ChatGPT

SQLite in C: Comprehensive Tutorial

Overview of SQLite in C

SQLite is an embedded **C-language** library that implements a **self-contained, fast, reliable SQL database engine** [1]. It is **serverless** (no separate server process) and **file-based**, making it ideal for local or embedded data storage. SQLite emphasizes simplicity and independence over networked scalability [2]. For example, it is built into virtually all mobile phones and many embedded devices, and is widely used for desktop or IoT applications [1] [2]. Use SQLite when you need a lightweight database for a single-application or low to medium traffic context. (Client/server databases like MySQL or PostgreSQL are better when you need heavy concurrency or large shared datasets [2].)

Installing and Linking SQLite in C

To use SQLite in a C project, you need the **SQLite library and headers**. On Linux, install the development package (e.g. `sudo apt-get install libsqlite3-dev`). Include the header `<sqlite3.h>` in your C code. When compiling, link against the SQLite library. For example:

```
gcc -o myprogram myprogram.c -lsqlite3
```

This tells GCC to link `-lsqlite3`, making SQLite functions available [3]. (On Windows, you can compile the amalgamation: include `sqlite3.c` and `sqlite3.h` in your project, or link against the SQLite DLL.)

Example:

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

int main(void) {
    sqlite3 *db;
    int rc = sqlite3_open("test.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        return 1;
    }
    printf("Database opened successfully\n");
    sqlite3_close(db);
    return 0;
}
```

Compiling with `gcc program.c -lsqlite3` will create and open `test.db` if it doesn't exist [3]. Always link with `-lsqlite3` and include the header to use SQLite API functions.

Using `sqlite3_exec` for Queries

The function `sqlite3_exec` is a **convenience wrapper** that prepares and executes one or more SQL statements in a single call [4] [5]. Its signature is:

```
int sqlite3_exec(
    sqlite3 *db,                    /* An open database */
    const char *sql,                /* SQL to be evaluated */
    int (*callback)(void*,int,char**,char**), /* Callback for each row */
    void *callback_arg,             /* First arg to callback */
    char **errmsg                   /* Error message (out) */
);
```

Under the hood, `sqlite3_exec` calls `sqlite3_prepare_v2()`, `sqlite3_step()`, `sqlite3_column_*()`, and `sqlite3_finalize()` for you [4] [5]. If the callback pointer is not NULL, it is invoked for each row of the result set [5].

Example of `sqlite3_exec`

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

static int callback(void *NotUsed, int argc, char **argv, char **azColName) {
    // Print each column of the row
    for(int i = 0; i < argc; i++) {
        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }
    printf("\n");
    return 0;
}

int main(void) {
    sqlite3 *db;
    char *err_msg = 0;

    // Open (or create) the database
    int rc = sqlite3_open("test.db", &db);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "Cannot open database: %s\n", sqlite3_errmsg(db));
        return 1;
    }
```

```
    // Create a table using sqlite3_exec
    const char *sql = "CREATE TABLE IF NOT EXISTS COMPANY("
                      "ID INT PRIMARY KEY     NOT NULL,"
                      "NAME           TEXT    NOT NULL,"
                      "AGE            INT     NOT NULL,"
                      "ADDRESS        CHAR(50),"
                      "SALARY         REAL );";
    rc = sqlite3_exec(db, sql, callback, 0, &err_msg);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "SQL error: %s\n", err_msg);
        sqlite3_free(err_msg);  // free error message
    } else {
        printf("Table created successfully\n");
    }

    sqlite3_close(db);
    return 0;
}
```

Output:

```
Table created successfully
```

This code opens `test.db`, creates a table, and prints success or error [6]. If there's an error, `err_msg` is set and must be freed with `sqlite3_free()` [7] [6].

Callback Functions with `sqlite3_exec`

The **callback function** used with `sqlite3_exec` must match the type:

```
int callback(void *data, int argc, char **argv, char **azColName);
```

- `data` is the same pointer you pass as the 4th argument to `sqlite3_exec`.
- `argc` is the number of columns in the result row.
- `argv` is an array of strings holding each column's value (always text) or NULL.
- `azColName` is an array of strings with the column names [8] [9].

SQLite calls this function for each row in a `SELECT`. For example:

```
static int my_callback(void *data, int argc, char **argv, char **azColName){
    printf("%s:\n", (const char*)data);
    for(int i = 0; i < argc; i++){
```

```c
        printf("  %s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    }
    return 0;
}

// ...

const char *sql = "SELECT ID, NAME FROM COMPANY;";
const char *msg = "Callback function called";
rc = sqlite3_exec(db, sql, my_callback, (void*)msg, &err_msg);
```

This will invoke `my_callback` once per row, printing each column. The `data` pointer (here a message string) is passed to the callback each time [9].

