G

C API Reference

This appendix describes the C language application programming interface for the MySQL client library. The API consists of a set of functions for communicating with MySQL servers and accessing databases, and a set of data types used by those functions. The client library functions group into these categories:

- Routines to initialize and terminate the client library
- Connection management routines to establish and terminate connections to the server
- Error-reporting routines to get error codes and messages
- Routines to construct SQL statements and send them to the server
- Result set processing routines to handle results from statements that return data
- Routines to process multiple result sets
- Routines that provide information about the client, server, protocol version, and the current connection
- Transaction control routines
- Routines for server-side prepared statements
- Administrative routines to control server operation
- Thread routines for writing threaded clients
- Routines that generate debugging information

Unless otherwise indicated, the data types and functions listed here have been present in the client library at least as early as MySQL 5.5.0. Changes made since then are so noted.

The examples in this appendix are only brief code fragments. For complete programs and instructions for writing them, see Chapter 7, “Writing MySQL Programs Using C.”

G.1 Compiling and Linking

At the source level, the interface to the C client library is defined in a set of header files. Generally, MySQL programs include at least the following three files:

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

To tell the C compiler where to find these files, you might need to specify an `-I`*`path_name`* option, where *`path_name`* is the pathname to the directory where the MySQL header files are installed. For example, if your MySQL header files are installed in `/usr/include/mysql` or `/usr/local/mysql/include`, compile a source file `my_func.c` by using commands that look something like this:

```
% gcc -I/usr/include/mysql -c my_func.c
% gcc -I/usr/local/mysql/include -c my_func.c
```

If you need to access other MySQL header files, they are located in the same directory as `mysql.h`. For example, `mysql_com.h` contains constants and macros for interpreting query result metadata. The header files `errmsg.h` and `mysqld_error.h` contain constants for error codes. Note that although you might want to look at `mysql_com.h` to see what's in it, you need not include this file explicitly. `mysql.h` does so, so including `mysql.h` gives your program access to the contents of `mysql_com.h` as well.

A MySQL program can communicate as a client to a standalone MySQL server using the regular client/server protocol, or it can use an embedded server that is linked directly into the program binary. By proper use of the C API `mysql_library_init()` and `mysql_library_end()` initialization and termination routines, a program can be written so that either server type can be used (see Section G.3.1, "Client Library Initialization and Termination Routines"). The type of server to use is determined by which library you link the program against to produce the executable image:

- A program acts as a client of a standalone server if you link it against the `libmysqlclient` library. To link this library into your program, specify `-lmysqlclient` on the link command. You'll probably also need to tell the linker where to find the library using a `-L`*`path_name`* option, where *`path_name`* is the pathname to the directory where the library is installed. For example:

  ```
  % gcc -o myprog my_main.o my_func.o -L/usr/local/mysql/lib
      -lmysqlclient
  ```

- A program uses the embedded server if you link it against the `libmysqld` library. To link this library into your program, specify `-lmysqld` on the link command:

  ```
  % gcc -o myprog my_main.o my_func.o -L/usr/local/mysql/lib -lmysqld
  ```

If a link command fails with "unresolved symbol" errors, you'll need to specify additional libraries for the linker to search. Common examples include the math library (`-lm`) and the `zlib` library (`-lz` or `-lgz`).

The `mysql_config` utility provides an easy way to determine the proper header file directories for compiling or library flags for linking. Invoke it as follows to find out which flags are appropriate for your system:

- Compilation flags:

```
% mysql_config --include
-I'/usr/local/mysql/include/mysql'
```

- Flags for linking a client program:

```
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

- Flags for linking the embedded server:

```
% mysql_config --libmysqld-libs
-L'/usr/local/mysql/lib/mysql' -lmysqld -lz -lcrypt -lnsl -lm
```

The output shown is illustrative, but likely will differ on your system.

G.2 C API Data Structures

Data structures for the MySQL client library represent the kinds of information you deal with in the course of a session with the server. There are structures for the connection itself, for results from a query, for a row within a result, and for metadata (descriptive information about the columns making up a result). The terms "column" and "field" are synonymous in the following discussion.

G.2.1 Scalar Data Types

MySQL's scalar data types represent values such as very large integers, boolean values, and field or row offsets.

- `my_bool`

  A boolean type, used for the return value of `mysql_change_user()` and `mysql_thread_init()`.

- `my_ulonglong`

  A long integer type, used for the return value of functions that return row counts or other potentially large numbers, such as `mysql_affected_rows()`, `mysql_num_rows()`, and `mysql_insert_id()`. To print a `my_ulonglong` value, cast it to `unsigned long` and use a format of `%lu`. For example:

  ```
  printf ("Row count = %lu\n", (unsigned long) mysql_affected_rows (conn));
  ```

  The value will not print correctly on some systems if you don't do this. However, if the value to be printed might exceed the maximum permitted by `unsigned long` (2^{32}–1), `%lu` won't work, either. You'll need to check your `printf()` documentation to see

whether there is some implementation-specific means of printing the value. For example, a `%llu` format specifier might be available.

- `MYSQL_FIELD_OFFSET`

  This data type is used by functions such as `mysql_field_seek()` and `mysql_field_tell()` to represent offsets within the set of `MYSQL_FIELD` structures for the current result set.

- `MYSQL_ROW_OFFSET`

  This data type is used by functions such as `mysql_row_seek()` and `mysql_row_tell()` to represent offsets within the set of rows for the current result set.

G.2.2 Nonscalar Data Structures

MySQL's nonscalar data structures represent structures or arrays. Any instance of a `MYSQL`, `MYSQL_RES`, or `MYSQL_STMT` structure should be considered a "black box." That is, you should refer only to the structure itself, not to members within the structure. The `MYSQL_ROW`, `MYSQL_FIELD`, `MYSQL_BIND`, and `MYSQL_TIME` structures do not have the same restriction. Each one has members that you can access freely to obtain data and metadata returned as a result of a query. The `MYSQL_BIND` and `MYSQL_TIME` structures are used both for transmitting data to the server and receiving results from the server.

- `MYSQL`

  The primary client library type is the `MYSQL` structure, which is used for connection handlers. A handler contains information about the state of a connection with a server. To open a session with the server, initialize a `MYSQL` structure with `mysql_init()`, then pass it to `mysql_real_connect()`. After you've established the connection, use the handler to issue SQL statements, generate result sets, get error information, and so forth. When you're done with the connection, pass the handler to `mysql_close()`, after which you should no longer use it.

- `MYSQL_FIELD`

  The client library uses `MYSQL_FIELD` structures to represent metadata about the columns in the result set, one structure per column. To determine the number of `MYSQL_FIELD` structures in the set, call `mysql_num_fields()`. To access successive field structures, call `mysql_fetch_field()`. To move back and forth among structures, call `mysql_field_tell()` and `mysql_field_seek()`. As of MySQL 5.5.3, `MYSQL_FIELD` also serves to provide metadata about `IN` and `OUT` stored-procedure parameters when a procedure is invoked using a prepared `CALL` statement.

  The `MYSQL_FIELD` structure is useful for presenting or interpreting the contents of data rows. It looks like this:

```
typedef struct st_mysql_field {
  char *name;
  char *org_name;
  char *table;
```

```
  char *org_table;
  char *db;
  char *catalog;
  char *def;
  unsigned long length;
  unsigned long max_length;
  unsigned int name_length;
  unsigned int org_name_length;
  unsigned int table_length;
  unsigned int org_table_length;
  unsigned int db_length;
  unsigned int catalog_length;
  unsigned int def_length;
  unsigned int flags;
  unsigned int decimals;
  unsigned int charsetnr;
  enum enum_field_types type;
} MYSQL_FIELD;
```

`MYSQL_FIELD` structure members have the following meanings. String-valued members are null-terminated.

- `char *name`

  The column name. For a column calculated as the result of an expression, `name` is that expression in string form. If a column or expression is given an alias, `name` is the alias name. For example, the following query results in `name` values of `"mycol"`, `"4*(mycol+1)"`, `"mc"`, and `"myexpr"`:

  ```
  SELECT mycol, 4*(mycol+1), mycol AS mc, 4*(mycol+1) AS myexpr ...
  ```

  For a procedure parameter, `name` is the parameter name.

- `char *org_name`

  This member is like `name`, except that column aliases are ignored. That is, `org_name` represents the original column name. For a column calculated as the result of an expression, `org_name` is an empty string.

- `char *table`

  The name of the table that the column comes from. For a column selected from a view, `table` is the view name. If the table or view was given an alias, `table` is the alias name. For a column calculated as the result of an expression, `table` is an empty string. For example, if you issue a query like the following, the table name for the first column is `mytbl`, whereas the table name for the second column is the empty string:

  ```
  SELECT mycol, mycol+0 FROM mytbl ...
  ```

  For a procedure parameter, `table` is the procedure name.

- `char *org_table`

  This member is similar to `table`, except that table aliases are ignored. That is, `org_table` represents the original table name. For a column selected from a view, `table` is the underlying table name. For a column calculated as the result of an expression, `org_table` is an empty string.

- `char *db`

  The database in which the table containing the column is located. For a column calculated as the result of an expression, `db` is an empty string. For a procedure parameter, `db` is the database containing the procedure.

- `char *catalog`

  The catalog name. This value is always `"def"`.

- `char *def`

  The default value for the column. This member of the `MYSQL_FIELD` structure is set only for result sets obtained by calling `mysql_list_fields()` (a deprecated function), and is `NULL` otherwise.

  Default values for table columns also can be obtained by executing a `SHOW COLUMNS FROM` *`tbl_name`* statement and examining the result set.

- `unsigned long length`

  The length of the column, as specified in the `CREATE TABLE` statement used to create the table. For a column calculated as the result of an expression, the length is determined from the elements in the expression.

- `unsigned long max_length`

  The length of the longest column value present in the result set. For example, if a string column in a result set contains the values `"Bill"`, `"Jack"`, and `"Belvidere"`, the value of `max_length` for the column is 9.

  Result set values are returned as strings, so this length refers to the longest string representation of the values in the result, even for nonstring columns.

  Because the `max_length` value can be determined only after all the rows have been seen, it is meaningful only for result sets created with `mysql_store_result()`. `max_length` is 0 for result sets created with `mysql_use_result()`.

- `unsigned int name_length`, `org_name_length`, `table_length`, `org_table_length`, `db_length`, `catalog_length`, `def_length`

  The lengths of the `name`, `org_name`, `table`, `org_table`, `db`, `catalog`, and `def` members, respectively.

- `unsigned int flags`

  The `flags` member specifies attributes for the columns. Within the `flags` value, attributes are represented by individual bits, which may be tested using the bitmask constants shown in Table G.1. For example, to determine whether a column's values are `UNSIGNED`, test the `flags` value like this:

