CSCI 466/566

Using MySQL in C/C++

Jon Lehuta

Northern Illinois University

April 12, 2017

Motivation

- We have already looked at how to interact with a MySQL Database from PHP using the PHP Data Objects (PDO) API. Now we will look at the API you can use to do the same thing in C/C++ programs.
- The MySQL API is implemented in C, so it is provided as a set of functions to call, as opposed to a set of classes, as may be found in more object oriented programming.

Reference Material

The list of functions that are discussed in this document is not exhaustive. There are others available. If you'd like to see the full list, you can view them all at the link below:

`https://dev.mysql.com/doc/refman/5.7/en/c-api-function-overview.html`

HEADER FILE

To use any of these functions, you need to include the appropriate header file.

```
#include <mysql.h>
```

This file provides declarations for the functions and new types that make up the MySQL API.
Later, during the linking stage, you will specify the `mysqlclient` library, where the implementations of the functions are found.

Some new variable types

The MySQL library defines some new types. Some of them are data structures that store more complicated data, which is dealt with using the library functions, others are just a different way of referring to types you already know

- `MYSQL` - a data structure containing data about a connection to a MySQL server
- `MYSQL_RES` - a data structure that contains data about a result set
- `MYSQL_ROW` - a data structure that contains data about a single row from a result set
- `my_ulonglong` - used to store large unsigned integers

HANDLING ERRORS

Various functions in the MySQL API have specific return codes for when an error has occurred. In other situations, it may not be clear.

When an error occurs, there are two functions you can use to find out what happened:

```
unsigned int mysql_errno(MYSQL *mysql);
const char *mysql_error(MYSQL *mysql);
```

- `mysql_errno` - returns a numeric error code that identifies the last error that occurred
- `mysql_error` - returns a human-readable error message describing the last error that occurred.

The `mysql` parameter is a pointer to the connection object you would like to check the error for. We will discuss that later.

Overview, Initializing and Closing

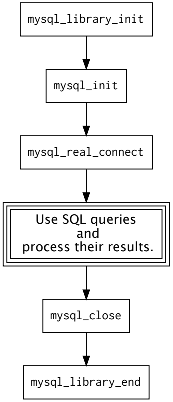

Figure: Overview of a MySQL program in C/C++, focused on initialization and cleanup.

Overview, Processing

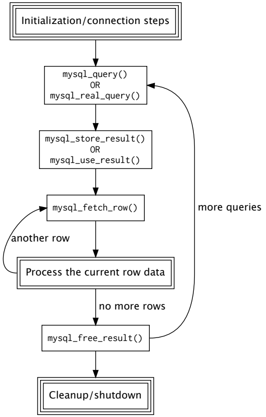
Initialization/connection steps
mysql_query() OR mysql_real_query()
mysql_store_result() OR mysql_use_result()
mysql_fetch_row()
another row
Process the current row data
more queries
no more rows
mysql_free_result()
Cleanup/shutdown

Figure: Overview of a MySQL program in C/C++, focused on dealing with the data

Northern Illinois University

Library Initialization

```
int mysql_library_init(int argc, char **argv, char **groups)
```

This function initializes the MySQL library. It is generally only needed if you want your application to be threadsafe, but you should call it in your programs for this class anyway.

- `argc` - argument count, use 0 unless you have a reason not to
- `argv` - argument vector, use `NULL` unless you have a reason not to
- `groups` - `NULL`-terminated array of strings listing which options to read, use `NULL` unless you have a reason not to

https://dev.mysql.com/doc/refman/5.7/en/mysql-library-init.html

Library Deinitialization

```
void mysql_library_end(void)
```

This function cleans up any memory allocated during the use of the MySQL library. You should make sure to call it after you are completely finished using the library to communicate with the DBMS.

`https://dev.mysql.com/doc/refman/5.7/en/mysql-library-end.html`

Initializing a Connection Object

```
MYSQL *mysql_init(MYSQL *mysql)
```

This function initializes (allocating memory as well, if needed) a `MYSQL` object, which is suitable for use with `mysql_real_connect()`. If `mysql` is `NULL`, then function will dynamically allocate memory for the object. If it is a pointer to an already allocated `MYSQL` object, it will set it up in that place. Either way, it will return a pointer to where the initialized object is. If there is an error, `NULL` will be returned.

- `mysql` - a pointer to a `MYSQL` object to initialize, or `NULL`

https://dev.mysql.com/doc/refman/5.7/en/mysql-init.html

Establishing a DB Connection

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host,
          const char *user, const char *passwd,
          const char *db, unsigned int port,
          const char *unix_socket, unsigned long client_flag)
```

Establishes a connection to the specified MySQL database.

- `mysql` - pointer to an existing `MYSQL` object, can be used to set options
- `host` - the hostname of the server to connect to item `user` - the username to authenticate with
- `passwd` - the password for the specified user
- `db` - the name of the database to use on that MySQL server
- `port` - set to zero unless you want to use a non-default port
- `unix_socket` - set this to `NULL`
- `client_flag` - set this to zero (can be used for special flags)

Returns a pointer to a `MYSQL` connection object on success, `NULL` on failure.

https://dev.mysql.com/doc/refman/5.7/en/mysql-real-connect.html

Northern Illinois University

Closing a DB Connection

