Using SQLite In Your C Program

Contents

Introduction

SQLite is a very useful tool to include in an embedded application. True to its name, SQLite is an extremely lightweight alternative to common Database Management Systems like PostgreSQL or mySQL. Where these programs run as a daemon serving up a database connection, SQLite contains its entire database in a single file. It can, alternatively, be used as a purely in-memory database in your application's process. This can be extremely useful for scenarios where constantly reading and writing from a filesystem is not ideal. It's also a relatively lightweight replacement for creating complicated collection data structures within your application, which may be buggy or require significant resources to implement. In short, SQLite can be a very useful resource in almost any situation.

Including SQLite

SQLite is widely available in most linux distributions. The best way to install it is to use the package manager installed.

Ubuntu:
```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

Arch Linux:
```
$ sudo pacman -S sqlite3
```

Once installed, using it in your C program is as easy as adding the following line to your program and then compiling with -lsqlite3 -ldl -pthread to link the library. -ldl and -lpthread are required for the dynamic linking of extensions and to ensure thread safety in SQLite. They can both be disabled but this is not recommended.

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

In most cases, this is sufficient to get started working with SQLite. However, in some cases, linking the shared library is not simple. If compiling using arm-linux-gnueabihf-gcc, it may require linking against the shared library from the target platform. This can be quite a hassle to set up.

Alternatively, SQLite is small and self-contained enough that it is simple to include the entire source code in your project and avoid any linking altogether. This source code can be obtained by downloading the SQLite amalgamation zip file from https://www.sqlite.org/download.html. Copy the sqlite3.c and sqlite3.h files into your project and use them like any other C file you have. Using the Amalgamation approach still requires the use of -ldl in your compilation command unless -DSQLITE_OMIT_LOAD_EXTENSION is used. This will disable dynamic loading of extensions in SQLite and remove the dependency on the dynamic linking library.

A downside to including the SQLite Amalgamation in your project is the increase in compilation times due to its size. Compiling SQLite every time you recompile your program will add seconds to each compilation and quickly become frustrating. In this case, I recommend refactoring your Makefile to rely on compiling object files. This will allow make to notice that the SQLite source files have not been modified and only compile them once. A sample Makefile implementing this is shown below.

```
CC = arm-linux-gnueabihf-gcc
CFLAGS = -ldl -lpthread
SRC = $(wildcard src/*.c)
OBJECTS=$(patsubst %.c, %.o, $(SRC))
OUTDIR = bin
OUT_NAME = sampleProgram

EXEC=$(OUTDIR)/$(OUT_NAME)

all: $(EXEC)

$(EXEC): $(OBJECTS)
	$(CC) $(OBJECTS) -o $@ $(CFLAGS)

$(OBJECTS): src/%.o : src/%.c
	$(CC) -c $< -o $@ $(CFLAGS)
```

Reading from SQLite database using callbacks

Reading from a SQLite database in C can be quite confusing and unfamiliar at first. For those of you who are familiar with Javascript, it is actually quite similar in style to a lot of constructs in that language.

To get things started, a connection to a database needs to be formed. This is as simple as opening up your database file.

```
#define DB_NAME "/path/to/db/file"
sqlite3* db;
int ret = sqlite3_open( DB_NAME,  &db );
```

Note that the second argument is actually a pointer to a pointer to a sqlite3 data-type. After initializing the sqlite3 pointer db, we must check to see if any errors prevented the database from opening.