  ```
  if (field->flags & UNSIGNED_FLAG)
    printf ("%s values are UNSIGNED\n", field->name);
  ```

Table G.1 **`MYSQL_FIELD flags` Member Values**

`flags` Value	**Meaning**
`AUTO_INCREMENT_FLAG`	Column has the `AUTO_INCREMENT` attribute
`BINARY_FLAG`	Column has the `BINARY` attribute
`MULTIPLE_KEY_FLAG`	Column is a part of a nonunique index
`NOT_NULL_FLAG`	Column cannot contain `NULL` values
`NO_DEFAULT_VALUE_FLAG`	Column definition has no `DEFAULT` clause
`NUM_FLAG`	Column is numeric
`PRI_KEY_FLAG`	Column is a part of a `PRIMARY KEY`
`UNIQUE_KEY_FLAG`	Column is a part of a `UNIQUE` index
`UNSIGNED_FLAG`	Column has the `UNSIGNED` attribute
`ZEROFILL_FLAG`	Column has the `ZEROFILL` attribute

`NUM_FLAG` is true for columns that have a type of `MYSQL_TYPE_DECIMAL`, `MYSQL_TYPE_NEWDECIMAL`, `MYSQL_TYPE_TINY`, `MYSQL_TYPE_SHORT`, `MYSQL_TYPE_LONG`, `MYSQL_TYPE_FLOAT`, `MYSQL_TYPE_DOUBLE`, `MYSQL_TYPE_NULL`, `MYSQL_TYPE_LONGLONG`, `MYSQL_TYPE_INT24`, or `MYSQL_TYPE_YEAR`.

The `NO_DEFAULT_VALUE_FLAG` is true if there is no `DEFAULT` clause in the column definition, except for columns that permit `NULL` or that have the `AUTO_INCREMENT` attribute. Such columns have an implicit default of `NULL` or the next sequence value, respectively.

A few `flags` constants indicate column data types rather than column attributes; they are now deprecated because you should use `field->type` to determine the data type. Table G.2 lists these deprecated constants.

Table G.2 **Deprecated `MYSQL_FIELD flags` Member Values**

`flags` Value	Meaning
`BLOB_FLAG`	Column contains `BLOB` or `TEXT` values
`ENUM_FLAG`	Column contains `ENUM` values
`SET_FLAG`	Column contains `SET` values
`TIMESTAMP_FLAG`	Column contains `TIMESTAMP` values

- `unsigned int decimals`

  The number of decimals for numeric columns, zero for nonnumeric columns. For example, the `decimals` value is 3 for a `DECIMAL(8,3)` column, but 0 for a `BLOB` column. As of MySQL 5.6.4, this member also indicates fractional seconds precision for temporal columns.

- `unsigned int charsetnr`

  The character set/collation number. If you must distinguish whether a string column contains binary or nonbinary (character) data, `charsetnr` is 63 for binary strings.

- `enum enum_field_types type`

  The data type. For a column calculated as the result of an expression, the type is determined from the types of the elements in the expression. For example, if `mycol` is a `VARCHAR(20)` column, `type` is `MYSQL_TYPE_VAR_STRING`, whereas `type` for `LENGTH(mycol)` is `MYSQL_TYPE_LONGLONG`. The possible `type` values are listed in `mysql_com.h` and shown in Table G.3.

Table G.3 **`MYSQL_FIELD type` Member Values**

`type` Value	SQL Data Type
`MYSQL_TYPE_TINY`	`TINYINT`
`MYSQL_TYPE_SHORT`	`SMALLINT`
`MYSQL_TYPE_INT24`	`MEDIUMINT`
`MYSQL_TYPE_LONG`	`INT`
`MYSQL_TYPE_LONGLONG`	`BIGINT`
`MYSQL_TYPE_DECIMAL`	`DECIMAL`, `NUMERIC`
`MYSQL_TYPE_NEWDECIMAL`	`DECIMAL`, `NUMERIC`
`MYSQL_TYPE_DOUBLE`	`DOUBLE`, `REAL`
`MYSQL_TYPE_FLOAT`	`FLOAT`

type Value	SQL Data Type
MYSQL_TYPE_STRING	CHAR
MYSQL_TYPE_VAR_STRING	VARCHAR
MYSQL_TYPE_BLOB	BLOB, TEXT
MYSQL_TYPE_ENUM	ENUM
MYSQL_TYPE_SET	SET
MYSQL_TYPE_DATE	DATE
MYSQL_TYPE_DATETIME	DATETIME
MYSQL_TYPE_TIME	TIME
MYSQL_TYPE_TIMESTAMP	TIMESTAMP
MYSQL_TYPE_YEAR	YEAR
MYSQL_TYPE_GEOMETRY	Spatial type
MYSQL_TYPE_BIT	BIT
MYSQL_TYPE_NULL	NULL

- MYSQL_RES

  Statements such as SELECT or SHOW that return data to the client do so by means of a result set, represented as a MYSQL_RES structure. This structure contains information about the rows returned by the query.

  After a statement generates a result set, you can call API functions to get result data (the data values in each row of the set) or metadata (information about the result, such as how many columns there are, their types, their lengths, and so forth).

- MYSQL_ROW

  The MYSQL_ROW type contains the values for one row of data, represented as an array of strings. All values are returned in string form (even numbers), except that if a value in a row is NULL, it is represented in the MYSQL_ROW structure by a C NULL pointer.

  The number of values in a row is given by mysql_num_fields(). The *i*-th column value in a row is given by row[*i*]. Values of *i* range from 0 to mysql_num_fields(res_set)−1, where res_set is a pointer to a MYSQL_RES result set.

  The MYSQL_ROW type is already a pointer, so you define a row variable like this:

  ```
  MYSQL_ROW row;        /* correct */
  ```

  Not like this:

  ```
  MYSQL_ROW *row;       /* incorrect */
  ```

Values in a MYSQL_ROW array have a terminating null byte, so nonbinary values may be treated as null-terminated strings. However, data values that may contain binary data might contain null bytes internally and should be treated as counted strings. To get a pointer to an array that contains the lengths of the values in the row, call mysql_fetch_lengths() like this:

```
unsigned long *length;
length = mysql_fetch_lengths (res_set);
```

The length of the *i*-th column value in a row is given by length[*i*]. If the column value is NULL, the length is zero.

- MYSQL_STMT

  A prepared statement handler. To create a handler, call mysql_stmt_init(). This function returns a pointer to the new handler, which can be used to prepare a statement, execute it, and so on. When you're done with the handler, pass it to mysql_stmt_close(), after which the handler should no longer be used.

- MYSQL_BIND

  This structure is used with prepared statements and has both input and output purposes.

  For input, MYSQL_BIND structures contain data to be transmitted to the server to be bound to the parameters of a prepared statement before the statement is executed. Set up an array of structures, then bind them to the statement by calling mysql_stmt_bind_param() before calling mysql_stmt_execute() to execute the statement. The array must contain one MYSQL_BIND structure per parameter.

  Input strings are assumed to be represented in the character set indicated by the character_set_client system variable. If this differs from the character set of the column into which the value is stored, conversion into the column character set occurs on the server side.

  For output, after a prepared statement that produces a result set is executed, MYSQL_BIND structures are used to fetch data values from the result set. Set up an array of structures, and then bind them to the statement by calling mysql_stmt_bind_result() before fetching result set rows with mysql_stmt_fetch(). The array must contain one MYSQL_BIND structure per column of the result set.

  Output strings are represented using the character set indicated by the character_set_results system variable.

  The MYSQL_BIND structure contains several members, but only some of them should be considered public. The public members are shown here:

```
typedef struct st_mysql_bind
{
  unsigned long         *length;
  my_bool               *is_null;
  void                  *buffer;
```

```
  my_bool               *error;
  unsigned long         buffer_length;
  enum enum_field_types buffer_type;
  my_bool               is_unsigned;
  ...
} MYSQL_BIND;
```

One `MYSQL_BIND` structure should be bound to each parameter of a prepared statement. The following list describes the purpose of each `MYSQL_BIND` member, for both input and output. True indicates a nonzero value; false indicates a zero value.

- `enum enum_field_types buffer_type`

  The data type of the C language variable bound to the parameter. This member must always be set to a `MYSQL_TYPE_`*`XXX`* value.

  For input, this is the type of the variable containing the value that you are sending to the server.

  For output, this is the type of the variable into which you want to receive the value returned by the server.

  Table G.4 and Table G.5 show the `buffer_type` values that correspond to C variable data types for input and output, respectively. In both directions, if the C variable type does not correspond to the SQL type of the value on the server side, conversion occurs when possible. If the C and SQL types are directly compatible, no conversion need be performed, which increases performance.

- `void *buffer`

  A pointer to the variable used to send or receive a data value.

  For input, this points to the variable that holds the data value to be sent to the server.

  For output, this points to the variable where the value returned by the server should be stored.

  `buffer` is always the address of the storage variable. For numeric types, `buffer` points to a scalar variable. For string types, it points to a `char` buffer. For temporal types, it points to a `MYSQL_TIME` structure. The variable type is indicated by the `buffer_type` value. If the variable is `unsigned`, set the `is_unsigned` value to true.

- `unsigned long buffer_length`

  The actual size in bytes of the buffer pointed to by `buffer`, both for input and output. This applies to string types, either binary or nonbinary, which can vary in length, and to output `BIT` values. For other data types, the `buffer_type` value determines the buffer length.

- `unsigned long *length`

  A pointer to a variable that indicates the number of bytes in the transferred data value. Like `buffer_length`, this member needs to be set only for string types and output `BIT` values. For numeric and temporal types, the data type determines the length.

  For input, set the pointed-to variable to indicate how many bytes to send to the server.

  For output, the pointed-to variable will be set by `mysql_stmt_fetch()`, and the return value of that function determines how to interpret the variable value. If `mysql_stmt_fetch()` returns 0 (success), `*length` is the actual length of the returned data value. If `mysql_stmt_fetch()` returns `MYSQL_DATA_TRUNCATED`, `*length` is the length the value would have had no truncation occurred, and the actual length is the minimum of `*length` and `buffer_length`.

- `my_bool *is_null`

  A pointer to a variable that indicates whether the data value corresponds to a `NULL` value.

  For input, set the pointed-to variable true or false to indicate whether the value being sent to the server is `NULL` or `NOT NULL`. Special cases: If the value bound to this parameter will never be `NULL`, you can set `is_null` to zero rather than to the address of a `my_bool` variable. If the value will always be `NULL`, set `buffer_type` to `MYSQL_TYPE_NULL` and the other `MYSQL_BIND` members do not matter.

  For output, the pointed-to variable will be set true or false to indicate whether the value returned by the server is `NULL` or `NOT NULL`.

- `my_bool is_unsigned`

  A flag that indicates whether the variable pointed to by `buffer` is an `unsigned` C variable, both for input and output. This member need be used only for C data types that can be `unsigned` (`char` and the integer types).

  `is_unsigned` applies to the C variable bound to the `MYSQL_BIND` structure, not to the SQL value on the server side. The client library uses `is_unsigned` to determine whether to perform sign conversion between the C and SQL values.

- `my_bool *error`

  For output, this points to a variable that indicates whether a value was fetched without truncation. After fetching a row, the pointed-to variable is false if there was no error, true if there was data truncation such as for a numeric value that is out of range or a string value that is too long. Truncation checking occurs by default; this can be changed by calling `mysql_options()` with the `MYSQL_REPORT_DATA_TRUNCATION` option.

Table G.4 shows the buffer_type values to use for C language variables used to send data values from the server. If the variable is unsigned, set the is_unsigned value to true. If the SQL value on the server side has the data type shown in the table, the input value can be used without conversion. For example, if you use a short int to supply a value for a SMALLINT, no conversion need be done. If short int supplies a value for a DECIMAL, a conversion is done.

Table G.4 **Input MYSQL_BIND buffer_type Values**

Input C Variable Type	buffer_type Value	Compatible SQL Value Type
signed char	MYSQL_TYPE_TINY	TINYINT
short int	MYSQL_TYPE_SHORT	SMALLINT
int	MYSQL_TYPE_LONG	INT
long long int	MYSQL_TYPE_LONGLONG	BIGINT
float	MYSQL_TYPE_FLOAT	FLOAT
double	MYSQL_TYPE_DOUBLE	DOUBLE
MYSQL_TIME	MYSQL_TYPE_TIME	TIME
MYSQL_TIME	MYSQL_TYPE_DATE	DATE
MYSQL_TIME	MYSQL_TYPE_DATETIME	DATETIME
MYSQL_TIME	MYSQL_TYPE_TIMESTAMP	TIMESTAMP
char[]	MYSQL_TYPE_STRING	TEXT, CHAR, VARCHAR
char[]	MYSQL_TYPE_BLOB	BLOB, BINARY, VARBINARY
	MYSQL_TYPE_NULL	NULL

MYSQL_TYPE_STRING and MYSQL_TYPE_BLOB are used for nonbinary and binary strings, respectively.

MYSQL_TYPE_NULL should be used only when an input parameter is always NULL. Otherwise, set the buffer_type value to one of the other MYSQL_TYPE_*XXX* values and set the is_null member appropriately each time you execute the statement to indicate whether the parameter is NULL.

Table G.5 shows the buffer_type values to use for C language variables used to receive data values from the server. If the variable is unsigned, set the is_unsigned value to true. If the C variable used to retrieve the value has the type shown in the table, the SQL value received from the server can be used without conversion. If you fetch a SMALLINT into a short int, no conversion need be done. If you fetch it into a char[], the value is converted to string form.

Table G.5 **Output `MYSQL_BIND buffer_type` Values**

Source SQL Value Type	`buffer_type` Value	Compatible C Variable Type
TINYINT	MYSQL_TYPE_TINY	signed char
SMALLINT	MYSQL_TYPE_SHORT	short int
MEDIUMINT	MYSQL_TYPE_INT24	int
INT	MYSQL_TYPE_LONG	int
BIGINT	MYSQL_TYPE_LONGLONG	long long int
FLOAT	MYSQL_TYPE_FLOAT	float
DOUBLE	MYSQL_TYPE_DOUBLE	double
DECIMAL	MYSQL_TYPE_NEWDECIMAL	char[]
YEAR	MYSQL_TYPE_SHORT	short int
TIME	MYSQL_TYPE_TIME	MYSQL_TIME
DATE	MYSQL_TYPE_DATE	MYSQL_TIME
DATETIME	MYSQL_TYPE_DATETIME	MYSQL_TIME
TIMESTAMP	MYSQL_TYPE_TIMESTAMP	MYSQL_TIME
CHAR, BINARY	MYSQL_TYPE_STRING	char[]
VARCHAR, VARBINARY	MYSQL_TYPE_VAR_STRING	char[]
TINYBLOB, TINYTEXT	MYSQL_TYPE_TINY_BLOB	char[]
BLOB, TEXT	MYSQL_TYPE_BLOB	char[]
MEDIUMBLOB, MEDIUMTEXT	MYSQL_TYPE_MEDIUM_BLOB	char[]
LONGBLOB, LONGTEXT	MYSQL_TYPE_LONG_BLOB	char[]
BIT	MYSQL_TYPE_BIT	char[]

DECIMAL and BIT values are returned as strings by default. If you specify a char[] variable to receive a DECIMAL value, you get the string representation of the numeric value. If you specify a numeric variable instead, the string is converted to numeric form. To receive a BIT value as a number, cast it to numeric form in your query (for example, SELECT my_bit_val+0 ...) and bind an integer variable to the MYSQL_BIND structure.

To distinguish nonbinary from binary string columns, use mysql_stmt_result_metadata() to get the result set metadata and check the column charsetnr member. A value of 63 indicates a binary string; anything else indicates a nonbinary string.

- MYSQL_TIME

  This structure is used to send temporal values to the server or receive them from the server. To associate a MYSQL_TIME structure with a MYSQL_BIND structure, set the buffer member of the MYSQL_BIND to the address of a MYSQL_TIME variable.

MYSQL_TIME is used for DATETIME, TIMESTAMP, DATE, and TIME types, but the structure members that do not apply to a given type are ignored. For example, the month, year, and day members do not apply to TIME values, and the hour, minute, and second members do not apply to DATE values.

The MYSQL_TIME structure contains several members, but only some of them should be considered public. The public members are shown here:

