24

C Reference

MySQL C API

The MySQL C API uses several defined datatypes beyond the standard C types. These types are defined in the 'mysql.h' header file that must be included when compiling any program that uses the MySQL library.

Datatypes

`MYSQL`

A structure representing a connection to the database server. The elements of the structure contain the name of the current database and information about the client connection among other things.

`MYSQL_FIELD`

A structure containing all of the information concerning a specific field in the table. Of all of the types created for MySQL, this is the only one whose member variables are directly accessed from client programs. Therefore it is necessary to know the layout of the structure:

*char *name*

The name of the field.

*char *table*

The name of the table containing this field. For result sets that do not correspond to real tables, this value is null.

*char *def*

The default value of this field, if one exists. This value will always be null unless `mysql_list_fields` is called, after which this will have the correct value for fields that have defaults.

enum enum_field_types type

The type of the field. The type is one of the MySQL SQL datatypes. The following field types (along with their corresponding MySQL SQL data type) are currently defined:

FIELD_TYPE_TINY (TINYINT)
FIELD_TYPE_SHORT (SMALLINT)
FIELD_TYPE_LONG (INTEGER)
FIELD_TYPE_INT24 (MEDIUMINT)
FIELD_TYPE_LONGLONG (BIGINT)
FIELD_TYPE_DECIMAL (DECIMAL or NUMERIC)
FIELD_TYPE_FLOAT (FLOAT)
FIELD_TYPE_DOUBLE (DOUBLE or REAL)
FIELD_TYPE_TIMESTAMP (TIMESTAMP)
FIELD_TYPE_DATE (DATE)
FIELD_TYPE_TIME (TIME)
FIELD_TYPE_DATETIME (DATETIME)
FIELD_TYPE_YEAR (YEAR)
FIELD_TYPE_STRING (CHAR or VARCHAR)
FIELD_TYPE_BLOB (BLOB or TEXT)
FIELD_TYPE_SET (SET)
FIELD_TYPE_ENUM (ENUM)
FIELD_TYPE_NULL (NULL)
FIELD_TYPE_CHAR (TINYINT) (Deprecated, replaced by FIELD_TYPE_TINY)

unsigned int length

The size of the field based on the field's type.

unsigned int max_length

If accessed after calling `mysql_list_fields`, this contains the length of the maximum value contained in the current result set. If the field is a BLOB-style field (e.g. BLOB, TEXT, LONGBLOB, MEDIUMTEXT, etc.) this value will always be 8000 (8kB) if called before the actual data is retrieved from the result set (by using `mysql_store_result()`, for example). Once the data has been retrieved this field will contain the actual maximum length.

unsigned int flags

Zero or more option flags. The following flags are currently defined:

NOT_NULL_FLAG
If defined, the field cannot contain a NULL value.

PRI_KEY_FLAG
If defined, the field is a primary key.

UNIQUE_KEY_FLAG
If defined, the field is part of a unique key.

MULTIPLE_KEY_FLAG
If defined, the field is part of a key.

BLOB_FLAG
If defined, the field is of type `BLOB` or `TEXT`.

UNSIGNED_FLAG
If defined, the field is a numeric type with an unsigned value.

ZEROFILL_FLAG
If defined, the field was created with the `ZEROFILL` flag.

BINARY_FLAG
If defined, the field is of type CHAR or VARCHAR with the BINARY flag.

ENUM_FLAG
If defined, the field is of type ENUM.

AUTO_INCREMENT_FLAG
If defined, the field has the AUTO_INCREMENT attribute.

TIMESTAMP_FLAG
If defined, the field is of type TIMESTAMP.

SET_FLAG
If defined, the field is of type SET.

NUM_FLAG
If defined, the field is a numeric type (e.g. INT, DOUBLE, etc.).

PART_KEY_FLAG
If defined, the field is of part of a key. This flag is not meant to be used by clients, and it's behavior may change in the future.

GROUP_FLAG
This flag is not meant to be used by clients, and it's behavior may change in the future.

UNIQUE_FLAG
This flag is not meant to be used by clients, and it's behavior may change in the future.

unsigned int decimals
When used with a numeric field, it lists the number of decimals used in the field.

The following macros are provided to help examine the MYSQL_FIELD data:

IS_PRI_KEY(flags)
Returns true if the field is a primary key. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_NOT_NULL(flags)
Returns true if the field is defined as NOT NULL. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_BLOB(flags)
Returns true if the field is of type BLOB or TEXT. This macro takes the 'flags' attribute of a MYSQL_FIELD structure as its argument.

IS_NUM(type)
Returns true if the field type is numeric. This macro takes the 'type' attribute of a MYSQL_FIELD structure as its argument.

IS_NUM_FIELD(field)
Returns true if the field is numeric. This macro takes a MYSQL_FIELD structure as its argument.

MYSQL_FIELD_OFFSET
A numerical type indicating the position of the “cursor” within a row.

MYSQL_RES
A structure containing the results of a SELECT (or SHOW) statement. The actual output of the query must be accessed through MYSQL_ROW elements of this structure.

`MYSQL_ROW`
: *A single row of data returned from a `SELECT` query. Output of all MySQL data types are stored in this type (as an array of character strings).*

`my_ulonglong`
: A numerical type used for MySQL return values. The value ranges from 0 to 1.8E19, with -1 used to indicate errors.

mysql_affected_rows

my_ulonglong mysql_affected_rows(MYSQL *mysql)

Returns the number of rows affected by the most recent query. When used with a non-`SELECT` query, it can be used after the `mysql_query` call that sent the query. With `SELECT`, this function is identical to `mysql_num_rows`. This function returns 0, as expected, for queries that affect or return no rows. In the case of an error, the function returns –1.

When an UPDATE query causes no change in the value of a row, it is not usually considered to be 'affected'. However, if the CLIENT_FOUND_ROWS flag is used when connecting to the MySQL server, any rows that match the 'where' clause of the UPDATE query will be considered affected.