```
if( ret ) {
     fprintf( stderr, "Failed to open database - error: %s\n",
sqlite3_errmsg( db ) );
     exit( EXIT_FAILURE );
}
```

SQLite includes a handy function called `sqlite3_errormsg( sqlite3* )` that will return a string explaining exactly what went wrong.

If our program is still running, then we have successfully connected to our database and are ready to read or write from it.

Executing SQL statements in the SQLite C interface is a multi-part process. It consists of `sqlite3_prepare()`, which takes a SQL statement string and processes it into a command to give to SQLite. We then take this command and give it to `sqlite3_step()`. This function actually runs the command until it returns a single row. This row is then given to `sqlite3_column()` which gives us a single column from the single row created by `sqlite3_step()`. `sqlite3_step()` and `sqlite3_column()` are repeated until we have processed every column of every row. Finally, we call `sqlite3_finalize()` which will destroy our processed command and free the memory used.

This is obviously an extremely inconvenient and cumbersome method, which is why `sqlite3_exec()` exists. This function is simply a wrapper that will handle executing all 4 of the previously mentioned functions for us. Its type signature is defined as follows:

```
int sqlite3_exec(
      sqlite3*,
      const char* sql,
      int (*callback)(void*, int, char**, char**),
      void*,
      char** errmsg
);
```

This function looks very intimidating at first glance but after walking through each step it will actually be quite intuitive. Before we can use this though, we should set up some structures that will be helpful for us. Let's first assume that we want to read from a table that has 3 columns: a numerical ID, a numerical timestamp, and a numerical value.

```
typedef struct {
      int id;
      int timestamp;
      int value;
} row_t;

typedef struct {
      row_t* rows;
      int num_rows;
      int max_rows;
} read_result_t;
```

Here we have created two data-structures. The first will represent a single row of our database table. The second will represent the result of our SQL read query. It contains a list of rows and two other fields to indicate how many rows were returned and the size of the array of rows. We will use these two structures to capture the result of our read operation.

Returning to the `sqlite3_exec()` function, we will now take a look at each argument.

The first, `sqlite3*`, is simply our `db` structure created earlier by opening the SQLite database.

The second, `const char* sql`, is a string containing the SQL statement we wish to execute.

The third, `int (*callback)(void*, int, char**, char**)`, represents a pointer to a function that returns an `int` and takes in a `void*`, `int`, `char**`, and `char**`. We will examine each of this callback function's arguments in the future. This function is called once for every row in the result of our query and is used to process the data.

The fourth, `void*`, will be the argument that is passed to our callback function. This is where we will provide our container for the results of our query. In our case, we will provide a pointer to a `read_result_t`.

Finally, the fifth argument, `char** errmsg`, is a pointer to a string that will contain an error message if anything goes wrong in the process of executing our query.

The function pointer argument in the `sqlite3_exec()` call is a pointer to a function of the type: `int callback( void*, int, char**, char** );`

We saw earlier that the `void*` will be our `read_result_t` pointer given to the original `sqlite3_exec()` function call.

Remembering that each time our callback function is executed represents a single row of our result, the `int` argument represents the number of columns in the current row. The first `char**` argument is an array of strings, one for each column's value. The last `char**` argument is also an array of strings. This array has one string for each column's name.

To continue the program started earlier, we can execute a sql query with the following statements.

```
char* sql_statement = “SQL STATEMENT”;
int max_rows = 1024;
char* err_msg = NULL;

read_result_t read_result;
read_result.max_rows = max_rows;
read_result.num_rows = 0;
read_result.rows = malloc( max_rows * sizeof( row_t ) );

int ret = sqlite3_exec(
      db,
      sql_statement,
      sql_read_callback,
      &read_result,
      &err_msg
);
if( ret != SQLITE_OK ) {
      fprintf( stderr, “Error executing query: %s\n”, err_msg );
      sqlite3_free( err_msg );
}
```

Notice how if there is an error message generated, SQLite will allocate memory to a string which must be freed with `sqlite3_free()`.

In order for this code to work, we must also have our callback function to process each row defined. In the above example, we are using a function like the following

