Intro to SQLite programming with C

CMPUT 391

The reading you need to do

Point your browser to: https://www.sqlite.org/cintro.html

Make sure you bookmark that page and read, today:

- Core Objects And Interfaces
- Typical Usage Of Core Routines And Objects

We suggest you follow the typical approach above, instead of the one in

- Convenience Wrappers Around Core Routines

How to Compile Your Code with SQLite3

- Go to the folder where you have sqlite3.c and sqlite3.h

tells gcc to add debugging information

- gcc -g <filename>.c sqlite3.c -lpthread -ldl

You should get a binary executable file called a.out

Accessing databases in your program

Typical way, used by every DBMS out there:

- Open a database **connection**
- Prepare a SQL **statement** for execution
- **Execute** the statement

If the statement is a query:

- Iterate through every tuple produced

If statement is an update:

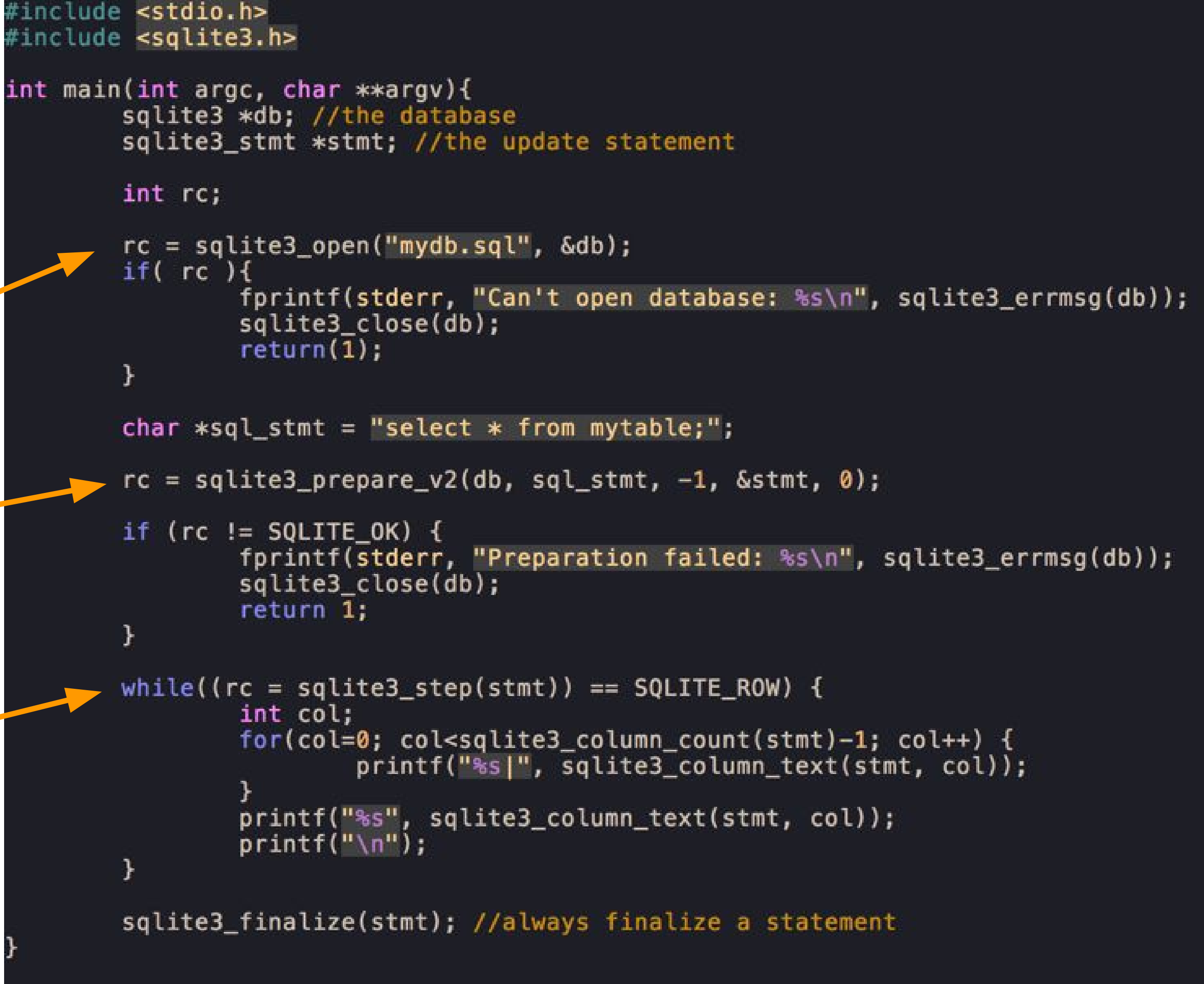
- Done after execution

Example

Open a database **connection**

Prepare a **statement**

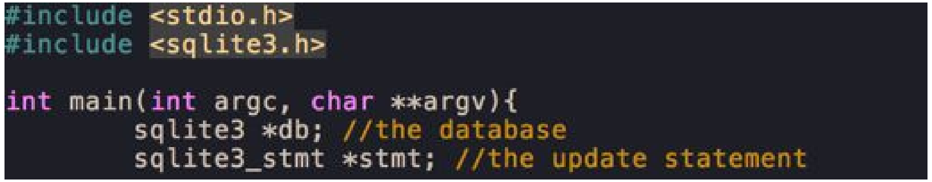
Execute the **statement**



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

int main(int argc, char **argv){
        sqlite3 *db; //the database
        sqlite3_stmt *stmt; //the update statement

        int rc;

        rc = sqlite3_open("mydb.sql", &db);
        if( rc ){
                fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
                sqlite3_close(db);
                return(1);
        }

        char *sql_stmt = "select * from mytable;";

        rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);

        if (rc != SQLITE_OK) {
                fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
                sqlite3_close(db);
                return 1;
        }

        while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
                int col;
                for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
                        printf("%s|", sqlite3_column_text(stmt, col));
                }
                printf("%s", sqlite3_column_text(stmt, col));
                printf("\n");
        }

        sqlite3_finalize(stmt); //always finalize a statement
}
```

How your program talks to SQLite3

Local variables shared with the SQLite3 code



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

int main(int argc, char **argv){
        sqlite3 *db; //the database
        sqlite3_stmt *stmt; //the update statement
```

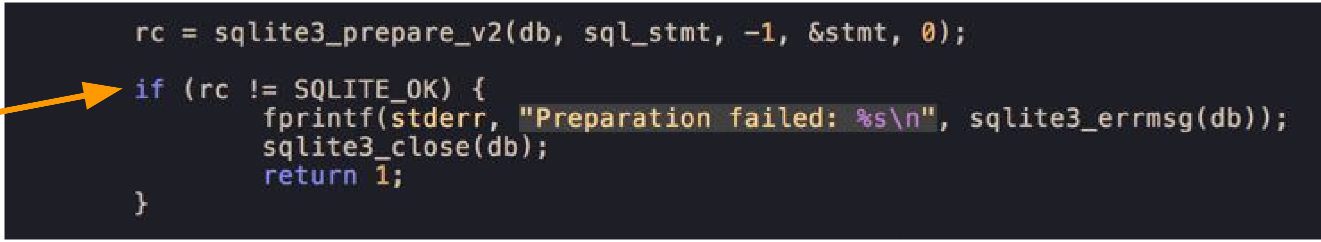
- `sqlite3 *db` https://www.sqlite.org/capi3ref.html#sqlite3 keeps information about the state of the connection
- `sqlite3_stmt *stmt` https://www.sqlite.org/capi3ref.html#sqlite3_stmt keeps information about the statement being executed