Example

```
/* Insert a row into the people table */
mysql_query(&mysql, "INSERT INTO people VALUES ('', 'Illyana Rasputin', 16)";
num = mysql_affected_rows(&mysql);
/* num should be 1 if the INSERT (of a single row) was successful, and -1 if
   there was an error */

/* Make any of 'HR', 'hr', 'Hr', or 'hR' into 'HR'. This is an easy way to
   force a consistent capitalization in a field.
mysql_query(&mysql, "UPDATE people SET dept = 'HR' WHERE dept LIKE 'HR'");
affected = mysql_affected_rows(&mysql);
/* By default, 'affected' will contain the number of rows that were changed.
   That is, the number of rows that had a dept value of 'hr', 'Hr' or 'hR'.
   If the CLIENT_FOUND_ROWS flag was used, 'affected' will contain the number
   of rows that matched the where (same as above plus 'HR'). */
```

mysql_change_user

my_bool mysql_change_user(MYSQL *mysql, char *username, char *password, char *database)

Changes the currently authenticated user and database. This function, re-authenticates the current connection using the given username and password. It also changes the default database to the given database (which can be NULL if no default is desired). If the password is incorrect for the given username or if the new user does not have rights to access the given database, a false value is returned and no action is taken. Otherwise, the rights of the new user take effect, the default database is selected and a true value is returned.

Example

```
if (! mysql_change_user( &mysql, new_user, new_pass, new_db ) ) {
      printf("Change of User unsuccessful!");
      exit(1);
}
/* At this point, the connection is operating under the access rights of the
   new username, and the new database is the default. */
```

mysql_character_set_name

char* mysql_character_set_name(MYSQL *mysql)

Returns the name of the default character set used by the MySQL server. A generic installation of the MySQL source uses the ISO-8859-1 character set by default.

Example

```
printf("This server uses the %s character set by default\n",
         mysql_character_set_name(&mysql));
```

mysql_close

void mysql_close(MYSQL *mysql)

Ends a connection to the database server. If there is a problem when the connection is broken, the error can be retrieved from the `mysql_err` function.

Example

```
mysql_close(&mysql);
/* The connection should now be terminated */
```

mysql_connect

MYSQL *mysql_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd)

Creates a connection to a MySQL database server. The first parameter must be a predeclared `MYSQL` structure. The second parameter is the hostname or IP address of the MySQL server. If the host is an empty string or `localhost`, a connection will be made to the MySQL server on the same machine. The final two parameters are the username and password used to make the connection. The password should be entered as plain text, not encrypted in any way. The return value is the MYSQL structure passed as the first argument, or NULL if the connection failed. (Because the structure is contained as an argument, the only use for the return value is to check if the connection succeeded.)

> This function has been deprecated in the newer releases of MySQL and the `mysql_real_connect` function should be used instead.

Example

```
/* Create a connection to the local MySQL server using the name "bob" and
   password "mypass" */
MYSQL mysql;
if(!mysql_connect(&mysql, "", "bob", "mypass")) {
```

```
                    printf("Connection error!\n");
                    exit(0);
}
/* If we've reached this point we have successfully connected to the database
   server. */
```

mysql_create_db

int mysql_create_db(MYSQL *mysql, const char *db)

Creates an entirely new database with the given name. The return value is zero if the operation was successful and nonzero if there was an error.

This function has been deprecated in the newer releases of MySQL. MySQL now supports the `CREATE DATABASE` SQL statement. This should be used, via the *mysql_query* function, instead.

Example

```
/* Create the database 'new_database' */
result = mysql_create_db(&mysql, "new_database");
```

mysql_data_seek

void mysql_data_seek(MYSQL_RES *res, unsigned int offset)

Moves to a specific row in a group a results. The first argument is the `MYSQL_RES` structure that contains the results. The second argument is the row number you wish to seek to. The first row is 0. This function only works if the data was retrieved using `mysql_store_result` (datasets retrieved with `mysql_use_result` are not guaranteed to be complete).

Example

```
/* Jump to the last row of the results */
mysql_data_seek(results, mysql_num_rows(results)-1);
```

mysql_debug

mysql_debug(char *debug)

Manipulates the debugging functions if the client has been compiled with debugging enabled. MySQL uses the Fred Fish debugging library, which has far too many features and options to detail here.

Example

```
/* This is a common use of the debugging library. It keeps a trace of the
   client program's activity in the file "debug.out" */
mysql_debug("d:t:O,debug.out");
```

mysql_drop_db

int mysql_drop_db(MYSQL *mysql, const char *db)

Destroys the database with the given name. The return value is zero if the operation was successful and nonzero if there was an error.

> This function has been deprecated in the newer releases of MySQL. MySQL now supports the `DROP DATABASE` SQL statement. This should be used, via the `mysql_query` function, instead.

Example

```
/* Destroy the database 'old_database' */
result = mysql_drop_db(&mysql, "old_database");
```

mysql_dump_debug_info

int mysql_dump_debug_info(MYSQL *mysql)

This function causes the database server to enter debugging information about the current connection into its logs. You must have Process privilege in the current connection to use this function. The return value is zero if the operation succeeded and nonzero in the case of an error.

Example

```
result = mysql_dump_debug_info(&mysql);
/* The server's logs should now contain information about this connection
   If something went wrong so that this is not the case, 'result' will have
   a false value.*/
```

mysql_eof

my_bool mysql_eof(MYSQL_RES *result)

Returns a nonzero value if there is no more data in the group of results being examined. If there is an error in the result set, zero is returned. This function only works of the result set was retrieved with the `mysql_use_result` function (`mysql_store_result` retrieves the entire result set, making this function unnecessary).

> This function has been deprecated in the newer releases of MySQL. The `mysql_errno` and `mysql_error` functions return more information about any errors that occur and they are more reliable.

Example

```
/* Read through the results until no more data comes out */
while((row = mysql_fetch_row(results))) {
       /* Do work */
}

if(!mysql_eof(results))
   printf("Error. End of results not reached.\n");
```

mysql_errno

unsigned int mysql_errno(MYSQL *mysql)

Returns the error number of the last error associated with the current connection. If there have been no errors in the connection, the function returns zero. The actual text of the error can be retrieved using the `mysql_error` function. The defined names for the client errors can be found in the errmsg.h header file. The defined names for the server error can be found in the mysqld_error.h header file.