```
typedef struct st_mysql_time
{
  unsigned int  year;
  unsigned int  month;
  unsigned int  day;
  unsigned int  hour;
  unsigned int  minute;
  unsigned int  second;
  unsigned long second_part;
  my_bool       neg;
  ...
} MYSQL_TIME
```

The members are used as follows:

- year, month, day

  The year, month, and day parts of temporal values that contain a date part.

- hour, minute, second, second_part

  The hour, minute, second, and fractional second parts of temporal values that contain a time part.

- neg

  A flag that indicates whether the temporal value contained in the MYSQL_TIME structure is negative.

G.2.3 Accessor Macros

mysql.h contains a few macros that enable you to test MYSQL_FIELD members more conveniently. IS_NUM() tests the type member; the others listed here test the flags member.

- IS_BLOB() is true if the column is one of the BLOB or TEXT types. However, this macro tests the deprecated BLOB_FLAG bit of the flags member, so IS_BLOB() is deprecated as well.
- IS_NOT_NULL() is true if the column cannot contain NULL values:

  ```
  if (IS_NOT_NULL (field->flags))
    printf ("Field %s values cannot be NULL\n", field->name);
  ```

- `IS_NUM()` is true (nonzero) if values in the column have a numeric type:

```
if (IS_NUM (field->type))
  printf ("Field %s is numeric\n", field->name);
```

- `IS_PRI_KEY()` is true if the column is part of a `PRIMARY KEY`:

```
if (IS_PRI_KEY (field->flags))
  printf ("Field %s is part of primary key\n", field->name);
```

G.3 C API Functions

Client library functions for the C API are described in detail in the following sections, grouped by category and listed alphabetically within category. Certain parameter names recur throughout the function descriptions and have the following conventional meanings:

- `conn` is a pointer to the `MYSQL` connection handler for a server connection.
- `res_set` is a pointer to a `MYSQL_RES` result set structure.
- `field` is a pointer to a `MYSQL_FIELD` column information structure.
- `row` is a `MYSQL_ROW` data row from a result set.
- `row_num` is a row number within a result set, from 0 to one less than the number of rows.
- `col_num` is a column number within a row of a result set, from 0 to one less than the number of columns.
- `stmt` is a handler for a prepared statement.

For brevity, where these parameters are not mentioned in the descriptions of functions in which they occur, assume the meanings just given.

G.3.1 Client Library Initialization and Termination Routines

This section describes routines that initialize and terminate the C API library. There are actually two such libraries, but the interface to them is the same so that a given program can use either one depending on which library you link the program against to produce the executable image:

- `libmysqlclient` is for programs that connect to a standalone MySQL server.
- `libmysqld` is for programs that include an embedded server in the program itself.

Using the `mysql_library_init()` and `mysql_library_end()` routines within your program to initialize and terminate the client library makes it possible to use the same source code to produce a client for a standalone server or one that uses the embedded server, depending on which library you select at link time. For information about linking in the appropriate C API library, see Section G.1, “Compiling and Linking.”

- ```
  void
  mysql_library_end (void);
  ```

  Terminates the client library. Call this function after you're done communicating with the server. If the program uses the embedded server library, this routine shuts down the embedded server.

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

  Initializes the client library. Returns zero for success and nonzero otherwise. This function must be called before calling any other `mysql_xxx()` functions.

  If the program uses the embedded server library, this routine initializes the embedded server. In this case, the `argc` and `argv` arguments are used like the standard arguments passed to `main()` in C programs: `argc` is the argument count; if there are none, `argc` should be zero. Otherwise, `argc` should be the number of arguments passed to the server. `argv` is an array of null-terminated strings containing the arguments. `argv[0]` will be ignored.

  The `groups` argument is an array of null-terminated strings indicating which option file groups the embedded server should read. The final element of the array should be `NULL`. If `group` itself is `NULL`, the server reads the `[server]` and `[embedded]` option file groups by default. Specify group names in the `groups` array without the surrounding '`[`' and '`]`' characters.

- ```
  void
  mysql_server_end (void);
  ```

  This routine is a synonym for `mysql_library_end()`.

- ```
  int
  mysql_server_init (int argc, char **argv, char **groups);
  ```

  This routine is a synonym for `mysql_library_init()`.

G.3.2 Connection Management Routines

These functions enable you to establish and terminate connections to a server, to set options affecting how connection establishment occurs, to re-establish connections that have timed out, and to change aspects of the connection such as the current username or character set.

A typical sequence involves calling `mysql_init()` to initialize a connection handler, `mysql_real_connect()` to establish the connection, and `mysql_close()` to terminate the connection when you are done with it. If it's necessary to indicate special options or set up an encrypted SSL connection, call `mysql_options()` or `mysql_ssl_set()` after `mysql_init()` and before `mysql_real_connect()`.

- ```
  my_bool
  mysql_change_user (MYSQL *conn,
                     const char *user_name,
                     const char *password,
                     const char *db_name);
  ```

  Changes the user and the default database for the connection specified by `conn`. The database becomes the default for table references that do not include a database specifier. If `db_name` is `NULL`, no default database is selected.

  `mysql_change_user()` returns zero for success (the user is permitted to connect to the server and, if a database was specified, has permission to access it) and nonzero if an error occurred. If the function fails, the current user and database remain unchanged.

  It is faster to use `mysql_change_user()` to change the current user than to close the connection and open it again with different parameters. This function can also be used to implement persistent connections for a program that serves different users during the course of its execution.

- ```
  void
  mysql_close (MYSQL *conn);
  ```

  Closes the connection specified by `conn`. Call this routine when you are done with a server session. If the connection handler was allocated automatically by `mysql_init()`, `mysql_close()` deallocates it.

  It is unnecessary to call `mysql_close()` if the attempt to open a connection fails. However, you might want to do so if `mysql_init()` allocated the handler, so that it can be disposed of.

- ```
  void
  mysql_get_character_set_info (MYSQL *conn,
                                MY_CHARSET_INFO *cs_info);
  ```

  Retrieves information about the current client character set. `cs_info` points to the `MY_CHARSET_INFO` structure into which the information should be placed. The structure looks like this:

  ```
  typedef struct character_set
  {
    unsigned int      number;     /* character set number              */
    unsigned int      state;      /* character set state               */
    const char        *csname;    /* collation name                    */
    const char        *name;      /* character set name                */
    const char        *comment;   /* comment                           */
    const char        *dir;       /* character set directory           */
    unsigned int      mbminlen;   /* min. length for multibyte strings */
    unsigned int      mbmaxlen;   /* max. length for multibyte strings */
  } MY_CHARSET_INFO;
  ```
  ```

- ```
  const char *
  mysql_get_ssl_cipher (MYSQL *conn);
  ```

  Returns a null-terminated string containing the name of the SSL cipher used for the connection, or `NULL` if there is no cipher.

- ```
  MYSQL *
  mysql_init (MYSQL *conn);
  ```

  Initializes a connection handler and returns a pointer to it. If the parameter points to an existing `MYSQL` handler structure, `mysql_init()` initializes it and returns its address:

  ```
  MYSQL conn_struct, *conn;
  conn = mysql_init (&conn_struct);
  ```

  If the parameter is `NULL`, `mysql_init()` allocates a new handler, initializes it, and returns its address:

  ```
  MYSQL *conn;
  conn = mysql_init (NULL);
  ```

  The second approach is preferable over the first; letting the client library allocate and initialize the handler itself avoids problems that may arise with shared libraries if you upgrade MySQL to a newer version that uses a different internal organization for the `MYSQL` structure.

  If `mysql_init()` fails, it returns `NULL`. This may happen if `mysql_init()` cannot allocate a new handler.

  If `mysql_init()` allocates the handler, `mysql_close()` deallocates it automatically when you close the connection.

- ```
  int
  mysql_options (MYSQL *conn,
                 enum mysql_option option,
                 const void *arg);
  ```

  This function enables you to tailor connection behavior more precisely than is possible with `mysql_real_connect()` alone. Call it after `mysql_init()` and before `mysql_real_connect()`. You can call `mysql_options()` multiple times to set several options. If you call `mysql_options()` multiple times to set a given option, the most recent option value applies.

  The `option` argument specifies which connection option to set. Additional information needed to set the option, if any, is specified by the `arg` argument, which is always interpreted as a pointer. Pass an `arg` value of `NULL` for options that require no additional information. (Assume that this is the case for option descriptions that say nothing about `arg`.) String values of `arg` should be null-terminated.

  `mysql_options()` returns zero for success and nonzero if the `option` value is unknown.
  ```

The mysql_options() calls in the following example have the effect of setting connection options so that mysql_real_connect() reads C:\my.ini.extra for information from the [client] and [mygroup] groups, connects using a named pipe and a timeout of 10 seconds, and executes a SET NAMES 'utf8' statement after the connection has been established.

```
MYSQL *conn;
unsigned int timeout;

