C API

In this book,, we examine several different programming languages, Python, Java, Perl, PHP and C. Of these languages. C/C++ is by far the most challenging. With the other languages, your primary concern is the formulation of SQL, the passing of that SQL to a function call, and the manipulation of the resulting data. C adds the very complex issue of memory management into the mix.

MySQL provides C libraries that enable the creation of MySQL database applications. MySQL derives its API very heavily from mSQL, and older database server still used to back-end many Internet web sites. However, due to it's extensive development, MySQL is much more feature-rich than mSQL. In this chapter, we will examine the details of the MySQL C API by building an object-oriented C++ API than can be used to interface C++ programs to a MySQL database server.

The API

Whether you are using C or C++, the MySQL API is the gateway into the database. How you use it, however, can be very different depending on whether you are using C or the object-oriented features of C++. C database programming must be attacked in a linear fashion, where you step through your application process to understand where the database calls are made and where clean up needs to occur. Object-oriented C++, on the other hand, requires an OO interface into the API of your choice. The objects of that API can then take on some of the responsibility for database resource management.

Table 13-1 shows the function calls of the MySQL C API. We will go into the details of how these functions are used later in the chapter. Right now, you should just take a minute to see what is available to you. Naturally, the reference section lists each of these methods with detailed prototype information, return values, and descriptions.

Table 13-1. The C API for MySQL

MySQL
mysql_affected_rows()
mysql_close()
mysql_connect()

mysql_create_db()
mysql_data_seek()
mysql_drop_db()
mysql_eof()
mysql_error()
mysql_fetch_field()
mysql_fetch_lengths()
mysql_fetch_row()
mysql_field_count()
mysql_field_seek()
mysql_free_result()
mysql_get_client_info()
mysql_get_host_info()
mysql_get_proto_into()
mysql_get_server_info()
mysql_init()
mysql_insert_id()
mysql_list_dbs()
mysql_list_fields()
mysql_list_processes()
mysql_list_tables()
mysql_num_fields()
mysql_num_rows()
mysql_query()
mysql_real_query()
mysql_reload()
mysql_select_db()
mysql_shutdown()
mysql_stat()
mysql_store_result()
mysql_use_result()

You may notice that many of the function names do not seem directly related to access database data. In many cases, MySQL is actually only providing an API interface into database administration functions. By just reading the function names, you might have gathered that any database application you write might minimally look something like this:

Connect

Select DB
Query
Fetch row
Fetch field
Close

Example 13-1 shows as simple select statement that retrieves data from a MySQL database using the MySQL C API.

Example 13-1. A Simple Program that Select All Data in a Test Database and Displays the Data

```
#include <sys/time.h>
#include <stdio.h>
#include <mysql.h>
int main(char **args) {
    MYSQL_RES *result;
    MYSQL_ROW row;
    MYSQL *connection, mysql;
    int state;

    /* connect to the MySQL database at my.server.com */
    mysql_init(&mysql);
    connection = mysql_real_connect(&mysql,
           "my.server.com", 0, "db_test", 0, 0);
    /* check for a connection error */
    if (connection == NULL) {
           /* print the error message */
           printf(mysql_error(&mysql));
           return 1;
    }
    state = mysql_query(connection,
           "SELECT test_id, test_val FROM test");
    if (state != 0) {
           printf(mysql_error(connection));
           return 1;
    }
    /* must call mysql_store_result() before can issue
      any other query calls */
     result = mysql_store_result(connection);
     printf("Rows: %d\n", mysql_num_rows(result));
     /* process each row in the result set */
     while ( ( row = mysql_fetch_row(result)) != NULL ) {
         printf("id: %s, val: %s\n",
              (row[0] ? row[0] : "NULL"),
              (row[1] ? Row[1] : "NULL"));
      }
      /* free the result set */
      mysql_free_result(result);
      /* close the connection */
      mysql_close(connection);
      printf("Done.\n");
}
```

Of the #include files, both mysql.h and stdio.h should be obvious to you. The mysql.h header contains the prototypes and variables required for MySQL, and the stdio.h the prototype for printf(). The sys/time.h header, on the other hand, is not actually used by this application. It is instead required by the mysql.h header as the MySQL file uses

definitions from sys/time.h without actually including it. To compile this program using the GNU C compiler, use the command line:

```
gcc -L/usr/local/mysql/lib -I/usr/local/mysql/include -o select select.c\
       -lmysql -lnsl -lsocket
```

You should of course substitute the directory where you have MySQL installed for /usr/local/mysql in the preceding code.