Example

```
error = mysql_errno(&mysql);
printf("The last error was number %d\n", error);
```

mysql_error

char *mysql_error(MYSQL *mysql)

Returns the error message of the last error associated with the current connection. If there have been no errors in the connection, the function returns an empty string. Error messages originating on the server will always be in the language used by the server (chosen at startup time with the `--language` option). The language of the client error messages can be chosen when compiling the client library. At the time of this writing MySQL supported the following languages: Czech, Danish, Dutch, English, Estonian, French, German, Greek, Hungarian, Italian, Japanese, Korean, Norwegian (standard and 'ny'), Polish, Portuguese, Romanian, Russian, Slovak, Spanish, and Swedish.

Example

```
printf("The last error was ’%s’\n", mysql_error(&mysql));
```

mysql_escape_string

unsigned int mysql_escape_string(char *to, const char *from, unsigned int length)

Encodes a string so that it is safe to insert it into a MySQL table. The first argument is the receiving string, which must be at least one character greater than twice the length of the second argument, the original string. (That is, to >= from*2+1.) The third argument indicates that only that many bytes are copied from the originating string before encoding it. The function returns the number of bytes in the encoded string, not including the terminating null character.

> While not officially deprecated, this function is generally inferior to the `mysql_real_escape_string` function which does everything this function does, but also takes into account the character set of the current connection, which may affect certain escape sequences.

Example

```
char name[15] = "Bob Marley’s";
char enc_name[31];
mysql_escape_string(enc_name, name);
/* enc_name will now contain "Bob Marley\’s" (the single quote is escaped).
```

mysql_fetch_field

MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)

Returns a `MYSQL_FIELD` structure describing the current field of the given result set. Repeated calls to this function will return information about each field in the result set until there are no more fields left, and then it will return a null value.

Example

```
MYSQL_FIELD *field;

while((field = mysql_fetch_field(results)))
{
    /* You can examine the field information here */
}
```

mysql_fetch_field_direct

MYSQL_FIELD * mysql_fetch_field_direct(MYSQL_RES * result, unsigned int fieldno)

This function is the same as `mysql_fetch_field`, except that you specify which field you wish to examine, instead of cycling through them. The first field in a result set is 0.

Example

```
MYSQL_FIELD *field;

/* Retrieve the third field in the result set for examination */
field = mysql_fetch_field_direct(results, 2);
```

mysql_fetch_fields

MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES * result)

The function is the same as `mysql_fetch_field`, except that it returns an array of `MYSQL_FIELD` structures containing the information for every field in the result set.

Example

```
MYSQL_FIELD *field; /* A pointer to a single field */
MYSQL_FIELD *fields; /* A pointer to an array of fields */

/* Retrieve all the field information for the results */
fields = mysql_fetch_fields(results);
/* Assign the third field to 'field' */
field = fields[2];
```

mysql_fetch_lengths

unsigned long *mysql_fetch_lengths(MYSQL_RES *result)

Returns an array of the lengths of each field in the current row. A null value is returned in the case of an error. You must have fetched at least one row (with `mysql_fetch_row`) before you can call this function. This function is the only way to determine the lengths of variable length fields, such as `BLOB` and `VARCHAR`, before you use the data.

This function is especially useful when reading binary data from a BLOB. Since all MySQL data is retrieved as strings (char *), it is common to use the `strlen()` function to determine the length of a data value. However, for binary data `strlen()` returns inaccurate results because it stops at the first null character. In these cases use can use `mysql_fetch_lengths` to retrieve the accurate length for a data value.

Example

```
unsigned long *lengths;

row = mysql_fetch_row(results);
lengths = mysql_fetch_lengths(results);
printf("The third field is %d bytes long\n", lengths[2]);
```

mysql_fetch_row

MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)

Retrieves the next row of the result and returns it as a `MYSQL_ROW` structure. A null value is returned if there are no more rows or there is an error. In the current implementation, the `MYSQL_ROW` structure is an array of character strings that can be used to represent any data. If a data element is NULL within the database, the MYSQL_ROW array element for that data element will be a null pointer. This is necessary to distinguish between a value that is NULL and a value that is simply an empty string (which will be a non-null pointer to a null value).

Example

```
MYSQL_ROW row;

row = mysql_fetch_row(results);
printf("The data in the third field of this row is: %s\n", row[2]);
```

mysql_field_count

unsigned int mysql_field_count(MYSQL *mysql)

Returns the number of columns contained in a result set. This function is most useful to check the type of query last executed. If a call to `mysql_store_result` returns a null pointer for a result set, either the query was a non-SELECT query (such as an UPDATE, INSERT, etc.) or there was an error. By calling `mysql_field_count` you can determine which was the case, since a non-SELECT query will always have zero fields returned and a SELECT query will always have at least one.

Example

```
MYSQL_FIELD field;
MYSQL_RES *result;

// A query has been executed and returned success
result = mysql_store_result();
if (! result ) {
     // Ooops, the result pointer is null, either the query was a non-SELECT
     // query or something bad happened!
     if ( mysql_field_count(&mysql) ) {
          // The number of columns queried is greater than zero, it must have
          // been a SELECT query and an error must have occurred.
     } else {
          // Since the number of columns queried is zero, it must have been
          // a non-SELECT query, so all is well...
     }
}
```

mysql_field_seek

MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset)

Seeks a result set to the given field of the current row. The position set by this function is used when `mysql_fetch_field` is called. The `MYSQL_FIELD_OFFSET` value passed should be the return value of a `mysql_field_tell` call (or another `mysql_field_seek`). Using the value 0 will seek to the beginning of the row. The return value is the position of the cursor before the function was called.

Example

```
MYSQL_FIELD field;

/* result is a MYSQL_RES structure containing a result set */
/* ... do some stuff */
/* Seek back to the beginning of the row */
old_pos = mysql_field_seek(results, 0);
/* Fetch the first field of the row */
field = mysql_fetch_field(results);
/* Go back to where you where */
mysql_field_seek(results, old_pos);
```

mysql_field_tell

MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)

Returns the value of the current field position within the current row of the result set. This value is used with `mysql_field_seek`.