if ((conn = mysql_init (NULL)) == NULL)
  ... deal with error ...
mysql_options (conn, MYSQL_READ_DEFAULT_FILE, "C:/my.ini.extra");
mysql_options (conn, MYSQL_READ_DEFAULT_GROUP, "mygroup");
mysql_options (conn, MYSQL_OPT_NAMED_PIPE, NULL);
timeout = 10;
mysql_options (conn, MYSQL_OPT_CONNECT_TIMEOUT, (char *) &timeout);
mysql_options (conn, MYSQL_INIT_COMMAND, "SET NAMES 'utf8'");
if (mysql_real_connect (conn, ...) == NULL)
   ... deal with error ...
```

The following options are available. Those indicated as applying to use of an embedded server are ignored if the program is linked against libmysqlclient rather than libmysqld.

- MYSQL_DEFAULT_AUTH

  The authentication plugin to use. arg points to a string naming the plugin. This option was introduced in MySQL 5.5.7.

- MYSQL_INIT_COMMAND

  Specifies a statement to execute after connecting to the server. arg points to a string containing the statement. The statement is executed after reconnecting as well (for example, if you call mysql_ping()). Any result set returned by the statement is discarded.

- MYSQL_OPT_BIND

  For client hosts that have multiple network interfaces, this option enables the program to specify which one to use for connecting to the server. arg points to a string containing the hostname or IP address. This option was introduced in MySQL 5.6.1.

- MYSQL_OPT_COMPRESS

  Requests use of the compressed client/server communication protocol if the client and server both support it.

  It is also possible to specify compression when you call mysql_real_connect().

- MYSQL_OPT_CONNECT_TIMEOUT

  Specifies the connection timeout, in seconds. arg points to an unsigned int containing the timeout value.

- MYSQL_OPT_GUESS_CONNECTION

  If the program includes an embedded server, this option enables the server library to choose whether to use the embedded server library or a remote server. It "guesses" the use of a remote server if the hostname is set and is not localhost. "Guessing" is the default. Use MYSQL_OPT_USE_EMBEDDED_CONNECTION or MYSQL_OPT_USE_REMOTE_CONNECTION to force the type of connection.

- MYSQL_OPT_LOCAL_INFILE

  Enables or disables the use of LOAD DATA LOCAL. arg is NULL to disable this capability, or a pointer to an unsigned int that is zero or nonzero to disable or enable this capability. Attempts to enable LOAD DATA LOCAL are ineffective if the server has been configured to prohibit it.

- MYSQL_OPT_NAMED_PIPE

  Specifies that the connection to the server should use a named pipe. This option is for Windows clients only, and only for connections to Windows servers with named-pipe support enabled.

- MYSQL_OPT_PROTOCOL

  Specifies the protocol to use for connecting to the server, assuming that the server supports the protocol. arg points to an unsigned int value containing the protocol code. The permitted codes are MYSQL_PROTOCOL_MEMORY (shared memory), MYSQL_PROTOCOL_PIPE (Windows named pipe), MYSQL_PROTOCOL_SOCKET (Unix socket file), and MYSQL_PROTOCOL_TCP (TCP/IP).

- MYSQL_OPT_READ_TIMEOUT

  The timeout for reading from the server, in seconds. arg points to an unsigned int containing the timeout value. The effective timeout is three times the option value due to retries if the initial read fails.

- MYSQL_OPT_RECONNECT

  Enables or disables automatic reconnection behavior (off by default) if the connection goes down. arg points to a my_bool that is set true or false.

- MYSQL_OPT_SSL_CA, MYSQL_OPT_SSL_CAPATH, MYSQL_OPT_SSL_CERT, MYSQL_OPT_SSL_CIPHER, MYSQL_OPT_SSL_CRL, MYSQL_OPT_SSL_CRLPATH, MYSQL_OPT_SSL_KEY

  These options specify parameters for SSL connections. The arg value is a string corresponding to the value for the similarly named command-line option --ssl-ca, --ssl-capath, and so forth. See Section F.2.1.1, "Standard SSL Options."

  These options were introduced in MySQL 5.6.3.

- MYSQL_OPT_SSL_VERIFY_SERVER_CERT

  Enables or disables verification of the Common Name in the server's certificate (disabled by default). The value must match the hostname used for connecting to the server or the connection attempt fails. This helps prevent man-in-the-middle exploits. arg points to a my_bool that is set true or false.

- MYSQL_OPT_USE_EMBEDDED_CONNECTION

  If the program includes an embedded server, this option tells the server library to use the embedded server library rather than a remote server.

- MYSQL_OPT_USE_REMOTE_CONNECTION

  If the program includes an embedded server, this option tells the server library to use a remote server rather than the embedded server library.

- MYSQL_OPT_WRITE_TIMEOUT

  The timeout for writing to the server, in seconds. arg points to an unsigned int containing the timeout value. The effective timeout is net_retry_count times the option value due to retries if the initial write fails.

- MYSQL_PLUGIN_DIR

  The directory where client plugins are located. arg points to a string naming the directory. This option was introduced in MySQL 5.5.7.

- MYSQL_READ_DEFAULT_FILE

  Specifies an option file to read for connection parameters, rather than the usual option files that are searched by default if option files are read. arg points to a string containing the filename. Options will be read from the [client] group in the file. If you also use MYSQL_READ_DEFAULT_GROUP to specify a group name, options from that group will be read from the file, too.

- MYSQL_READ_DEFAULT_GROUP

  Specifies an option file group in which to look for option values. arg points to a string containing the group name. (Specify the group name without the surrounding '[' and ']' characters.) The named group will be read in addition to the [client] group. If you also name a particular option file with MYSQL_READ_DEFAULT_FILE, options are read from that file only. Otherwise, the client library looks for the options in the standard option files.

  If you specify neither MYSQL_READ_DEFAULT_FILE nor MYSQL_READ_DEFAULT_GROUP, no option files are read.

- MYSQL_REPORT_DATA_TRUNCATION

  Controls whether to report data truncation errors by means of the error member of MYSQL_BIND structures when the binary protocol for prepared statements is used. arg points to a my_bool variable that is zero or nonzero to disable or enable truncation reporting. Reporting is enabled by default.

- MYSQL_SECURE_AUTH

  Controls whether to require secure authentication. arg points to a my_bool variable that is zero or nonzero to permit or prohibit connecting to a server that does not support the password hashing improvements implemented in MySQL 4.1.

- MYSQL_SET_CHARSET_DIR

  Specifies the pathname of the directory where character set files are located. arg points to a string containing the pathname of a directory on the client host. Use this option when the client needs to access character sets that aren't compiled into the client library but for which definition files are available.

- MYSQL_SET_CHARSET_NAME

  Indicates the name of the default character set to use. arg points to a string containing the character set name, or can be MYSQL_AUTODETECT_CHARSET_NAME to enable autodetection of the character set from the operating system (for example, if the client user has the LANG or LC_ALL environment variable set).

- MYSQL_SET_CLIENT_IP

  If the program includes an embedded server that has authentication support, this option causes the server to treat the connection as having originated from the IP address given by arg, which points to the number specified as a string (for example, "192.168.3.12").

- MYSQL_SHARED_MEMORY_BASE_NAME

  Indicates the shared-memory name to use for shared-memory connections. arg points to a string containing the name. This option is for Windows clients only, and only for connections to Windows servers with shared-memory support enabled.

For Windows pathnames that are specified with the MYSQL_READ_DEFAULT_FILE or MYSQL_SET_CHARSET_DIR options, '\' characters can be given either as '/' or as '\\'.

If you use the MYSQL_READ_DEFAULT_FILE or MYSQL_READ_DEFAULT_GROUP options with mysql_options() to cause mysql_real_connect() to read option files, the following options are recognized:

```
bind-address=address (as of 5.6.1)
character-sets-dir=charset_directory_path
compress
connect-timeout=seconds
database=db_name
debug
default-auth=name (as of 5.5.7)
default-character-set=charset_name
disable-local-infile
host=host_name
```

```
init-command=stmt
interactive-timeout=seconds
local-infile[={0|1}]
max-allowed-packet=size
multi-queries
multi-results
multi-statements
password=your_pass
pipe
plugin-dir=dir_name (as of 5.5.7)
port=port_num
protocol=protocol_type
report-data-truncation
return-found-rows
secure-auth
shared-memory-base-name=name
socket=socket_name
ssl-ca=file_name
ssl-capath=dir_name
ssl-cert=file_name
ssl-cipher=str
ssl-crl=file_name (as of 5.6.3)
ssl-crlpath=dir_name (as of 5.6.3)
ssl-key=file_name
timeout=seconds
user=user_name
```

Instances of the `host`, `user`, `password`, `database`, `port`, or `socket` options found in option files are overridden if the corresponding argument to `mysql_real_connect()` is non-`NULL`.

The `multi-results` option is equivalent to passing `CLIENT_MULTI_RESULTS` in the `flags` argument to `mysql_real_connect()`. Either `multi-queries` or `multi-statements` is equivalent to passing `CLIENT_MULTI_STATEMENTS` in the `flags` argument to `mysql_real_connect()` (which also enables `CLIENT_MULTI_RESULTS`).

`timeout` is recognized but obsolete; use `connect-timeout` instead.

- ```
  int
  mysql_ping (MYSQL *conn);
  ```

  Checks whether the connection indicated by `conn` is still up. If not, and auto-reconnect is enabled, `mysql_ping()` reconnects using the same parameters that were used initially to make the connection. Thus, you should not call `mysql_ping()` without first successfully having called `mysql_real_connect()`. Returns zero if the connection was up or was successfully re-established, or nonzero if an error occurred or if the connection is down and auto-reconnect is disabled.
  ```

- ```
  MYSQL *
  mysql_real_connect (MYSQL *conn,
                      const char *host_name,
                      const char *user_name,
                      const char *password,
                      const char *db_name,
                      unsigned int port_num,
                      const char *socket_name,
                      unsigned long flags);
  ```

  Connects to a server and returns a pointer to the connection handler. `conn` should be a pointer to an existing connection handler that has been initialized by `mysql_init()`. The return value is the address of the handler for a successful connection, or `NULL` if an error occurred.

  If the connection attempt fails, you can pass the `conn` handler value to `mysql_errno()` and `mysql_error()` to obtain error information. However, do not pass the `conn` value to any client library routine that assumes a connection has been established successfully.

  The remaining arguments indicate how to connect to the server. For arguments specified as `NULL` or zero, the value can be supplied by options found in an option file that `mysql_real_connect()` reads. The client can cause `mysql_real_connect()` to read option files by calling `mysql_options()` with the `MYSQL_READ_DEFAULT_FILE` or `MYSQL_READ_DEFAULT_GROUP` options.

  `host_name` indicates the name of the MySQL server host. Table G.6 shows the connection protocol that the client uses for different kinds of `host_name` values for Unix and Windows clients. The table applies unless you have called `mysql_options()` with the `MYSQL_OPT_PROTOCOL` option to specify the protocol explicitly. The name `"localhost"` is special for Unix systems. It indicates that you want to connect using a Unix socket rather than a TCP/IP connection. To connect to a server running on the local host using TCP/IP, pass `"127.0.0.1"` (a string containing the IP address of the local host's loopback interface) for the `host_name` value, rather than passing the string `"localhost"`.

Table G.6 **Client Connection Protocol by Server Hostname Type**

Hostname Value	Unix Connection Protocol	Windows Connection Protocol
hostname	TCP/IP connection to the named host	TCP/IP connection to the named host
IP address	TCP/IP connection to the named host	TCP/IP connection to the named host
`localhost`	Unix socket file connection to the local host	Shared-memory connection (if available) to the local host; otherwise, a TCP/IP connection
`127.0.0.1`	TCP/IP connection to the local host	TCP/IP connection to the local host
  ```

Hostname Value	Unix Connection Protocol	Windows Connection Protocol
. (period)	Does not apply	Named-pipe connection to the local host
NULL	Unix socket file connection to the local host	A named-pipe connection is attempted first before falling back to TCP/IP

user_name is your MySQL username. If this is NULL, the client library sends a default name. Under Unix, the default is your login name. Under Windows, the default is your name as specified in the USER environment variable if that variable is set and "ODBC" otherwise.

password is your password. If this is NULL, you can connect only if the password is blank in the user grant table entry that matches your username and the host from which you are connecting.

db_name is the default database to use. If this is NULL, no default database is selected.

port_num is the port number to use for TCP/IP connections. If this is 0, the default port number is used.

socket_name is the Unix socket filename to use for connections to "localhost" under Unix, or the pipe name for named-pipe connections under Windows. If this is NULL, the default socket or pipe name is used.

The port number and socket filename are used according to the value of host_name, as described in Table G.6.

The flags value can be one or more of the values shown in the following list, or 0 to specify no options. These options affect the operation of the server. mysql_com.h lists other CLIENT_*XXX* values besides those in the list, but they are either unused or intended for internal use, so client programs should not specify them in the flags value.

- CLIENT_COMPRESS

  Requests use of the compressed client/server communication protocol if the client and server both support it.

- CLIENT_FOUND_ROWS

  For UPDATE statements and the ROW_COUNT() function, the server should return the number of rows matched rather than the number of rows changed. Use of this option may hinder the MySQL optimizer and make updates slower.

- CLIENT_IGNORE_SIGPIPE

  Prevents the client library from installing a handler for the SIGPIPE signal. This can be useful for an application that installs its own handler.

- `CLIENT_IGNORE_SPACE`

  Normally, names of built-in functions must be followed immediately by the parenthesis that begins the argument list, with no intervening spaces. This option tells the server to all spaces between the function name and the argument list, which also has the side effect of making all function names reserved words.

- `CLIENT_INTERACTIVE`

  Identifies the client as an interactive client. This tells the server that it can close the connection after a number of seconds of client inactivity equal to the server's `interactive_timeout` variable value. Normally, the value of the `wait_timeout` variable is used.

- `CLIENT_LOCAL_FILES`

  Enables the use of `LOAD DATA LOCAL`. This is ineffective if the server has been configured to always prohibit `LOAD DATA LOCAL`.

- `CLIENT_MULTI_RESULTS`

  Enables multiple result sets to be fetched with the `mysql_more_results()` and `mysql_next_result()` functions.

  You *must* specify this option if the program uses a `CALL` statement to invoke any stored procedures that return a result set. Otherwise, an error occurs. As of MySQL 5.5.3, this option is enabled by default.

- `CLIENT_MULTI_STATEMENTS`

  Enables multiple-statement execution. When this capability is turned on, you can send multiple statements to the server in a single string. This option also enables `CLIENT_MULTI_RESULTS` so that multiple result sets can be fetched.

- `CLIENT_NO_SCHEMA`

  Prohibits *`db_name.tbl_name.col_name`* syntax in SQL statements. If you specify this option, the server permits references only of the forms *`tbl_name.col_name`*, *`tbl_name`*, or *`col_name`*.

- `CLIENT_REMEMBER_OPTIONS`

  By default, if `mysql_real_connect()` fails, any `mysql_options()` calls made prior to the connection attempt must be repeated for the next attempt. This flag makes that unnecessary.

The flag values are bit values; combine them in additive fashion using either the `|` or the `+` operator. For example, the following expressions are equivalent:

```
CLIENT_COMPRESS | CLIENT_ODBC
CLIENT_COMPRESS + CLIENT_ODBC
```

- int
  mysql_select_db (MYSQL *conn, const char *db_name);

  Selects the database named by db_name as the default database, which becomes the default for table references that contain no explicit database specifier. If you do not have permission to access the database, mysql_select_db() fails.

  mysql_select_db() is most useful for changing databases within the course of a connection. Normally you specify the initial database when you call mysql_real_connect(), which is faster than calling mysql_select_db() after connecting.

  mysql_select_db() returns zero for success, and nonzero for failure.

- int
  mysql_set_character_set (MYSQL *conn, const char *cs_name);

  Sets the default character set for the connection (as though a SET NAMES statement had been executed). cs_name points to a string containing the character set name.

  mysql_set_character_set() returns zero for success, and nonzero for failure.

- my_bool
  mysql_ssl_set (MYSQL *conn,
  const char *key,
  const char *cert,
  const char *ca,
  const char *capath,
  const char *cipher);

  This function is used for setting up a secure connection over SSL to the MySQL server. If SSL support is not compiled into the client library, mysql_ssl_set() does nothing. Otherwise, it sets up the information required to establish an encrypted connection when you call mysql_real_connect(). In other words, to set up a secure connection, call mysql_ssl_set() first and then mysql_real_connect().

  mysql_ssl_set() always returns 0; any SSL setup errors result in an error at the time you call mysql_real_connect().

  key is the path to the key file. cert is the path to the certificate file. ca is the path to the certificate authority file. capath is the path to a directory of trusted certificates for certificate verification. cipher is a string listing the cipher or ciphers to use. Any parameter that is unused may be passed as NULL.

  For an example that shows how to write a client that can use secure connections, see Section 7.6, “Writing Clients That Include SSL Support.”

  mysql_ssl_set() requires some additional MySQL configuration ahead of time. See Section 13.5, “Setting Up Secure Connections Using SSL,” for the necessary background information.

G.3.3 Error-Reporting Routines

The functions in this section enable you to determine and report the causes of errors. The possible error codes and messages are listed in the `errmsg.h`, `mysqld_error.h`, and `sql_state.h` MySQL header files.