```
int sql_read_callback( void* result, int num_columns, char**
values, char** labels )
{
	read_result_t* read_result = (read_result_t*)result;

	row_t* row = malloc( sizeof( row_t ) );

	row->id = atoi( values[ 0 ] );
	row->timestamp = atoi( values[ 1 ] );
	row->value = atoi( values[ 2 ] );

	if( read_result->num_rows == read_result->max_rows ) {
		free( row );
		return 1;
	}

	int index = read_result->num_rows;
	read_result->rows[ index ] = row;
	read_result->num_rows++;

	return 0;
}
```

This function first casts our `void*` argument to the type we passed in, a `read_result_t*`. Since we know exactly how our data is laid out in the table we directly convert each value to an integer and save it. If our data was more general, we would need some extra logic based on the column labels and number of columns. If our array of rows in our result has enough room for our new row, we add it and return 0. Otherwise, we return 1. When `sqlite3_exec()` encounters a callback that returns 1, it will halt execution of the SQL query and return early. Thus, we should always make sure our array of rows has enough room to contain the result of our query.

Full Example

Combining all of these parts together, we are left with the following program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
#define DB_NAME “/path/to/db/file”
#define SQL_STATEMENT “sql query”
#define MAX_ROWS 1024

typedef struct {
     int id;
     int timestamp;
     int value;
} row_t;

typedef struct {
     row_t* rows;
     int num_rows;
     int max_rows;
} read_result_t;

int sql_read_callback( void* result, int num_columns, char**
values, char** labels )
{
     read_result_t* read_result = (read_result_t*)result;

     row_t* row = malloc( sizeof( row_t ) );

     row->id = atoi( values[ 0 ] );
     row->timestamp = atoi( values[ 1 ] );
     row->value = atoi( values[ 2 ] );

     if( read_result->num_rows == read_result->max_rows ) {
          free( row );
          return 1;
     }

     int index = read_result->num_rows;
     read_result->rows[ index ] = row;
     read_result->num_rows++;

     return 0;
}
```

```
int main( void )
{
      sqlite3* db;
      int ret = sqlite3_open( DB_NAME,  &db );
      if( ret ) {
            fprintf( stderr, “Failed to open database - error:
      %s\n”, sqlite3_errmsg( db ) );
            exit( EXIT_FAILURE );
      }

      char* sql_statement = SQL_STATEMENT;
      char* err_msg = NULL;

      read_result_t read_result;
      read_result.max_rows = MAX_ROWS;
      read_result.num_rows = 0;
      read_result.rows = malloc( MAX_ROWS* sizeof( row_t ) );

      ret = sqlite3_exec(
            db,
            sql_statement,
            SQL_STATEMENT,
            &read_result,
            &err_msg
      );
      if( ret != SQLITE_OK ) {
            fprintf(stderr, “Error executing: %s\n”, err_msg );
            sqlite3_free( err_msg );
      }
      else {
            // Use data
      }

      for( int i = 0; i < read_result.num_rows; i++ ) {
            free( read_result.rows[ i ];
      }
      free( read_result.rows );

      sqlite3_close( db );

      return 0;
}
```

Common Issues

A very common source of bugs is in the path of the database file. When creating the database object with `sqlite3_open()` it is crucial to double check whether or not the filepath you give it is correct. If it is not, SQLite will have no problem with this and instead of opening your existing database, it will create a new database at the location you entered. This means your program will not crash right away. When trying to read or write to a table that does not exist in this new database however, your SQL statements will fail.

An even more difficult to diagnose issue is when using a SQLite database file located in a folder mounted through NFS. As stated on the SQLite website, “SQLite uses reader/writer locks to control access to the database. (Under Win95/98/ME which lacks support for reader/writer locks, a probabilistic simulation is used instead.) But use caution: this locking mechanism might not work correctly if the database file is kept on an NFS filesystem. This is because fcntl() file locking is broken on many NFS implementations.”[1] This means that if you have multiple threads of execution or processes accessing a SQLite database at once, their reads or write may inexplicably fail if the database file is located on an NFS drive. Therefore it is recommended that NFS be used to transfer the database file to the target and then move the database file elsewhere in the target’s filesystem before use.

[1] https://sqlite.org/faq.html