Example

```
MYSQL_FIELD field1, field2, field3;
/* results is a MYSQL_RES structure containing a result set */

/* Record my current position */
old_pos = mysql_field_tell(results);
/* Fetch three more fields */
field1 = mysql_field_field(results);
field2 = mysql_field_field(results);
field3 = mysql_field_field(results);
/* Go back to where you where */
mysql_field_seek(results, old_pos);
```

mysql_free_result

void mysql_free_result(MYSQL_RES *result)

Frees the memory associated with a `MYSQL_RES` structure. This must be called whenever you are finished using this type of structure or else memory problems will occur. This should only be used on a pointer to an actual MYSQL_RES structure. For example, if a call to `mysql_store_result` returned a null pointer, this function should be used.

Example

```
MYSQL_RES *results;
/* Do work with results */
/* free results... we know it's not null since we just did work with
   it, but we'll check just to be safe. */
if (results)
    mysql_free_result(results);
```

mysql_get_client_info

char *mysql_get_client_info(void)

Returns a string with the MySQL library version used by the client program.

Example

```
printf("This program uses MySQL client library version %s\n",
       mysql_get_client_info()));
```

mysql_get_host_info

char *mysql_get_host_info(MYSQL *mysql)

Returns a string with the hostname of the MySQL database server and the type of connection used (e.g., Unix socket or TCP).

Example

```
printf("Connection info: %s", mysql_get_host_info(&mysql));
```

mysql_get_proto_info

unsigned int mysql_get_proto_info(MYSQL *mysql)

Returns the MySQL protocol version used in the current connection as an integer. As a general rule, the MySQL network protocol will only change between minor releases of MySQL. That is, all releases of MySQL 3.23.x should have the same protocol version number.

Example

```
printf("This connection is using MySQL connection protocol ver. %d\n",
        mysql_get_proto_info());
```

mysql_get_server_info

char *mysql_get_server_info(MYSQL *mysql)

Returns a string with the version number of the MySQL database server used by the current connection.

Example

```
printf("You are currently connected to MySQL server version %s\n",
        mysql_get_server_info(&mysql);
```

mysql_info

char *mysql_info(MYSQL *mysql)

Returns a string containing information about the most recent query, if the query was of a certain type. Currently, the following SQL queries supply extra information via this function: `INSERT INTO` (when used with a `SELECT` clause or a VALUES clause with more than one record); `LOAD DATA INFILE`; `ALTER TABLE`; and UPDATE. If the last query had no additional information (e.g., it was not one of the above queries), this function returns a null value.

The format of the returned string depends on which of the above queries is used:

INSERT INTO or ALTER TABLE: Records: *n* Duplicates: *n* Warnings: *n*

LOAD DATA INFILE: Records: *n* Deleted: *n* Skipped: *n* Warnings: *n*

UPDATE: Rows matched: *n* Changed: *n* Warnings: *n*

Example

```
/* We just sent LOAD DATA INFILE query reading a set of record from a file into
   an existing table */
printf("Results of data load: %s\n", mysql_info(&mysql));
/* The printed string looks like this:
Records: 30 Deleted: 0 Skipped: 0 Warnings: 0
*/
```

mysql_init

MYSQL *mysql_init(MYSQL *mysql)

Initializes a `MYSQL` structure used to create a connection to a MySQL database server. This, along with `mysql_real_connect`, is currently the approved way to initialize a server connection. You pass this function a `MYSQL` structure that you declared, or a null pointer, in which case a `MYSQL` structure will be created and returned. Structures created by this function will be properly freed when `mysql_close` is called. Conversly, if you passed your own pointer, you are responsible for freeing it when the time comes. A null value is returned if there is not enough memory to initialize the structure.

As of the current release of MySQL, MySQL clients will crash on certain platforms (such as SCO Unix) when you pass in a pointer to a MYSQL structure that you allocated yourself. If this is happening to you, just pass in NULL and use the pointer created by the MySQL library. As an added bonus, you don't have to worry about freeing it if you do that.

Example

```
MYSQL mysql;
```

```
if (!mysql_init(&mysql)) {
                 printf("Error initializing MySQL client\n");
                 exit(1);
}
/* Now you can call mysql_real_connect() to connect to a server... */

/* Alternative method: */
MYSQL *mysql;

mysql = mysql_init(NULL);
if (!mysql) {
     printf("Error initializing MySQL client\n");
     exit(1);
}
```

mysql_insert_id

my_ulonglong mysql_insert_id(MYSQL *mysql)

Returns the generated for an `AUTO_INCREMENT` field if the last query created a new row. This function is usually used immediately after a value is inserted into an `AUTO_INCREMENT` field, to determine the value that was inserted. This value is reset to 0 after any query that does not insert a new auto-increment row.

The MySQL-specific SQL function LAST_INSERT_ID() also returns the value of the most recent auto-increment. In addition, it is not reset after each query, and so can be called at any time to retrieve that value of the last auto-increment INSERT executed during the current session.

Example

```
/* We just inserted an employee record with automatically generated ID into
   a table */
id = mysql_insert_id(&mysql);
printf("The new employee has ID %d\n", id);
/* As soon as we run another query, mysql_insert_id will return 0 */
```

mysql_kill

int mysql_kill(MYSQL *mysql, unsigned long pid)

Attempts to kill the MySQL server thread with the specified Process ID. This function returns zero if the operation was successful and nonzero on failure. You must have Process privileges in the current connection to use this function.

The process IDs are part of the process information returned by the `mysql_list_processes` function.

Example

```
/* Kill thread 4 */
result = mysql_kill(&mysql, 4);
```

mysql_list_dbs

MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild)

Returns a `MYSQL_RES` structure containing the names of all existing databases that match the pattern given by the second argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all databases are listed. Like all `MYSQL_RES` structures, the return value of this function must be freed with `mysql_free_result`. This function returns a null value in the case of an error.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW databases'.