Storing Query Results in Custom Structs

To collect query results programmatically, pass a struct via the callback's `void*` argument. For example:

```c
struct Person {
    int id;
    char name[100];
    double salary;
};

static int store_person(void *data, int argc, char **argv, char **azColName) {
    struct Person *p = (struct Person*)data;
    if(argc >= 3) {
        p->id = atoi(argv[0]);                    // convert text to int
        strncpy(p->name, argv[1] ? argv[1] : "", sizeof(p->name));
        p->salary = argv[2] ? atof(argv[2]) : 0.0;
    }
    return 0;
}

int main(void) {
    // ... (open db)
    struct Person person = {0};
    const char *sql = "SELECT ID, NAME, SALARY FROM COMPANY WHERE ID=1;";
    rc = sqlite3_exec(db, sql, store_person, &person, &err_msg);
    if(rc == SQLITE_OK) {
        printf("Retrieved: ID=%d, Name=%s, Salary=%.2f\n",
               person.id, person.name, person.salary);
    }
    // ... (cleanup)
}
```

Here, `store_person` casts `data` to `Person*` and fills its fields. The main code passes `&person` as the callback argument. This technique ("use a custom struct") is recommended to handle results in a type-safe way [10].

`sqlite3_prepare_v2`: Preparing Statements

The function `sqlite3_prepare_v2` compiles SQL text into a **prepared statement** (`sqlite3_stmt *`) that you can execute and step through. Its signature is:

```
int sqlite3_prepare_v2(
    sqlite3 *db,            /* Database handle */
    const char *zSql,       /* SQL statement (UTF-8) */
    int nByte,              /* Max length of zSql in bytes */
    sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
    const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

- `db` is an open database.
- `zSql` is the SQL text to compile.
- `nByte` is the max number of bytes to read from `zSql`. If negative, `zSql` is read up to the null terminator [11].
- `ppStmt` returns a compiled `sqlite3_stmt*`.
- `pzTail` (often `NULL`) if not NULL will point to the portion of `zSql` after the first SQL statement [12].

On success, `*ppStmt` points to a compiled statement and the function returns `SQLITE_OK` [13]. If there is no valid SQL or an error occurs, `*ppStmt` is set to NULL [13]. **Always finalize** the statement with `sqlite3_finalize()` when done [13]. This prevents memory leaks.

When to use `sqlite3_prepare_v2`: Use it when you want to bind parameters, execute the statement multiple times, or retrieve typed results. Unlike `sqlite3_exec`, preparing a statement gives you control with `sqlite3_step()` and `sqlite3_column_*()` for each row [14].

Example: `sqlite3_prepare_v2` with Binding and Retrieval

Below is an example that prepares a SELECT with a parameter, binds an integer, steps through results, and finalizes:

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

int main(void) {
    sqlite3 *db;
    sqlite3_stmt *stmt;
    int rc = sqlite3_open("test.db", &db);
```

```c
    if(rc != SQLITE_OK) {
        fprintf(stderr, "Cannot open DB: %s\n", sqlite3_errmsg(db));
        return 1;
    }

    const char *sql = "SELECT ID, NAME FROM COMPANY WHERE ID = ?";
    rc = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
    if(rc != SQLITE_OK) {
        fprintf(stderr, "Failed to prepare stmt: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
    }

    sqlite3_bind_int(stmt, 1, 3);  // Bind integer value 3 to first parameter

    if(sqlite3_step(stmt) == SQLITE_ROW) {
        int id    = sqlite3_column_int(stmt, 0);
        const unsigned char *name = sqlite3_column_text(stmt, 1);
        printf("Id = %d, Name = %s\n", id, name);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

This code compiles the SQL with a `?` placeholder. `sqlite3_bind_int(stmt, 1, 3)` binds the integer 3 to that placeholder [15]. After calling `sqlite3_step(stmt)`, we fetch the row using `sqlite3_column_int/text()`.

Output for the above (assuming ID=3 exists):