- ```
  unsigned int
  mysql_errno (MYSQL *conn);
  ```

  Returns an error code for the most recently invoked client library routine that returned a status. The value is zero if no error occurred and nonzero otherwise.

  ```
  if (mysql_errno (conn) == 0)
    printf ("Everything is okay\n");
  else
    printf ("Something is wrong!\n");
  ```

- ```
  const char *
  mysql_error (MYSQL *conn);
  ```

  Returns a null-terminated string that contains an error message for the most recently invoked client library routine that returned a status. The return value is the empty string if no error occurred (this is the zero-length string `""`, not a `NULL` pointer). Although normally you call `mysql_error()` after you already know an error occurred, the return value itself can be used to detect the occurrence of an error:

  ```
  const char *err = mysql_error (conn);
  if (err[0] == '\0')                  /* empty string? */
    printf ("Everything is okay\n");
  else
    printf ("Something is wrong!\n");
  ```

- ```
  const char *
  mysql_sqlstate (MYSQL *conn);
  ```

  Returns a null-terminated string that contains an SQLSTATE error code for the most recently invoked client library routine that returned a status. This code is a five-character string. SQLSTATE values are taken from the ANSI SQL and ODBC standards. A value that begins with `"00"` means "no error." A value of `"HY000"` means "general error." This value is used for those MySQL errors that have not yet been assigned more-specific SQLSTATE codes.

  ```
  if (strncmp (mysql_sqlstate (conn), "00", 2) == 0)
    printf ("Everything is okay\n");
  else
    printf ("Something is wrong!\n");
  ```
  ```

G.3.4 Statement Construction and Execution Routines

The functions in this section enable you to send SQL statements to the server. `mysql_hex_string()` and `mysql_real_escape_string()` help you construct statements by encoding characters that need special treatment. Unless you have enabled multiple-statement execution as described later in Section G.3.6, "Multiple Result Set Routines," each string sent to the server for execution must consist of a single SQL statement, and should not end with a semicolon character (';') or a `\g` sequence. The ';' and `\g` terminators are conventions of the `mysql` client program, not of the C client library.

- ```
  unsigned long
  mysql_hex_string (char *to_str,
                    const char *from_str,
                    unsigned long from_len);
  ```

  Encodes a string that may contain special characters so that it can be used in an SQL statement.

  The buffer to encode is specified as a counted string. `from_str` points to the buffer, and `from_len` indicates the number of bytes in it. `mysql_hex_string()` encodes every character in the buffer using two hexadecimal digits, writes the encoded result into the buffer pointed to by `to_str`, and adds a terminating null byte. `to_str` must point to an existing buffer that is at least `(from_len*2)+1` bytes long.

  `mysql_hex_string()` returns the length of the encoded string, not counting the terminating null byte.

  Here's an example:

  ```
  to_len = mysql_hex_string (to_str, "\0\\'\"\n\r\032", 7);
  printf ("to_len = %lu, to_str = %s\n", to_len, to_str);
  ```

  The example produces the following output:

  ```
  to_len = 14, to_str = 005C27220A0D1A
  ```

  The encoded string returned by `mysql_hex_string()` contains no internal null bytes but is null-terminated, so you can use it with functions such as `strlen()` or `strcat()`. Note that the result value is not by itself legal as a hexadecimal constant in an SQL statement. To construct a legal constant, either add `"0x"` at the beginning, or add `"X'"` at the beginning and `"'"` at the end.

- ```
  int
  mysql_query (MYSQL *conn, const char *stmt_str);
  ```

  Given an SQL statement specified as a null-terminated string, `mysql_query()` sends the statement to the server to be executed. The string should not contain binary data; in particular, it should not contain null bytes, because `mysql_query()` interprets the first one as the end of the statement. If your statement does contain binary data, use `mysql_real_query()` instead.

`mysql_query()` returns zero for success, and nonzero for failure. A successful statement is one that the server accepts as legal and executes without error. Success implies nothing about the number of rows affected or returned.

- ```
  unsigned long
  mysql_real_escape_string (MYSQL *conn,
                            char *to_str,
                            const char *from_str,
                            unsigned long from_len);
  ```

  Encodes a string that may contain special characters so that it can be used in an SQL statement, taking into account the current character set when performing encoding. Table G.7 lists the characters that are considered special and how they are encoded. (Note that the list does not include the SQL pattern characters, '`%`' and '`_`'.)

Table G.7 **`mysql_real_escape_string()` Character Encodings**

Special Character	Encoding
NUL (zero-valued byte)	`\0` (backslash-zero)
Backslash	`\\` (backslash-backslash)
Single quote	`\'` (backslash-single quote)
Double quote	`\"` (backslash-double quote)
Newline	`\n` (backslash-'`n`')
Carriage return	`\r` (backslash-'`r`')
Control-Z	`\Z` (backslash-'`Z`')

The only characters that MySQL itself requires to be escaped within a string are the backslash and the quote character that surrounds the string (either '`'`' or '`"`'). `mysql_real_escape_string()` escapes the others to produce strings that are easier to read and to process in log files.

The buffer to be encoded is specified as a counted string. `from_str` points to the buffer, and `from_len` indicates the number of bytes in it. `mysql_real_escape_string()` writes the encoded result into the buffer pointed to by `to_str` and adds a terminating null byte. `to_str` must point to an existing buffer that is at least `(from_len*2)+1` bytes long. (In the worst-case scenario, every character in `from_str` might need to be encoded as a two-character sequence, and you also need room for the terminating null byte.)

`mysql_real_escape_string()` returns the length of the encoded string, not counting the terminating null byte.

The resulting encoded string contains no internal null bytes but is null-terminated, so you can use it with functions such as `strlen()` or `strcat()`.
  ```

When you write literal strings in your program, take care not to confuse the lexical escape conventions of the C programming language with the encoding done by `mysql_real_escape_string()`. Consider the following example source code, and the output produced by it:

```
to_len = mysql_real_escape_string (conn, to_str, "\0\\\'\"\n\r\032", 7);
printf ("to_len = %lu, to_str = %s\n", to_len, to_str);
```

The example produces the following output:

```
to_len = 14, to_str = \0\\\'\"\n\r\Z
```

The printed value of `to_str` in the output looks very much like the string specified as the third argument of the `mysql_real_escape_string()` call in the original source code, but is in fact quite different.

- ```
  int
  mysql_real_query (MYSQL *conn,
                    const char *stmt_str,
                    unsigned long length);
  ```

  Given an SQL statement specified as a counted string, `mysql_real_query()` sends the statement to the server to be executed. The statement text is given by `stmt_str` and its length by `length`. The string may contain binary data (including null bytes).

  `mysql_real_query()` returns zero for success, and nonzero for failure. A successful statement is one that the server accepts as legal and executes without error. Success implies nothing about the number of rows affected or returned.

G.3.5 Result Set Processing Routines

When a statement produces a result set, the functions in this section enable you to retrieve the set and access its contents. The `mysql_store_result()` and `mysql_use_result()` functions create the result set and one or the other must be called before using any other functions in this section. Table G.8 compares the two functions.

Table G.8 **Comparison of `mysql_store_result()` and `mysql_use_result()`**

`mysql_store_result()`	`mysql_use_result()`
`mysql_store_result()` itself retrieves all rows in the result set.	`mysql_use_result()` initializes the result set, but defers row retrieval to `mysql_fetch_row()`.
Uses more memory; all rows are buffered on the client side.	Uses less memory; one row at a time is stored on the client side.
Slower due to overhead involved in allocating memory for the entire result set.	Faster because memory need be allocated only for the current row.
  ```

mysql_store_result()	**mysql_use_result()**
A NULL return from mysql_fetch_row() indicates the end of the result set, not an error.	A NULL return from mysql_fetch_row() indicates the end of the result set or an error, because communications failure can disrupt retrieval of the current row.
mysql_num_rows() can be called any time after mysql_store_result() has been called.	mysql_num_rows() returns a correct row count only after all rows have been fetched.
mysql_affected_rows() is a synonym for mysql_num_rows().	mysql_affected_rows() cannot be used.
Random access to result set rows is possible with mysql_data_seek(), mysql_row_seek(), and mysql_row_tell().	No random access into result set; rows must be processed in order as returned by the server. mysql_data_seek(), mysql_row_seek(), and mysql_row_tell() should not be used.
Tables are read-locked for no longer than necessary to fetch the data rows.	Tables can stay read-locked if the client pauses in mid-retrieval, locking out other clients attempting to modify the tables.
The max_length member of each result set MYSQL_FIELD structure indicates the longest value length in the corresponding result set column.	max_length has no meaningful value because it cannot be known until all rows are retrieved.

- my_ulonglong
  mysql_affected_rows (MYSQL *conn);

  Returns the number of rows affected by the most recent statement, or –1 if an error occurred. A return value of –1 can also indicate that you (erroneously) called mysql_affected_rows() after issuing a statement that returns rows but before retrieving the result set. This is the C API equivalent of the ROW_COUNT() SQL function; see the description of that function in Appendix C, "Operator and Function Reference."

  mysql_affected_rows() returns an unsigned value, so you can detect a negative return value only by casting the result to a signed value before performing the comparison:

  ```
  if ((long) mysql_affected_rows (conn) == -1)
    fprintf (stderr, "Error!\n");
  ```

  If you specified that the client should return the number of rows matched for UPDATE statements, mysql_affected_rows() returns that value rather than the number of rows modified. (MySQL does not update a row if the columns to be modified are the same as the new values.) This behavior can be selected by passing CLIENT_FOUND_ROWS in the flags argument to mysql_real_connect().

  mysql_affected_rows() returns a my_ulonglong value; see the note about printing values of this type in Section G.2.1, "Scalar Data Types."

- `void`
  `mysql_data_seek` `(MYSQL_RES *res_set, my_ulonglong row_num);`

  Seeks to the result set row indicated by `row_num`, which can range from 0 to `mysql_num_rows(res_set)`–1. The result is unpredictable if `row_num` is out of range.

  `mysql_data_seek()` requires that the entire result set has been retrieved into client memory, so you can use it only for a result set created by `mysql_store_result()`, not by `mysql_use_result()`.

  `mysql_data_seek()` differs from `mysql_row_seek()`, which takes a row offset value as returned by `mysql_row_tell()` rather than a row number.

- `MYSQL_FIELD *`
  `mysql_fetch_field` `(MYSQL_RES *res_set);`

  Returns a structure containing information (metadata) about a column in the result set. Following successful execution of a statement that returns rows, the first call to `mysql_fetch_field()` returns information about the first column. Subsequent calls return information about successive columns following the first, or `NULL` when no more columns are left.

  Related functions are `mysql_field_tell()` to determine the current column position, or `mysql_field_seek()` to select a particular column to be returned by the next call to `mysql_fetch_field()`.

  The following example seeks to the first `MYSQL_FIELD`, then fetches successive column information structures:

```
MYSQL_FIELD   *field;
unsigned int  i;

mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  printf ("column %u: name = %s max_length = %lu\n",
          i, field->name, field->max_length);
}
```

- `MYSQL_FIELD *`
  `mysql_fetch_fields` `(MYSQL_RES *res_set);`

  Returns an array of all column information structures for the result set. These may be accessed as follows:

```
MYSQL_FIELD   *field;
unsigned int  i;
```

```
field = mysql_fetch_fields (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  printf ("column %u: name = %s max_length = %lu\n",
          i, field[i].name, field[i].max_length);
}
```

Compare this to the example shown for `mysql_fetch_field()`. Note that although both functions return values of the same type, those values are accessed using slightly different syntax for each function. `mysql_fetch_field()` returns a pointer to a single field structure; `mysql_fetch_fields()` returns a pointer to an array of field structures.