```
void mysql_close(MYSQL *mysql)
```

This function closes a previously opened connection. It also deallocates the connection handler pointed to by `mysql` if the handler was allocated automatically in either `mysql_init()` or `mysql_connect()`.

- `mysql` - a pointer to the `MYSQL` object for the connection to be closed

`https://dev.mysql.com/doc/refman/5.7/en/mysql-close.html`

Running SQL Queries

After you have initialized the API and connected to the server, you can begin to run SQL queries on it. There are a couple of functions that are designed to do this.

- `mysql_query()` allows you to send a query as a null-terminated string
- `mysql_real_query()` allows you to send a query as a string of a specified length

Queries using null-terminated string

```
int mysql_query(MYSQL *mysql, const char *stmt_str)
```

This function executes the SQL statement stored in the null-terminated string pointed to by `stmt_str`. Normally the string must consist of a single SQL statment without a terminating semicolon. If you explicitly enable multiple-statement execution (see the link), it can contain several statements separated by semicolons. It is recommended not to do this, normally, because of the potential for SQL injection.

- `stmt_str` - a pointer to the beginning of a null-terminated string containing the SQL statement(s) to run

Returns zero on success. Non-zero if an error has occurred.

https://dev.mysql.com/doc/refman/5.7/en/mysql-query.html

Queries using string with length

```
int mysql_real_query(MYSQL *mysql,
                     const char *stmt_str,
                     unsigned long length)
```

This function executes the SQL statement pointed to by `stmt_str`, which is interpreted to be `length` bytes long. The use of a length as opposed to a terminating null character is what distinguishes this from the `mysql_query()` function from before. This allows binary data, which may contain null characters as valid data, to be sent.

- `mysql` - pointer to the MySQL connection to run the query through
- `stmt_str` - pointer to the beginning of the string containing the SQL statement(s)
- `length` - the length, in bytes, of the string, `stmt_str`

It returns zero on success, non-zero if an error has occurred.

https://dev.mysql.com/doc/refman/5.7/en/mysql-real-query.html

Get information on a query's results

- After a query is run, there may be a result set, which you would expect from a query with, for example, a **`SELECT`** statement. There are a couple of ways to access the results of such a query.
- Other queries might not generate a result set, such as **`INSERT`**, **`UPDATE`**, or **`DELETE`**. There are some things you may want to know about them even though no data is returned.

Get information on a query's results

For queries that don't generate a result set, you may want to know some or all of the following

- If you want to know how many rows were returned in a result set, you can use `mysql_num_rows()`.
- If you want to know how many attributes (columns) exist in a result set, you can use `mysql_field_count()`.
- If you want to know what value was chosen for an `AUTO_INCREMENT` field, you can use `mysql_insert_id()`.

How many rows were affected?

`my_ulonglong mysql_affected_rows(MYSQL *mysql)`

This function can be called immediately after running a query on the server.

- `mysql` - a pointer to the server connection object you ran the query through

Greater than zero indicates number of rows affected or retrieved. Zero indicates no rows. Errors are indicated with a `-1` return code.

https://dev.mysql.com/doc/refman/5.7/en/mysql-affected-rows.html

How many rows were returned?

```
my_ulonglong mysql_num_rows(MYSQL_RES *result)
```

This function will return the number of rows in the specified result set. If you want to use this before doing things with your data, you will need to use `mysql_store_result()` instead of `mysql_use_result()` to retrieve the result set.

- `result` - a pointer to the result set object to inquire about

`https://dev.mysql.com/doc/refman/5.7/en/mysql-num-rows.html`

How many columns in the result?

```
unsigned int mysql_field_count(MYSQL *mysql)
```

This function will return the number of columns were in the result set of the most recent query on the specified connection.

- `mysql` - a pointer to the connection object you just queried

`https://dev.mysql.com/doc/refman/5.7/en/mysql-field-count.html`

AUTO_INCREMENT → What is the key of inserted row?

`my_ulonglong mysql_insert_id(MYSQL *mysql)`

This function returns the value generated for an `AUTO_INCREMENT` column by the previous **`INSERT`** or **`UPDATE`** statement.

- `mysql` - a pointer to the connection object we ran the query through

Will return zero unless a value has been stored in an `AUTO_INCREMENT` field. If multiple rows were affected, only the first one will be returned.
`https://dev.mysql.com/doc/refman/5.7/en/mysql-insert-id.html`

Getting info from result sets

There are two basic functions that you can use to retrieve the data from a result set.

- The `mysql_store_result()` will function similarly to the `PDOStatement::fetchAll()` function did, in that it will retrieve all the results at once.
- The `mysql_use_result()` works more similarly to the `PDOStatement::fetch()` function, grabbing the results one row at a time.
- Neither of these two will directly give you the values in the row; that can be done with the `mysql_fetch_row()` function.
- You should make a call to `mysql_free_result()` after finishing with the result sets, to free up memory used to store the results.

Download all rows of the result set immediately

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

This function is used to retrieve the result set generated by a query. It will download the whole result set from the server immediately and store it in your program's memory. The values can then be obtained row-by-row with `mysql_fetch_row()` or you can use `mysql_row_seek()` to jump to specific rows.

- `mysql` - pointer to the connection object we used to run the query

Returns a pointer to a result set structure if successful, `NULL` on error.

`https://dev.mysql.com/doc/refman/5.7/en/mysql-store-result.html`

Set up to download one row at a time

```
MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

This function sets up a result set that will fetch its rows one at a time from the server. Individual rows are obtained with calls to `mysql_fetch_row()`. This is more memory efficient than using `mysql_store_result()` would be.