```
Id = 3, Name = Skoda
```

`sqlite3_bind_int` and Parameter Binding

The function `sqlite3_bind_int(sqlite3_stmt*, int index, int value)` binds an integer value to a SQL parameter in a prepared statement [15] [16]. The **index** (1-based) specifies which `?` or named placeholder to replace, and **value** is the integer to bind. For example, after preparing `"WHERE ID = ?"`, calling `sqlite3_bind_int(stmt, 1, 42)` replaces the first `?` with 42 [15].

Binding is crucial for both convenience and security: it avoids manual string formatting of SQL and prevents SQL injection. (SQLite also provides `sqlite3_bind_text`, `sqlite3_bind_double`, etc. for other types.) Always bind parameters rather than concatenating user input into SQL. For instance, using `?` or named placeholders like `:name` is safer [17] [15].

`sqlite3_exec` vs `sqlite3_prepare_v2`

- `sqlite3_exec` executes one or more SQL statements directly. It is easy to use (a "one-stop" function) and suitable for simple cases [4] [5]. However, `sqlite3_exec` always treats returned values as text [18] and only works through a callback. It's not ideal for retrieving large result sets or using parameter binding. It's often used for quick scripting tasks, logging, or simple table creation.
- `sqlite3_prepare_v2` **+** `sqlite3_step` is a lower-level approach. You first compile the SQL into a statement, then call `sqlite3_step()` in a loop to retrieve rows, and finally `sqlite3_finalize()` [14]. This method supports parameter binding, type-safe retrieval (int, double, text, blob via `sqlite3_column_*`), and is better for error handling. It is the recommended method for complex or performance-sensitive code.

In summary: `sqlite3_exec` is convenient but limited to basic uses [18] [19]. For robust applications — especially those needing binding or precise control — use `sqlite3_prepare_v2`, `sqlite3_bind_*`, `sqlite3_step`, and `sqlite3_finalize` [18] [14].

Best Practices

- **Check return codes:** Always test the return value of SQLite calls. If a function returns something other than `SQLITE_OK` or `SQLITE_ROW`, use `sqlite3_errmsg(db)` to get the error text [6]. For example:

```
if (rc != SQLITE_OK) {
    fprintf(stderr, "SQL error: %s\n", sqlite3_errmsg(db));
    // handle error...
}
```

- **Finalize statements and close DB:** For every `sqlite3_prepare_v2` call that succeeds, you **must** call `sqlite3_finalize(stmt)` when done. Likewise, close the database with `sqlite3_close(db)` when your program is finished. Failing to finalize statements or close the database can lead to memory leaks. In fact, `sqlite3_close()` will report `SQLITE_BUSY` if there are unfinalized statements [20] [13].
- **Free error messages:** When using `sqlite3_exec`, you pass a `char **errmsg`. If an error occurs, SQLite allocates a string for the error message. After handling the error, always free this string with `sqlite3_free(errmsg)` to avoid memory leaks [7] [6].
- **Use parameter binding:** Never build SQL by concatenating untrusted input. Instead, use `?` or named parameters in your SQL and bind values using `sqlite3_bind_*`. This not only simplifies the code but also **prevents SQL injection** [17] [15]. For example: `sqlite3_bind_text(stmt, 1, user_input, -1, SQLITE_TRANSIENT);` safely includes a user string.
- **Memory management:** After `sqlite3_close(db)`, the database connection is closed and freed. Also, ensure that any other dynamic memory (e.g. data buffers) is managed. Use `sqlite3_finalize()` and `sqlite3_free()` as noted to release SQLite's resources.

By following these practices (error checking, freeing resources, binding parameters), you can avoid common bugs and security issues in C programs using SQLite.

References: SQLite official docs and tutorials provide detailed explanations and examples of the C API [4] [5] [14] [15] [17], which we have cited here. These sources ensure the accuracy of usage patterns and best practices.

---

[1] SQLite Home Page
https://sqlite.org/

[2] Appropriate Uses For SQLite
https://sqlite.org/whentouse.html

[3] [6] [9] [20] SQLite C/C++ Interface
https://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm

[4] An Introduction To The SQLite C/C++ Interface
https://sqlite.org/cintro.html

[5] [7] [8] One-Step Query Execution Interface
https://sqlite.org/c3ref/exec.html

[10] [18] [19] SQLite User Forum: sqlite3_exec
https://sqlite.org/forum/info/a3ec0c803cbef1592087ca85af1fe3134a8491ec6e72aa953cc05b2c6f02b458

[11] [12] [13] [14] Compiling An SQL Statement
https://www.sqlite.org/c3ref/prepare.html

[15] SQLite C - SQLite programming in C
https://zetcode.com/db/sqlitec/

[16] [17] Binding Values To Prepared Statements
https://www.sqlite.org/c3ref/bind_blob.html