The main() function follows the steps we outlines earlier – it connects to the server, selects a database, issues a query, processes the result sets, and cleans up the resources it used. We will cover each of these steps in detail as the chapter progresses. For now you should just take the time to read the code and get a feel for what it is doing.

As we discussed earlier in the book, MySQL supports a complex level of user authentication with user name and password combinations. The first argument of the connection API for MySQL is peculiar at first inspection. It is basically a way to track all calls not otherwise associated with a connection. For example, if you try to connect and the attempt fails, you need to get the error message associated with that failure. The MySQL mysql_error() function, however, requires a pointer to a valid MySQL connection. The null connection you allocate early on provides that connection. You must, however, have a valid reference to that value for the lifetime of your application—an issue of great importance in more structured environment than a straight "connect, query, close" application. The C++ examples later in the chapter will shed more light on this issue.

Object-oriented Database Access in C++

The C API works great for procedural C development. It does not, however, fit into the object-oriented world of C++ all that well. In order to demonstrate how the API works in real code, we will spend some time using it to create a C++ API for object-oriented database development.

Because we are trying to illustrate MySQL database access, we will focus on issues specific to MySQL and not try to create the perfect general C++ API. In the MySQL world, there are three basic concepts: the connection, the result set, and the rows in the result set. We will use the concepts as the core of the object model on which our library will be based. Figure 13-1 shows the objects in a UML diagram.

The Database Connection

Database access in any environment starts with the connection. We will start our object-oriented library by abstracting on that concept and creating a Connection object. A Connection object should be able to establish a connection to the server, select the appropriate database, send queries, and return results. Example 13-2 is the header file that declares the interface for the Connection object.

Example 13-2. The Connection Class Header


```
#ifndef l_connection_h
#define l_connection_h

#include <sys/time.h>
#include <mysql.h>

#include "result.h"

class Connection {
private:
    int affected_rows;
    MYSQL mysql;
    MYSQL *connection;

public:
    Connection(char *, char *);
    Connection(char *, char *, char *, char *);
    ~Connection();
    void Close();
    void Connect(char *host, char *db, char *uid, char *pw);
    int GetAffectedRows();
    char *GetError();
    int IsConnected();
    Result *Query(char *);
};

#endif // l_connection_h
```

The methods the Connection class will expose to the world are uniform no matter which database engine you use. Underneath the covers, however, the class will have private data members specific to the library you compile it against. For making a connection, the only distinct data members are those that represent a database connection. As we noted earlier, MySQL uses a MYSQL pointer with an addition MYSQL value to handle establishing the connection.

Connecting to the database

Any applications we write against this API now need only to create a new Connection instance using one of the associated constructors in order to connect to the database. Similarly, an application can disconnect by delete the Connection instance. In can even ruse a Connection instance by making direct calls to Close() and Connect(). Example 13-3 shows the implementation for the constructors and the Connect() method.

Example 13-3. Connecting to MySQL Inside the Connection Class

```
#include "connection.h"
Connection::Connection(char *host, char *db) {
    connection = (MYSQL *)NULL;
    Connect( host, db, (char *)NULL, (char *)NULL);
}
Connection::Connection(char *host, char *db, char *uid, char *pw) {
    connection = (MYSQL *)NULL;
    Connect( host, db, uid, pw );
}

void Connection::Connect(char *host, char *db, char *uid, char *pw) {

    if (IsConnected() ) {
```



```
            throw "Connection has already been established.";
    }

    mysql_init(&mysql);
    connection = mysql_real_connect(&mysql, host,
                            uid, pw,
                            db, 0, 0);
    if (!IsConnected() ) {
            throw GetError();
    }
}
```

The two constructors are designed to allow for different connection needs. In most circumstances, the username and password will be used and the four-argument constructor is called for. In some cases, however, it is not necessary to supply a username and password explicitly, and the two-argument constructor can be used. The actual database connectivity occurs in the Connect() method.

The Connect() method encapsulates all steps required for a connection. The mainly entails a call to mysql_real_connect(). If this fails, Connect() throws and exception.

Disconnecting from the database

A Connection's other logic function is to disconnect from the database and free up the resources it has hidden from the application. The functionality occurs in the Close() method. Example 13-4 provides all of the functionality for disconnecting from MySQL.