Everything you do with them must be through the SQLite3 methods as specified on those pages

Preparing a statement

To execute an SQL query, it must first be compiled into a byte-code program for the SQLite virtual machine https://www.sqlite.org/capi3ref.html#sqlite3_prepare

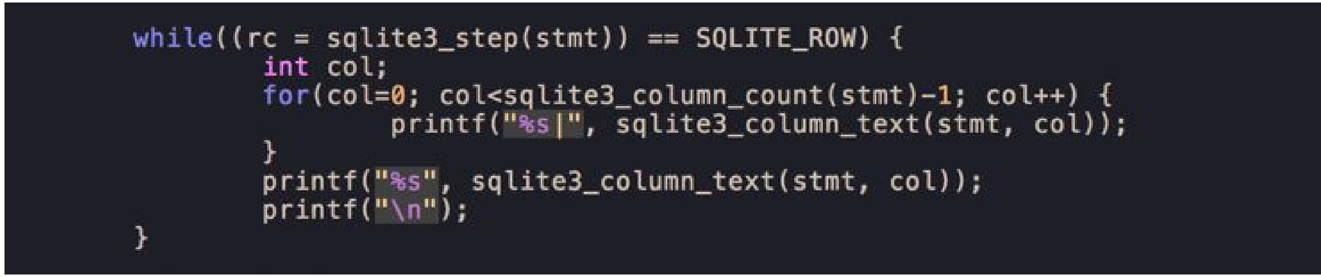
always check for errors!



```c
rc = sqlite3_prepare_v2(db, sql_stmt, -1, &stmt, 0);

if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
}
```

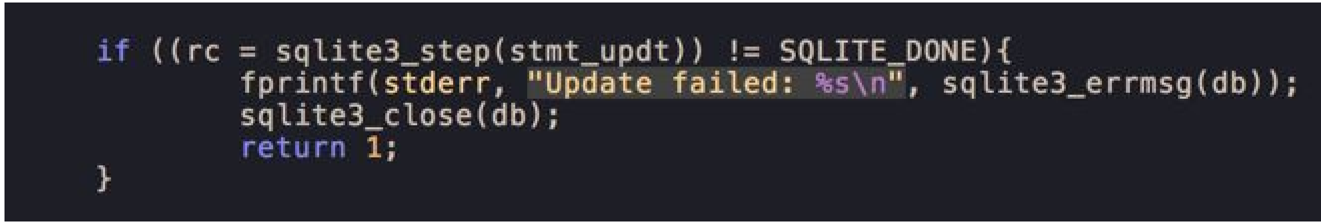
Executing statements

Queries: “step” through once for each tuple in the answer



```
while((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        int col;
        for(col=0; col<sqlite3_column_count(stmt)-1; col++) {
                printf("%s|", sqlite3_column_text(stmt, col));
        }
        printf("%s", sqlite3_column_text(stmt, col));
        printf("\n");
}
```

Updates: “step” through once and you’re done



```
if ((rc = sqlite3_step(stmt_updt)) != SQLITE_DONE){
        fprintf(stderr, "Update failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
}
```

Obtaining results from a query

Read up https://www.sqlite.org/capi3ref.html#sqlite3_column_blob

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);

double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);

sqlite3_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);

const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);

const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);

sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
```

Sample Programs

Sample program with a SQL query

https://sites.ualberta.ca/~denilson/files/cmput391/sample_code/select.c

Sample program with an update statement

https://sites.ualberta.ca/~denilson/files/cmput391/sample_code/update.c

Parameterized SQL

The reading you need to do

Point your browser to: https://www.sqlite.org/cintro.html

Make sure you bookmark that page and read, today:

- Binding Parameters and Reusing Prepared Statements
- Extending SQLite

We suggest you also read the section on

- Configuring SQLite

Database for the next examples

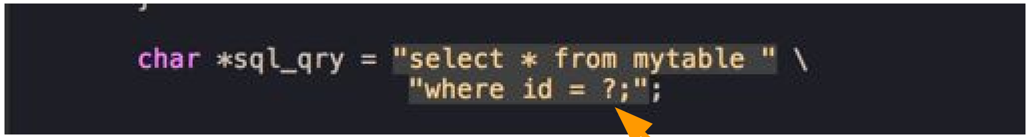
```
sqlite mydb.sql
```