- `mysql` - a pointer to the connection object that ran the query

Returns a pointer to the result set structure on success. `NULL` is returned on failure.

https://dev.mysql.com/doc/refman/5.7/en/mysql-use-result.html

Get the row data

`MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)`

This will fetch the next row in the result set as a `MYSQL_ROW` structure, if there is one to fetch. It will start at the beginning and advance by one row each time it is called. If you call this after you've fetched the last row, it will return `NULL`. It will also return `NULL` if an error has occurred.

- `result` - a pointer to the result set object to fetch rows from

If you have any binary data in any of your fields, you will need to use `mysql_fetch_lengths()` to get the lengths, as they may contain the null character as part of their valid data. Otherwise, you can treat a `MYSQL_ROW` as an array of null-terminated strings. The fields can be addressed as the elements of the array, in the order they appear in the result set, starting from element 0.

https://dev.mysql.com/doc/refman/5.7/en/mysql-fetch-row.html

Northern Illinois University

Get byte-lengths for the fields in a row

```
unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

This function returns an array of integers that contains the lengths of each of the fields in a row returned by a previous call to `mysql_fetch_row()`. The integer in a given element of the array returned is the length of the corresponding element in the `MYSQL_ROW`.

- `result` - a pointer to the result set you just got a row from

Returns `NULL` on error. The most common error will be that you haven't fetched a row, or the last fetch failed.

`https://dev.mysql.com/doc/refman/5.7/en/mysql-fetch-lengths.html`

COMPILATION AND LINKING

When you are building a C++ program, there are two steps that must occur:

1. Compilation - the source code files are evaluated and object files are created containing function implementations
2. Linking - the machine code from the object files and various libraries is gathered and put together to form the final program

We will be using `g++` to compile C++ programs, or `gcc` to compile C programs. The command line arguments for each should be very similar, as both come from the same package, the GNU Compiler Collection.

For the programs in class, you should be building and running your code on Turing and Hopper, using these tools. Building them on a different machine or with a different compiler might change what you need to type at the command line.

Compiling without Linking

To compile without linking, you can specify the `-c` flag to `gcc` or `g++`. This will take your source code, program.cc or program.c in this example, and make an object file from the code within it.

```
g++ -c -I/usr/include/mysql program.cc
gcc -c -I/usr/include/mysql program.c
```

Either of these statements will yield a new file called `program.o`, which is the object file compiled from the original source.

Compilation Errors

The compilation stage is concerned with declarations. If you have an error that says something is undeclared, that failure is happening in the compilation stage.
If you have an error during compilation, it is usually one of the following types

- You forgot to include a header file that contains necessary declarations
- You made a syntax error in your code, which the compiler's error message should help you correct
- You made a typo somewhere (misspelled identifiers, etc.)

Linking Alone

After your object files have been created through compilation, you can perform the linking stage with one of the following commands.

```
g++ -o program program.o -L/usr/lib/mysql -lmysqlclient
gcc -o program program.o -L/usr/lib/mysql -lmysqlclient
```

- The `-o` flag say to name the program whatever the next word is. In this case, your `program.o` would be linked to make an executable file named `program`. If your project had multiple source code files, you'd include all of the object files' names, separated by spaces, where `program.o` is now.
- The `-L` flag is followed by the path to the directory where the MySQL library can be found.
- the `-l` flag is used to specify the name of the library to link in.

Linking Errors

The linking stage is concerned with finding the compiled code in object files/libraries. If you see an error about an "undefined reference", then the failure is happening during the linking stage.
If an error occurs during the linking phase, it is usually one of the following

- You failed to list one of the object files that contains the implementations of your functions
- You failed to tell the linker to include a library that is needed
- The linker path does not include the directory your library is located in.
- Your header file has a different declaration than its implementation uses.

Compilation and Linking Together

It is possible to perform compilation and linking with the same command. This may be easier to type, but doing it separately allows you to skip recompiling files that haven't changed. The following commands are an example of how to do both stages together.

```
g++ -o program -I/usr/include/mysql \
    -L/usr/lib/mysql -lmysqlclient program.cc
gcc -o program -I/usr/include/mysql \
    -L/usr/lib/mysql -lmysqlclient program.c
```

Note: The backslash \ at the end of the line means that the command continues on the next line. If you type the command on a single line, you will not need the backslash.

Running your program

The commands in the above slides will create your program in the current working directory. Generally speaking, this directory will not be a part of the search path for commands, so when you run your command, you'll need to specify that you want to run the one in the current directory.

```
./program
```