Example

```
MYSQL_RES databases;
databases = mysql_list_dbs(&mysql, (char *)NULL);
/* 'databases' now contains the names of all of the databases in the
   MySQL server */
/* ... */
mysql_free_result( databases );
/* Find all databases that start with 'projectName' */
databases = mysql_list_dbs(&mysql, "projectName%");
```

mysql_list_fields

MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild)

Returns a `MYSQL_RES` structure containing the names of all existing fields in the given table that match the pattern given by the third argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all fields are listed. Like all `MYSQL_RES` structures, the return value of this function must be freed with `mysql_free_result`. This function returns a null value in the case of an error.

> The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW COLUMNS FROM table'.

Example

```
MYSQL_RES fields;
fields = mysql_list_fields(&mysql, "people", "address%");
/* 'fields' now contains the names of all fields in the 'people' table
    that start with 'address' */
/* ... */
mysql_free_result( fields );
```

mysql_list_processes

MYSQL_RES *mysql_list_processes(MYSQL *mysql)

Returns a `MYSQL_RES` structure containing the information on all of the threads currently running on the MySQL database server. This information contained here can be used with mysql_kill to remove faulty threads. Like all MYSQL_RES structures, the return value of this function must be freed with mysql_free_result. This function returns a null value in the case of an error.

The returned result set contains the information in the following order:

Process ID – The MySQL process ID. This is the ID used with mysql_kill to kill a thread.
Username – The MySQL username of the user executing a thread.
Hostname – The location of the client running a thread.
Database – The current database for the client running a thread.
Action – The type of action last run in a thread. All SQL queries of any type show up as 'Query', so this will be the most common value here.
Time – The amount of time taken (in seconds) to execute the last action in a thread.
State – The state of the current thread. This indicates whether the thread is active (currently executing a command) or idle.
Info – Any extra information about the thread. For SQL queries, this will contain the text of the query.

The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW PROCESSLIST'

Example

```
MYSQL_RES *threads;
MYSQL_ROW row
threads = mysql_list_processes(&mysql);

row = mysql_fetch_row( threads );
printf("The ID of the first active thread is %d\n", row[0]);
```

mysql_list_tables

MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)

Returns a MYSQL_RES structure containing the names of all existing tables in the current database that match the pattern given by the second argument. This argument may be any standard SQL regular expression. If a null pointer is passed instead, all tables are listed. Like all MYSQL_RES structures, the return value of this function must be freed with mysql_free_result. This function returns a null value in the case of an error.

The information obtained from this function can also be obtained through a SQL query using the statement 'SHOW TABLES'

Example

```
MYSQL_RES tables;
tables = mysql_list_tables(&mysql, "p%");
/* 'tables' now contains the names of all tables in the current database
    that start with 'p' */
```

mysql_num_fields

unsigned int mysql_num_fields(MYSQL_RES *result)

Returns the number of fields contained in each row of the given result set. This is different from the `mysql_field_count` function, in that it operates on an actual result set, which is known to contain data where `mysql_field_count` checks the last executed query (usually to determine if an error occurred).

Example

```
/* 'results' is a MYSQL_RES result set structure */
num_fields = mysql_num_fields(results);
printf("There are %d fields in each row\n", num_fields);
```

mysql_num_rows

int mysql_num_rows(MYSQL_RES *result)

Returns the number of rows of data in the result set. This function is only accurate if the result set was retrieved with `mysql_store_result`. If `mysql_use_result` was used, the value returned by this function will be the number of rows accessed so far.

Example

```
/* 'results' is a MYSQL_RES result set structure */
num_rows = mysql_num_rows(results);
printf("There were %d rows returned, that I know about\n", num_rows);
```

mysql_odbc_escape_string

char *mysql_odbc_escape_string(MYSQL *mysql, char *result_string, unsigned long result_string_length, char *original_string, unsigned long original_string_length, void *parameters, char *(*extend_buffer))

Creates a properly escaped SQL query string from a given string. It is intended for use with ODBC clients, and `mysql_real_escape_string` provides the same functionality with a simpler interface. This function takes the string given in the fourth argument (with the length given in the fifth argument, not including the terminating null character), and escapes it so that the resulting string (which is put into the address given in the second argument, with a maximum length given in the third argument) is safe to use as a MySQL SQL statement. This function a copy of the result string (the second argument). The seventh argument must be a pointer to a function that can be used to allocate memory for result string. The function must take three arguments: A pointer to a set of parameters that control how the memory is allocated (these parameters are passed in as the sixth argument to the original function), a pointer to the result string and a pointer to the maximum length of the result string.

```
char *data = "\000\002\001";
int data_length = 3;
char *result;
int result_length = 5; /* We don't want the final string to be longer than 5.
 extend_buffer() is a function that meets the criteria given above. */
mysql_odbc_escape_string( &mysql, result, result_length, data, data_length,
   NULL, extend_buffer );
/* 'result' now contains the string '\\\000\002\001'
 (that is, a backslash, followed by ASCII 0, then ASCII 2 then ASCII 1. */
```

mysql_odbc_remove_escape

void mysql_odbc_remove_escape(MYSQL *mysql, char *string)

Removes escape characters from a string. This function is intended for use by internal ODBC drivers, and not for general use. Given a string, this function will remove the escape character ('\') from in front of any escape characters. This will modify (and this shorten) the original string that is passed in.

```
char *escaped = "\\'an escaped quoted string.\\'";
/* escaped contains the string: \' and escaped quoted string.\' */
mysql_odbc_remove_escape(&mysql, escaped);
/* escaped now contains the string: 'an escaped quoted string.' */
```

mysql_options

int mysql_options(MYSQL *mysql, enum mysql_option option, void *value)

Sets a connect option for an upcoming MySQL connection. This function must be called after a MYSQL structure has been initialized using `mysql_init` and before a connection has actually been established using `mysql_real_connect`. The options affect the upcoming connection. This function can be called multiple times to set more than one option. The value of the third argument depends on the type of option. Some options require a character string as an argument while others take a point to an integer, or nothing at all. The options are as follows (the type of the third argument is given in parenthesis after the option name:

MYSQL_INIT_COMMAND (char *)

A SQL query to execute as soon as the connection is established. This query is re-executed if the connection is lost and automatically re-connected.