```
CREATE TABLE mytable(id int, name text, score double);
INSERT INTO "mytable" VALUES(1001,'Elaine',3.9);
INSERT INTO "mytable" VALUES(1002,'Jerry',3.5);
```

Parameterized SQL statements

What to do if the actual query depends on user input?

Example:



```
char *sql_qry = "select * from mytable " \
                 "where id = ?;";
```

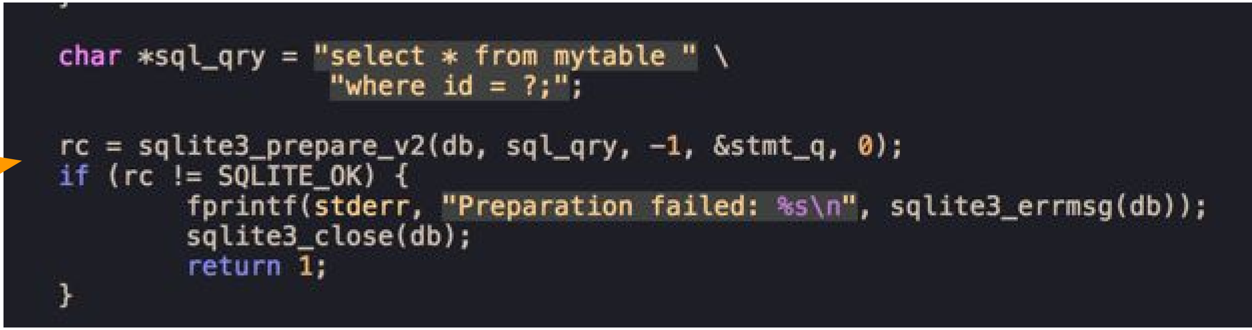
? is the parameter

Note: this is part of the SQL standard

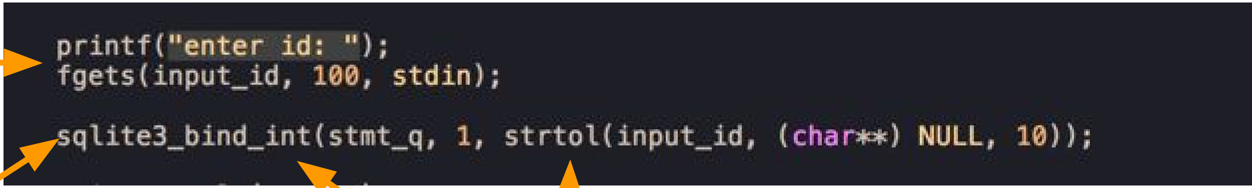
→ most DBMSs, including SQLite support it

Read up https://www.sqlite.org/c3ref/bind_blob.html

Binding parameters to a SQL query



```
char *sql_qry = "select * from mytable " \
                 "where id = ?;";

rc = sqlite3_prepare_v2(db, sql_qry, -1, &stmt_q, 0);
if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
}
```