- `MYSQL_FIELD *`
  `mysql_fetch_field_direct` `(MYSQL_RES *res_set, unsigned int col_num);`

  Given a column index, returns the information structure for that column. The value of `col_num` can range from 0 to `mysql_num_fields(res_set)`–1. The result is unpredictable if `col_num` is out of range.

  The following example accesses `MYSQL_FIELD` structures directly:

  ```
  MYSQL_FIELD   *field;
  unsigned int  i;

  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field_direct (res_set, i);
    printf ("column %u: name = %s max_length = %lu\n",
            i, field->name, field->max_length);
  }
  ```

- `unsigned long *`
  `mysql_fetch_lengths` `(MYSQL_RES *res_set);`

  Returns a pointer to an array of `unsigned long` values representing the lengths of the column values in the current result set row. You must call `mysql_fetch_lengths()` each time you call `mysql_fetch_row()` or your lengths will be out of synchrony with your data values.

  The length for `NULL` values is zero, but a zero length does not by itself indicate a `NULL` data value. An empty string also has a length of zero, so you must check whether the data value is a `NULL` pointer to distinguish between the two cases.

  The following example displays lengths and values for the current row, printing the word "NULL" if the value is `NULL`:

  ```
  unsigned long *length;

  length = mysql_fetch_lengths (res_set);
  ```

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  printf ("length is %lu, value is %s\n",
          length[i], (row[i] != NULL ? row[i] : "NULL"));
}
```

- `MYSQL_ROW`
  `mysql_fetch_row` `(MYSQL_RES *res_set);`

  Returns a pointer to the next result set row, represented as an array of strings (except that `NULL` column values are represented as `NULL` pointers). The *i*-th value in the row is the *i*-th member of the value array. Values of *i* range from 0 to `mysql_num_fields(res_set)`–1.

  Values for all data types, even numeric types, are returned as strings. To perform a numeric calculation with a value, you must convert it yourself—for example, with `atoi()`, `atof()`, or `sscanf()`.

  `mysql_fetch_row()` returns `NULL` when there are no more rows in the data set. (If you use `mysql_use_result()` to initiate a row-by-row result set retrieval, `mysql_fetch_row()` also returns `NULL` if a communications error occurred.)

  Data values are null-terminated, but you should not treat values that can contain binary data as null-terminated strings. Treat them as counted strings instead. To do this, you need the column value lengths, obtained by calling `mysql_fetch_lengths()`.

  The following code shows how to loop through a row of data values and determine whether each value is `NULL`:

  ```
  MYSQL_ROW       row;
  unsigned int    i;

  while ((row = mysql_fetch_row (res_set)) != NULL)
  {
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
      printf ("column %u: value is %s\n",
              i, (row[i] == NULL ? "NULL" : "not NULL"));
    }
  }
  ```

  To determine the types of the column values, use the column metadata stored in the `MYSQL_FIELD` column information structures, obtained by calling `mysql_fetch_field()`, `mysql_fetch_fields()`, or `mysql_fetch_field_direct()`.

- `unsigned int`
  `mysql_field_count` `(MYSQL *conn);`

Returns the number of columns for the most recent statement on the given connection. This function is normally used when `mysql_store_result()` or `mysql_use_result()` return `NULL`. `mysql_field_count()` tells you whether a result set should have been returned. A return value of zero indicates no result set and no error. (This happens for `INSERT` and `UPDATE` statements, for example.) A nonzero value indicates that columns were expected and that, because none were returned, an error occurred.

The following example illustrates how to use `mysql_field_count()` for error-detection purposes:

```
res_set = mysql_store_result (conn);
if (res_set)      /* a result set was returned */
{
  /* ... process rows here, then free result set ... */
  mysql_free_result (res_set);
}
else              /* no result set was returned */
{
  /*
   * does the lack of a result set mean that the statement didn't
   * return one, or that it should have but an error occurred?
   */
  if (mysql_field_count (conn) == 0)
  {
    /*
     * statement generated no result set (it was not a SELECT,
     * SHOW, DESCRIBE, etc.); just report rows-affected value.
     */
    printf ("Number of rows affected: %lu\n",
            (unsigned long) mysql_affected_rows (conn));
  }
  else  /* an error occurred */
  {
    printf ("Problem processing the result set\n");
  }
}
```

- `MYSQL_FIELD_OFFSET`
  `mysql_field_seek` `(MYSQL_RES *res_set, MYSQL_FIELD_OFFSET offset);`

  Seeks to the column information structure specified by `offset`. The next call to `mysql_fetch_field()` returns the information structure for that column. `offset` is *not* a column index; it is a `MYSQL_FIELD_OFFSET` value obtained from an earlier call to `mysql_field_tell()` or from `mysql_field_seek()`.

  To reset to the first column, use an `offset` value of zero.

- MYSQL_FIELD_OFFSET
  mysql_field_tell (MYSQL_RES *res_set);

  Returns the current column information structure offset. This value can be passed to mysql_field_seek().

- void
  mysql_free_result (MYSQL_RES *res_set);

  Deallocates the memory used by the result set. You must call mysql_free_result() for each result set you work with, typically generated by calling mysql_store_result() or mysql_use_result().

  For result sets generated by calling mysql_use_result(), mysql_free_result() automatically fetches and discards any unfetched rows.

- my_ulonglong
  mysql_insert_id (MYSQL *conn);

  Returns the value stored into an AUTO_INCREMENT column by the most recently executed INSERT or UPDATE statement on the given connection. This applies to an automatically generated AUTO_INCREMENT value or an explicit value stored in the column. (This differs from the LAST_INSERT_ID() SQL function, which returns only automatically generated values.)

  An INSERT can insert multiple rows, and different kinds of values can be stored in the AUTO_INCREMENT column (an automatically generated value, the result of LAST_INSERT_ID(*expr*), or an explicit value), so the precedence is as follows:

  - The first successfully inserted automatically generated value, if any.
  - The last LAST_INSERT_ID(*expr*) value, if any. (In this case, this value is returned even if the table contains no AUTO_INCREMENT column.)
  - The last explicit value.

  mysql_insert_id() returns zero if no statement has been executed or if the previous statement did not involve an AUTO_INCREMENT column or did not successfully insert any rows. (A zero return value is distinct from any valid AUTO_INCREMENT value because such values are positive.) The value of mysql_insert_id() is undefined if the previous statement produced an error.

  Call mysql_insert_id() immediately after issuing the statement that involves an AUTO_INCREMENT column. If you issue another statement before calling mysql_insert_id(), its value may be reset. Note that this behavior differs from the LAST_INSERT_ID() SQL function. mysql_insert_id() is maintained in the client and is set for each statement. The value of LAST_INSERT_ID() is maintained in the server and persists from statement to statement, until you generate another AUTO_INCREMENT value.

The value returned by `mysql_insert_id()` is session-specific and is not affected by `AUTO_INCREMENT` activity of other sessions.

`mysql_insert_id()` returns a `my_ulonglong` value; see the note about printing values of this type in Section G.2.1, "Scalar Data Types."

- `unsigned int`
  `mysql_num_fields` `(MYSQL_RES *res_set);`

  Returns the number of columns in the result set. `mysql_num_fields()` is often used to iterate through the columns of the current row of the set, as illustrated by the following example:

```
MYSQL_ROW     row;
unsigned int  i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    /* do something with row[i] here ... */
  }
}
```

- `my_ulonglong`
  `mysql_num_rows` `(MYSQL_RES *res_set);`

  Returns the number of rows in the result set. If you generate the result set with `mysql_store_result()`, you can call `mysql_num_rows()` any time thereafter:

```
if ((res_set = mysql_store_result (conn)) != NULL)
{
  /* mysql_num_rows() can be called now */
}
```

  If you generate the result set with `mysql_use_result()`, `mysql_num_rows()` doesn't return the correct value until you have fetched all the rows:

```
if ((res_set = mysql_use_result (conn)) != NULL)
{
  /* mysql_num_rows() cannot be called yet */
  while ((row = mysql_fetch_row (res_set)) != NULL)
  {
    /* mysql_num_rows() still cannot be called */
  }
  /* mysql_num_rows() can be called now */
}
```

  `mysql_num_rows()` returns a `my_ulonglong` value; see the note about printing values of this type in Section G.2.1, "Scalar Data Types."

- MYSQL_ROW_OFFSET
  mysql_row_seek (MYSQL_RES *res_set, MYSQL_ROW_OFFSET offset);

  Seeks to a particular result set row. mysql_row_seek() is similar to mysql_data_seek(), except that the offset value is not a row number, but either a MYSQL_ROW_OFFSET value obtained from a call to mysql_row_tell() or mysql_row_seek(), or zero to seek to the first row.

  mysql_row_seek() returns the previous row offset.

  mysql_row_seek() requires that the entire result set has been retrieved into client memory, so you can use it only for a result set created by mysql_store_result(), not by mysql_use_result().

- MYSQL_ROW_OFFSET
  mysql_row_tell (MYSQL_RES *res_set);

  Returns an offset representing the current row position in the result set. This is not a row number; the value may be passed only to mysql_row_seek(), not to mysql_data_seek().

  mysql_row_tell() requires that the entire result set has been retrieved into client memory, so you can use it only for a result set created by mysql_store_result(), not by mysql_use_result().

- MYSQL_RES *
  mysql_store_result (MYSQL *conn);

  Following successful statement execution, returns the result set and stores it in the client. Returns NULL if the statement returns no data or an error occurred. When mysql_store_result() returns NULL, call mysql_field_count() or one of the error-reporting functions to determine whether a result set was not expected or whether an error occurred. For an example, see the description of mysql_field_count().

  When you are done with the result set, pass it to mysql_free_result() to deallocate it.

  See the comparison of mysql_store_result() and mysql_use_result() in Table G.8.

- MYSQL_RES *
  mysql_use_result (MYSQL *conn);

  Following successful statement execution, initiates result set retrieval but retrieves no data rows itself. You must call mysql_fetch_row() to fetch the rows one by one. Returns NULL if the statement returns no data or an error occurred. When mysql_use_result() returns NULL, call mysql_field_count() or one of the error-reporting functions to determine whether a result set was not expected or whether an error occurred. For an example, see the description of mysql_field_count().

When you are done with the result set, pass it to `mysql_free_result()` to deallocate it. That is all that is necessary to finish statement processing, because `mysql_free_result()` automatically retrieves and discards any unfetched rows before releasing the result set.

See the comparison of `mysql_store_result()` and `mysql_use_result()` in Table G.8.

G.3.6 Multiple Result Set Routines

The routines in this section are used when multiple-statement execution capability is enabled. To use this capability, specify the `CLIENT_MULTI_STATEMENTS` flag when you open the connection with `mysql_real_connect()`. To enable multiple-statement execution for an already-open connection, call the `mysql_set_server_option()` function.

To send to the server the statements to be executed, call `mysql_real_query()` or `mysql_query()`. The statements should be sent as a single string, separated by semicolons.

For an example that shows how to use these routines, see Section 7.7, "Using Multiple-Statement Execution."

- ```
  my_bool
  mysql_more_results (MYSQL *conn);
  ```

  Returns nonzero if more statement results exist to be read and zero otherwise. To begin processing the next result, you must call `mysql_next_result()`.

- ```
  int
  mysql_next_result (MYSQL *conn);
  ```

  Initiates processing for the next result if one exists. After calling this function, process the result as you normally would for single-statement execution.

  `mysql_next_result()` returns zero if more results are available, –1 if not, and a value greater than zero if an error occurred.

G.3.7 Information Routines

These functions provide information about the client, server, protocol version, and the current connection. The values returned by most of these are retrieved from the server at connect time and stored within the client library.

- ```
  const char *
  mysql_character_set_name (MYSQL *conn);
  ```

  Returns a null-terminated string containing the name of the default character set for the given connection; for example, `"latin1"`.
  ```

- `const char *`
  `mysql_get_client_info` `(void);`

  Returns a null-terminated string describing the client library version; for example, `"5.5.30"`.

- `unsigned long`
  `mysql_get_client_version` `(void);`

  Returns an integer that indicates the client library version. The format of the return value is the same as for `mysql_get_server_version()`.

- `const char *`
  `mysql_get_host_info` `(MYSQL *conn);`

  Returns a null-terminated string describing the given connection, such as `"Localhost via Unix socket"`, `"host3.example.com via TCP/IP"`, `". via named pipe"`, or `"Shared memory"`.

- `unsigned int`
  `mysql_get_proto_info` `(MYSQL *conn);`

  Returns an integer indicating the client/server protocol version used for the given connection.

- `const char *`
  `mysql_get_server_info` `(MYSQL *conn);`

  Returns a null-terminated string describing the server version; for example, `"5.5.30-debug-log"`. The value consists of a version number, possibly followed by one or more suffixes. This is the same information returned by the `VERSION()` SQL function.