MYSQL_OPT_COMPRESS (none)

Causes the connection to use a compressed protocol with the server, to increase speed.

MYSQL_OPT_CONNECT_TIMEOUT (unsigned int *)

The number of seconds waited before giving up on connecting to the server.

MYSQL_OPT_NAMED_PIPE (none)

Causes the connection to use named pipes, as opposed to TCP, to connect to a local MySQL server running on Windows NT.

MYSQL_READ_DEFAULT_FILE (char *)

The name of the file to read for default options, in place of the default 'my.cnf'.

MYSQL_READ_DEFAULT_GROUP (char *)

The name of a group within the configuration file to read for the connection options, in place of the default 'client'. See Chapter XX: Configuration for information about the configuration file options.

Example

```
MYSQL mysql;
```

```
mysql_init( &mysql );
/* Prepare this connection to use the compressed protocol, execute the
   query "SHOW tables" upon connection, and read addition options from the
   'startup' stanze in the file .mysqlrc */
mysql_options(&mysql, MYSQL_OPT_COMPRESS, 0 );
mysql_options(&mysql, MYSQL_INIT_COMMAND, "SHOW tables" );
mysql_options(&mysql, MYSQL_READ_DEFAULT_FILE, ".mysqlrc" );
mysql_options(&mysql, MYSQL_READ_DEFAULT_GROUP, "startup" );
/* Now it is time to call mysql_real_connect() to make the connection using
   these options */
```

mysql_ping

int mysql_ping(MYSQL *mysql)

Checks to see if the connection to the MySQL server is still alive. If it is not, the client will attempt to reconnect automatically. This function returns zero if the connection is alive and nonzero if it cannot successful contact the server.

Example

```
while(mysql_ping(&mysql)) printf("Error, attempting reconnection...\n");
```

mysql_query

int mysql_query(MYSQL *mysql, const char *query)

Executes the SQL query given in the second argument. If the query contains any binary data (particularly the null character), this function cannot be used and `mysql_real_query` should be used instead. The function returns zero if the query was successful and nonzero in the case of an error.

Once a query has been executed using this function, the result set can be retrieved using the `mysql_store_result` or `mysql_use_result` functions.

Example

```
error = mysql_query(&mysql, "SELECT * FROM people WHERE name like 'Bill%'");
if (error) {
     printf("Error with query!\n");
     exit(1);
}
```

mysql_read_query_result

int mysql_read_query_result(MYSQL *mysql)

Processes the result of a query execute with the `mysql_send_query` command. Any data processed this way is not returned. Therefore, this function is only useful when running non-SELECT queries, or for debugging (since the return value still accurately reports if there was an error). The function return zero on success, and a non-zero number if an error occurred.

Example

```
mysql_send_query(&mysql, "SELECT * INTO OUTFILE results.out FROM mytable");
```

```
/* This executes the query, but does not process the results, which is necessary
   in order to write the values into the outfile */
mysql_read_query_result(&mysql);
/* Now the results have been processed and the data written to the outfile */
```

mysql_real_connect

MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user,

const char *passwd, const char *db, uint port, const char *unix_socket,

uint client_flag)

Creates a connection with a MySQL database server. There are eight arguments to this function:

- An initialized `MYSQL` structure, created with `mysql_init`.
- The hostname or IP address of the MySQL database server (use an empty string or `localhost` to connect to the local MySQL server over a Unix socket).
- The username used to connect to the database server (an empty string may be used assuming the Unix login name of the person running the client).
- The password used to authenticate the given user. If an empty string is used, only users with no passwords are checked for authentication.
- The initial database selected when you connect (an empty string may be used to not initially choose a database).
- The port used to remotely connect to a MySQL database server over TCP (0 may be used to accept the default port).
- The filename of the Unix socket used to connect to a MySQL server on the local machine (an empty string may be used to accept the default socket).
- Zero or more of a set of flags used under special circumstances:

`CLIENT_FOUND_ROWS`
: When using queries that change tables, returns the number of rows found in the table, not the number of rows affected.

CLIENT_IGNORE_SPACE
: Allows spaces after built-in MySQL functions in SQL queries. Traditionally, functions must be followed immediately by their arguments in parenthesis. If this option is used, the function names become reserved words and cannot be used for names of tables, columns or databases.

CLIENT_INTERACTIVE
: Causes the server to wait for a longer amount of time (the value of the interactive_timeout server variable) before automatically disconnecting a connection. This is useful for interactive clients where the user may not enter any data for significant periods of time.

`CLIENT_NO_SCHEMA`
: Prevent the client from using the full `database.table.column` form to specify a column from any database.

CLIENT_COMPRESS
: Use compression when communicating with the server.

CLIENT_ODBC
: Tell the server the client is an ODBC connection.

CLIENT_SSL
: Use SSL encryption to secure the connection. The server must be compiled to support SSL.

Example

```
MYSQL *mysql;
mysql = mysql_init( NULL );
/* Connect to the server on the local host with standard options. */
if (! mysql_real_connect(&mysql, "localhost", "bob", "mypass", "", 0, "", 0))
{ print "Error connecting!\n";
  exit(1);
}

/* or... */
/* Connect to the server at my.server.com using a compressed, secure protocol */
if (! mysql_real_connect(&mysql, "my.server.com", "bob", "mypass",
                          "", 0, "", CLIENT_COMPRESS|CLIENT_SSL)) {
          print "Error connecting!\n";
          exit(1);
}
```

mysql_real_escape_string

unsigned long mysql_real_escape_string(MYSQL *mysql, char *result_string, char *original_string, unsigned long orginal_string_length)

Creates a properly escaped SQL query string from a given string. This function takes the string given in the third argument (with the length given in the fourth argument, not including the terminating null character), and escapes it so that the resulting string (which is put into the address given in the second argument) is safe to use as a MySQL SQL statement. This function returns the new length of the resulting string (not including the terminating null character. To be completely safe, the allocated space for the result string should be at least twice as big as the original string (in case each character has to be escaped) plus one (for the terminating null character).