```
printf("enter id: ");
fgets(input_id, 100, stdin);

sqlite3_bind_int(stmt_q, 1, strtol(input_id, (char**) NULL, 10));
```

prepare the statement as usual

read the input from STDIN

bind the input to the SQL parameter

convert the parameter to an **int**

Re-using a prepared statement

Preparing a statement means compiling it into SQLite3 byte-code (and takes time)

If you need to issue the same statement many times:

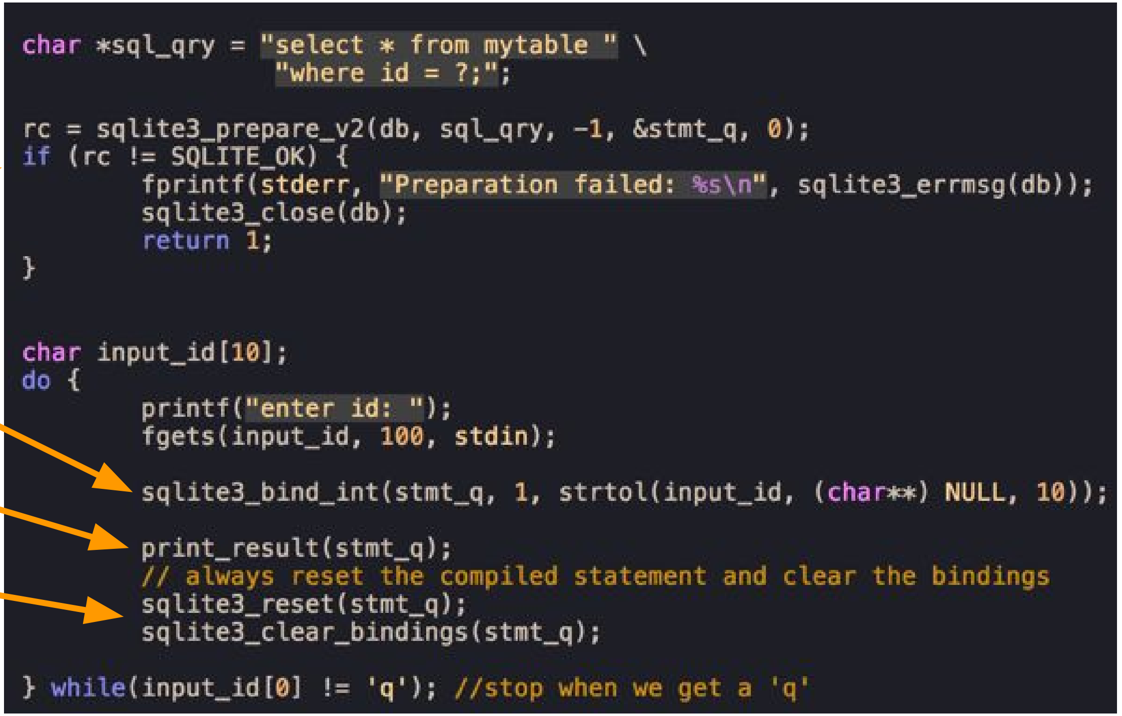
prepare it **once**

call many times

bind parameters

execute

reset bindings



```c
char *sql_qry = "select * from mytable " \
                 "where id = ?;";

rc = sqlite3_prepare_v2(db, sql_qry, -1, &stmt_q, 0);
if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
}


char input_id[10];
do {
        printf("enter id: ");
        fgets(input_id, 100, stdin);

        sqlite3_bind_int(stmt_q, 1, strtol(input_id, (char**) NULL, 10));

        print_result(stmt_q);
        // always reset the compiled statement and clear the bindings
        sqlite3_reset(stmt_q);
        sqlite3_clear_bindings(stmt_q);

