

```
#include <sqlite3.h>
#include <stdio.h>

int main(void) {

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("test.db", &db);

    if (rc != SQLITE_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    char *sql = "DROP TABLE IF EXISTS Friends;"
                "BEGIN TRANSACTION;"
                "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
                "INSERT INTO Friends(Name) VALUES ('Tom');"
                "INSERT INTO Friends(Name) VALUES ('Rebecca');"
                "INSERT INTO Friends(Name) VALUES ('Jim');"
                "INSERT INTO Friend(Name) VALUES ('Robert');"
                "COMMIT;";


    rc = sqlite3_exec(db, sql, 0, 0, &err_msg);

    if (rc != SQLITE_OK ) {

        fprintf(stderr, "SQL error: %s\n", err_msg);

        sqlite3_free(err_msg);
        sqlite3_close(db);
        return 1;
    }


    sqlite3_close(db);

    return 0;
}
```

We continue working with the `Friends` table.

```
char *sql = "DROP TABLE IF EXISTS Friends;"
            "BEGIN TRANSACTION;"
            "CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);"
            "INSERT INTO Friends(Name) VALUES ('Tom');"
            "INSERT INTO Friends(Name) VALUES ('Rebecca');"
            "INSERT INTO Friends(Name) VALUES ('Jim');"
            "INSERT INTO Friend(Name) VALUES ('Robert');"
            "COMMIT;";
```

The first statement drops the `Friends` table if it exists. The other statements are placed within a transaction. Transactions work in an all-or-nothing mode. Either everything or nothing is committed.

```
sqlite> .tables
Cars     Images
```

Since the last statement had an error, the transaction was rolled back and the `Friends` table was not created.

Sources

The SQLite documentation was used to create this tutorial.

This was SQLite C tutorial. ZetCode has a complete *e-book* for SQLite Python:
SQLite Python e-book.