> This function is safe to use with binary data. The string can contain null characters or any other binary data. This is why it is necessary to include the length of the string. Otherwise, the MySQL library would not be able to determine how long the string was if any null characters were present.

Example

```
# Properly escape a query that contains binary data.
char *data = "\002\001\000";
int original_length = 4 # 3 characters plus one for the null.
char real_data[7]; # Twice as big as the original string (3)
                   # plus one for the null.
int new_length;
```

```
new_length = mysql_real_escape_string(&mysql, data, real_data, original_length);
/* real_query can now be safely used in as a SQL query. */
/* The returned length is '4' since the only character that needed escaping
   was \000 (the null character) */
```

mysql_real_query

int mysql_real_query(MYSQL *mysql, const char *query, unsigned int length)

Executes the SQL query given in the second argument. The length of the query (not including any terminating null character) must be given in the third argument. By supplying the length, you can use binary data, including null characters, in the query. This function is also faster than `mysql_query`. The function returns zero if the query was successful and nonzero in the case of an error.

Once a query has been executed using this function, the result set can be retrieved using the `mysql_store_result` or `mysql_use_result` functions.

Example

```
error = mysql_real_query(&mysql, "SELECT * FROM people WHERE name like 'Bill%'",
         44);
if (error) {
     printf("Error with query!\n");
     exit(1);
}
```

mysql_refresh

int mysql_refresh(MYSQL *mysql, unsigned int options)

Instructs the server to refresh various system operations. What exactly is refreshed depends upon the second argument which is a bitwise combination of any of the following options:

REFRESH_GRANT – Reload the permissions tables (same as mysql_reload() or the 'FLUSH PRIVILEGES' SQL command).
REFRESH_LOG – Flushes the logs files to disk and then closes and re-opens them.
REFRESH_TABLES – Flushes any open tables to disk.
REFRESH_READ_LOCK – Re-instates the read-lock on the database files stored on disk.
REFRESH_HOSTS – Reloads the internal cache MySQL keeps of known hostnames to limit DNS lookup calls.
REFRESH_STATUS – Refreshes the status of the server by checking the status of various internal operations.
REFRESH_THREADS – Clears out any dead or inactive threads from the internal cache of threads.
REFRESH_MASTER – Flushes, closes and reopens the binary log that tracks all changes to MySQL tables. This log is sent to slave server for replication.
REFRESH_SLAVE – Resets the connection with any slave servers that are replicating the master MySQL server.

Example

```
/* Flush the log files and the table data to disk */
if (!mysql_refresh( &mysql, REFRESH_LOG|REFRESH_TABLES )) {
     printf("Error sending refresh command...\n");
}
```

mysql_reload

int mysql_reload(MYSQL *mysql)

Reloads the permission tables on the MySQL database server. You must have Reload permissions on the current connection to use this function. If the operation is successful, zero is returned otherwise a nonzero value is returned.

> This function is deprecated and will be removed in a future version of the API. The exact same functionality can be obtained by using the SQL query 'FLUSH PRIVILEGES'.

Example

```
/* Make some changes to the grant tables... */
result = mysql_reload(&mysql);
/* The changes now take effect... */
```

mysql_row_seek

MYSQL_ROW_OFFSET mysql_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset)

Moves the pointer for a result set (MYSQL_RES structure) to a specific row. This function requires that the offset be an actual MYSQL_ROW_OFFSET structure, not a simple row number. If you only have a row number, use the `mysql_data_seek` function. A MYSQL_ROW_OFFSET can be obtained from a call to either `mysql_row_tell` or this function (which returns the row the result set was at before it was changed).

This function is only useful if the result set contains all of the data from the query. Therefore it should be used in conjunction with `mysql_store_result` and not used with `mysql_use_result`.

Example

```
/* result is a result set pointer created with mysql_store_result() */
MYSQL_ROW_OFFSET where, other_place;
where = mysql_row_tell( result );
/* Do some more work with the result set... */
/* Go back to where you were before, but remember where you are now: */
other_place = mysql_row_seek( result, where );
/* Do some more work... */
/* Go back to the second marker: */
mysql_row_seek( result, other_place );
```

mysql_row_tell

MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)

Returns the value of the cursor used as `mysql_fetch_row` reads the rows of a result set. The return value of this function can used with `mysql_row_seek` to jump to a specific row in the result set. Values from the function are only useful if the result set was created with `mysql_store_result` (which contains all of the data) and not `mysql_use_result` (which does not).

Example

```
/* results is a result set pointer created with mysql_store_result() */
MYSQL_ROW_OFFSET saved_pos = mysql_row_tell(results);
/* I can now jump back to this row at any time using mysql_row_seek() */
```

mysql_select_db

int mysql_select_db(MYSQL *mysql, const char *db)

Changes the current database. The user must have permission to access the new database. The function returns zero if the operation was successful and nonzero in the case of an error.

Example

```
if ( mysql_select_db(&mysql, "newdb") ) {
     printf("Error changing database, you probably don't have permission.\n");
}
```

mysql_send_query

int mysql_send_query(MYSQL *mysql, char *query, unsigned int query_length)

Executes a single MySQL query, without providing any results to the user. This function is useful for quickly executing a non-SELECT statement that does not return any data to the user. This function is also useful for debugging, since the return value still accurately reports whether an error occurred. The function return zero on success, and a non-zero number if an error occurred.

Example

```
/* Quickly insert a row into a table */
mysql_send_query(&mysql, "INSERT INTO mytable VALUES ('blah', 'fnor')");
```

mysql_shutdown

int mysql_shutdown(MYSQL *mysql)

Shutdown the MySQL database server. The user must have Shutdown privileges on the current connection to use this function. The function returns zero if the operation was successful and nonzero in the case of an error.