} while(input_id[0] != 'q'); //stop when we get a 'q'
```

Read carefully

Among other important things, https://www.sqlite.org/c3ref/bind_blob.html states:

“The second argument is the index of the SQL parameter to be set. The leftmost SQL parameter has an index of 1.”

Also read:

https://www.sqlite.org/c3ref/clear_bindings.html

https://www.sqlite.org/c3ref/reset.html

User-defined Functions

Using custom functions in SQL statements

SQL has a very limited list of built-in functions

Most DBMSs offer ways for you to add custom functions to be used in SQL statements

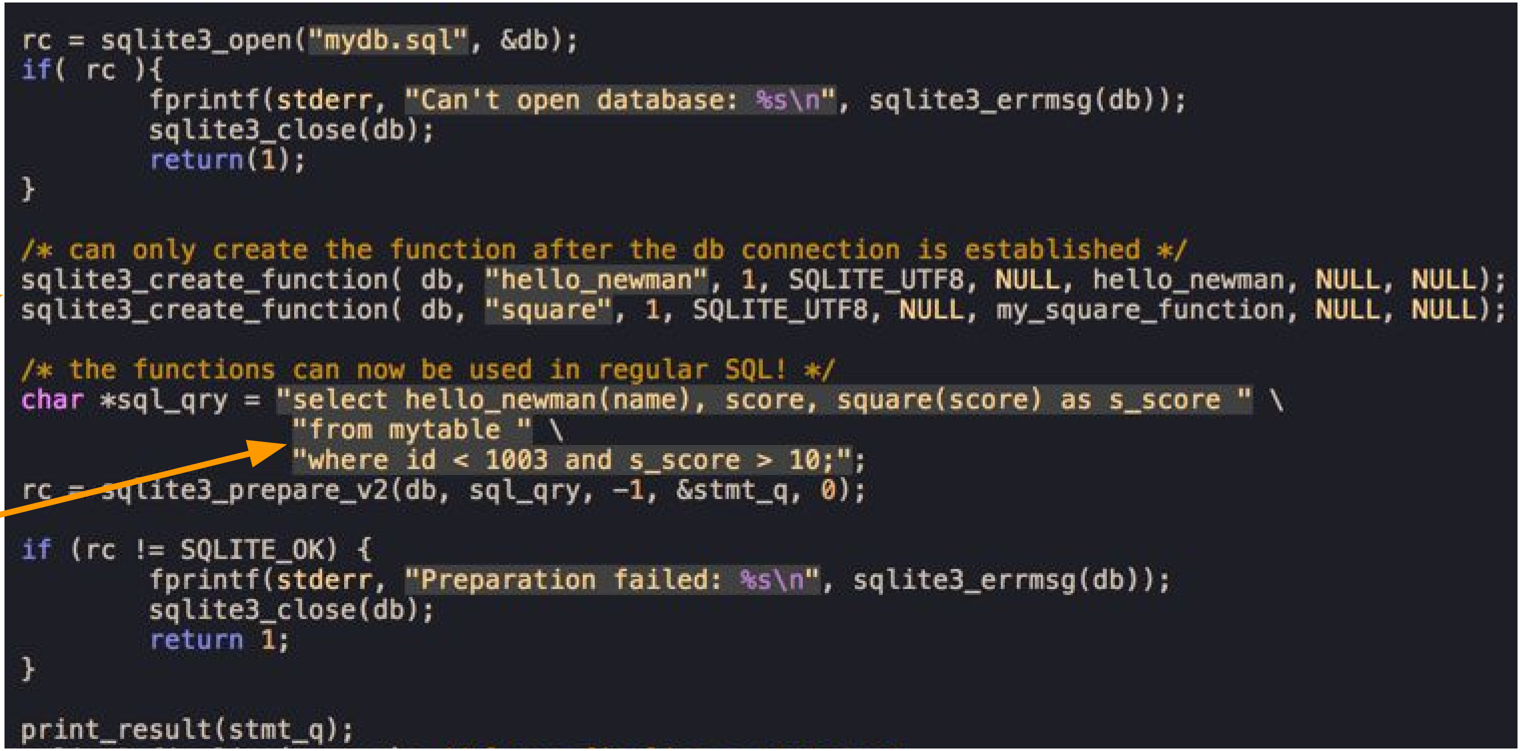
SQLite does not support the official SQL programming standard :(

But it allows the same functionality

Examples of scalar functions

declare the functions

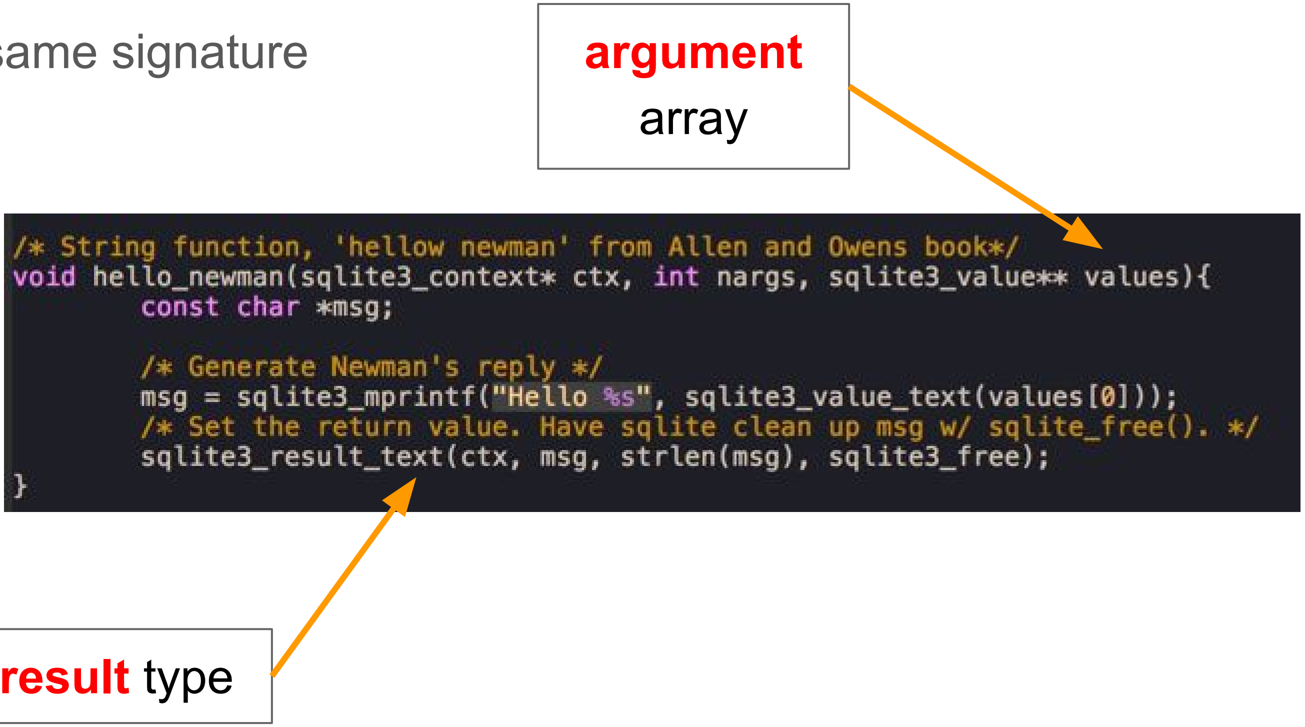
use the functions in SQL