- `unsigned long`
  `mysql_get_server_version` `(MYSQL *conn);`

  Returns an integer that indicates the server version in *`XYYZZ`* format, where *`X.YY`* and *`ZZ`* represent the series number, and release within the series. For example, if the version is MySQL 5.5.30, this function returns `50530`.

- `const char *`
  `mysql_info` `(MYSQL *conn);`

  Returns a null-terminated string containing information about the effect of the most recently executed statement of the following types. The string format is given immediately following each statement:

  ```
  ALTER TABLE ...
    Records: 0 Duplicates: 0 Warnings: 0
  INSERT INTO ... SELECT ...
  ```

```
  Records: 0 Duplicates: 0 Warnings: 0
INSERT INTO ... VALUES (...),(...),...
  Records: 0 Duplicates: 0 Warnings: 0
LOAD DATA ...
  Records: 0 Deleted: 0 Skipped: 0 Warnings: 0
UPDATE ...
  Rows matched: 0 Changed: 0 Warnings: 0
```

The numbers vary according to the statement you execute, of course.

`mysql_info()` returns non-`NULL` for `INSERT INTO ... VALUES` only if the statement contains more than one value list. For statements not shown in the preceding list, `mysql_info()` always returns `NULL`.

The string returned by `mysql_info()` is in the language used by the server, so you can't necessarily count on being able to parse it by looking for certain words.

- ```
  const char *
  mysql_stat (MYSQL *conn);
  ```

  Returns a null-terminated string containing server status information, or `NULL` if an error occurred. The format of the string is subject to change. Currently it looks something like this:

  ```
  Uptime: 2153150 Threads: 6 Questions: 1306220 Slow queries: 271 Opens: 1260
  Flush tables: 1 Open tables: 64 Queries per second avg: 0.607
  ```

  Interpret these values as follows:

  - `Uptime`: The number of seconds the server has been running
  - `Threads`: The number of threads running in the server
  - `Questions`: The number of statements the server has executed
  - `Slow queries`: The number of statements that took longer to process than the time indicated by the server's `long_query_time` variable
  - `Opens`: The number of tables the server has opened
  - `Flush tables`: The number of `FLUSH`, `REFRESH`, and `RELOAD` statements that have been executed
  - `Open tables`: The number of tables the server has open
  - `Queries per second`: The ratio of `Questions` to `Uptime`

  Not coincidentally, the information returned by the `mysql_stat()` function is the same as that reported by the `mysqladmin status` command. (`mysqladmin` itself invokes this function to get the information.)
  ```

- ```
  unsigned long
  mysql_thread_id (MYSQL *conn);
  ```

  Returns the connection ID the server associates with the current session (the same value returned by the `CONNECTION_ID()` SQL function). You can use this value as an identifier for the `KILL` statement.

  Unless you know that automatic reconnect is disabled, do not invoke `mysql_thread_id()` until just before you need its value. If you retrieve the value and store it, the value may be incorrect when you use it later. This can happen if your session goes down and then is re-established (for example, with `mysql_ping()`) because the server assigns the new session a different identifier.

- ```
  unsigned int
  mysql_warning_count (MYSQL *conn);
  ```

  Returns the number of warnings generated by the most recent statement that generates such messages.

G.3.8 Transaction Control Routines

The functions in this section provide control over transaction processing.

- ```
  my_bool
  mysql_autocommit (MYSQL *conn, my_bool mode);
  ```

  Enables autocommit for the current session if `mode` is true (nonzero), and disables autocommit otherwise. Returns zero for success, and nonzero otherwise.

- ```
  my_bool
  mysql_commit (MYSQL *conn);
  ```

  Commits the current transaction. Returns zero for success, and nonzero otherwise. This function is affected by the value of the `completion_type` system variable.

- ```
  my_bool
  mysql_rollback (MYSQL *conn);
  ```

  Rolls back the current transaction. Returns zero for success, and nonzero otherwise. This function is affected by the value of the `completion_type` system variable.

G.3.9 Prepared Statement Routines

The routines in this section implement the binary client/server protocol for the prepared statement API. They are grouped into the following sections:

- Error-reporting routines to get error codes and messages
  ```

- Routines to construct SQL statements and send them to the server
- Result set processing routines to handle results from statements that return data

The initial implementation of prepared statements supported only the following statements: `CREATE TABLE`, `DELETE`, `DO`, `INSERT`, `REPLACE`, `SELECT`, `SET`, `UPDATE`, and most variations of `SHOW`. The list of supported statements has expanded since. See the MySQL Reference Manual for the current list.

G.3.9.1 Prepared Statement Error-Reporting Routines

The functions in this section enable you to determine and report the causes of prepared statement errors. To see the possible error codes and messages, check the `errmsg.h`, `mysqld_error.h`, and `sql_state.h` MySQL header files.

- ```
  unsigned int
  mysql_stmt_errno (MYSQL_STMT *stmt);
  ```

  Returns an error code for the most recently invoked prepared statement routine that returned a status. The value is zero if no error occurred and nonzero otherwise.

  ```
  if (mysql_stmt_errno (stmt) == 0)
    printf ("Everything is okay\n");
  else
    printf ("Something is wrong!\n");
  ```

- ```
  const char *
  mysql_stmt_error (MYSQL_STMT *stmt);
  ```

  Returns a null-terminated string that contains an error message for the most recently invoked prepared statement routine that returned a status. The return value is the empty string if no error occurred (this is the zero-length string `""`, not a `NULL` pointer). Although normally you call `mysql_stmt_error()` after you already know an error occurred, the return value itself can be used to detect the occurrence of an error:

  ```
  const char *err = mysql_stmt_error (stmt);
  if (err[0] == '\0')                   /* empty string? */
    printf ("Everything is okay\n");
  else
    printf ("Something is wrong!\n");
  ```

- ```
  const char *
  mysql_stmt_sqlstate (MYSQL_STMT *stmt);
  ```

  Returns a null-terminated string that contains an SQLSTATE error code for the most recently invoked prepared statement routine that returned a status. This code is a five-character string. SQLSTATE values are taken from the ANSI SQL and ODBC standards.
  ```

A value that begins with "00" means "no error." A value of "HY000" means "general error." This value is used for those MySQL errors that have not yet been assigned more-specific SQLSTATE codes.

```
if (strncmp (mysql_stmt_sqlstate (stmt), "00", 2) == 0)
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

G.3.9.2 Prepared Statement Construction and Execution Routines

The functions in this section enable you to send prepared SQL statements to the server. Each string must consist of a single SQL statement, and should not end with a semicolon character (';') or a \g sequence. ';' and \g are conventions of the mysql client program, not of the C client library.

For an example program that demonstrates many of these functions, see Section 7.8, "Using Server-Side Prepared Statements."

- ```
  my_bool
  mysql_stmt_bind_param (MYSQL_STMT *stmt, MYSQL_BIND *bind_array);
  ```

  Given a prepared statement handler, stmt, the mysql_stmt_bind_param() function binds a set of data values to the '?' placeholders in the statement. bind_array is the address of an array of MYSQL_BIND structures. There must be one structure in the array for each placeholder in the prepared statement. mysql_stmt_bind_param() returns zero if the bind operation was successful and nonzero otherwise.

- ```
  my_bool
  mysql_stmt_close (MYSQL_STMT *stmt);
  ```

  Closes the prepared statement handler, deallocates any resources associated with it, and cancels any results that might be pending for it. mysql_stmt_close() returns zero for success and nonzero otherwise.

  After closing a statement handler, do not attempt to use it for further operations.

  If the server still has prepared statements associated with a given client session when the session terminates, it discards those statements.

- ```
  MYSQL_STMT *
  mysql_stmt_init (MYSQL *conn);
  ```

  Allocates and initializes a MYSQL_STMT handler. Returns a pointer to the handler, or NULL if the handler could not be allocated.

  You should release the handler with mysql_stmt_close() when you are done with it.

- ```
  int
  mysql_stmt_execute (MYSQL_STMT *stmt);
  ```

Executes the prepared statement associated with the given statement handler. Returns zero if the statement was executed successfully and nonzero otherwise.

Before executing the statement, you must bind data values to it by calling `mysql_stmt_bind_param()` if the statement contains any '`?`' placeholders.

Following successful statement execution, process the statement result according to whether it returns a result set. For statements that return no result set, call `mysql_stmt_affected_rows()` to determine the number of rows inserted, deleted, or updated. For statements that return a result set, metadata becomes available and can be retrieved with `mysql_stmt_result_metadata()`. To fetch the results, use `mysql_stmt_bind_result()` to bind result buffers to columns, `mysql_stmt_fetch()` to retrieve rows, and `mysql_stmt_free_result()` to free the result set.

- ```
  int
  mysql_stmt_prepare (MYSQL_STMT *stmt,
                      const char *stmt_str,
                      unsigned long length);
  ```

  Given an SQL statement specified as a counted string, `mysql_stmt_prepare()` sends the statement to the server to be prepared for later execution and associates the statement handler, `stmt`, with the prepared statement. The statement text is given by `stmt_str` and its length by `length`. The string may contain binary data (including null bytes).

  `mysql_stmt_prepare()` returns zero for success, and nonzero for failure.

  The statement can contain '`?`' characters as parameter markers to indicate where data values should be bound to the statement when it is executed later.

- ```
  my_bool
  mysql_stmt_reset (MYSQL_STMT *stmt);
  ```

  Reset the prepared statement handler to the state that it has after calling `mysql_stmt_prepare()`.

- ```
  MYSQL_RES *
  mysql_stmt_result_metadata (MYSQL_STMT *stmt);
  ```

  After a successful call to `mysql_stmt_execute()`, `mysql_stmt_result_metadata()` returns metadata about the columns that result from the statement if it is one that returns a result set. The return value is a pointer to a `MYSQL_RES` result set structure. The structure is similar to that for a nonprepared statement that you obtain after invoking `mysql_store_result()`, except that it contains no data. To obtain information about the columns, pass the structure pointer to functions that take a `MYSQL_RES` argument such as `mysql_fetch_field()`, `mysql_fetch_fields()`, and `mysql_num_fields()`. When you are done with the structure, pass it to `mysql_free_result()` to dispose of it.

  If the prepared statement is not one that returns a result set, `mysql_stmt_result_metadata()` returns `NULL` to indicate that no metadata information is available.
  ```

- my_bool
  mysql_stmt_send_long_data (MYSQL_STMT *stmt,
  unsigned int param_num,
  const char *data,
  unsigned long length);

  This function can be used to send long BLOB or TEXT values a piece at a time. The param_num value indicates which parameter the call applies to, in the range from 0 to mysql_stmt_param_count(stmt)–1. data is a pointer to the buffer containing the data to send, and length indicates how many bytes to send.

G.3.9.3 Prepared Statement Result Set Processing Routines

When prepared statement execution produces a result set, the functions in this section enable you to retrieve the set and access its contents.

For an example program that demonstrates many of these functions, see Section 7.8, "Using Server-Side Prepared Statements."

- my_ulonglong
  mysql_stmt_affected_rows (MYSQL_STMT *stmt);

  This function is the prepared statement equivalent of mysql_affected_rows(), except that you call it after invoking mysql_stmt_execute(). For statements that return no result set, mysql_stmt_affected_rows() returns the number of rows inserted, deleted, or updated by executing the statement. For statements that return a result set, this function acts like mysql_num_rows().

  mysql_stmt_affected_rows() returns a my_ulonglong value; see the note about printing values of this type in Section G.2.1, "Scalar Data Types."

- my_bool
  mysql_stmt_attr_get (MYSQL_STMT *stmt,
  enum enum_stmt_attr_type attr_type,
  void *attr);

  Gets a prepared statement handler attribute. See the description of mysql_stmt_attr_set() for a description of the permitted attr_type attribute values. attr is a pointer to a variable into which the attribute value should be written.

  ```
  my_bool attr;
  if (mysql_stmt_attr_get (stmt, STMT_ATTR_UPDATE_MAX_LENGTH, &attr) == 0)
    printf ("Attribute gotten successfully\n");
  else
    printf ("Attribute not gotten successfully\n");
  ```

  mysql_stmt_attr_get() returns zero if the attribute was obtained successfully, and nonzero if the attribute type is unknown.