Example

```
if ( mysql_shutdown(&mysql) ) {
       printf("Server not shut down... Check your permissions...\n");
} else {
       printf("Server successfully shut down!\n");
}
```

mysql_ssl_cipher

char *mysql_ssl_cipher(MYSQL *mysql)

Returns the name of the SSL cipher that is used (or going to be used) with the current connection. This could be RSA, blowfish, or any other cipher supported by the server's SSL library (MySQL uses OpenSSL by default, if SSL is enabled). The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
printf("This connection is using the %s cipher for security.\n",
             Mysql_ssl_cipher(&mysql));
```

mysql_ssl_clear

int mysql_ssl_clear(MYSQL *mysql)

Clears any SSL information associated with the current connection. If called before the connection is made, this will cause the connection to made without SSL, even if SSL options had been selected previously. The function returns zero on success and a non-zero number if an error occurs. It must be called before `mysql_real_connect` to have any effect. The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
/* init a MYSQL structure and set SSL options...*/
/* Changed my mind, I don't want this connection to use SSL: */
mysql_ssl_clear(&mysql);
```

mysql_ssl_set

int mysql_ssl_set(MYSQL *mysql, char *key, char *certificate, char *authority, char *authority_path)

Sets SSL information for the current connection and causes the connection to be made using SSL for encryption. This function must be called before `mysql_real_connect` to have any effect. The arguments are (beyond the pointer to the MYSQL structure) the text of the SSL public key being used for the connection, the filename of the certificate being used, the name of the authority that issued the certificate, and the directory that contains the authority's certificates.

The function returns zero on success and a non-zero number if an error occurs. It must be called before `mysql_real_connect` to have any effect. The MySQL server and client both must have been compiled with SSL support for this function to work properly.

Example

```
/* 'key' contains an SSL public key.
   'cert' contains the filename of a certificate
   'ca' contains the name of the certificate authority
   'capath' contains the directory containing the certificate
*/
/* Create an initialized MYSQL structure using mysql_init */
mysql_ssl_set(&mysql, key, cert, ca, capath);
/* Now, when mysql_real_connect is called, the connection will use SSL for
   encryption. */
```

mysql_stat

char *mysql_stat(MYSQL *mysql)

Returns information about the current operating status of the database server. This includes the uptime, the number of running threads, and the number of queries being processed, among other information.

Example

```
printf("Server info\n-----------\n%s\n", mysql_stat(&mysql));
/* Output may look like this:
Server info
-----------
Uptime: 259044  Threads: 1  Questions: 24  Slow queries: 0  Opens: 6
Flush tables: 1 Open tables: 0 Queries per second avg: 0.000
^^^^^ This is all on one line
*/
```

mysql_store_result

MYSQL_RES *mysql_store_result(MYSQL *mysql)

Reads the entire result of a query and stores in a `MYSQL_RES` structure. Either this function or `mysql_use_result` must be called to access return information from a query. You must call `mysql_free_result` to free the `MYSQL_RES` structure when you are done with it.

The function returns a null value in the case of an error. The function will also return a null value if the query was not of a type that returns data (such as an INSERT or UPDATE query). If receive a null pointer and are not sure if the query was supposed to return data or not, you can call `mysql_field_count` to find out the number of fields the query was supposed to return. If zero, then it was a non-SELECT statement and the pointer is supposed to be null. Otherwise, an error has occurred.

If the query was a SELECT-type statement, but happens to contain no data, this function will still return a valid (but empty) MYSQL_RES structure (it will not be a null-pointer).

Example

```
MYSQL_RES results;
mysql_query(&mysql, "SELECT * FROM people");
results = mysql_store_result(&mysql);
/* 'results' should now contain all of the information from the 'people' table */
if (!results) { printf("An error has occurred!\n"); }

/* 'query' is some query string we obtained elsewhere,
    we're not sure what it is... */
mysql_query(&mysql, query);
results = mysql_store_result(&mysql);
if (!results) { /* An error might have occurred,
                   or maybe this is just a non-SELECT statement */
     if (! mysql_field_count(&mysql) ) { /* Aha! This is zero so it was
                                            a non-SELECT statement */
               printf("No error here, just a non-SELECT statement...\n");
     } else {
               printf("An error has occurred!\n");
     }
}
```

mysql_thread_id

unsigned long mysql_thread_id(MYSQL * mysql)

Returns the thread ID of the current connection. This value can be used with *mysql_kill* to terminate the thread in case of an error. This value will be different if you disconnect from the server and then reconnect (which may happen automatically, without warning, if you use `mysql_ping`). Therefore, you should call this function immediately before you are going to use the value.

Example

```
thread_id = mysql_thread_id(&mysql);
/* This number can be used with mysql_kill() to terminate the current thread. */
```

mysql_thread_safe

unsigned int mysql_thread_safe(void)

Indicates whether the MySQL client library is safe to use in a threaded invironment. The function returns a true value if the library is thread safe, and zero (false) if it is not.

Example

```
if (mysql_thread_safe()) {
     printf("This library is thread safe... thread away!\n");
} else {
     printf("This library is *not* thread safe, be careful!\n");
}
```

mysql_use_result

MYSQL_RES *mysql_use_result(MYSQL *mysql)

Reads the result of a query row by row and allows access to the data through a `MYSQL_RES` structure. Either this function or `mysql_store_result` must be called to access return information from a query. Because this function does not read the entire data set at once, it is faster and more memory efficient than `mysql_store_result`. However, when using this function you must read all of the rows of the dataset from the server or else the next query will receive the left over data. Also, you cannot run any other queries until you are done with the data in this query. Even worse, no other threads running on the server can access the tables used by the query until it is finished. For this reason, you should only use this function when you are certain you can read the data in a timely manner and release it. You must call `mysql_free_result` to free the `MYSQL_RES` structure when you are done with it.

The function returns a null value in the case of an error. The function will also return a null value if the query was not of a type that returns data (such as an INSERT or UPDATE query). If receive a null pointer and are not sure if the query was supposed to return data or not, you can call `mysql_field_count` to find out the number of fields the query was supposed to return. If zero, then it was a non-SELECT statement and the pointer is supposed to be null. Otherwise, an error has occurred.

If the query was a SELECT-type statement, but happens to contain no data, this function will still return a valid (but empty) MYSQL_RES structure (it will not be a null-pointer).

Example

```
MYSQL_RES results;
mysql_query(&mysql, "SELECT * FROM people");
results = mysql_store_result(&mysql);
/* 'results' will now allow access (using mysql_fetch_row) to the table
   data, one row at a time */
```