```c
rc = sqlite3_open("mydb.sql", &db);
if( rc ){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return(1);
}

/* can only create the function after the db connection is established */
sqlite3_create_function( db, "hello_newman", 1, SQLITE_UTF8, NULL, hello_newman, NULL, NULL);
sqlite3_create_function( db, "square", 1, SQLITE_UTF8, NULL, my_square_function, NULL, NULL);

/* the functions can now be used in regular SQL! */
char *sql_qry = "select hello_newman(name), score, square(score) as s_score " \
                 "from mytable " \
                 "where id < 1003 and s_score > 10;";
rc = sqlite3_prepare_v2(db, sql_qry, -1, &stmt_q, 0);

if (rc != SQLITE_OK) {
        fprintf(stderr, "Preparation failed: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
}

print_result(stmt_q);
```

The function implementation

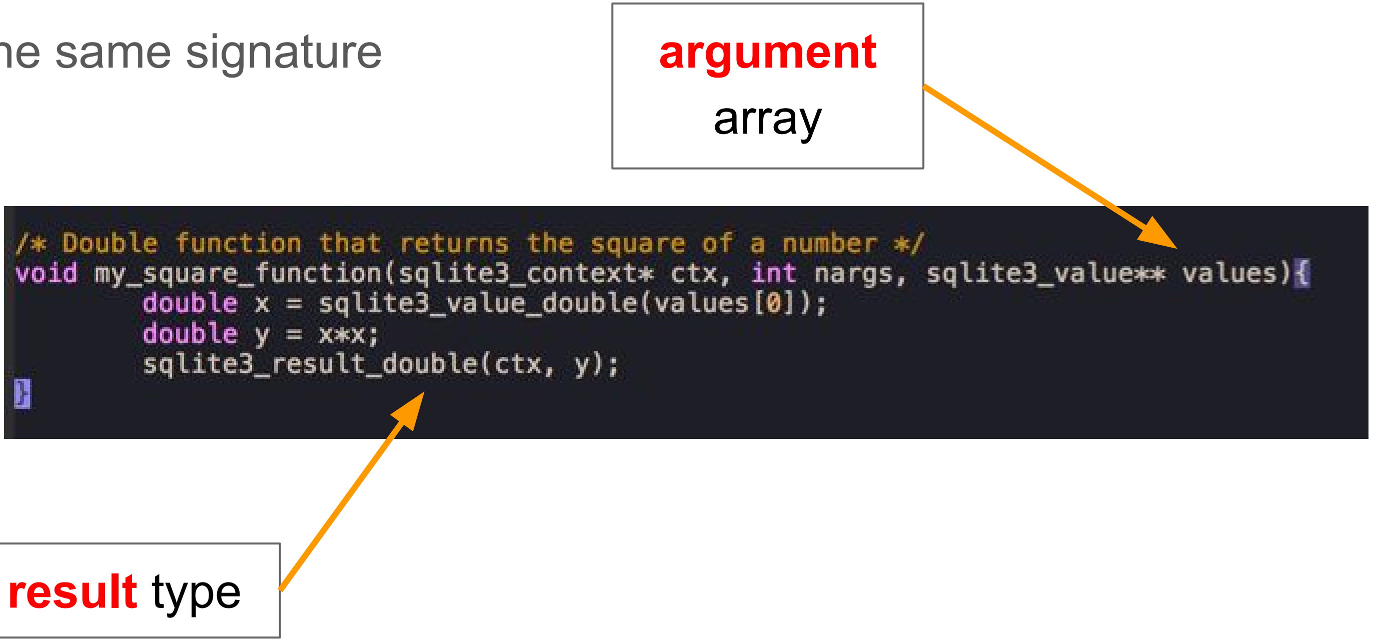
Always the same signature



```
/* String function, 'hellow newman' from Allen and Owens book*/
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values){
        const char *msg;

        /* Generate Newman's reply */
        msg = sqlite3_mprintf("Hello %s", sqlite3_value_text(values[0]));
        /* Set the return value. Have sqlite clean up msg w/ sqlite_free(). */
        sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

The function implementation

Always the same signature



```
/* Double function that returns the square of a number */
void my_square_function(sqlite3_context* ctx, int nargs, sqlite3_value** values){
        double x = sqlite3_value_double(values[0]);
        double y = x*x;
        sqlite3_result_double(ctx, y);
}
```

Read carefully

Among other important things https://www.sqlite.org/c3ref/create_function.html explains how to create **aggregate** functions as well.

Sample code

Parameterized SQL

https://sites.ualberta.ca/~denilson/files/cmput391/sample_code/parameterized_sql.c

Sample functions

https://sites.ualberta.ca/~denilson/files/cmput391/sample_code/sample_functions.c