Example 13-4. Freeing up Database Resources

```
Connection::~Connection() {
    if (IsConnected()) {
            Close();
    }
}

void Connection::Close() {
    if ( !IsConnected() ) {
            return;
    }
    mysql_close(connection);
    connection = (MYSQL *)NULL;
}
```

The mysql_close() function frees up the resources associated with connections to MySQL.

Making Calls to the database

In between opening a connection and closing it, you generally want to send statements to the database. The Connection class accomplished this via a Query() method that takes a SQL statement as an argument. If the statement was a query, it return an instance of the Result class from the object model in Figure 13-1. If, on the other hand, the statement was an update, the method will return NULL and set the affected_rows value to the number of rows affected by the update. Example 13-5 shows how the Connection class handles queries against MySQL databases.

Example 13-5. Querying the Databases

```
Result *Connection::Query(char *sql) {
    T_RESULT *res;
    int state;

    // if not connectioned, there is nothing we can do
    if ( !IsConnected() ) {
            throw "Not connected.";
    }
    // execute the query
    state = mysql_query(connection, sql);
    // an error occurred
    if ( state > 0 ) {
            throw getError();
    }
    // grab the result, if there was any
    res = mysql_store_result(connection);
    // if the result was null, if was an update or an error occurred
    if (res == (T_RESULT *)NULL ) {
            // field_count != 0 means an error occurred
            int field_count = mysql_num_fields(connection);

            if (field_count != 0) {
                    throw GetError();
            } else {
                    // store the affected rows
                    affected_rows = mysql_affected_rows(connection);
            }

            // return NULL for updates
            return (Result *)NULL;
    }
    // return a Result instance for queries
    return new Result(res);
```

The first part of a making-a-database call is calling mysql_query() with the SQL to be executed. Both APIs return a nonzero on error. The next step is to cal mysql_store_result() to check if results were generated and make those result usable by your application.

The mysql_store_result() function is used to place the results generated by a query into storage managed by the application. To trap errors from this call, you need to wrapper mysql_store_result() with some exception handling. Specifically, a NULL return value from mysql_store_result() can mean either the call was a non-query or an error occurred in storing the results. A call to mysql_num_fields() will tell you which is in fact the case. A field count not equal to zero means an error occurred. The number of affected rows, on the other hand, may be determined by a call to mysql_affected_rows(). *

* One particular situation behaves differently. MySQL is optimized for cases where you delete all records in a table. This optimization incorrectly causes some versions of MySQL to return 0 for a mysql_affacted_rows() call.

Other Connection Behaviors

Throughout the Connection class are calls to two support methods, IsConnected() and GetError(). Testing for connection status is simple – you just check the value of the connection attribute. If should always be non-NULL for MySQL. Error messages, on the other hand, require some explanation.

Because MySQL is a multithreaded application, it needs to provide threadsafe access to any error messages. If manages to make error handling work in a multithreaded environment by hiding error messages behind the mysql_error() function. Example 13-6 shows MySQL error handling in the GetError() method as well as a connection testing in IsConnected().

Example 13-6. Reading Error and Other Support Tasks of the Connection Class

```
int Connection::GetAffectedRows() {
    return affected_rows;
}

char *Connection::GetError() {
    if (IsConnected() ) {
            return mysql_error(connection);
    } else {
            return mysql_error(&mysql);
    }
}

int Connection::IsConnected() {
    return !(!connection);
}
```

Error Handling Issues

While the error handling above is rather simple because we have encapsulated it into a simple API call in the Connection class, you should be aware of some potential pitfalls you can encounter.

MySQL manages the storage of error messages inside the API. Because you have no control over that storage, you may run into another issue regarding the persistence of error messages. In our C++ API, we are handling the error messages right after they occur – before the application makes any other database calls. If we wanted to move on with other processing before dealing with an error message, we would need to copy the error message into storage manage by our application.

Result Sets

The Result class is an abstraction on the MySQL result concept. Specifically, should provide access to the data in a result set as well as the meta-data surrounding the result set. According to the object model from Figure 13-1, our Result class will support looping through the rows of a result set and getting the row count of a result set. Example 13-7 is the header file for the Result class.