- ```
  my_bool
  mysql_stmt_attr_set (MYSQL_STMT *stmt,
                       enum enum_stmt_attr_type attr_type,
                       const void *attr);
  ```

  Sets a prepared statement handler attribute. `attr_type` indicates which attribute to set, and `attr` is a pointer to a variable that contains the value of the attribute.

  `attr_type` may be any of the following values:

  - `STMT_ATTR_UPDATE_MAX_LENGTH` controls whether `mysql_stmt_store_result()` calculates the `max_length` metadata value for result set columns. To enable or disable this attribute, pass an `attr` value that points to a `my_bool` that is set to true or false. By default, `max_length` calculation is disabled.
  - `STMT_ATTR_CURSOR_TYPE` indicates the type of cursor to use for the statement when `mysql_stmt_execute()` is called. `arg` points to an `unsigned long` that can be set to `CURSOR_TYPE_NO_CURSOR` (the default) or `CURSOR_TYPE_READ_ONLY`.
  - `STMT_ATTR_PREFETCH_ROWS` indicates now many rows to fetch at a time from the server when a cursor is used. `arg` points to an `unsigned long` that is set to the number of rows. The value should be at least 1 (the default).

  The following example enables `max_length` calculations for result sets:

  ```
  my_bool attr = 1;
  if (mysql_stmt_attr_set (stmt, STMT_ATTR_UPDATE_MAX_LENGTH, &attr) == 0)
    printf ("Attribute set successfully\n");
  else
    printf ("Attribute not set successfully\n");
  ```

  `mysql_stmt_attr_set()` returns zero if the attribute was set successfully, and nonzero if the attribute is unknown.

- ```
  my_bool
  mysql_stmt_bind_result (MYSQL_STMT *stmt, MYSQL_BIND *bind_array);
  ```

  Given a prepared statement handler, `stmt`, the `mysql_stmt_bind_result()` function specifies an array of `MYSQL_BIND` structures to be used for fetching result set rows. `bind_array` is the address of an array of `MYSQL_BIND` structures. There must be one structure in the array for each column in the result set. Each time you call `mysql_stmt_fetch()` to retrieve a result set row, the column values are returned in the `MYSQL_BIND` structures. `mysql_stmt_bind_result()` returns zero if the bind operation was successful and nonzero otherwise.

  You must bind the structures to the result set columns before retrieving rows, and the buffers pointed to by the structures must be large enough to store the retrieved values. It is permitted to call `mysql_stmt_bind_result()` during result set retrieval to bind columns to different `MYSQL_STMT` structures; `mysql_stmt_fetch()` uses the most recent bindings.

- void
  mysql_stmt_data_seek (MYSQL_STMT *stmt, my_ulonglong row_num);

  Seeks to the result set row indicated by row_num, which can range from 0 to mysql_stmt_num_rows(stmt)–1. The result is unpredictable if row_num is out of range.

  mysql_stmt_data_seek() requires that the entire result set has been retrieved into client memory, so you can use it only if you have called mysql_stmt_store_result() after executing the statement.

  mysql_stmt_data_seek() differs from mysql_stmt_row_seek(), which takes a row offset value as returned by mysql_stmt_row_tell() rather than a row number.

- unsigned int
  mysql_stmt_field_count (MYSQL_STMT *stmt);

  This function can be called after invoking mysql_stmt_prepare() with the statement handler. It returns the number of columns in the result set that will be generated when you execute the statement. If the statement will not produce a result set (for example, if it is an INSERT or UPDATE), mysql_stmt_field_count() returns zero.

- int
  mysql_stmt_fetch (MYSQL_STMT *stmt);

  After a successful call to mysql_stmt_execute() to execute a prepared statement that returns a result set, optionally followed by a call to mysql_stmt_store_result() to retrieve the result set into client memory, call mysql_stmt_fetch() to retrieve rows of the result. The buffers into which you want to fetch result columns first must be bound to MYSQL_BIND structures by calling mysql_stmt_bind_result().

  mysql_stmt_fetch() returns zero if a row was fetched successfully, MYSQL_NO_DATA if there are no more rows to fetch, MYSQL_DATA_TRUNCATED if data truncation occurred, and 1 if an error occurred. After a successful fetch, the column values are available in the MYSQL_BIND structures bound to the result. Truncation checking occurs by default; this can be changed by calling mysql_options() with the MYSQL_REPORT_DATA_TRUNCATION option.

- int
  mysql_stmt_fetch_column (MYSQL_STMT *stmt,
  MYSQL_BIND *bind,
  unsigned int col_num,
  unsigned long offset);

  This function fetches data for a single column from the current result set row. Returns zero for success and nonzero if an error occurred. bind is a MYSQL_BIND structure that should be set up to indicate the kind of value to retrieve, the buffer into which to retrieve it, and the length (amount) of the data to retrieve. col_num indicates which

column to fetch. Its value can range from 0 to `mysql_stmt_field_count(stmt)`–1. `offset` indicates the offset into the column value at which value retrieval should begin; 0 indicates the start of the value.

- `my_bool`
  `mysql_stmt_free_result` `(MYSQL_STMT *stmt);`

  Deallocates the memory used by the result set associated with the given statement handler. Returns zero for success and nonzero otherwise. Any unfetched rows are discarded. You must call `mysql_stmt_free_result()` for each result set generated by the handler.

- `my_ulonglong`
  `mysql_stmt_insert_id` `(MYSQL_STMT *stmt);`

  This function is the prepared-statement equivalent of `mysql_insert_id()`. It is used after you call `mysql_stmt_execute()` to execute a statement that generates an `AUTO_INCREMENT` value.

  `mysql_stmt_insert_id()` returns a `my_ulonglong` value; see the note about printing values of this type in Section G.2.1, “Scalar Data Types.”

- `int`
  `mysql_stmt_next_result` `(MYSQL_STMT *stmt);`

  This function is used following execution of a prepared `CALL` statement that invokes a stored procedure. The results from the procedure begin with result sets from statements such as `SELECT`, if there are any. Then, if the procedure has `OUT` or `INOUT` parameters, there is a single-row result set that contains their final values, in the order they appear in the procedure definition. Finally, there is always a status packet that signals the end of the results.

  Following retrieval of each part of the procedure results, invoke `mysql_stmt_next_result()` to determine whether there are any more results to process. It returns zero if more results are available, –1 if not, and a value greater than zero if an error occurred. For an example program that shows how this works, see Section 7.9, “Using Prepared `CALL` Support.”

  This function was introduced in MySQL 5.5.3.

- `my_ulonglong`
  `mysql_stmt_num_rows` `(MYSQL_STMT *stmt);`

  Returns the number of rows in the result set, if you have fetched the result into client memory by calling `mysql_stmt_store_result()`. Otherwise, `mysql_stmt_num_rows()` returns zero.

  `mysql_stmt_num_rows()` returns a `my_ulonglong` value; see the note about printing values of this type in Section G.2.1, “Scalar Data Types.”

- `int`
  `mysql_stmt_store_result` `(MYSQL_STMT *stmt);`

  Normally, result sets produced by executing a prepared statement are unbuffered and calling `mysql_stmt_fetch()` fetches rows one at a time from the server. Calling `mysql_stmt_store_result()` after executing the statement and before fetching the result set causes the result to be retrieved and buffered in client memory, so that calls to `mysql_stmt_fetch()` return rows from the buffered result. Calling `mysql_stmt_store_result()` also makes the result set "seekable", and enables you to use `mysql_stmt_data_seek()`, `mysql_stmt_row_seek()`, and `mysql_stmt_row_tell()`. These functions operate by positioning the row cursor of a result set buffered in client memory.

  For performance reasons, the `max_length` value in the result set metadata for each column is not calculated by default. To have this value calculated when you call `mysql_stmt_store_result()`, use the `mysql_stmt_set_attr()` function to enable the statement handler's `STMT_ATTR_UPDATE_MAX_LENGTH_FLAG` attribute.

  You can fetch rows of the result set by calling `mysql_stmt_fetch()` without calling `mysql_stmt_store_result()` first. In this case, rows are retrieved from the server one by one.

  Calling `mysql_stmt_store_result()` after executing a statement that produces no result set has no effect.

- `unsigned long`
  `mysql_stmt_param_count` `(MYSQL_STMT *stmt);`

  After a successful call to `mysql_stmt_prepare()` to prepare a statement, `mysql_stmt_param_count()` returns the number of parameters in the statement (indicated by '`?`' placeholders), and zero if there are none.

- `MYSQL_ROW_OFFSET`
  `mysql_stmt_row_seek` `(MYSQL_STMT *stmt, MYSQL_ROW_OFFSET offset);`

  Seeks to a particular result set row. `mysql_stmt_row_seek()` is similar to `mysql_stmt_data_seek()`, except that the `offset` value is not a row number, but either a `MYSQL_ROW_OFFSET` value obtained from a call to `mysql_stmt_row_tell()` or `mysql_stmt_row_seek()`, or zero to seek to the first row.

  `mysql_stmt_row_seek()` returns the previous row offset.

  `mysql_stmt_row_seek()` requires that the entire result set has been retrieved into client memory, so you can use it only if you have called `mysql_stmt_store_result()` after executing the statement.

- `MYSQL_ROW_OFFSET`
  `mysql_stmt_row_tell` `(MYSQL_STMT *stmt);`

Returns an offset representing the current row position in the result set. This is not a row number; the value may be passed only to `mysql_stmt_row_seek()`, not to `mysql_stmt_data_seek()`.

`mysql_stmt_row_tell()` requires that the entire result set has been retrieved into client memory, so you can use it only if you have called `mysql_stmt_store_result()` after executing the statement.

G.3.10 Administrative Routines

The functions in this section enable you to control aspects of server operation.

- ```
  int
  mysql_refresh (MYSQL *conn, unsigned int options);
  ```

  This function is similar in effect to the SQL `FLUSH` and `RESET` statements, except that you can tell the server to flush several kinds of things at once. `mysql_refresh()` returns zero for success, and nonzero for failure.

  The `options` value should be composed of one or more of the values shown in the following list. You must have the `RELOAD` privilege to perform these operations.

  - `REFRESH_GRANT`

    Reloads the grant table contents. This is equivalent to issuing a `FLUSH PRIVILEGES` statement.

  - `REFRESH_HOSTS`

    Flushes the host cache. This is equivalent to issuing a `FLUSH HOSTS` statement.

  - `REFRESH_LOG`

    Flushes the log files by closing and reopening them. This applies to whatever logs the server has open, and is equivalent to issuing a `FLUSH LOGS` statement.

  - `REFRESH_MASTER`

    Tells a replication master server to delete the binary log files listed in the binary log index file and to truncate the index. This is equivalent to issuing a `RESET MASTER` statement.

  - `REFRESH_SLAVE`

    Tells a replication slave server to forget its position in the master logs. This is equivalent to issuing a `RESET SLAVE` statement.

  - `REFRESH_STATUS`

    Reinitializes the status variables to zero. This is equivalent to issuing a `FLUSH STATUS` statement.
  ```

- REFRESH_TABLES

  Closes all open tables. This is equivalent to issuing a FLUSH TABLES statement.

- REFRESH_THREADS

  Flushes the thread cache. There is no equivalent SQL statement for this operation.

The option flags are bit values; combine them in additive fashion using either the | or the + operator. For example, the following expressions are equivalent:

```
REFRESH_LOG | REFRESH_TABLES
REFRESH_LOG + REFRESH_TABLES
```

- ```
  int
  mysql_set_server_option (MYSQL *conn,
                           enum enum_mysql_set_option option);
  ```

  Sets a server option and returns zero if the option was set successfully, or nonzero otherwise. The permitted options are MYSQL_OPTION_MULTI_STATEMENTS_ON or MYSQL_OPTION_MULTI_STATEMENTS_OFF, which enable or disable multi-statement execution capability, respectively.

  Enabling multiple-statement execution with MYSQL_OPTION_MULTI_STATEMENTS_ON does *not* also enable multiple result sets. This differs from the CLIENT_MULTI_STATEMENTS option to mysql_real_connect(), which also enables CLIENT_MULTI_RESULTS.

- ```
  int
  mysql_shutdown (MYSQL *conn, enum mysql_enum_shutdown_level level);
  ```

  Instructs the server to shut down. You must have the SHUTDOWN privilege to do this. The value of the second argument should be SHUTDOWN_DEFAULT. mysql_shutdown() returns zero for success, and nonzero for failure.

G.3.11 Threaded Client Routines

The routines in this section are used for writing multi-threaded clients.

- ```
  void
  mysql_thread_end (void);
  ```

  Frees any thread-specific variables initialized by mysql_thread_init(). To avoid memory leaks, you should call this function explicitly to terminate any threads that you create.

- ```
  my_bool
  mysql_thread_init (void);
  ```

Initializes thread-specific variables. This function should be called for any thread you create that will call MySQL functions. In addition, you should call `mysql_thread_end()` before terminating the thread.

- ```
  unsigned int
  mysql_thread_safe (void);
  ```

  Returns 1 if the client library was compiled to be thread-safe, and 0 otherwise.

G.3.12 Debugging Routines

These functions enable you to generate debugging information on either the client or server end of the connection. To use them, MySQL must have been built with debugging support.

- ```
  void
  mysql_debug (const char *debug_str);
  ```

  Performs a `DBUG_PUSH` operation using the string `debug_str`. The format of the string is described in the MySQL Reference Manual.

- ```
  int
  mysql_dump_debug_info (MYSQL *conn);
  ```

  Instructs the server to write debugging information to the log. You must have the `SUPER` privilege to do this. `mysql_dump_debug_info()` returns zero for success, and nonzero for failure.
  ```