Exmaple XX-7. The interface for a Result Class in result.h

```
#ifndef l_result_h
#define l_result_h
#incude <sys/time.h>
#include <mysql.h>
#include "row.h"

class Result {
    private:
            int row_count;
            T_RESULT *result;
            Row *current_row;

    public:
            Result(T_RESULT *);
            ~Result();'

            void Close();
            Row *GetCurrentRow();
            int GetRowCount();
            int Next();
    };

#endif // l_result_h
```

Navigating results

Our Result class enables a developer to work through a result rest one row at a time. Upon getting a Result instance from a call to Query(), an application should call Next() and GetCurrentRow() in succession until Next() return 0. Example XX-8 shows how this functionality looks for MySQL

Example XX-8. Result Set Navigation

```
int Result::Next() {
    T_ROW row;

    if( result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }

    row = mysql_fetch_row(result);

    if (!row) {
            current_row = (Row *)NULL;
            return 0;
    } else {
            current_row = new Row(result, row);
            return 1;
    }
}

Row *Result::GetCurrentRow() {
    if( result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }
    return current_row;
}
```

The row.h header file in Example 13-10 defines T_ROW and T_RESULT as abstractions of the MySQL-specific ROW and RESULT structures. The functionality for moving to the next row is simple. You simply call mysql_fetch_row(). If the call returns NULL, there are no more rows left to process.

In an object-oriented environment, this is the only kind of navigation you should ever use. A database API in an OO world exists only to provide you access to the data – not as a tool for the manipulation of that data. Manipulation should be encapsulated in domain objects. Not all applications, however, are object-oriented applications. MySQL provides a function that allows you to move to specific rows in the database. That method is mysql_data_seek().

Cleaning up and row count

Database applications need to clean up after themselves. In talking about the Connection class, we mentioned how the result sets associated with a query are moved into storage managed by the application. The Close() method in the Result class frees the storage associated with that result. Example 13-9 shows how to clean up results and get a row count for a result set.

Example 13-9. Clean up and Row Count

```
void Result::Close() {
    if (result == (T_RESULT *)NULL) {
            return;
    }

    mysql_free_result(result);
    result = (T_RESULT *)NULL;
}

int Result::GetRowCount() {
    if (result == (T_RESULT *)NULL ) {
            throw "Result set closed.";
    }
    if ( row_count > -1 ) {
            return row_count;
    } else {
            row_count = mysql_num_rows(result);
    }
}
```

Rows

An individual row from a result set is represented in our object model by the Row class. The Row class enables an application to get at individual fields in a row. Example 13-10 shows the declaration of a Row class.

Example 13-10. The Row Class from row.h

```
#ifndef l_row_h
#define l_row_h

#include <sys/types.h>
```



```
#include <mysql.h>
#define T_RESULT MYSQL_RES
#define T_ROW    MYSQL_ROW

class Row {
   private:
           T_RESULT *result;
           T_ROW fields;

   public:
           Row(T_RESULT *, T_ROW);
           ~Row();

           char *GetField(int);
           int GetFieldCount();
           int IsClosed();
           void Close();
};
```

Both APIs have macros for datatypes representing a result set and a row within that result set. In both APIs, a row is really nothing more than an array of strings containing the data from that row. Access to that data is controlled by indexing on that array based on the query order. For example, if your query was SELECT user_id, password FROM users, then index 0 would contain the user ID and index 1 the password. Our C++ API makes this indexing a little more user friendly. GetField(1) will actually return the first field, or fields[0]. Example 13-11 contains the full source listing for the Row class.

Example 13-11. The implementation of the Row Class

```
#include <malloc.h>

#include "row.h"

Row::Row(T_RESULT *res, T_ROW row) {
    fields = row;
    result = res;
}

Row::~Row() {
    if (!IsClosed()) {
           Close();
    }
}

void Row::Close() {
    if (! IsClosed() ) {
           throw "Row closed.";
    }
    fields = (T_ROW)NULL;
    result = (T_RESULT *)NULL;
}

int Row::GetFieldCount() {
    if ( IsClosed() ) {
           throw "Row closed.";
    }

           return mysql_num_fields(result);
}
```



```
// Caller should be prepared for a possible NULL
// return value from this method.
char *Row::GetField(int field) {
    if ( IsClosed() ) {
            throw "Row closed.";
    }

    if ( field < 1 || field > GetFieldCount() ) {
            throw "Field index out of bounds.";
    }
    return fields[field-1];
}

int Row::IsClosed() {
    return (fields == (T_ROW)NULL);
}
```

As example application using these C++ classes is packages with the examples from this book.

